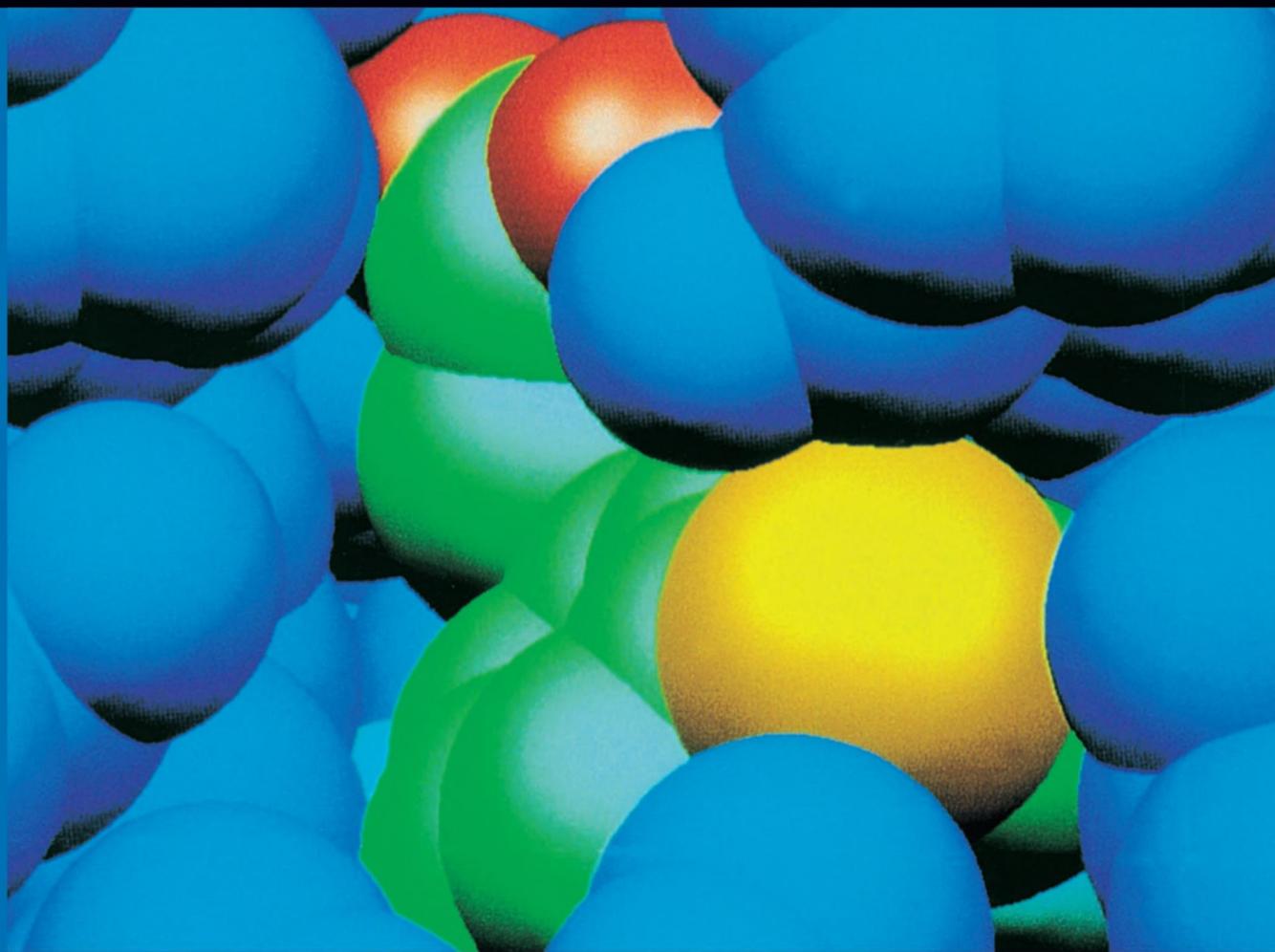
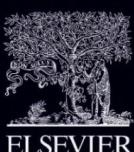


THEORETICAL AND COMPUTATIONAL CHEMISTRY 23



Python for Quantum Chemistry

A Full Stack Programming Guide



Qiming Sun

Theoretical and Computational Chemistry

Python for Quantum Chemistry

A Full Stack Programming Guide

Volume 23

Qiming Sun
Quantum Engine LLC
Lacey, WA, United States



ELSEVIER

Elsevier
Radarweg 29, PO Box 211, 1000 AE Amsterdam, Netherlands
125 London Wall, London EC2Y 5AS, United Kingdom
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Copyright © 2025 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

For accessibility purposes, images in electronic versions of this book are accompanied by alt text descriptions provided by Elsevier. For more information, see <https://www.elsevier.com/about/accessibility>.

Publisher's note: Elsevier takes a neutral position with respect to territorial disputes or jurisdictional claims in its published content, including in maps and institutional affiliations.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-443-23837-6

ISSN: 1380-7323

For information on all Elsevier publications
visit our website at <https://www.elsevier.com/books-and-journals>

Publisher: Peter Llewelyn
Acquisitions Editor: Charles Bath
Editorial Project Manager: Emerald Li
Production Project Manager: Paul Prasad Chandramohan
Cover Designer: Miles Hitchen

Typeset by VTeX



Working together
to grow libraries in
developing countries

www.elsevier.com • www.bookaid.org

Contents

Preface	xv
List of acronyms	xxiii

PART 1 Python ecosystems and tools for scientific computing

CHAPTER 1 Python programming environment.....	3
--	----------

1.1 Python package system.....	4
1.1.1 Python runtime	4
1.1.2 Package, sub-package, and module	6
1.1.3 Python program as CLI tools	12
1.1.4 Virtual environment.....	16
1.1.5 Conda.....	17
1.2 Python programs in Docker	18
1.3 Managing a Python project.....	23
1.3.1 Version control	23
1.3.2 Testing a Python program	29
1.3.3 Coding style and static checks	31
1.3.4 Releasing a Python package	32
1.3.5 CI and DevOps workflow.....	38
1.4 Modular design and object-oriented programming	42
1.4.1 Method resolution order in multiple inheritances.....	43
1.4.2 Mixins.....	47
1.5 Working with IPython and Jupyter	49
1.5.1 Startup configurations	49
1.5.2 Extensions	50
1.5.3 Magics	50
1.5.4 Remote execution	53
Summary.....	53
References.....	54

CHAPTER 2 Data processing.....	57
---------------------------------------	-----------

2.1 Vectorized data processing with NumPy	58
2.1.1 The basics of NumPy	58
2.1.2 Universal functions (ufunc)	60
2.1.3 Inplace operations	61
2.1.4 Broadcasting	62

2.1.5	Fancy indexing	64
2.1.6	Mask array	67
2.1.7	Data structure of NumPy ndarray	70
2.1.8	Array views	75
2.1.9	The <code>reshape</code> function	78
2.2	Data types in NumPy	79
2.2.1	Type casting	79
2.2.2	Scalar type and zero-dimensional array	81
2.2.3	Infinity (<code>inf</code>) and not-a-number (<code>nan</code>)	83
2.2.4	Data with high precision	85
2.2.5	Structured array	90
2.3	Data with labels: Pandas	91
2.3.1	Pandas data objects	92
2.3.2	Broadcasting	96
2.3.3	Indexing	99
2.3.4	<code>query</code> and <code>eval</code> methods	104
2.3.5	Altering structure of <code>DataFrame</code>	105
2.3.6	Data types	110
2.3.7	Missing data	111
2.3.8	Grouping and aggregation	112
2.3.9	View and copy	114
	Summary	116
	References	116
CHAPTER 3	Visualization	119
3.1	Matplotlib	119
3.2	Pandas visualization	124
3.3	Mayavi for 3D plotting	127
3.4	Quantum chemistry visualization	135
3.4.1	Jinja template	136
3.4.2	Molden format	137
3.4.3	Cube format	140
	Summary	142
	References	142
CHAPTER 4	Scientific computing tools	145
4.1	Linear algebra	145
4.2	Sparse matrices	147
4.2.1	Storage formats	148
4.2.2	Linear algebra for sparse matrix	152
4.2.3	Linear operator	153

4.3	Tensor contractions.....	154
4.4	Discrete Fourier transforms	157
4.4.1	SciPy FFT	158
4.4.2	Intel FFT.....	160
4.4.3	PyFFTW	160
4.4.4	FFT performance benchmark.....	162
4.4.5	How to choose FFT libraries?	169
	Summary.....	170
	References.....	171
CHAPTER 5	Meta-programming and non-numerical computation	173
5.1	Code generation.....	173
5.1.1	eval and exec	174
5.1.2	Composing code programmatically	175
5.1.3	Manipulating classes and functions at runtime	175
5.1.4	Python AST	176
5.2	Symbolic computation.....	181
5.3	Automatic differentiation.....	186
5.3.1	PyTorch	187
5.3.2	JAX	190
	Summary.....	193
	References.....	193
CHAPTER 6	Input and output.....	195
6.1	Serialization.....	196
6.1.1	Pickle	197
6.1.2	JSON	199
6.1.3	YAML.....	201
6.2	File I/O	202
6.2.1	The npy format	203
6.2.2	HDF5 storage.....	205
6.2.3	Memory mapping.....	207
6.3	I/O buffering	207
6.4	In-memory I/O	208
6.5	Network I/O.....	209
6.5.1	Network requests over HTTP.....	210
6.5.2	REST API	212
6.5.3	RPC over HTTP	213
6.5.4	gRPC	216
6.5.5	Apache arrow.....	218

Summary.....	221
References.....	221
CHAPTER 7 Working with cloud.....	223
7.1 Utilizing cloud computing.....	224
7.1.1 Comparison between cloud and supercomputers	224
7.1.2 Designing a workflow	225
7.1.3 Computing resources	227
7.1.4 Communications among cloud services.....	234
7.1.5 Function-as-a-Service	241
7.2 Distributed job executors	246
7.2.1 Celery	247
7.2.2 Dask	253
7.2.3 Ray	256
Summary.....	259
References.....	260
PART 2 High performance computing with Python	
CHAPTER 8 Foreign language interfaces.....	265
8.1 Inter-process communication interfaces.....	266
8.2 C/C++ interfaces.....	267
8.2.1 Data types and type conversion	268
8.2.2 Cython	275
8.2.3 pybind11	277
8.2.4 Compiling C++ code into a Python module	278
8.3 Foreign function interfaces	283
8.3.1 Ctypes	283
8.3.2 CFFI	289
8.3.3 Memory leaks	292
8.3.4 Duplicated function names in extensions	294
8.4 Fortran interfaces	296
8.4.1 ctypes for Fortran	296
8.4.2 f2py compiler.....	299
8.5 Rust interfaces.....	301
Summary.....	303
References.....	303
CHAPTER 9 Program performance optimization	305
9.1 Principles for performance optimization	306

9.1.1	Cost comparison between Python and C/C++ operations	307
9.1.2	Hardware and operating system overhead	311
9.1.3	Latency and throughput	313
9.1.4	Strategies for optimizing latency and throughput	315
9.1.5	Computation bound and I/O bound	316
9.1.6	Instruction level parallelism	324
9.2	Profiling	326
9.2.1	Benchmark tests	327
9.2.2	Python built-in profiling tools	327
9.2.3	Line profiler	330
9.2.4	Sampling profiler	332
9.2.5	Perf	334
9.3	Python level optimization	338
9.3.1	The <code>dis</code> module	338
9.3.2	Performance-friendly Python code	340
9.3.3	Utilizing tensor operations	344
9.3.4	Optimizing tensor indexing efficiency	346
9.4	Compiling Python code	347
9.4.1	Numba	348
9.4.2	Cython	355
9.4.3	Pythran	359
9.4.4	Comparison of Cython, Pythran, and Numba	360
9.5	Optimization with compiled languages	362
9.5.1	GCC compiler	363
9.6	Optimization for I/O	365
9.6.1	Storage layout	366
9.6.2	Compressing data	370
9.6.3	Overlapping computation and I/O	372
9.7	Precomputation and memoization	374
9.7.1	LRU cache	375
9.7.2	Functional programming	377
9.7.3	Dynamic programming	380
9.8	Optimization with lazy evaluation	384
Summary	390	
References	393	
CHAPTER 10	Parallel computation	395
10.1	Multithreading	396
10.1.1	The <code>threading</code> module	396
10.1.2	<code>ThreadPoolExecutor</code>	398

10.2	Lock and thread synchronization	401
10.2.1	Mutex	401
10.2.2	Event and condition variable	402
10.3	Producer-consumer model.....	406
10.4	Pipeline executor.....	408
10.5	Asynchronous program	416
10.5.1	Similarity between asynchronous programming and lazy evaluation	416
10.5.2	Asynchronous program with coroutines and <code>asyncio</code> ..	417
10.6	Multiprocessing	419
10.6.1	Data communication in <code>multiprocessing.Process</code>	419
10.6.2	Data communication in <code>ProcessPoolExecutor</code>	422
10.6.3	Locks in the multiprocessing program	425
10.7	Inter-process communication	426
10.8	OpenMP	435
10.9	MPI	436
10.9.1	Naming conventions in MPI4Py	437
10.9.2	SPMD MPI programs in Python.....	439
10.9.3	Integrating MPI into serial program	440
10.9.4	Shared memory	445
Summary.....	446	
References.....	447	
CHAPTER 11	GPU programming	449
11.1	Configuring GPU runtime environment	449
11.2	Architecture-independent optimization	452
11.2.1	CuPy.....	453
11.2.2	PyTorch.....	454
11.2.3	JAX	456
11.3	Architecture-aware optimization.....	456
11.3.1	Pinned memory	457
11.3.2	CUDA stream	458
11.4	Custom GPU kernels in Python	460
11.4.1	Kernel in CUDA code.....	460
11.4.2	Numba CUDA JIT	464
11.4.3	CuPy custom kernel	466
11.4.4	Integrating CUDA kernels using <code>ctypes</code>	468
Summary.....	472	
References.....	473	

PART 3 Quantum chemistry applications with Python

CHAPTER 12 Integral evaluation.....	477
12.1 Analytical integral evaluation for Gaussian type orbitals	478
12.1.1 Data structure for GTO basis	478
12.1.2 Basic types for analytical GTO integrals	484
12.1.3 Recurrence relations and the dynamic programming implementation	486
12.1.4 Converting recursion to iteration	502
12.1.5 Eliminating branches and adjusting iterations.....	510
12.1.6 Optimization with Numba JIT compilation.....	515
12.1.7 Optimization with Cython compilation	519
12.1.8 Optimization with meta-programming techniques	520
12.1.9 Performance benchmark.....	529
12.1.10 Other performance factors	531
12.2 Numerical integration.....	537
12.2.1 Gaussian quadrature	538
12.2.2 Generating quadrature roots and weights.....	538
12.2.3 Approximating quadratures.....	544
12.2.4 Numerical integration with Fourier transform	549
12.3 Integral transformation	552
Summary.....	557
References.....	557
CHAPTER 13 Mean-field methods	559
13.1 The self-consistency iteration program.....	560
13.2 Coulomb and exchange matrices.....	567
13.2.1 Integral screening for Coulomb and exchange matrices.....	572
13.2.2 J-engine.....	573
13.3 Integrals for exchange-correlation functionals	576
13.4 DIIS	584
13.5 Design of Python classes for mean-field methods	589
13.5.1 New mean-field classes via class inheritances	590
13.5.2 Mean-field classes via the visitor pattern	590
13.5.3 Dynamic patches to mean-field objects	592
13.5.4 Dynamic mean-field classes	594
13.6 Speeding up mean-field calculations.....	597
13.6.1 Cholesky decomposition and resolution of identity methods.....	598
13.6.2 Improving convergence using DIIS	604

13.6.3	Improving initial guess	607
Summary.....	609	
References.....	610	
CHAPTER 14	Post Hartree-Fock I: full configuration interaction..	613
14.1	Theory of full configuration interaction	614
14.2	The string representation.....	615
14.3	Davidson diagonalization	617
14.4	Direct CI algorithm	623
14.5	Optimizing tensor contractions.....	625
14.6	Optimization for memory efficiency	628
14.7	Parallel computation	633
Summary.....	639	
References.....	640	
CHAPTER 15	Post Hartree-Fock II: coupled cluster	641
15.1	Coupled cluster theory	642
15.2	CCD program.....	643
15.3	Optimizing data transferring	647
15.4	Just-in-time compilation for I/O readahead	652
15.5	Symbolic programming for coupled cluster theory.....	656
15.5.1	Defining fundamental symbolic elements	657
15.5.2	Implementing computation rules	658
15.5.3	Evaluating expressions symbolically	663
15.5.4	Generating CCD equations symbolically.....	665
15.5.5	Applying permutation symmetry to CCD tensors	666
Summary.....	672	
References.....	672	
CHAPTER 16	Molecular properties	675
16.1	Molecular properties for single-particle operators	675
16.2	Analytical nuclear gradients	677
16.2.1	Basic equations of Hartree-Fock derivatives.....	677
16.2.2	Derivatives of integrals	679
16.2.3	Nuclear gradients for RHF energy	681
16.2.4	Nuclear gradients with finite difference	683
16.2.5	First order molecular orbitals.....	684
16.3	Krylov subspace linear equation solver	687
16.4	Nuclear gradients with automatic differentiation	690
16.4.1	Differentiable Python class	700
16.4.2	Implicit differentiation	703

16.4.3 Higher-order derivatives	707
16.4.4 Conversion between JVP and VJP	709
Summary.....	710
References.....	711
Index	713

Preface

Python has been widely adopted in scientific computation and numerical data processing. Despite its popularity, certain stereotypes persist regarding Python's capability in quantum chemistry computation. Some believe that Python is inefficient, only suitable for toy projects, and cannot compete with compiled languages. Some argue that Python's parallel performance is poor. These views are true and false.

Due to its interpreted nature, Python requires additional workloads to execute elemental operations, making it slower in some scenarios than compiled languages such as C++ and Fortran. However, interpreted execution is not the only factor that influences the performance of quantum chemistry programs. In many scenarios, hardware, the operating system, and characteristics of the programming language can also impact the execution speed of programs. Furthermore, in computationally intensive scenarios, Python offers extensive libraries and tools that can enhance the performance of Python programs, potentially making them as fast as those written in compiled languages. In this book, we will analyze various factors that affect the performance of Python programs and demonstrate techniques to optimize Python programs. We hope that by reading this book, readers will change their view that *Python is slow* and will be able to analyze and utilize various Python tools to write efficient quantum chemistry programs.

Learning the basics of Python programming is generally easier than learning most other programming languages. However, developing a practical computation project in Python can still pose significant challenges. On one hand, one would face technical challenges in program development, such as managing Python packages, ensuring maintainability, optimizing performance, and utilizing third-party libraries. On the other hand, developers may encounter techniques not covered in standard textbooks, such as concurrent programming, heterogeneous programming, automatic differentiation, dynamic programming, symbolic programming, and cloud computing. These technical challenges are quite different from the basic programming concepts found in introductory textbooks.

Many technical issues may appear complicated at first glance. In real applications, some of these technologies may manifest in various forms or be entangled with extensive code details, making them difficult to identify. In fact, their underlying principles are often straightforward. Understanding these principles from a fundamental perspective is not as challenging as it might seem. By reading this book, we hope readers will not feel frustrated or discouraged when facing these complex technological challenges in the future.

In recent years, the performance of GPT and various AI (Artificial Intelligence) large language models (LLMs) has experienced several significant advancements, demonstrating an unprecedented level of “intelligence”. One might wonder, can we rely on AI to address these technical challenges?

It is undeniable that AI LLMs are exceptionally good at summarizing basic knowledge, making it convenient for us to learn the fundamentals of a field rapidly and interactively. However, the summarization capabilities of AI LLMs have led to a misconception, causing people to believe that a comprehensive technological summary by AI equates to an understanding of a particular field or even the capability of solving a challenge. While AI might be able to accomplish 80% of the detailed coding work, it still requires our understanding of the nature of technical challenges and our guidance on what to do. Various experiences with LLMs have shown that to effectively utilize AI, the key is to provide AI with clear task objectives, intuitive guidance, and explicit instructions [1]. When these conditions are met, AI can “write” professional code, sometimes even more standardized than human-written code. The involvement of AI in code development actually requires a more comprehensive and profound understanding of technology, rather than merely dictating answers. In the AI era, understanding the *why* behind a programming technology becomes more important than merely knowing the *how*. This book will guide you through the methodologies to analyze technologies and understand their logic. With a solid understanding of the principles, you will be able to comprehend how a technology is designed, how a system can be optimized, when a particular tool can fit in, and how to use them effectively.

The scope of this book

This book is neither a textbook on Python programming, nor a tutorial for a specific tool, nor a textbook on quantum chemistry theory. It represents a combining experience from both academic research and industrial practice. The content of this book is designed to help you better understand advanced Python programming technologies and how to apply these techniques to practical quantum chemistry programs.

- This book is not intended as a tutorial on quantum chemistry theories. When presenting applications in quantum chemistry, we will briefly describe the background and formalism of quantum chemistry theory, but detailed derivations will be omitted. For a deeper understanding of the theory, it is recommended to consult more specialized quantum chemistry textbooks, such as *Modern Quantum Chemistry* by Szabo [2], *Molecular Electronic-Structure Theory* by Helgaker [3], and *Methods of Molecular Quantum Mechanics* by McWeeny [4].
- This book does not teach Python programming syntax. Before diving into this book, it is recommended that readers acquire some basic knowledge of Python programming. For the basics of Python coding, readers can refer to the Python documentation or other Python programming textbooks, such as *Fluent Python* by Ramalho [5] and *Think Python* by Downey [6]. Additionally, this book involves some knowledge of C/C++, networking, and operating systems. Although not prerequisites, some knowledges in these areas will enhance the understanding of the content presented in this book.

- This book does not intend to present detailed instructions on how to install or use specific software packages. Instead, we discuss the principles, designs, and trade-offs of various Python packages.
- Although many of the techniques and insights in this book are derived from the Python quantum chemistry package PySCF [7], this book is not intended to serve as a programming reference or a user guide for PySCF.
- The examples discussed in this book may not represent the latest or most efficient algorithms and implementations. Our focus is not on the theoretical models, methodologies, or optimal algorithms. Instead, we aim to demonstrate how a technology or a Python feature can be applied in quantum chemistry. Some technologies may appear different from traditional implementations of quantum chemistry programs. We hope this approach will inspire you to use these technologies in your own projects.

When selecting content for this book, we considered the influence of many new technologies on modern software, such as heterogeneous computation, cloud computing. We have intentionally included discussions on new technologies to enrich the content. If you are interested in these areas, you will find relevant application examples in the book.

We strongly encourage the use of AI tools to explore technologies. Some very basic knowledge can be delegated to ChatGPT or other AI tools, which is an effective method for quickly learning new technologies. Although this book is not specifically about how to use AI tools, in some chapters, we demonstrate the AI prompts directly. These examples can serve as a reference for using AI coding tools. The content of this book is written in depth, and we have intentionally omitted many fundamental topics. We strive to avoid presenting knowledge and techniques that can be easily acquired through the use of search engines or AI tools.

Who is the audience

This book is written to provide insights into Python programming technologies using examples from quantum chemistry. It can be used as a technical reference for people studying Python programming in the context of quantum chemistry. Despite its title, *Python for Quantum Chemistry*, the techniques discussed are also applicable to other areas of scientific computing and high-performance Python programming. Additionally, this book will be a useful resource for answering your questions if you encounter any of the following scenarios:

- You extensively use numerical computational tools, such as NumPy, SciPy, and Pandas, in your applications. You wish to gain a deeper understanding of their design to enhance the efficiency and robustness of your program.
- You are developing a Python package for solving specific scientific problems. As more features are added to the package and more people become involved in the

project, the codebase may become complex and disorganized. You want to learn how to effectively manage, maintain, share, and publish your work.

- You are interested in optimizing the performance of scientific applications. By searching on Google or Stack Overflow, you can find several suggested options and technologies, such as Cython compilation, JIT compilation, GPU acceleration, JAX distributed computation, and multiprocessing parallelization. You wonder about the pros and cons of these options and which technology may be best suited for your current scenario.
- You want to utilize parallel computation in your Python program. There are various parallel computation methods and techniques in Python. You are curious about how each parallel computation method works, along with their advantages and disadvantages.
- You are interested in newly merged techniques such as heterogeneous computing, cloud computing, and automatic differentiation. You want to understand what these tools are like and how they can be utilized in scientific computing work.
- Python seems far away from low-level systems and hardware. You are curious about how the underlying systems and hardware impact program efficiency and how low-level systems coordinate with the Python interpreter to affect the efficiency and the design of Python programs.

Outline of this book

This book is structured in three parts:

- Part 1 focuses on useful Python tools for developing quantum chemistry and scientific applications.
 - Chapter 1 covers the Python runtime knowledges, the Python package system, isolated Python environments, and methods for managing Python projects.
 - Chapter 2 explores the design of the NumPy library and offers practical advices for utilizing both NumPy and the Pandas library.
 - Chapter 3 demonstrates techniques for 2D and 3D visualization of quantum chemistry results.
 - Chapter 4 provides an overview of various scientific computation tools, with a focus on linear algebra, tensor operations, and Fourier transforms.
 - Chapter 5 explores meta-programming techniques, including code generation, symbolic computation, and automatic differentiation.
 - Chapter 6 addresses input and output issues in Python, focusing on serialization, file I/O, and network I/O.
 - Chapter 7 briefly introduces cloud computing concepts and use cases.
- Part 2 focuses on program optimization techniques in Python.
 - Chapter 8 discusses interfacing between Python and other programming languages, particularly compiled languages such as C/C++ and Fortran. These

techniques offering possibilities for optimizing Python through hybrid programming approaches.

- Chapter 9 delves into code optimization techniques, including how to identify performance issues through cost estimation and profiling, how to analyze Python bytecode, how to use Python compilation to enhance execution speed, and how to reduce I/O overhead. Additionally, this chapter explores special optimization techniques such as precomputation, memoization, and lazy evaluation.
- Chapter 10 provides a concise discussion and comparison of various parallel programming approaches, including multi-threading, multi-processing, asynchronous computation, and MPI (Message Passing Interface). It addresses race conditions associated with each parallel computation method and demonstrates how to resolve these issues using synchronization primitives and the producer-consumer model.
- Chapter 11 briefly discusses how to incrementally integrate GPU computation into Python programs. It examines Python libraries that leverage GPU computation and provides insights on customizing GPU kernels and embedding them into Python applications.
- Part 3 presents applications of the previously discussed knowledge and technologies, utilizing examples from quantum chemistry.
 - Chapter 12 introduces the computation of integrals in quantum chemistry programs, extensively utilizing the optimization techniques discussed in Chapter 9.
 - Chapter 13 explores the development of mean-field programs, including code design and organization, as well as technical considerations for enhancing program performance.
 - Chapter 14 addresses the challenges of program optimization in configuration interaction, serving as an example of program profiling from Chapter 9 and parallel computation methods from Chapter 10.
 - Chapter 15 demonstrates the application of meta-programming techniques using Coupled Cluster theory. It details the development steps of symbolic programming.
 - Chapter 16 focuses on computing analytical nuclear gradients. This chapter provides a detailed demonstration of using the JAX package to customize automatic differentiation programs.

Chapters in Part 1 are largely independent of each other and can be explored in any order. Chapters in Part 2 exhibit a certain level of dependence. It is advisable to read them sequentially for a comprehensive understanding. Although Part 3 builds upon the knowledge and technologies presented in Parts 1 and 2, it is not strictly necessary to read these parts before proceeding to Part 3. If your primary interest lies in the quantum chemistry programming examples, you may choose to start directly with the chapters in Part 3, referring back to earlier chapters as needed. When techniques from the first two parts are used in Part 3, we will list the relevant earlier chapters therein.

Where to find support information

All code examples used in this book are accessible at the GitHub repository <https://github.com/sunqm/py-qc-book/>. Readers are encouraged to post comments and questions on the GitHub issues board of this repository.

Acknowledgments

Throughout the writing of this book, I engaged in frequent technical discussions with Xiaojie Wu. After completing the initial draft, he carefully reviewed the draft and provided many valuable suggestions. I would like to express my special thanks to Xiaojie Wu for his invaluable assistance.

Many people have contributed by reviewing the draft and providing feedback. I am deeply thankful to Matthew Chan, Leopold Talirz, Bin Shao, Chong Sun, and Kori Elizabeth Smyser, and Sandeep Sharma for their valuable feedback and suggestions. Their advices have greatly assisted me in completing this book.

Many technical problems explored in this book originate from questions and discussions on the PySCF project's issue board on GitHub. I would like to express my gratitude to the PySCF community for providing the material for this book.

I also learnt a lot of knowledge and experience in Python project management, DevOps workflows, cloud computing and data processing technologies from the research conducted by Boxiao Zheng, Jing Chen, and their teams in AxiomQuant Investment Management LLC. I am grateful for their insights and inspiration on these technical matters.

To make this book become real, I would like to thank my Acquisitions Editor Charles Bath. Initially, I was uncertain about whether to write such a book focused on Python technology in quantum chemistry. His advice and encouragement were instrumental in my decision to proceed with the project.

References

- [1] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E.H. Chi, Q.V. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, in: Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22, Curran Associates Inc., Red Hook, NY, USA, 2024, pp. 24824–24837.
- [2] A. Szabo, N.S. Ostlund, Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory, 1st edition, Dover Publications, Inc., Mineola, 1996.
- [3] T. Helgaker, P. Jørgensen, J. Olsen, Molecular Electronic Structure Theory, John Wiley & Sons, LTD, Chichester, 2000, <https://www.amazon.com/Molecular-Electronic-Structure-Theory-Trygve-Helgaker/dp/1118531477/>.
- [4] R. McWeeny, B. Sutcliffe, Methods of Molecular Quantum Mechanics, Pure and Applied Mathematics, Academic Press, 1969, https://books.google.com/books?id=D_Ph2OZu0YC.
- [5] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, O'Reilly Media, 2015, <https://books.google.co.in/books?id=bIZHCgAAQBAJ>.

- [6] A. Downey, Think Python. How to Think Like a Computer Scientist, 2nd edition, Green Tea Press, Needham, Massachusetts, 2014.
- [7] The PySCF Developers, Quantum chemistry with Python, <https://pyscf.org/>, 2024.

List of acronyms

AD	Automatic Differentiation
AI	Artificial Intelligence
AO	Atomic Orbital
API	Application Programming Interface
AST	Abstract Syntax Tree
BLAS	Basic Linear Algebra Subprograms
CC	Coupled Cluster
CCD	Coupled Cluster with Double Excitation
CD	Cholesky Decomposition
CI	Configuration Interaction
CI/CD	Continuous Integration/Continuous Deployment
CLI	Command Line Interface
CPHF	Coupled Perturbed Hartree-Fock
CPU	Central Processing Unit
DF	Density Fitting
DFT	Density Functional Theory
DIIS	Direct Inversion in the Iterative Subspace
DMA	Direct Memory Access
DP	Dynamic Programming
DevOps	Development and Operations
ERI	Electron Repulsion Integral
FCI	Full Configuration Interaction
FFI	Foreign Function Interface
FFT	Fast Fourier Transform
FIFO	First In, First Out
FLOP	Floating Point Operation
FMA	Fused Multiply-Add
FP	Functional Programming
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
GTO	Gaussian-type Orbital
GUI	Graphical User Interface
HF	Hartree-Fock
HPC	High-performance Computer
HDF5	Hierarchical Data Format Version 5
IO	Input and Output
IDE	Integrated Development Environments
IPC	Inter-Process Communication
JIT	Just-In-Time (compiler)
JVP	Jacobian-vector Product
KS	Kohn-Sham
LLVM	Low-Level Virtual Machine
LRU	Least Recently Used
MD	McMurchie-Davidson

MEP	Molecular Electrostatic Potential
MKL	Intel Math Kernel Library
MO	Molecular Orbital
MPI	Message Passing Interface
MRO	Method Resolution Order
OOP	Object-oriented Programming
OS	Operating System
PEP	Python Enhancement Proposal
PES	Potential Energy Surface
QM/MM	Quantum Mechanics/Molecular Mechanics
RPC	Remote Procedure Call
RR	Recurrence Relations
SCF	Self-Consistent Field
SIMD	Single Instruction Multiple Data
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
VJP	Vector-Jacobian Product
XC	Exchange-Correlation

PART

Python ecosystems and tools for scientific computing

1

Python programming environment

1

Python program has straightforward syntax and a highly interpretable coding structure. These characteristics make it easy to get started with Python programming and write Python programs. As you progress further into Python, you might often face challenges associated with Python package systems and Python project management, such as:

- I have multiple versions of a Python library installed in the system, how do I know which library I'm using?
- I have multiple modules that reference each other and sometimes my program crashes due to import errors. How does this happen?
- I would like to manage my work as a Python package. How should I organize my project?
- I have developed a Python package. How can I publish it for others to use? What is the best option to distribute my program package?
- I would like to run my Python program as a command-line tool. How can I accomplish this?
- I need to run a Python program on different machines. How can I ensure they operate in the same environment?
- I have a class that inherited from multiple base classes. How do I know if a particular method from a base class is correctly invoked?
- I would like to try some cool Python packages, but installing new packages disrupts my environment. How can I keep my runtime environment stable and unaffected?

These issues, as they are not directly related to programming skills, are typically omitted in Python programming textbooks. Yet, they greatly affect the Python programming capabilities and efficiency. We will start our Python programming learning journey from these technical aspects.

In this chapter, we will examine Python runtime systems and package management systems. These systems are closely related to the methodologies for managing Python projects and distributing Python packages. To effectively manage a Python project, we need a variety of tools, not limited to Python-specific ones. We will also discuss the use of modern software development tools, such as Git and Docker. The use of Docker, although not a Python tool, is particularly important in the Python ecosystem. It will be utilized throughout this book.

1.1 Python package system

1.1.1 Python runtime

Python is powerful due to its extensive package resources. To use a package in a Python program, we simply `import` the desired module from the package. To execute a Python program, we simply need to type the following command:

```
python my_script.py
```

However, have you ever wondered how the computer interprets our needs and executes the program accordingly? The computer cannot magically understand our needs. Specifically, three questions, aside from the coding itself, are related to the correct execution of a Python program:

- Which Python executable is being invoked?
- Are there any configurations or environment variables that affect the execution of the program?
- Which dependent libraries are actually imported?

1.1.1.1 Python executables

The term `python` in command line is merely a shortcut for a specific Python executable. A system may have multiple CPython executables of various versions installed. This includes the default Python executable installed by the operating system (typically a legacy version), as well as Python executables from other sources, such as Anaconda.

Generally, interpreters for different versions of Python 3 behave consistently with most Python code. However, in each new release of Python 3, Python continuously introduces new syntax and modifies the functions in its built-in libraries. This can occasionally lead to errors and reproducibility issues when using executables from different Python versions.

A more significant issue that arises from using the wrong Python executable is the risk of accessing packages from incorrect paths. Each Python executables may have installed different packages or versions of those packages. This can lead to import errors or compatibility issues with the packages used in our program.

If our program uses new programming syntax that requires new versions of Python, we can check the Python version in the program using the variable `sys.version_info`. For instance, the code below ensures that the program runs only on Python 3.10:

```
assert (3, 9) < sys.version_info < (3, 11)
```

Throughout this book, our discussions and examples are based on the Python 3.10 standard.

1.1.1.2 Python runtime paths

In Python, the runtime paths refer to the locations where Python searches for modules and packages. Here are some typical locations where Python searches for modules:

- The current directory.
- Standard library directories, such as `/usr/lib/python3.10`.
- Site-packages directories, such as `~/.local/lib/python3.10/site-packages`.

The runtime paths are stored in a list called `sys.path`. When importing a library, Python sequentially searches for the library in each path specified in `sys.path`. Python will load the first match it finds, and any potential matches in subsequent paths will be ignored.

If multiple copies of a package are installed in different location on the system, how do we determine which version is actually used in a program? One method is to import the module and inspect its `__file__` attribute. For example,

```
In [1]: import numpy
        numpy.__file__
Out[1]:
'/home/ubuntu/miniconda3/lib/python3.10/site-packages/numpy/__init__.py'
```

By running the command `python -h` in the terminal, you can find a list of environment variables with the prefix `PYTHON`. Some environment variables can alter the default behavior of Python. For instance, the `PYTHONHOME` and `PYTHONPATH` environment variables can affect the locations where Python searches packages. Although adjusting these variables is convenient for testing a new package, it can lead to unintended consequences, such as the accidental import of buggy or incompatible modules into the Python runtime.

1.1.1.3 Inspecting dependencies

On Linux systems, the `ldd` command can reveal the exact paths of dependent shared libraries. Is there a similar command to show the location of dependent libraries for a Python program?

Inspired by the `ldd` command, we can create a simple tool `pyldd` to inspect all dependent libraries of a Python package. The basic idea is to walk through all files in a package and extract import statements from those files. This functionality can be implemented using the following code snippet:

```
import importlib
from pipreqs import pipreqs

imports = pipreqs.get_all_imports(path)
for name in imports:
    mod = importlib.import_module(name)
    print(mod)
```

For a specified path, we utilize the tool `pipreqs` to extract `import` statements from a package located at that path. We then use the `importlib` library to dynamically execute the import statements. This process enables us to collect the locations of the imported modules.

Some Python modules are shared libraries that are implemented in C or C++. These modules can be found in the Python `site-packages` path with a name following the convention

```
{module_name}.cpython-{python_version}-{platform_info}.so
```

For instance, a module named `_yaml.cpython-310m-x86_64-linux-gnu.so` can be found in the `site-packages/yaml/` directory. Such modules can be directly imported into Python.

```
from yaml import _yaml
```

1.1.1.4 The `importlib` library

`importlib.import_module` function is the tool that allows dynamically loading modules at runtime. Using the statement

```
b = importlib.import_module('a')
```

to load a module has the same effect as the standard import statements

```
import a as b
```

No matter how many times, and where the `import_module` function is called, the module, as well as the functions and global variables registered in the module are *initialized only once*, during the first time the module is imported. Python stores all active modules in the dictionary `sys.modules`. To tell whether or not a module has been imported, one can check if its name exists in `sys.modules`.

The dynamic module loader `importlib.import_module` is flexible. It allows us to load modules on demand based on the module name at runtime. It can reduce the overhead of Python start-up and the risk of import failures. Consider the scenario of loading GPU kernels in Python. To optimize GPU performance, distinct GPU kernels may be available for different hardware. Importing all GPU kernels simultaneously when initializing modules could lead to compatibility issues or substantial overhead. By using dynamic imports, the program can detect the GPU device at runtime and selectively load the appropriate kernels.

However, the use of `importlib.import_module` can potentially compromise the clarity of Python programs as it introduces an additional layer of indirection. The dynamic import also complicates static code analysis. Static code analyzers, such as `flake8` and `pylint`, may fail to analyze the dynamic modules, overlook relevant modules, or raise inaccurate warnings. To maintain code clarity and facilitate effective static code analysis, it is recommended to use the standard `import` statement if the module name can be determined in advance.

1.1.2 Package, sub-package, and module

Python packages and modules are structured based on the file system. A module in Python usually corresponds to a single `.py` file. A package, on the other hand, is a

directory comprising several `.py` files along with a special file named `__init__.py`, which is executed when the package is imported. Packages can be nested and included within other packages, and they are known as sub-packages.

Managing packages and modules can present challenges in programming:

- How do we add a package or a module to the system?
- How can dependencies and conflicts between different modules be resolved?
- What are the best practices for creating and structuring a package?

1.1.2.1 Adding packages

The commands `pip install` and `conda install` are the most commonly used methods to install new Python packages. The `pip` command can install Python packages from

- The official PyPI (Python Package Index) server <https://pypi.org> or any PyPI mirrors.
- Third party PyPI servers, or self-hosted PyPI servers via the option

```
--extra-index-url <host-url>
```

- Git repositories with the prefix `git+` in the package URL, such as

```
pip install git+https://github.com/pypa/pip
```

- Packages on local file systems.

If downloading packages from `pypi.org` is slow due to poor internet connection, packages can be installed from mirrors with the option

```
--index-url <mirror-host-url>
```

This option can be permanently added to the `pip` command. For example,

```
$ pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

This command writes configurations to files `~/.config/pip/pip.conf` or `~/.pip/pip.conf` on Linux systems. The contents of the configuration files [1] would be like

```
[global]
index-url = https://pypi.tuna.tsinghua.edu.cn/simple
```

When installing large packages such as TensorFlow or PyTorch, even using mirrors may not provide the download speed we desire. In such scenarios, caching PyPI packages on a self-hosted PyPI server may be considered. The process of setting up a self-hosted PyPI server is not complicated. It only requires a basic web server to manage files and index pages for the file system. For detailed guidance, you can refer to the instructions provided on the Packaging Python website [2]. This can be accomplished manually, or more simply, by deploying a server using Docker containers.

We will present an example of hosting a private PyPI server using Docker in Section 1.2. When operating a self-hosted PyPI server, it might be necessary to specify the `--trusted-host` option to bypass SSL (Secure Sockets Layer) validation errors.

When working on a Python project that is under development, it is often necessary to use packages that are not installed in the `site-packages` directory. In such scenarios, the editable mode `pip install -e` is a useful method to install a package from a local path. In editable mode, Python can immediately detect any changes made to the package. The editable mode relies on Python `.pth` files (short for path files). These files are usually located in the `site-packages` folder. Each line in the `.pth` file specifies a directory that will be appended to the module searching path, `sys.path`. For example, there may exist a `easy_install.pth` file in the directory

```
~/.local/lib/python3.10/site-packages
```

Suppose that the `easy_install.pth` file contains the following contents

```
/home/ubuntu/workspace/pytest/src  
/home/ubuntu/workspace/geomeTRIC
```

The two directories will be appended to the end of the `sys.path` list, thereby extending the package searching paths. Please note that Python searches for modules and packages in the following order:

1. The directory of the input script;
2. Directories in `PYTHONPATH`;
3. The `site-packages` directories;
4. The directories listed in the `.pth` files.

1.1.2.2 Package dependencies

When working with complex Python package-module system, we may encounter two types of dependency problems:

- Version conflicts;
- Circular dependency.

In a Python project, the dependent libraries are typically listed in the `pyproject.toml` configuration file, the `setup.py` script, and the `requirements.txt` file. These files specify the direct dependencies of a Python package. Additionally, there are *indirect dependencies*, which are introduced recursively by the dependent libraries. Indirect dependencies can sometimes lead to version conflicts. Unlike shared libraries in Linux, which can encode the version and path information of dependencies within their files, the Python import mechanism does not support specifying version information. If version conflicts arise due to the coexistence of unrelated projects in the same Python environment, using virtual environments could potentially reduce the risk of version conflicts. *However, Python lacks a standard solution to resolve version conflicts among indirect dependencies.*

Circular dependency among modules is another common issue in a Python project. Consider a package with the layout as below

```
mypkg
├── __init__.py
└── module_a.py
    └── module_b.py
```

The implementations of `module_a` and `module_b` are

```
$ cat module_a.py
from . import module_b                                # (1)
def factory():
    return module_b.B()

$ cat module_b.py
from . import module_a                                # (2)
class B:
    def new():
        return module_a.factory()                      # (3)
```

They reference each other in their import statements, thus create a circular dependency. However, importing either module will not result in import errors.

When Python processes the statement `import module_a`, it first locates the `module_a.py` file, creates a module object for `module_a.py`, and registers this object in `sys.modules`. Following this, Python begins the initialization of `module_a`. In `module_a.py`, the import statement at line (1) is executed. Here, Python creates a module object for `module_b` since it is not available in `sys.modules`. Next, Python proceeds with the initialization of `module_b`. In `module_b.py`, the statement at line (2) requires `module_a`, which has been created and registered in `sys.modules` but not yet completed its initialization at this point. The interpreter detects that the symbol `module_a` has already been defined. It continues to process the initialization of the remaining code in `module_b.py`. Inside the `B.new` method, a dependency to `module_a` is introduced, as shown in line (3). However, this does not break the initialization of `module_b` as the dependency to `module_a` is only realized when the method is invoked. Up to this point, Python has successfully completed the creation and initialization of `module_b`, as well as line (1) in `module_a.py`. It then proceeds with the remaining initialization steps in `module_a`, which include loading the `factory` function into `module_a`'s namespace. Although `module_b` is used within the `factory` function of `module_a`, this does not prevent the Python interpreter from parsing and loading the function during the initialization of `module_a`.

The code example below shows a different situation of circular dependency between `module_a` and `module_b` which will lead to an `ImportError`.

```
$ cat module_a.py
from . import module_b                                # (1)
class A:
    pass
def factory():

$ cat module_b.py
from . import module_a
class B:
    def new():
        return module_a.A()                          # (2)
```

```
    return module_b.B()

$ cat module_b.py
from . import module_a
class B(module_a.A):
    pass
```

When

```
import module_a
```

is executed, Python creates a module object for `module_a.py` in `sys.modules`. This action yields to the module creation and initialization for `module_b.py` due to the statement at line (1). During the initialization of `module_b`, loading class `B` triggers access to the class `A` in `module_a`. An `ImportError` will be raised because class `A` has not been initialized in `module_a`'s namespace at this point.

There is a workaround, although not very intuitive, to circumvent this error. When using this package, one can import `module_b` first. This will trigger a complete initialization of `module_a`, thereby making class `A` accessible in the namespace of `module_a`. This approach prevents import errors when loading class `B` in `module_b`. Importing `module_a` afterwards will also succeed as both modules are initialized and accessible in `sys.modules`.

In this example, import order matters for resolving circular dependencies. To ensure imports are handled in the correct order, one method is to explicitly manage the importation of the two modules during the package initialization stage. This can be achieved by adding the following import statements to the package's `__init__.py` file:

```
from . import module_b
from . import module_a
```

When importing modules from this package, no matter which module is imported first, the `__init__.py` file will be executed first, where the two modules are imported in the correct order.

While adjusting the import order to resolve circular imports can sometimes work, this method has drawbacks. It solves the issue in an implicit manner, which can make the solution fragile and the code difficult to understand. Furthermore, such a solution may become problematic when the program is being refactored or when new features are being added. To address these problems, an alternative approach is to import modules closer to the place where they are actually used, rather than at the module's top level. For example, in the following code snippet, `module_b` is not imported at the initialization stage of `module_a`. Instead, it is imported only when the factory function in `module_a` is invoked. This lazy-import approach helps decoupling the circular dependencies during the module initialization phase.

```
$ cat module_a.py
class A:
    pass
def factory():
    from .module_b import B
    return B()

$ cat module_b.py
from . import module_a
class B(module_a.A):
    pass
```

According to the PEP (Python Enhancement Proposal) 8 coding style guide, lazy imports are not recommended. PEP 8 advises placing import statements at the beginning of the code file. This practice helps to identify any `ImportError` early, rather than encountering errors when the specific function is called. Circular imports can cause problems in the import chain and should generally be avoided when designing a project.

1.1.2.3 Namespace packages

When managing a complex software project, we can organize modules into sub-packages based on their functionalities. Modules and sub-packages of a package are not necessarily placed in a single folder. Python allows for a package to be divided into smaller components and distributed in different locations.

Every Python package has an attribute `__path__`. This attribute is a list of directories indicating where its sub-packages and sub-modules are located. Similar to the functionality of `sys.path`, Python searches through the directories listed in `__path__` in the order they are listed and selects the first module or sub-package that matches the requested name.

For instance, consider a project that includes a core repository and an extension, with a file structure shown below

```
...
├── core_repo
│   └── mypkg
│       └── __init__.py
└── extensions
    └── mypkg_ext
        └── module_a.py
...
```

The parent package is named `mypkg` and it includes an `__init__.py` file. When `mypkg` is imported, the `mypkg.__path__` attribute holds all directories of the package. At this point, Python is unable to import `module_a.py` because it does not know where to

find the module. However, once the directory containing `module_a.py` is added to `mypkg.__path__`, Python can successfully locate and import this module. If the modified `__path__` is defined in the `core_repo/mypkg/__init__.py` file, every time `mypkg` is imported, the extension module will be automatically registered within the project.

```
In [1]: import mypkg
        print(mypkg.__path__)
Out[1]:
['ubuntu/home/core_repo/mypkg']

In [2]: import mypkg.module_a
...
ModuleNotFoundError: No module named 'mypkg.module_a'

In [3]: mypkg.__path__.append('/ubuntu/home/extensions/mypkg_ext')
        import mypkg.module_a
        print(mypkg.module_a.__file__)
Out[3]:
'/ubuntu/home/extensions/mypkg_ext/module_a.py'
```

This feature in Python is referred to as a namespace package, which enables multiple packages to share a common namespace prefix. There are various methods to create a namespace package, as detailed in the Python document [3]. Furthermore, PEP 420 [4] introduced a strategy for organizing namespace packages. In these methods, the `__path__` attribute is managed or generated in a more systematic and automatic manner. Nonetheless, the fundamental principles are similar to the process we described here.

1.1.3 Python program as CLI tools

When executing data processing or workflow automation tasks, the command-line interface (CLI) is an efficient method to interact with Python programs. Python libraries can be easily integrated as CLI tools. There are several approaches to creating a CLI in Python:

- The Python command `python -m`.
- The standard library `argparse`.
- Third party tools such as `click` [5], `Typer` [6], and `Fire` [7].

The `python -m` command is the simplest method to enable a CLI. It only requires to define the code block

```
if __name__ == '__main__':
    ...
```

in a module. If such a module is accessible in `sys.path`, executing the command `python -m package.module.name` will import the module and run the code within this block. For instance, the standard Python library `json.tool` offers the CLI

```
python -m json.tool
```

for formatting JSON data (further details on JSON data format are discussed in Chapter 6).

The libraries `argparse` and `click` can offer more advanced CLI functionalities. `argparse` is included in the Python standard library, which makes it readily available in any Python environment. For projects where additional dependencies are acceptable, the `click` library is an excellent option due to its intuitive and straightforward approach to CLI creation.

The `argparse` library tends to involve more coding as it requires specifying all details of CLI configurations. Conversely, `click` can efficiently define commands and options by leveraging Python decorators. Moreover, `click` includes several additional features, such as shell-completion [8], to improve the CLI functionality. In the following examples, a CLI for the previously developed dependency inspection tool `pyldd.py` is implemented using the two libraries. The `pyldd` CLI accepts one argument `path` for the target package and an optional flag `--depth` to control the depth of the dependency analysis. Here is the implementation using `argparse`:

```
#!/usr/bin/env python
...
Find and print all dependent libraries required by a package.
...
import importlib
import argparse
import pathlib

def main():
    parser = argparse.ArgumentParser('pyldd', description=__doc__)
    parser.add_argument('path', type=pathlib.Path)
    parser.add_argument('--depth', type=int, default=0,
                       help='The maximum depth of sub-packages to inspect')
    args = parser.parse_args()

    imports = get_all_imports(args.path)
    imports = [imp for imp in imports if imp.count('.') <= args.depth]
    for name in set(imports):
        try:
            mod = importlib.import_module(name)
            print(mod)
        except Exception:
```

```

        print(f'Module {name} not found')

if __name__ == '__main__':
    main()

```

The following version utilizes click:

```

#!/usr/bin/env python
...
Find and print all dependent libraries required by a package.
...
import importlib
import click

@click.command(name='pyldd', help=__doc__)
@click.option('--depth', type=int, default=0, help='The maximum depth of
    sub-packages to inspect')
@click.argument('path', type=click.Path(exists=True, dir_okay=True))
def main(path, depth):
    imports = get_all_imports(path)
    imports = [imp for imp in imports if imp.count('.') <= depth]
    for name in set(imports):
        try:
            mod = importlib.import_module(name)
            print(mod)
        except Exception:
            print(f'Module {name} not found')

if __name__ == '__main__':
    main()

```

If you wish to rapidly develop a CLI with minimal coding effort, libraries like `Fire` and `Typer` are excellent options. These libraries can automatically generate a CLI by mapping the positional and keyword arguments of a function to CLI arguments and optional flags, respectively. We will not delve into details here.

Normally, to execute a Python script from the command line, you can pass the script file as an argument to the Python interpreter, such as

```
python script.py arg1 arg2
```

Alternatively, the command

```
python -m module.name
```

can be used to execute the CLI script. However, these methods are somewhat inconvenient as they require us to repeatedly type the term `python` along with the script path or the module name. Can executing a Python script from the CLI be simplified?

One simple method to execute a Python script is to grant executable permissions to the script:

```
$ chmod +x script.py
```

Additionally, we need to include the so-called *shebang* (short for sharp-bang `#!`) line at the top of the CLI script. The shebang line is essential as it informs the Unix shell how the script should be executed. The shebang

```
#!/usr/bin/env python
```

is generally sufficient in most scenarios. If multiple Python interpreters are installed on the system and there is a need to use a specific one, the desired Python executable can be specified directly in the shebang line. For example:

```
#!/usr/bin/python3.10
```

If the CLI tool is intended as a feature of a Python package, more standardized methods, such as using the `setup.py` script or the package configuration file `pyproject.toml`, can be employed to create an executable command.

In the `setup.py` script, as shown in the following example, the optional keyword `entry_points` for `setup()` defines the name of the executable (`pyldd`) and the corresponding function (`main` in the `pyldd.cli` module) for the CLI tool:

```
from setuptools import setup, find_packages
setup(
    name='pyldd',
    packages=find_packages(),
    entry_points={
        "console_scripts": [
            'pyldd = pyldd.cli:main'
        ]
    }
)
```

Alternatively, in the package configuration file `pyproject.toml`, the `[project.scripts]` section instructs the Python package system to create a command for the CLI. For example,

```
[project.scripts]
pyldd = "pyldd.cli:main"
```

More details on the `pyproject.toml` configurations will be explored in Section 1.3.4.

After installing the package, the `pyldd` command can be executed directly from the command line.

1.1.4 Virtual environment

In C/C++ or Fortran, to isolate a program from shared libraries installed in the runtime environment, one can build a statically linked executable. Unlike these compiled languages, Python does not have a mechanism like static compilation to isolate dependencies. It relies on the correct installation of packages and a properly configured runtime environment. To address this issue, the concept of *virtual environments* was introduced in Python to isolating the runtime environment. Running Python programs in a virtual environment has several benefits, such as

- Separating testing environments from production environments.
- Ensuring reproducibility on different platforms.
- Reducing dependency conflicts between different projects.

Using virtual environments is a highly recommended practice when developing Python software projects. Generally, there are two popular options for managing Python virtual environments: the Python `venv` module and `conda env`.

Managing virtual environments with `venv` is straightforward. To create a new virtual environment in a directory, we can execute the command:

```
$ python -m venv new-env-dir
```

To activate the virtual environment, we can execute the shell command:

```
$ source new-env-dir/bin/activate
```

After activating the virtual environment, any packages installed using `pip` are restricted to the virtual environment's directory. Specifically, newly installed packages are placed in the `new-env-dir/lib/site-packages` directory, and executables are created in the `new-env-dir/bin` directory. Furthermore, `pip uninstall` is limited to removing packages found within the `new-env-dir` directory. It can prevent the accidental un-installation of system-level packages. Meanwhile, changes made in the system-wide `site-packages` directory, such as updating or uninstalling packages, do not affect the virtual environment. Each `venv` is self-contained and independent.

Purging a virtual environment is also straightforward. One can simply delete the directory of the virtual environment, which removes all packages and their dependencies without affecting other projects or system-level packages.

While a virtual environment solves the issue of runtime environment isolation, can it provide the reproducibility that a statically linked executable offers? Furthermore, if we need to run Python programs on another machine using a specific environment, how should we proceed?

The `venv`-created environment cannot be renamed or moved like a statically linked executable. Moving the `venv` directory to a different location will break the `venv` environment. This means that `venv` is intended for local use only. It is not suitable for archiving or distributing a project.

To transfer a `venv` environment to a different folder or a different machine, the main challenge is to ensure that the libraries installed in the new `venv` environment

remain consistent, and their versions match those in the original environment. We can execute the following command to list all installed packages from the current `venv` and save them into a `requirements.txt` file.

```
$ pip freeze --local > requirements.txt
```

Next, create a new `venv` on a different machine or directory and replicate the installation by executing:

```
$ pip install -r requirements.txt
```

1.1.5 Conda

Scientific software often involves the use of many tools and libraries beyond Python alone. The Python package system might not be sufficient to manage projects that involve multiple programming languages or various system libraries. Conda presents a solution for managing scientific computing projects that are developed with multiple programming languages. Furthermore, Conda provides access to a wide range of scientific computing libraries through the Anaconda distribution and Anaconda cloud [9].

Conda is not merely a Python package, although a CLI tool named `conda` can be installed via the command

```
pip install conda
```

Conda offers a comprehensive system that includes Python executables, packages, and system libraries. The Conda system can be installed via Miniconda or Anaconda distributions. Miniconda is a light-weight application which includes only the Python executable. The Anaconda distribution comes with many pre-installed libraries. For example, the Python environment used in this book is provided by Miniconda, which is installed using the command:

```
$ curl https://repo.anaconda.com/miniconda/Miniconda3-py310_23.3.1-0-Linux-x86_64.sh | bash
```

Installing, updating, and uninstalling packages using the `conda` command are very similar to how one would use `pip`. However, the package dependency resolution in Conda involves not only the Python packages but also the relevant system libraries. This comprehensive system tends to make Conda operations slower than those of `pip`. To address this issue, an independent project named Mamba [10] was developed to enhance the speed of the Conda package management system. It can be used as a drop-in replacement for Conda.

Conda also offers virtual environments [11] to isolate the runtime environment. Its concept and usage are similar to `venv`. An empty Conda environment can be created using the following command:

```
$ conda create -y --name env-name
```

This environment can be activated with the command:

```
$ source activate env-name
```

Similar to `venv`, the directory of a Conda environment is not portable. To transfer a Conda environment across different machines, one needs to export the installed Conda packages to a configuration file:

```
$ conda env export --file environment.yaml
```

Then, a new environment can be created using the exported configuration:

```
$ conda env create -y --name new_env_name --file environment.yaml
```

Here, you might have a question: Can Conda and PyPI packages be used together, interchangeably?

Python packages installed by the `conda` command have their own records in the Conda package management system. Typically, `pip` can uninstall or upgrade `conda`-installed Python packages without errors. However, Conda is not able to detect the changes made by `pip`. Mixing `pip` and `conda` installations may break Conda package management systems.

Conda advises using it as the primary tool for installing Python packages whenever possible. It suggests to use Conda to install any packages that are available in Conda. If a package is not available in Conda, the recommendation is to first use Conda to install as many dependent packages as possible, and then use `pip` for the remaining packages.

In practice, sorting out the Conda-available packages, as suggested, is inconvenient. To simplify this process, some simple rules can be followed to determine when to use `conda` or `pip`. For instance, one can use Conda to set up the base Python environment, manage several performance-sensitive Python packages such as NumPy, SciPy and `scikit-learn`, along with all system (non-Python) libraries. For other Python packages, `pip` can always be used.

1.2 Python programs in Docker

Docker offers a lightweight, isolated, and reproducible runtime environment called a container for running applications. Compared to traditional virtual machines, Docker containers consume fewer resources and exhibit a negligible performance overhead [12,13]. The Container technology simplifies the packaging and distribution of Python applications. It allows an application to *build once, run anywhere*. Developers can bundle an application along with all its dependencies into a container image, which can then be deployed and executed across various machines and operating systems. Due to the container's inherent isolation features, the use of virtual environments, such as `venv` or `conda env`, within Docker is unnecessary.

In this section, we demonstrate some scenarios that involve using Docker in Python program development and applications. For the technical background and the basic concepts of Docker, readers can refer to the Docker official documentation [14]. Docker and its underlying *virtualization* technologies have a significant impact on cloud technologies. In Chapter 7, you will find more examples of Docker and its utilization in cloud computing applications.

Running Docker as a non-root user

On most Linux distributions, Docker can be installed using the package management system, such as the following command on Ubuntu:

```
$ sudo apt install docker
```

However, Docker installed in this manner requires `sudo` privileges to execute the `docker` command. This approach may lead to a security risk on a cluster that is accessible to multiple users. Additionally, granting `sudo` privileges to regular users may not always be practical.

To enable Docker operating as a non-root user, the Docker rootless mode [15] can be installed with the following command:

```
$ curl -fsSL https://get.docker.com/rootless | sh
```

During the installation process, you will be prompted to set environment variables such as `DOCKER_HOST` in the shell startup configuration file (for example, `~/.bashrc`). After installation, you can verify the rootless Docker installation by executing the `hello world` service without `sudo`:

```
$ docker run hello-world
```

To ensure that rootless Docker starts as a daemon during system boot, the following command should be executed for additional configurations:

```
$ systemctl --user enable docker
$ sudo loginctl enable-linger $USER
```

If you are interested in trying out new tools without disrupting your existing environment, Docker offers a convenient solution. For example, suppose we want to assess the performance of the PyPy interpreter [16], a JIT compiler known for superior performance compared to the CPython interpreter. With Docker, we can easily test this interpreter, using the following command

```
$ docker run --rm -it -v /path/to/applications:/opt/app pypy:latest bash
```

This command creates a Docker container with a `bash` session and maps the `/path/to/applications` directory on the host machine to `/opt/app` in the container. Inside the container, we can use the PyPy interpreter in the same manner as a standard Python interpreter, such as running an application with

```
pypy application.py
```

Accessing an active container

A Docker container is a light-weight virtual machine. Just like a traditional virtual machine, containers can be shut down and restarted. The status of available containers, whether they are running or stopped, can be inspected using the command

```
$ docker ps -a
```

This command will display the status as well as the ID for each container. To stop a running container, we can use the command

```
$ docker stop <container_id>
```

To start a container that has been stopped, we can use the command

```
$ docker start <container_id>
```

We might want to use the same container to execute various jobs. In that case, we need to log into a container from our terminal. Unlike a traditional virtual machine which requires a remote login shell (like OpenSSH), it is not necessary to install or configure any login tools within the Docker container. We can execute the following command to log into a container.

```
$ docker exec -it <container_id> bash
```

Here, the command `bash` can be changed to other commands.

Inspecting an image

Essentially, a Docker image is a snapshot of a Docker container. It can be used as a template for creating new containers. To inspect how an image is created or what has been installed in the image, we can use the `docker history` command:

```
$ docker history --no-trunc <image_id>
```

where the `image_id` can be obtained with the command:

```
$ docker images
```

The `docker history` command reveals all the layers that comprise the image. From its output, we can identify the scripts or operations that were executed in each layer.

Creating a Docker image

A Docker image can be created by saving a snapshot of a running Docker container using the `docker commit` command. A more conventional method for creating a Docker image is by using a `Dockerfile`. Suppose we want to create a Docker image for a private PyPI server [17]. The `Dockerfile` for this service can be created as follows:

```
FROM python:3.10
RUN pip install --no-cache pypiserver && \
    mkdir /data/packages
ENV PACKAGES_DIRECTORY=/data/packages
CMD pypi-server run -v -a . -P .
```

Next, we can use the `docker build` command to create an image based on this Dockerfile:

```
$ docker build -t local-pypi:0.1 -f Dockerfile .
```

Here, the `-t` option specifies a tag for this image, which typically includes two parts: the image name (`local-pypi`) and a version ID (`0.1`).

Defining a Dockerfile is a crucial step in building a Docker image. Several keywords in a Dockerfile are required to specify how the Docker image should be constructed [18]. In this example, the Dockerfile specifies that the new image is based on the `python:3.10` base image, as indicated by the `FROM` line. The line starting with the keyword `RUN` specifies the commands that will be executed in a shell during the `docker build` process. The `ENV` line configures new environment variables. The `CMD` keyword defines the default command to execute when launching a Docker container for the image.

Deploying a private PyPI server

As we mentioned in Section 1.1.2.1, a self-hosted PyPI service can be utilized to distribute private packages or serve as a cache to accelerate the downloading of large Python packages. Using the image we previously created, we can launch a private PyPI server using the following `docker run` command:

```
$ docker run -d -p 8080:8080 -v /data/pypi-mirror:/data/packages local-pypi
:0.1
```

The `-d` option runs the container in daemon mode. The `-p` option exposes port 8080 from the container, allowing us to access the service via the URL

`http://localhost:8080`

The `-v` option mounts the `/data/pypi-mirror` directory from the host machine to `/data/packages` inside the container.

Python package `twine` can be used to test the local PyPI service. `twine` requires a configuration file, `~/.pypirc`, to define the PyPI server, including the URL and the login method.

```
[distutils]
index-servers =
    local
```

```
[local]
repository: http://localhost:8080
username: placeholder
password: placeholder
```

Then we can upload a package to the server, such as the command:

```
$ twine upload -r local dist/pyldd-0.1.0.tar.gz
```

File `dist/pyldd-0.1.0.tar.gz` is a source distribution (sdist) release of the `pyldd` package. More details about releasing a Python package will be discussed in Section 1.3.4. If the private PyPI service is functioning correctly, the uploaded packages should be found in the directory `/data/pypi-mirror` on the host machine. This PyPI service can also function as a PyPI mirror server. To cache additional packages on this server, we can download the wheel files from `pypi.org` and then use the `twine upload` command to upload the packages to the private PyPI server.

The `docker run` command mentioned above can be converted into a more readable and manageable configuration file in YAML format, named `docker-compose.yml`. `docker-compose` is designed for defining and running multi-container Docker applications. Below is an example of a `docker-compose.yml` file that achieves the same outcome as the previous `docker run` command:

```
version: '3.7'
services:
  pypi-local-mirror:
    image: local-pypi:0.1
    env:
      - PACKAGES_DIRECTORY=/data/packages
    ports:
      - 8080:8080
    volumes:
      - /data/pypi-mirror:/data/packages
    command: pypi-server run -v -a . -P .
    restart: always
```

We can use the `docker-compose` command to launch the service:

```
$ docker-compose up -d
```

To stop or restart the service, we can call:

```
$ docker-compose stop pypi-local-mirror
$ docker-compose restart pypi-local-mirror
```

To terminate and clean up the service, we can execute:

```
$ docker-compose down
```

Executing legacy Python code

In quantum chemistry research, we may encounter scenarios that require running legacy programs. In the newest operating systems or Python environments, these programs may fail for various reasons, such as

- The program is no longer maintained and cannot be installed on new systems.
- The program relies on older versions of libraries, compilers, Python interpreters, or packages that are not available on new systems.
- The program only produces correct results on older platforms.

Docker container can serve as a time machine, enabling us to recreate old computing environments and run legacy code. For instance, let's assume we need to run a Python program that was developed for Python 2.7. To use this program, we can start a Docker container that runs Python 2.7.

```
$ docker run -it python:2.7
```

The Docker image `python:2.7` is built on Debian Linux 10. When using `pip` commands to install Python packages inside this container, it automatically installs older versions of the packages.

1.3 Managing a Python project

1.3.1 Version control

Git is the most commonly used version control tool in a Python project, despite not being a Python-specific tool. It can be used to manage various aspects of a project, including data, configuration, and source code. In the following, we will use some common scenarios in Python program development to demonstrate Git commands and features. We will not delve into Git concepts (such as Git index, Git worktree) or the details of each Git command, as they are beyond the scope of this book. For those interested in mastering advanced Git technologies, the book *Pro Git* [19] is an excellent reference.

Downloading projects from GitHub

We can use the command `git clone` along with the URL of the repository to create a local copy of the remote repository. For example, the code examples in this book can be downloaded via

```
$ git clone https://github.com/sunqm/py-qc-book
```

The `git clone` command may be slow if the repository has a long commit history or contains large files. To speed up the cloning process, we can use the option `--depth 1` to download only the latest commit and ignore the commit history.

Synchronizing remote repository to local repository

A simple method to apply changes from a remote repository into a local repository is to use the `git pull` command. For instance, to merge the new features from the `dev` branch of the remote repository into our local repository, we can execute the command:

```
$ git pull https://github.com/sunqm/py-qc-book dev
```

If we have made changes in the local repository, `git pull` may prompt conflicts between the local and the remote repository, requiring us to resolve the conflicts. In this scenario, we can break down the `git pull` command into distinct steps and resolve conflicts at each step.

```
# Set up a remote branch
git remote add upstream https://github.com/sunqm/py-qc-book

# git fetch can download all remote changes without modifying local branch
git fetch upstream dev

# Merge the changes made in the dev branch of the remote repository.
git merge upstream/dev

# Conflicts may be raised by "git merge".
# Edit the problematic files then create a commit to save the modification.
git commit -a

# Finally, clean up the commit history if needed
git rebase -i upstream/dev
```

Here, the `git rebase` command rebuilds the commit history by re-applying the commits recorded in the local branch onto the HEAD of another branch (`upstream/dev` branch in this case). The option `-i` enables interactive manipulation of the commit history in an editor (Vim by default). In the interactive mode, Git presents a list of commits and allows us to choose how they should be modified or combined. After executing `git rebase`, the local branch will have a linear commit history on top of the target branch.

If there are uncommitted changes in the local branch, `git pull` or `git merge` will be interrupted. To address this issue, `git stash` command can be employed to temporarily store the local changes. Later, executing `git stash pop` will restore the uncommitted files.

Contributing to a GitHub repository

To add features or fix bugs in a remote repository, we can begin by creating a local branch with the command:

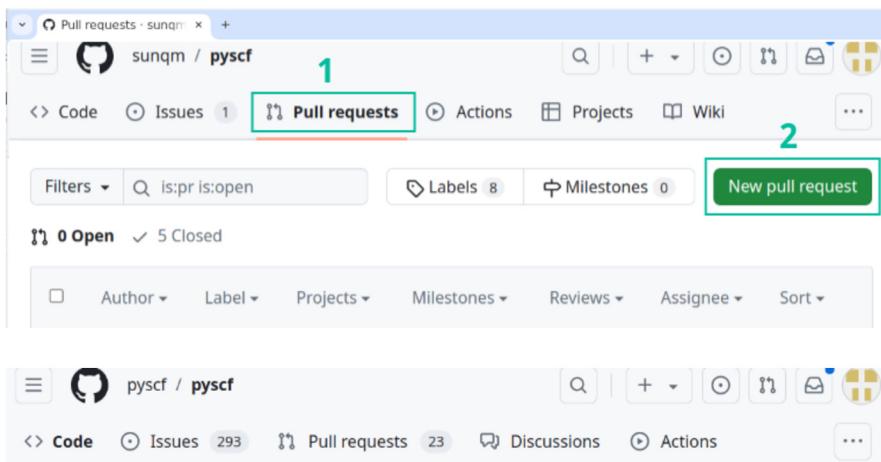
```
$ git checkout -b <branch_name>
```

Git branches enable us to work on various features of a project simultaneously without affecting each other. We can use the command

```
git checkout <branch_name>
```

to switch between different branches. After modifying the source code, we can utilize Git commands like `git status`, `git add`, and `git commit` etc. to save the changes in the local branch.

```
# Create and switch to a new branch  
git checkout -b new-feature  
  
# Make changes and commit them  
git add .  
git commit -m "Add new feature"
```



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff comparisons.

FIGURE 1.1

Procedure for creating a GitHub pull request.

Before uploading local changes, it is recommended to execute the command `git fetch origin` to retrieve all changes from the remote repository. Then, as previously discussed, we can execute `git merge` and resolve any potential conflicts

between the changes made in the local branch and those in the remote branch. Finally, upload local commits using the command `git push`. If the remote repository is a forked repository on GitHub or GitLab, we can contribute updates from our local repository by creating a pull request (PR), as illustrated in Fig. 1.1. When submitting PRs, we should check the target repository, local repository, and their respective branches. The automatically generated repositories and branches in a GitHub PR sometimes are incorrect.

When multiple contributors work on the same project, using the standard merging operation to merge PRs can complicate the commit history. This issue becomes particularly severe when conflicting changes are made in different PRs. After merging one PR, in another PR, we may need to synchronize these updates, resolve conflicts, and then merge changes into the main branch. This process can lead to a tangled commit history, making it challenging to track the relationships between commits. Maintaining a linear commit history can help prevent this chaos. This requires PRs to be merged using either a squash merge or a rebase merge [20]. Although enforcing a linear commit history can result in the loss of intermediate commits, it effectively reduces the complexity of the commit history, making it suitable for managing projects with multiple contributors.

Maintaining a linear commit history requires cooperation between the repository manager and contributors. For instance, suppose we are contributing to the `py-qc-book` project. When we need to add or modify code, we can fork the repository and create a branch for new features. After completing the functionality on the new branch, we can merge the possible updates from the remote repository to our local branch using the command:

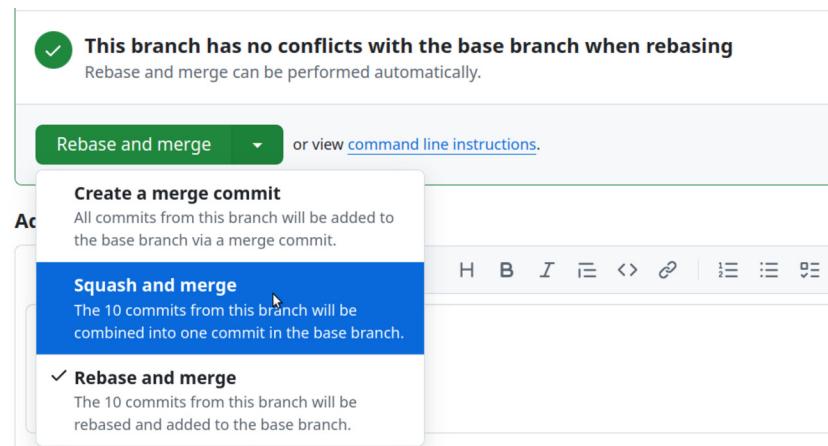
```
git pull https://github.com/sunqm/py-qc-book main
```

This merge process might generate conflicts. After resolving conflicts, we can push the local branch to our forked GitHub repository and then submit a pull request. The project manager, when merging this PR, should use a squash merge or a rebase merge (Fig. 1.2). After the new feature is merged, we can synchronize the changes made in the parent repository with our forked repository [21].

To prevent synchronization issues between the forked repository and the remote repository, we should avoid from making direct modifications to the main branch. If all contributors and the project manager adhere to this process for maintaining the project, it can ensure a clean commit history and minimize code conflicts.

Inspecting changes made in another branch

In some projects, we may have been working on a local branch for a while. The local branch may have diverged from the main branch. To find out what changes would be applied to the main branch if merging the two branches, we can first synchronize the repository and then use the `git diff` command to display the difference, as shown below:

**FIGURE 1.2**

Options for merging a GitHub pull request.

```
# Download new commits made in the remote repository  
git fetch origin main  
  
# Compare the local branch to the latest commit on the main branch.  
git diff ...origin/HEAD
```

It is usually more readable to view differences in a side-by-side mode. To achieve this, the command

```
git difftool -t vimdiff ...origin/HEAD
```

can be used. It launches `vimdiff` to display the differences between the two branches.

Reverting modifications

Git provides several commands to undo modifications and revert to previous states. If you wish to remove changes in the source code and roll back the commit history, use

```
git reset --hard <commit-id> <file-or-dir>
```

If you only need to roll back the commit history while keeping the changes in the source code, use

```
git reset --mixed <commit-id> <file-or-dir>
```

To undo a previous commit, use

```
git revert <commit-id>
```

This creates a new commit that undoes the changes made in <commit-id> while preserving the commit history. To discard all local changes or revert to the latest committed state, use

```
git checkout -- <file-or-dir>
```

In case you accidentally perform any of the undo operations above and wish to revert them, use `git reflog` to find the correct states for rollback. For example:

```
# Undo an accidental git reset operation using git reflog
git reflog
git reset HEAD@{2}
```

Identifying the commit where a bug was introduced

`git bisect` is a tool for tracing the problematic commit. Assuming we have a testing script that worked correctly in the past, we can use `git bisect run` to identify the problematic commit.

```
$ git bisect run python test.py
```

This command will switch to a specific commit and then execute the command we have specified, such as the command `python test.py` in this example.

Alternatively, we can manually run the `bisect` command to search for the problematic commit. For example:

```
git bisect start
# Mark the current commit as bad
git bisect bad
# Mark the commit v2.0.0 as good (no bug)
git bisect good v2.0.0
```

The `bisect` command will perform a binary search through the commit history between `v2.0.0` and the latest commit, producing a state for testing. We can label each produced state as either

```
git bisect good
```

or

```
git bisect bad
```

`git bisect` will continue the search until it identifies the commit causing the issue. Upon discovery, it will display a message specifying the problematic commit and the files that were altered by that commit.

Sub-packages distributed in different repositories

In certain projects, some modules are developed in separate repositories. The Git submodule feature allows for the inclusion of other repositories as subdirectories within the main repository. The details of these submodules are stored in the `.gitmodules` file.

We can use the `--recurse-submodules` flag for the `git clone` command to clone a repository that contains submodules. This command first clones the main project. If the project includes a `.gitmodules` file, submodules will be downloaded based on the contents of the `.gitmodules` file. For example,

```
$ git clone --recurse-submodules https://github.com/numpy/numpy
```

1.3.2 Testing a Python program

In Python programming, testing is particularly important due to the dynamic nature of the language. Unlike compiled languages, Python does not have a compilation stage to check for syntax and datatype correctness. Many errors only reveal at runtime. When developing a quantum chemistry program, a developer often needs to add new features or optimize existing algorithms. Test cases are crucial for verifying that the newly implemented code produces the intended outputs and does not break the existing functionalities. Testing has its own theory and methodology. If you are interested in this aspect, *Lessons Learned in Software Testing* [22] is an excellent book for learning how to design and manage tests effectively.

1.3.2.1 Testing tools

There are several testing frameworks available for managing tests in Python programs. `pytest` is one of the most popular testing framework. To use `pytest` [23], one just needs to put the test code in a function and prefix the function name with `test_`. `pytest` can discover these test functions automatically when executing the following command in the terminal:

```
$ pytest <filename_or_dir>
```

If the argument is a directory, `pytest` can automatically find and run all tests within that directory. Upon completion, `pytest` can generate a comprehensive report, including the number of discovered tests, passed tests, and the details of failed tests.

Developers would create *unit tests* and *integration tests* to verify the correctness and functionality of the code. Unit tests are designed to test individual functions or components of the code, ensuring that they produce the expected results. Typically, unit tests focus on confirming that functions developed earlier still operate correctly after modifications. An example of unit tests for developing quantum chemistry analytical integrals can be found in Section 12.1.3.6 of Chapter 12. On the other hand, integration tests focus on verifying that various components of the application can work together. Moreover, there are additional aspects to consider in testing:

- Code coverage, which can help identify areas that may need more testing. There are various metrics to measure code coverage, such as statement coverage which measures the number of executed lines, and branch coverage which evaluates the branches that have been executed among all possible branches. The `pytest` plugin, `pytest-cov`, can be used to report these coverage metrics.
- Doctests, which leverage code snippets within doc-strings to test the functionality of APIs. The command `pytest --doctest-modules` can launch doctests.
- Static analysis, which exams the source code to identify problems such as coding style, unused variables, syntax errors, type-related errors, and other potential issues. There are several popular static analysis tools available for Python, such as Pylint [24], Ruff [25], Flake8 [26], mypy [27].

Tests should be executed in a clean and reproducible environment. This can be achieved by working with a Python virtual environment or a Docker container. For example, to perform tests in a Docker container, we can create a test script named `test_all.sh`

```
#!/bin/bash
pip install .
pytest .
```

and execute the script with the docker command

```
$ docker run --rm --volume `pwd`:/src --workdir /src python:3.10 bash
test_all.sh
```

1.3.2.2 Testing automation

Tools such as `pytest` facilitate the execution of testing tasks and the summarization of results. Testing automation is another an important matter to consider in practice. If a project is hosted on GitHub or GitLab, testing tasks are typically set up as Continuous Integration (CI) jobs, utilizing GitHub Actions or GitLab CI/CD runners. We will discuss CI in more detail in Section 1.3.5.

1.3.2.3 Configuring tests for selective execution

Running an entire test suite for a package can be a time-consuming process. This may impact the code development efficiency and the effective utilization of computational resources. To reduce the duration of testing, testing jobs can be configured to terminate early upon receiving any errors. This is achievable in `pytest` by using the `pytest -x` option. Additionally, tests can be organized into subsets according to their characteristics. By selectively executing tests, we can bypass resource-intensive tests or less critical tests. In `pytest`, this requires a configuration section `[tool.pytest.ini_options]` in the `pyproject.toml` file which defines special labels to mark tests, for example,

```
[tool.pytest.ini_options]
markers = [
    "quick": marks tests as quick,
    "slow": marks tests as slow,
    "high_mem": marks tests with high memory usage,
]
```

Individual tests can be labeled with these markers, for example,

```
import pytest
@pytest.mark.quick
def test_small_array():
    ...

import pytest
@pytest.mark.slow
def test_sparse_array():
    ...
```

When the following command is executed:

```
$ pytest -m quick
```

only the tests annotated with `@pytest.mark.quick` will be processed.

1.3.3 Coding style and static checks

PEP 8 [28] is a coding style guideline recommended by the Python community. It covers aspects such as indentation, line length, whitespace, naming conventions, and so forth, to enhance code readability and consistency. Tools like Flake8 [26], Ruff [25] and Pylint [24] can be used to check our Python programs against these recommended guidelines. Additionally, these tools can serve as static code analyzers to detect potential errors within the Python code.

These tools are straightforward to use. In the command line, we can execute commands such as `flake8 path/to/your/code` or `pylint path/to/your/code` to report style violations and any potential issues in the code. Also, they can be configured in IDEs (integrated development environments) or code editors, providing real-time feedback and suggestions [29].

The default style rules enforced by these tools are quite strict. In practice, it may be necessary to ignore certain rules. Lint rules can be customized through a configuration file. Additionally, a common practice is to add specific derivatives in the source code to bypass certain checks. For instance, if the marker

```
# noqa: rule-ID
```

is added at the end of a statement, a specific rule in Flake8 and Ruff [30,31] is bypassed. The customization for Pylint employs a different marker:

```
# pylint: disable=rule-name
```

where the `rule-name` is specified in the Pylint documentation [32].

1.3.4 Releasing a Python package

If we wish to distribute our program for others to use, there are several methods for distributing a Python library. In the simplest setup, users can copy the source code and configure the environment variable `PYTHONPATH` to access new Python features. However, this method is not scalable. It is recommended to release a program project through a standardized process, distributing the library as a PyPI package or a Conda package. Below are the steps to distribute a library as a PyPI package:

1. Organize the source code according to Python's packaging conventions.
2. Create a configuration file in the root directory of the project to store the metadata, such as the package name, version, and dependencies.
3. Build the program and generate a distribution file.
4. Upload the distribution file to PyPI.

For Conda packages, the process is similar. The difference is that we need to use the Conda build system to create the package file and upload the package file to a Conda repository.

1.3.4.1 Python project structure

Conventionally, it is recommended that the source code of a Python project follows a structured layout (such as <https://github.com/pypa/sampleproject>). A very good reference for the file structure of Python projects can be found in the official document *Python Packaging User Guide* [33]. The following illustrates a common layout for a Python package.

```
.
├── doc
│   ├── conf.py
│   └── index.rst
├── LICENSE.txt
├── README.md
└── package_name
    ├── __init__.py
    ├── module1.py
    └── subpackage
        ├── __init__.py
        └── module2.py
    ...
└── ...
    ├── pyproject.toml
    └── requirements.txt
```

```
└── setup.py  
└── tests
```

- The `doc` directory contains documentation files written in reStructuredText or Markdown format. Documents in web pages or PDFs can be generated from these files using the documentation tool `Sphinx` [34].
- The `README.md` file should include a brief introduction to the package, concise installation instructions, and a simple usage example.
- The `pyproject.toml` file specifies metadata of the package, including project information, the build environment, and the install options. It can also be used to store configurations for testing, CI jobs, and other related tools.
- The `setup.py` file stores the project metadata, the dependencies, and the configurations for building the project. The `setup.py` script was the standard package configuration file for a long period. It is still widely found in many Python packages as the only configuration to manage the project. However, it is now more encouraged to separate project metadata and the building script, and manage the metadata using `pyproject.toml`.
- Many Python projects contain a `requirements.txt` file, which explicitly records all dependencies, including the names and versions of both direct and indirect dependencies. The `requirements.txt` file is primarily used to set up reproducible environments.
- Optionally, a `MANIFEST.in` file can be created to specify the files and directories that should be included (or excluded) in the package.
- The source code of the package, sub-packages, and modules is structured and stored in an `src` folder or a folder named after the project name.

1.3.4.2 `pyproject.toml` and `setup.py` for Python package configuration

The metadata of a package and installation options can be specified in the `pyproject.toml` file, the `setup.cfg` file, or directly in the arguments of the `setup()` function within the `setup.py` file. Nowadays, the recommended configuration method is the `pyproject.toml` file. Let's use the `pyldd` tool as an example to demonstrate the `pyproject.toml` [35] configuration.

The `pyproject.toml` file for `pyldd` can be configured as follows:

```
[project]  
name = "pyldd"  
version = "0.1.0"  
description = "Tool for inspecting Python package dependencies"  
readme = "README.md"  
keywords = ["pychem-book", "ldd"]  
dependencies = [  
    "pipreqs>=0.4",  
    "click"  
]
```

```
[project.scripts]
pyldd = "pyldd.cli:main"
```

When a package is published on PyPI, the fields specified in the [project] section is displayed on the project's PyPI page, which can help users understand the general information of the package. In the [project] section, the name and version fields are mandatory. Other fields, such as description, readme, and keywords, are optional. The dependencies field specifies the packages required for the project to operate properly. To avoid potential conflicts among different packages, typically, the versions of the dependencies are not strictly specified.

Let's assume that we have activated a virtual environment /home/ubuntu/chem-env. Given a pyproject.toml defined above, running pip install will install the pyldd package in the /home/ubuntu/chem-env/lib/python3.10/site-packages directory. Furthermore, thanks to the field defined in the [project.scripts] section, an executable script named /home/ubuntu/chem-env/bin/pyldd will be created.

When a Python project includes modules written in C or C++, it is necessary to configure the compilation settings in the setup.py file. Below is a minimal example of a setup.py file for C/C++ extensions.

```
from setuptools import setup, Extension
setup(ext_modules=[
    Extension(
        'example.simple_cpp',
        sources=['src/cpp/example_A.cpp', 'src/cpp/example_B.cpp'],
        language='c++',
    ),
])
```

In the setup.py file, we can utilize the setup() function from the setuptools library to define the extension module. The ext_modules option of setup() specifies how these extensions should be compiled. It accepts a list of Extension objects, which define the source files, compiler options, and other settings to build each extension module. Once the setup.py file is configured, running python setup.py build_ext will compile and link them into shared libraries, which can be imported and used like regular Python modules.

Please note that the setup() function and some other functionalities, such as Extension, are available in both the setuptools package and the Python standard library distutils. The setuptools package offers functionalities that are based on distutils but are more suitable for processing by the pip command. Although setuptools and distutils are closely related, using them together in the same setup.py script can lead to compatibility issues. It is recommended to exclusively use the functions from the setuptools package. We will explore the methods of compiling extension modules in more details in Chapter 8.

In some projects, Cython may be required to compile extension modules. This means that Cython must be installed before running the setup.py file for compilation.

The build dependencies can be specified in the `pyproject.toml` file [35]. For instance, the prerequisites `setuptools`, `wheel`, and `cython` can be defined in the `[build-system]` section like this:

```
[build-system]
requires = ['setuptools', 'wheel', 'cython']
```

The build dependencies should be distinguished from the installation requirements listed in the `dependencies` field of the `[project]` section. Packages mentioned in the `[build-system]` section are installed only during the build stage. These packages will not be included in the final distribution of the package.

1.3.4.3 Building wheels

Python packages have two types of distribution formats:

- Source distribution (`sdist`).
- Binary distribution (`bdist`).

An `sdist` is a `.tar.gz` file that contains all the source code and necessary files for building and installing the package. If the package is managed by `setup.py`, we can run the following command to create a `sdist` file, such as `pyldd-0.1.0.tar.gz`, under the `dist/` directory within the project.

```
$ python setup.py sdist
```

For `pyproject.toml`-managed projects, we need to use the third-party library `build` package to generate the `sdist` file.

```
$ pip install build
$ python -m build -s .
```

A `bdist` is also called a *wheel*, which is a ZIP-archived file with the suffix `.whl` in its filename. This implies the contents of a wheel file can be extracted using the `unzip` command. A wheel file contains pre-built binary files for specific platforms. If a Python project includes C/C++ extensions, installing its `sdist` package requires compiling the extensions at installation time, which can be inconvenient. The wheel release eliminates the need of compiling extensions during the installation process, simplifying the installation.

To generate a wheel file, we can use the following `bdist_wheel` command. It produces a wheel file `pyldd-0.1.0-py3-none-any.whl` in the `dist/` directory.

```
$ pip install wheel
$ python setup.py bdist_wheel
```

For projects managed by `pyproject.toml`, `bdist_wheel` can be compiled using the `build` package:

```
$ pip install build
$ python -m build -w .
```

Please note that the wheel file generated by the above commands may not be runnable by other users in their local Python environments. The C extensions in the wheel file might depend on specific system libraries that are only exist on the system where the wheel package was built. Although installing the `bdist` package file in a different Python environment may not display any errors, the program might crash during runtime due to missing dependent libraries.

To ensure compatibility across different platforms, before releasing the wheel for public use, we need to utilize a tool called `auditwheel` to verify and repair the binary dependencies in the wheel file:

```
$ auditwheel -v repair dist/pyldd-0.1.0-py3-none-any.whl -w wheelhouse/
```

This command will generate a new wheel file in the `wheelhouse/` directory with a platform-specific filename. The filename of the generated wheel cannot be arbitrarily changed. As described by PEP-0427 [36], the naming convention for wheels reflects the platforms that the wheel supports. The `pip install` command relies on this naming convention to search for compatible wheels. By reading the filename of a wheel, we can identify the compatible operating systems and Python versions. For example, the filename `pyldd-0.1.0-py3-none-any.whl` indicates that it is a pure Python package, which can be executed on any platforms by any Python 3 interpreters. Conversely, the filename `numpy-1.26.4-cp310-cp310-macosx_11_0_arm64.whl` indicates that this wheel is specific to Apple Silicon platform and is only compatible with Python 3.10. All packages hosted on the `pypi.org` server follow this convention.

Both `sdist` and `bdist` can be uploaded to PyPI server using tools such as `twine`.

```
$ twine upload dist/pyldd-0.1.0.tar.gz
```

PyPI server enforces very strict rules on package versions. If a version already exists in the upload history, the server will reject any new uploads with that same version number, even if the original file has been deleted from the server. The version check is conducted based on the metadata specified in `pyproject.toml` or `setup.py`. Simply changing the filename cannot bypass the version check. Therefore, you should ensure the correctness of a package before uploading it to the PyPI server.

1.3.4.4 Files in the PyPI package

When packaging a project in `sdist` or `wheel` formats, the `setuptools.find_packages` function in `setup.py`, or the `[options.packages.find]` section in `pyproject.toml`, can automatically locate and include all Python modules in the package, as long as the `__init__.py` file is present. Therefore, to ensure that all Python source code is included, we can create the `__init__.py` file in all directories, even if it is an empty file.

These files can be specified in `MANIFEST.in`. Beyond individual files, the `MANIFEST.in` configuration also allows for providing rules to match files that need

to be included or excluded [37]. We will not delve into the details of the `MANIFEST.in` configuration in this book. Thanks to the progress of AI coding assistants, it is now feasible to use AI tools to generate these rules based on descriptions provided in human languages.

1.3.4.5 Building Conda packages

In addition to packaging a wheel release for PyPI, Conda is another popular option for distributing packages. Although wheels provide the capability to package pre-compiled binary libraries, they cannot effectively manage complex dependency trees for binary libraries beyond Python extensions. Due to the differences between packaging and runtime environments, `auditwheel` imposes strict restrictions on binary libraries to ensure compatibility across various systems. Many `bdist` wheels fail in `auditwheel` checks due to these restrictions. In contrast, Conda can manage not only Python libraries but also system libraries. The Conda package manager can provide a consistent environment for both package building and runtime. There are far fewer restrictions on binary libraries within a Conda package.

To build a Conda package, we need a *Conda recipe* file, `meta.yaml`, to specify the metadata of the package. Similar to the metadata recorded in `pyproject.toml`, we can define the name, version, description, dependencies, and other information in the Conda recipe file. The `conda-build` tool needs this information to construct a Conda package.

Taking the `pyldd` project as an example, we created a `meta.yaml` file in the `conda-recipe` directory, which is located in the root directory of the `pyldd` repository. The minimal configuration for the `meta.yaml` file is as follows.

```
package:
  name: pyldd
  version: 0.1.0

source:
  path: ..

build:
  script: pip install --no-deps .

requirements:
  host:
    - python
  run: # List runtime dependencies here
    - python
    - click
```

Next, we can install the `conda-build` package and execute the following command in the `pyldd` root directory to compile the recipe.

```
$ conda-build conda-recipe
```

This command generates a Conda package named `pyldd-0.1.0-py310_0.tar.bz2`. To distribute the package for others to use, we can create a channel on the Conda cloud and upload the new package to this channel. For more details on how to upload a package, please refer to the Conda documentation [38].

You might have noticed that in the `pyproject.toml` file (Section 1.3.4.2), the `pipreqs` package is presented as a necessary dependency in the `dependency` field. However, in `Conda meta.yaml`, this package is not specified as a dependency. We bypass this library because the dependencies of a Conda package must be Conda packages. Conda packages cannot reference external packages, such as those distributed on PyPI. For instance, `pipreqs` is not available in Conda, and therefore, it cannot be listed in the `requirements` entry.

So, how can we manage external dependencies in a Conda package?

One strategy is to include additional channels, such as the `conda-forge` channel. The `conda-forge` channel hosts a huge amount of packages. We can configure the `~/.condarc` file to include new channels in addition to the default channel, as shown below:

```
channels:
  - defaults
  - conda-forge
```

Please notice, due to the extensive collection of packages hosted on `conda-forge`, enabling this channel can significantly slow down the performance of Conda commands.

Another approach is to create a Conda package for the `pipreqs` library and release it to a specific channel on the Conda cloud. This channel can then be added to the `~/.condarc` file. Subsequently, the `pipreqs` package can be included as the runtime dependencies in `meta.yaml`.

1.3.5 CI and DevOps workflow

Continuous Integration and Continuous Deployment (CI/CD) are workflows frequently utilized in software projects to simplify engineering operations. A CI/CD workflow can include various types of jobs. Common components of a workflow include:

- *Automated testing*, which ensures that any changes made to the codebase are validated before they are merged into the main branch.
- *Code quality analysis*, which maintains code readability and consistency.
- *Automated build*, which compiles the code and resolves dependencies. If the code is correctly built, this job generates application artifacts for testing or releasing.
- *Code release*, which generates documentation and release notes, and pushes the artifact generated from the build stage to certain repositories.

- *Notification*, which reports errors, warnings, or other information generated by the workflow to specific users.

When hosting a codebase on platforms like GitHub, it is straightforward to execute CI workflow with GitHub Actions. GitHub Actions are defined in a YAML file located in the `.github/workflows` directory of the repository. For instance, let us consider a use case where multiple developers are working on different features of a project, and new features are merged into the main branch through GitHub pull requests (PR). To ensure that new pull requests do not break the codebase, we can create an action that runs automatically whenever a pull request is issued. The `.github/workflows/pr_test.yml` can be configured as:

```
name: Testing for PR
on: pull_request
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4           # (1)
      - name: Set up Python
        uses: actions/setup-python@v4       # (2)
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: pip install -e . pytest         # (3)
      - name: Static analysis
        uses: py-actions/flake8@v2         # (4)
        with:
          max-line-length: 120
          path: src
      - name: Test with pytest             # (5)
        run:
          - |
            pytest tests/
```

This workflow defines a single job called `test` that runs on an Ubuntu environment. Each job can include multiple steps. In each step, one can either use a predefined GitHub Action to execute a series of processes, as shown in line (1), (2), (4), or run a shell script specified by the key `run` in line (3) and (5). Predefined GitHub Actions can be found in GitHub Marketplace.

There are several additional points to keep in mind when using GitHub Actions:

- Tokens and security risk. In some workflows, we may need to visit private resources with passwords or tokens. Passwords and tokens should not be hardcoded in the workflows or anywhere in the source code of the project. The recommended method is to use GitHub's built-in secrets feature (Fig. 1.3) to store and manage tokens [39].

- The cost of running workflows. GitHub offers a certain amount of free usage quota for workflows. When designing unit tests or other types of tests to run in GitHub Actions, it is advisable to minimize their resource usage.

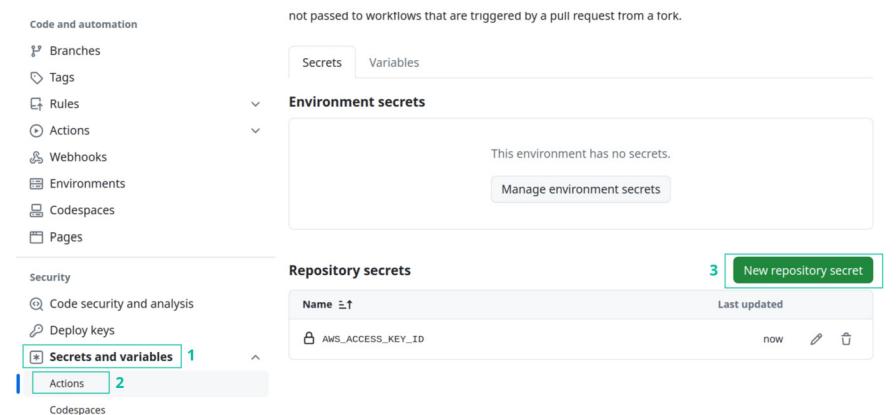


FIGURE 1.3

Managing passwords and tokens using GitHub secrets.

1.3.5.1 Self-hosted runners

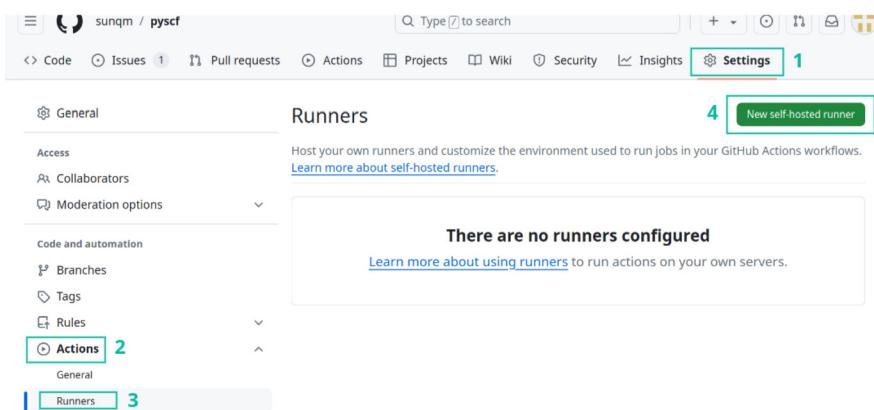
GitHub Actions can be configured to execute on the Microsoft Azure cloud. However, the Microsoft Azure cloud may not be the ideal choice if a job requires specialized hardware (such as GPU) or needs to access data that are inconvenient to transfer over the internet. In these instances, self-hosted runners can be a useful alternative.

To illustrate how self-hosted runners can be used, in the following example, we will create a dedicated runner that is configured to run GPU-dependent jobs:

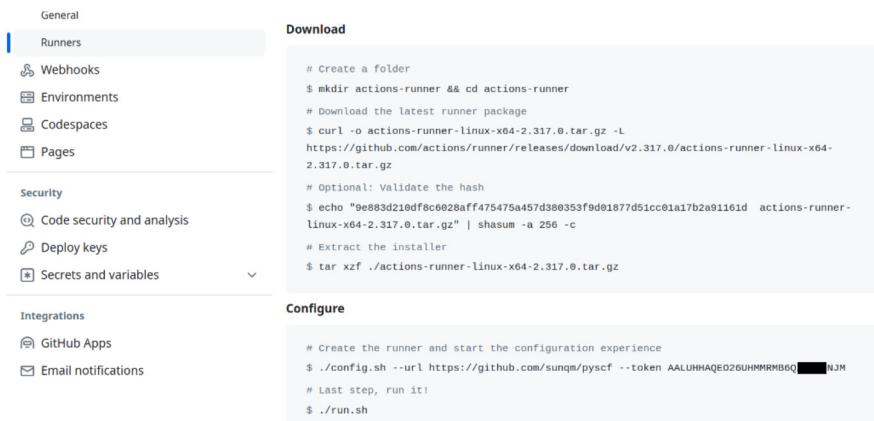
- Configurations on GitHub web, as shown in Fig. 1.4.
- Download and install the runner package following the instructions provided by the GitHub page (Fig. 1.5).
- Build a Docker image for the local runner which includes Nvidia GPU runtime dependencies.

```
FROM nvidia/cuda:12.1.1-base-ubuntu22.04
RUN apt update && \
    apt install -y ca-certificates curl git python3 && \
    mkdir -p /home/runner
WORKDIR /home/runner
```

The Docker image with named `local-gpu-runner:latest` can be created locally:

**FIGURE 1.4**

Configuring GitHub self-hosted runner.

**FIGURE 1.5**

Install instruction of GitHub self-hosted runner.

```
$ docker build -t local-gpu-runner:0.1 .
```

4. Create a docker-compose.yml file to manage the self-hosted runner.

```
version: "3.7"
services:
  runner:
    image: local-gpu-runner:latest
    volumes:
```

```

    - /data/github-runner:/home/runner
    - /data/modelA:/data
  command: /home/runner/run.sh
  runtime: nvidia

```

The directory `/data/github-runner` is the folder where we unzip the actions-runner package. This directory contains a script file `run.sh` that can start the local action runner. Please note that the `runtime` key in the `docker-compose` file is set to `nvidia`, which is a custom Docker runtime for Nvidia GPUs. The Docker runtime requires the separate installation of the `nvidia-container-runtime` package on the operating system [40]. We can then start the runner by executing:

```
$ docker-compose up -d
```

1.3.5.2 PyPI release workflow

Building wheels and releasing them to PyPI via GitHub Action is a very common task. Therefore, there are some predefined actions in the GitHub Marketplace to automate this job. In the configuration below, we utilize predefined actions to automate the release of PyPI packages:

```

name: Wheels
on:
  release:
    types:
      # trigger the workflow when a new release is tagged
      - released
jobs:
  build_wheels:
    name: Wheels for Mac OS
    runs-on: macos-13
    steps:
      - uses: actions/checkout@v4
      - name: Build wheels
        uses: pypa/cibuildwheel@v2.14.1
      - name: Upload package to PyPI
        uses: pypa/gh-action-pypi-publish@release/v1
        with:
          password: ${{ secrets.PYPI_API_TOKEN }}

```

1.4 Modular design and object-oriented programming

As a program project evolves and expands with more functionalities, these features can be organized based on their common characteristics. This leads to a modular de-

sign and sometimes an object-oriented code structure. *Object-oriented programming* (OOP) is a very successful programming paradigm that is adopted in many programming projects. In various scenarios, it provides a natural way to map programming abstractions to the problems being solved. Given its popularity, it is not difficult to find books and courses on OOP in Python, such as the book *Fluent Python* [41] by Luciano Ramalho or *Think Python* [42] by Allen B. Downey.

In the realm of quantum chemistry, various objects and concepts can be adapted to the OOP framework. For instance, entities like atoms, molecules, wavefunctions, and Hamiltonians can be effectively modeled using OOP classes. Moreover, quantum chemistry methods often exhibit close connections. A clear hierarchy can often be observed among quantum chemistry methods. Therefore, it is logical to organize these methods using OOP classes through inheritance. How can the OOP design be applied to quantum chemistry programs? A detailed discussion on this topic will be presented in Chapter 13, where we will use the design for mean-field programs as an example.

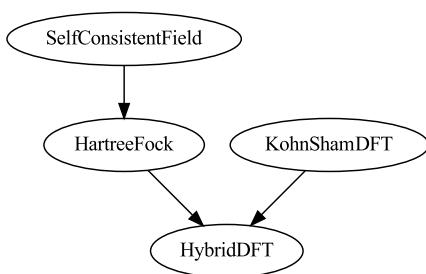
Although OOP is a very useful paradigm, applying OOP can sometimes introduce various challenges. One such challenge is *multiple inheritance*. Given the complex relationships among quantum chemistry methods, there may be a desire to use multiple inheritance to model these relationships. However, multiple inheritance can lead to increased complexity and make the code difficult to understand.

Another challenge in OOP arises in the context of *parallel programming*. Parallel computation is an important aspect in quantum chemistry programs. When dealing with mutable objects, such as wavefunctions, one would have to carefully synchronize the states of objects across threads or processes to ensure their consistency. The cost of synchronization can have a significant impact on the efficiency of parallel programs. This issue will be discussed further in Chapter 10.

1.4.1 Method resolution order in multiple inheritances

Class inheritance allows the child class to access methods that are developed in the parent classes. In the case of single inheritance, to determine which method a child class will use, we can simply track the parent classes recursively until finding the relevant method. In the case of multiple inheritances, determining the correct method to call is more complicated. When method name collisions occur in parent classes, it can be ambiguous which methods are actually called.

For instance, let's assume we have an Self-Consistent Field (SCF) base class with a method to check SCF convergence. We can define a subclass for the Hartree-Fock method that inherits from the SCF base class. Additionally, we might have a class for the Kohn-Sham Density Functional Theory (DFT) method, which uses a different convergence checker. To implement a hybrid DFT method that combines the Hartree-Fock and DFT methods, we create a child class that inherits from both the Hartree-Fock and Kohn-Sham DFT classes (Fig. 1.6), which leads to the use of multiple inheritance.

**FIGURE 1.6**

A simple class inheritance for Hartree-Fock, Kohn-Sham, and hybrid DFT classes.

```

class SelfConsistentField:
    def check_convergence(self):
        print('Default convergence checker')

class HartreeFock(SelfConsistentField):
    pass

class KohnShamDFT:
    def check_convergence(self):
        print('Convergence checker for DFT')

class HybridDFT(HartreeFock, KohnShamDFT):
    pass
  
```

When calling the `check_convergence` method in an instance of the `HybridDFT` class, which parent class does it inherit?

As a DFT class, we might hope the `HybridDFT` class to reuse the methods defined in the parent class `KohnShamDFT`. However, this expectation is not met in this class with multiple inheritances. By executing the code below, we can observe that the `check_convergence` method is actually resolved to the method implemented in the base class `SelfConsistentField`.

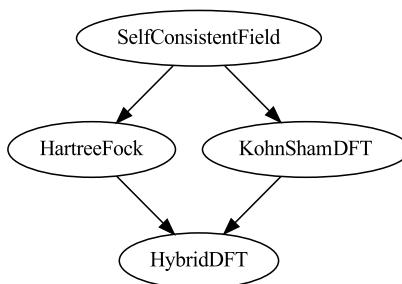
```

In [1]: HybridDFT().check_convergence()
Out[1]:
Default convergence checker
  
```

If we make a small change to the implementation of the `KohnShamDFT` class (Fig. 1.7):

```

class KohnShamDFT(SelfConsistentField):
    def check_convergence(self):
        print('Convergence checker for DFT')
  
```

**FIGURE 1.7**

A modified version of class inheritance for Hartree-Fock, Kohn-Sham, and hybrid DFT classes.

the `check_convergence` method will then defer to the implementation in the `KohnShamDFT` class

```
In [2]: HybridDFT().check_convergence()
Out[2]:
Convergence checker for DFT
```

How can this occur for such a minor difference?

Here is what happens in Python's inheritance mechanism. When the method `HybridDFT.check_convergence` is called, Python searches for the method following a specific sequence known as the Method Resolution Order (MRO). If the method name is found in multiple classes, Python selects the one that appears first in the MRO.

The MRO can be accessed via the `__mro__` attribute of a class. In this example, using the first version of the `KohnShamDFT` class, the MRO for `HybridDFT` class is

```
In [3]: HybridDFT.__mro__
Out[3]:
(<class 'HybridDFT'>, <class 'HartreeFock'>, <class 'SelfConsistentField'>,
 <class 'KohnShamDFT'>, <class 'object'>)
```

Following the MRO, Python first finds the `check_convergence` method in the `SelfConsistentField` base class. When the second version of `KohnShamDFT` class is employed, the MRO for the `HybridDFT` class turns to

```
(<class 'HybridDFT'>, <class 'HartreeFock'>, <class 'KohnShamDFT'>, <class
 'SelfConsistentField'>, <class 'object'>)
```

In this case, Python locates the first available `check_convergence` method in the `KohnShamDFT` class.

Python constructs MRO by a specific algorithm known as the C3 linearization [43] to uniquely determine the MRO. This algorithm is designed to establish a consis-

tent and reproducible MRO for multiple inheritance classes. Although it guarantees that MRO is deterministic, this does not imply that ambiguity within the multiple inheritance classes is reduced. For example, the difference in the two MROs of `HybridDFT` class we presented above is caused by a minor hierarchy difference in the secondary parent class `KohnShamDFT`, which may be easily overlooked when working on the child class `HybridDFT`.

To reduce ambiguity and confusion in an OOP program, it is advisable to limit the use of multiple inheritances whenever possible. If multiple inheritance is inevitable, it is recommended to explicitly define any ambiguous methods in the child class to ensure clarity. This can be achieved by directly calling the implementation from the desired parent class. For instance, the `KohnShamDFT.check_convergence` method can be explicitly called within the `HybridDFT.check_convergence` method:

```
class HybridDFT(HartreeFock, KohnShamDFT):
    def check_convergence(self):
        return KohnShamDFT.check_convergence(self)
```

Alternatively, the method from the parent class can be explicitly assigned to an attribute of the child class.

```
class HybridDFT(HartreeFock, KohnShamDFT):
    check_convergence = KohnShamDFT.check_convergence
```

It is worth noting that in this example, we explicitly spell the name of the parent class instead of using the Python built-in function `super()` to determine the parent class. Why don't we use the `super()` function here? This is because `super()` relies on the MRO to locate methods in the superclass. In this example, we aim to call the method of a specific parent class without relying on the MRO. Using `super()` would reintroduce the MRO and cause ambiguity.

This raises a related question: In which scenarios is the use of `super()` advantageous? Consider a situation where we need to access each overloaded method from the parent classes. Using `super()` in this scenario can automatically locate and execute the available methods along the MRO chain, as shown in the following example:

```
class SelfConsistentField:
    def check_convergence(self):
        print('Default convergence checker')
        # Some code to test convergence
        converged = ...
        return converged

class HartreeFock(SelfConsistentField):
    pass

class KohnShamDFT(SelfConsistentField):
    def check_convergence(self):
```

```
print('Convergence checker for DFT')
converged = ...
return super().check_convergence() and converged

class HybridDFT(HartreeFock, KohnShamDFT):
    def check_convergence(self):
        print('Convergence checker for HybridDFT')
        converged = ...
        return super().check_convergence() and converged
```

There is no need to specify which parent class to refer to in the child class. When the `check_convergence` method is called from the child class, it triggers the execution of each `check_convergence` method within the parent classes.

```
In [1]: HybridDFT().check_convergence()
Out[1]:
Convergence checker for HybridDFT
Convergence checker for DFT
Default convergence checker
```

1.4.2 Mixins

Mixin is a design strategy in OOP programs to facilitate code reuse and minimize method conflicts in inheritance hierarchies. Essentially, a Mixin is a set of methods to augment the functionality of other classes. Technically, to provide complex functionalities, multiple Mixins can be combined through class multiple inheritances. Ideally, each Mixin within the class should offer a distinct functionality. Different Mixins are expected to deliver orthogonal functionalities, so as to reduce the risk of method collisions.

For instance, by adopting the design of Mixins, the `HybridDFT` class and its parent classes can be structured as

```
class SelfConsistentField:
    pass

class HartreeFockMixin:
    pass

class KohnShamDFTMixin:
    pass

class SCFOptimizationMixin:
    def check_convergence(self):
        print('Default convergence checker')
```

```

class DFTOptimizationMixin(SCFOptimizationMixin):
    def check_convergence(self):
        super().check_convergence()
        print('Convergence checker for DFT')

class HartreeFock(SelfConsistentField, HartreeFockMixin,
                  SCFOptimizationMixin):
    pass

class KohnShamDFT(SelfConsistentField, KohnShamDFTMixin,
                   DFTOptimizationMixin):
    pass

class HybridDFT(SelfConsistentField, HartreeFockMixin, KohnShamDFTMixin,
                DFTOptimizationMixin):
    pass

```

The `check_convergence` methods are placed in classes dedicated to wave-function optimization. Similarly, other features for each theoretical model are grouped and implemented in their respective Mixins. When developing a new model, such as the `HybridDFT` class, the new class is derived by assembling the Mixins of relevant theory models. This class hierarchy eliminates the ambiguity of `check_convergence` method in the `HybridDFT` class, since the only available candidate is provided by the `DFTOptimizationMixin` class.

It should be noted that Mixins do not automatically resolve the ambiguities in multiple inheritances. It requires careful design and implementations to ensure that every parent class, including both the base class and Mixins, contributes a distinct set of methods. The Mixins approach is essentially a more complex form of multiple inheritance. The use of Mixins can potentially complicate the scenario of multiple inheritance even further and lead to various unintended name collisions within subclasses.

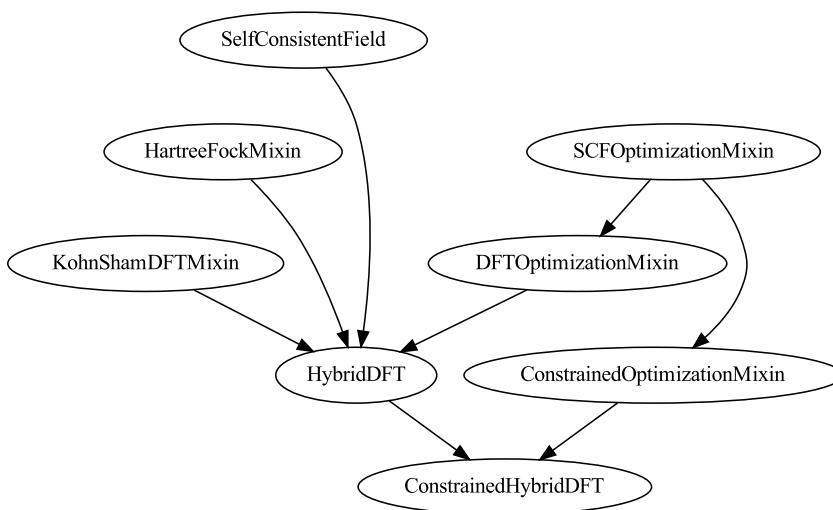
For instance, suppose we have developed a new model `ConstrainedHybridDFT` based on `HybridDFT` with a new optimization algorithm that supports constraints on the locality of electron density. If the new classes are implemented as below, we will encounter ambiguity in the `check_convergence` method. This is due to the diamond inheritance structure in the `OptimizationMixin` series (Fig. 1.8).

```

class ConstrainedOptimizationMixin(SCFOptimizationMixin):
    def check_convergence(self):
        super().check_convergence()
        print('constrained optimization convergence checker')

class ConstrainedHybridDFT(HybridDFT, ConstrainedOptimizationMixin):
    pass

```

**FIGURE 1.8**

Diamond inheritances in the `ConstrainedHybridDFT` class.

1.5 Working with IPython and Jupyter

IPython and Jupyter are extensively used in data science and scientific computing. IPython is an enhanced Python shell with additional functionalities, such as smart tab-completion, and editable history. Jupyter is a web-based interactive computing platform that supports multiple programming languages, such as Python, R, and Julia. Jupyter offers a notebook environment that allows for interactive code execution in a web browser.

The usage of IPython and Jupyter is intuitive. Typing one question mark

`<keyword>?`

allows us to access the *docstring* of the function, while two question marks

`<keyword>??`

additionally display the source code. It is easy to find lively demonstrations of IPython and Jupyter online. If you have never used IPython or Jupyter before, it is recommended to begin by following online tutorials for these excellent tools [44].

1.5.1 Startup configurations

IPython and Jupyter Notebook share the same startup configurations, which are Python scripts located in the folder `~/.ipython/profile_default/startup/`. We can create a script in this directory to import commonly used libraries, such as

```
# Import regularly used libraries
import numpy as np
import pandas as pd
```

Python scripts located in the startup folder will be executed in sequence, based on the order of their filenames. If startup actions are configured differently in various scripts, all of the configurations will be executed. However, for variables that are defined in the global namespace, the definitions from scripts executed later will overwrite those from earlier scripts.

1.5.2 Extensions

Jupyter offers a variety of extensions to improve the functionality of Jupyter Notebooks. To activate these extensions in Jupyter, we need to install additional libraries:

```
$ pip install jupyter jupyter_contrib_nbextensions "notebook~=6.0"
```

Please be aware that an older version of the `notebook` package may be required, as the latest version is not fully compatible with the Jupyter Notebook extension. There are many common extensions available, including those for debugging, autocompletion, auto indentation, visualization, Git, etc. To enable these extensions within Jupyter, the following commands can be used:

```
$ jupyter nbextension install extension-name --user
$ jupyter nbextension enable extension-name --user
$ jupyter nbextension list
```

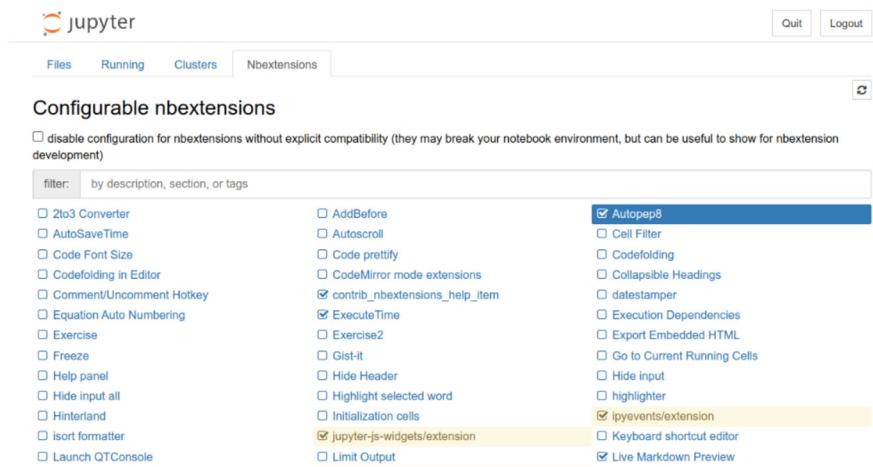
Additionally, we can choose which extensions to load within the Jupyter Notebook, as illustrated in Fig. 1.9.

On certain systems, you might actually have installed JupyterLab [45] instead of Jupyter Notebook. Although the two projects are closely related, their extensions are not interchangeable. The previous commands `jupyter nbextension` are not applicable to JupyterLab. To manage extensions in JupyterLab, the `jupyter labextension` command should be used.

1.5.3 Magics

A notable feature in IPython (and Jupyter) is known as *magics*. IPython magics are special commands that take the contents of an IPython cell as input and execute specific tasks. For example, we can measure the execution period of a code block using the `%time` magic.

```
In [1]: %time
      a = set()
      for i in range(100000):
          a.add(i)
```

**FIGURE 1.9**

Configuring extensions in Jupyter Notebook.

```
CPU times: user 24 ms, sys: 4.05 ms, total: 28.1 ms
Wall time: 27.6 ms
```

What magics does IPython offer? We can list all available magics using the `%lsmagic` command. In addition to the IPython built-in magics, we can load magic extensions using the `%load_ext` command. For example, a magic for the Cython compiler (more details in Chapter 9) can be loaded into IPython. Using Cython in the standard way usually involves a complex setup. The `%cython` magic enables us to compile functions within the IPython shell in a simple manner.

```
In [2]: %load_ext cython

In [3]: %%cython
def incremental_set():
    a = set()
    for i in range(100000):
        a.add(i)

In [4]: %time incremental_set()
CPU times: user 1.74 ms, sys: 8.2 ms, total: 9.94 ms
Wall time: 10 ms
```

Moreover, we can customize magics through the interface provided by IPython [46]. For instance, we can create a magic to integrate AI tools into our programming workflow. Here, we use OpenAI's ChatGPT as an example, customizing a magic to function as an AI programming assistant:

```

try:
    import openai
except ImportError:
    pass
else:
    from IPython.core.magic import register_cell_magic

@register_cell_magic
def hello_gpt(line, cell):
    client = openai.OpenAI(api_key='YOUR OPENAI TOKEN STRING')
    prompts = [ {'role': 'system', 'content': ''},
    - You are an AI programming assistant.
    - The output should be in a single code block.
    - Skip any installation instructions or basic usages.
    - When explaining the techniques, write them as comments within the
        generated code.
    - Avoid wrapping the response in triple backticks.
    ''', {'role': 'user', 'content': f'Create a Python function\n{n{cell}}'}]
    ret = client.chat.completions.create(
        model='gpt-4-turbo',
        messages=prompts,
        max_tokens=1000,
        timeout=20,
        temperature=0.7,
    )
    print(ret.choices[0].message.content)

```

This code should be placed in a script file within IPython's startup directory. When IPython or a Jupyter Notebook is launched, the new cell magic `%%hello_gpt` will be automatically registered. Within the Jupyter Notebook, we can use this cell magic command to generate code based on the requirements specified in the cell block.

```

In [2]: %%hello_gpt
        # This function uses chatgpt API to generate code as requested
        def query_gpt(request):
Out[2]:
def query_gpt(request):
    import openai
    ...
    try:
        response = openai.Completion.create(
            engine=model,
            prompt=request,
            max_tokens=150

```

```
)  
...  
}
```

1.5.4 Remote execution

Jupyter Notebooks are not limited to use on local machines. We can launch a Jupyter kernel on a remote machine and interact with it via a local web browser. The following command can initiate a Jupyter Notebook kernel and display the login method, which includes the server URL and a password token for login. Using this URL and password token, we can access the remote Jupyter Notebook server.

```
$ jupyter notebook --no-browser --ip "*"
```

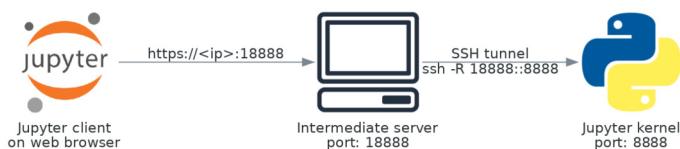


FIGURE 1.10

Remote execution of Jupyter Notebook via an intermediate server.

If the Jupyter kernel is launched on a cluster with limited public network access, we can access the Jupyter Notebook kernel via SSH tunnel, as shown in Fig. 1.10. Assuming that there exists an *intermediate server* to which both the Jupyter kernel and the client can connect, we can establish an SSH tunnel that forwards requests from the intermediate server to the remote computing server. To configure this tunnel, the following command can be executed on the *remote server*:

```
$ ssh -fN -R 18888:localhost:8888 intermediate_server_ip
```

Then, by visiting the URL `https://<intermediate_server_ip>:18888`, we can access the remote Jupyter Notebook.

Summary

This chapter explores various aspects of the Python programming environment and the Python package ecosystem, including:

- The mechanism of the Python runtime.
- Python package management systems.
- How to create and manage an isolated Python runtime environment.
- How to manage Python projects.

- How to publish Python packages.
- The design principles of object-oriented Python projects.
- The configuration of IPython and Jupyter Notebook.

These topics are frequently overlooked during the journey of learning Python programming. However, in practice, they are common issues that one may frequently encounter. Investing time to understand these technical aspects is beneficial. Technologies like Git, Docker, and CI/CD workflows, although not exclusive to Python programming, are commonly used in the development of Python applications. Being familiar with these technologies can greatly enhance the efficiency of both program development and scientific research.

References

- [1] The pip developers, Configuration of pip, <https://pip.pypa.io/en/stable/topics/configuration/>, 2024.
- [2] Python Packaging Authority, Hosting your own simple repository, <https://packaging.python.org/en/latest/guides/hosting-your-own-index/>, 2024.
- [3] Python Packaging Authority, Packaging namespace packages, <https://packaging.python.org/en/latest/guides/packaging-namespace-packages/>, 2024.
- [4] E.V. Smith, Pep 420 – implicit namespace packages, <https://peps.python.org/pep-0420/>, Apr. 2012.
- [5] A. Ronacher, Click documentation, <https://click.palletsprojects.com/en/8.1.x/>, 2024.
- [6] S. Ramírez, Typer, <https://typer.tiangolo.com/>, 2024.
- [7] Google, Python fire, <https://google.github.io/python-fire/>, 2024.
- [8] A. Ronacher, Click documentation - shell completion, <https://click.palletsprojects.com/en/8.1.x/shell-completion/>, 2024.
- [9] Conda Development Team, Anaconda documentation, <https://docs.anaconda.com/anaconda/index.html>, 2024.
- [10] QuantStack & mamba contributors, Mamba documentation, <https://mamba.readthedocs.io/en/latest/>, 2024.
- [11] Conda Development Team, Manage environments - conda documentation, <https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>, 2023.
- [12] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and Linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171–172, <https://doi.org/10.1109/ISPASS.2015.7095802>.
- [13] P. Saha, A. Beltre, P. Uminski, M. Govindaraju, Evaluation of docker containers for scientific workloads in the cloud, in: Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity, PEARC ’18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–8, <https://doi.org/10.1145/3219104.3229280>.
- [14] Docker, Inc., Docker documentation, <https://docs.docker.com/manuals/>, 2024.
- [15] Docker, Inc., Run the docker daemon as a non-root user (rootless mode), <https://docs.docker.com/engine/security/rootless/>, 2024.
- [16] The PyPy Team, PyPy documentation, <https://pypy.org/features.html>, 2024.
- [17] M. Herman, Setting up a private PyPI server, <https://testdriven.io/blog/private-pypi/>, Jul. 2022.

- [18] Docker, Inc., Dockerfile reference, <https://docs.docker.com/reference/dockerfile/>, 2024.
- [19] S. Chacon, B. Straub, Pro Git: Everything You Need to Know About Git, 2nd edition, Apress, 2014, <https://git-scm.com/book/en/v2>.
- [20] GitHub, Inc., Incorporating changes from a pull request, <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request>, 2024.
- [21] GitHub, Inc., Syncing a fork, <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/syncing-a-fork>, 2024.
- [22] C. Kaner, J. Bach, B. Pettichord, Lessons Learned in Software Testing, John Wiley & Sons, Inc., USA, 2001.
- [23] D. Hillard, Effective Python testing with pytest, <https://realpython.com/pytest-python-testing/>, 2020.
- [24] Logilab and Pylint contributors, Pylint documentation, <https://pylint.readthedocs.io/en/stable/>, 2024.
- [25] Astral, Ruff documentation, <https://docs.astral.sh/ruff/>, 2024.
- [26] I. Cordasco, Flake8 documentation, <https://flake8.pycqa.org/en/latest/>, 2024.
- [27] J. Lehtosalo, Mypy documentation, <https://mypy.readthedocs.io/en/stable/>, 2024.
- [28] G. van Rossum, B. Warsaw, N. Coghlan, Pep 8 – style guide for Python code, <https://peps.python.org/pep-0008/>, 2001.
- [29] Microsoft, Pylint extension for visual studio code, <https://marketplace.visualstudio.com/items?itemName=ms-python pylint>, 2024.
- [30] G. McConaughey, The big ol' list of rules, <https://www.flake8rules.com/>, 2024.
- [31] Astral, The Ruff Linter - error suppression, <https://docs.astral.sh/ruff/linter/#error-suppression>, 2024.
- [32] Logilab and Pylint contributors, Pylint checkers' options and switches, https://pylint.readthedocs.io/en/latest/user_guide/checkers/features.html, 2024.
- [33] Python Packaging Authority, Packaging Python projects, <https://packaging.python.org/en/latest/tutorials/packaging-projects/>, 2024.
- [34] Read the Docs, Inc & contributors, Getting started with sphinx, <https://docs.readthedocs.io/en/stable/intro/getting-started-with-sphinx.html>, 2024.
- [35] Python Packaging Authority, pyproject.toml specification, <https://packaging.python.org/en/latest/specifications/pyproject-toml/#pyproject-toml-spec>, 2024.
- [36] D. Holth, Pep 427 – the wheel binary package format 1.0, <https://peps.python.org/pep-0427/#file-name-convention>, Sep. 2012.
- [37] Python Packaging Authority, Controlling files in the distribution, <https://setuptools.pypa.io/en/latest/userguide/miscellaneous.html>, 2024.
- [38] Anaconda, Inc., Channels and packages, <https://enterprise-docs.anaconda.com/en/latest/data-science-workflows/packages/channels.html#uploading-a-package-to-a-channel>, 2024.
- [39] GitHub, Inc., Using secrets in GitHub actions, <https://docs.github.com/en/actions/security-guides/using-secrets-in-github-actions>, 2024.
- [40] NVIDIA Corporation, Installing the NVIDIA container toolkit, <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html>, 2024.
- [41] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, O'Reilly Media, 2015, <https://books.google.co.in/books?id=bIZHCgAAQBAJ>.
- [42] A. Downey, Think Python. How to Think Like a Computer Scientist, 2nd edition, Green Tea Press, Needham, Massachusetts, 2014.
- [43] M. Simionato, The Python 2.3 method resolution order, <https://www.python.org/download/releases/2.3/mro/>, 2003.

- [44] Jupyter Team, The Jupyter Notebook, <https://jupyter-notebook.readthedocs.io/en/latest/notebook.html>, 2024.
- [45] Jupyter Project, Jupyterlab documentation, <https://jupyterlab.readthedocs.io/en/latest/>, 2024.
- [46] The IPython Development Team, IPython documentation - defining custom magics, <https://ipython.readthedocs.io/en/stable/config/custommagics.html#define-custom-magics>, 2024.

Data processing

2

NumPy and Pandas are the cornerstones of the Python data processing and scientific computing ecosystem. Using NumPy and Pandas for data processing is generally straightforward. You may have been using them in your work for a long time. However, when working with these tools in your applications, you might have encountered various problems, such as:

- When I modify one array, why does the data of another array change as well?
- How can I access a small block of a high-dimensional array without using loops?
- I know NumPy fancy indexing is powerful, but I find it challenging to understand. How can I read and write NumPy fancy indexing code more effectively?
- Why does my NumPy program use more memory than expected? Where is all the memory going?
- I occasionally encounter `nan` in my program. Where are the `nan` values coming from, and how can I avoid them?
- Sometimes I can access the elements of a Pandas object using indexing methods similar to those of a NumPy array, but sometimes this either fails or behaves incorrectly. Is there a connection between Pandas and NumPy indexing methods?

To address these questions, we need a deeper understanding of how NumPy and Pandas work. This understanding involves two key aspects:

1. How is data organized in NumPy and Pandas objects?
2. How are data types managed?

The discussion in this chapter is built on these considerations.

Understanding the data structures and data type management in NumPy and Pandas is essential. This knowledge not only enables us to use these tools more effectively but also serves as foundational knowledge for further optimization using compilation techniques. The technique of compilation optimization for numerical computation will be discussed in more detail in Chapter 8 and Chapter 9.

In this chapter, we will occasionally analyze numerical computation techniques from the perspective of hardware and operating systems. Understanding these technical backgrounds is helpful in writing efficient programs. However, if you are not interested in the details of the underlying systems, you can skip the technical dis-

cussion. These insights are intended to deepen our understanding of the design of numerical computation tools, but they are not essential for using these tools.

2.1 Vectorized data processing with NumPy

When working with a Python list, processing each element involves multiple operations beyond the basic arithmetic computation. These include type checks, accessing underlying methods, and managing the object life-cycle. In numerical computations, the elements of a list are usually of the same data type. It is more efficient to process them as a NumPy array in a vectorized fashion. The high efficiency of NumPy can be attributed to several key aspects:

- The elimination of overhead associated with type checking.
- The execution of arithmetic operations in compiled code.
- The accelerated access to elements thanks to the fixed data types and sizes.
- Improved cache utilization due to more efficient memory usage.

The NumPy module is typically imported with the alias `np`:

```
import numpy as np
```

We will adhere to this import convention throughout this book.

2.1.1 The basics of NumPy

If you are a beginner to NumPy, there are numerous resources available to quickly learn its fundamentals. Besides the official NumPy tutorial at https://numpy.org/doc/stable/user/absolute_beginners.html, there are several useful materials you can refer to.

- *Python Numpy Tutorial (with Jupyter and Colab)* by Justin Johnson, accessible at <https://cs231n.github.io/python-numpy-tutorial/>. This is an interactive tutorial that allows you to work with NumPy code examples in a Jupyter Notebook.
- *Scientific Python Lectures* offers a comprehensive introduction of both basic and advanced NumPy features, accompanied by many illustrative diagrams. This book is available at <https://lectures.scientific-python.org/>
- NumPy learning documentation available at <https://numpy.org/learn/>. This web collects a variety of learning materials for different levels of NumPy (and SciPy) users.

We do not intend to replicate the content of the existing excellent learning resources available for NumPy. In the following, we briefly categorize the basic functionalities of NumPy and list some of the most frequently used functions within each category:

- *Creating an array to store data.* The most direct method to create an array is the `np.array` function, which can convert a Python list or tuple to a NumPy array. Additionally, there are various functions for array creation, such as
 - `np.zeros`
 - `np.zeros_like`
 - `np.empty`
 - `np.empty_like`
 - `np.ones`
 - `np.full`
 - `np.identity`
 - `np.eye`
 - `np.random.rand`
 - `np.arange`
- *Reorganizing an array.* Common functions include
 - `np.reshape`
 - `np.split`
 - `np.roll`
 - `np.transpose`
 - `np.concatenate`
 - `np.stack`
 - `np.hstack`
 - `np.vstack`
 - `np.append`
 - `np.ravel`
 - `np.flatten`
 - `np.swapaxes`
 - `np.rollaxis`
 - `np.sort`
 - `np.partition`
- *Accessing certain elements of an array,* such as
 - `np.diag`
 - `np.tril`
 - `np.triu`
- *Indices of certain elements within an array,* such as
 - `np.where`
 - `np.argwhere`
 - `np.argmax`
 - `np.argmin`
 - `np.argpartition`
 - `np.argsort`
 - `np.diag_indices`
 - `np.tril_indices`
 - `np.triu_indices`
 - `np.unique`

- *Element-wise arithmetic operations*, such as
 - `np.exp`
 - `np.sum`
 - `np.cumsum`
 - `np.any`
 - `np.min`
- *Linear algebra operations*, such as
 - `np.dot`
 - `np.roots`
 - `np.linalg.norm`
 - `np.linalg.eigh`
 - `np.linalg.solve`
- *Sub-modules for additional functionalities*
 - The `linalg` module for linear algebra functions.
 - The `polynomial` module for special polynomials.
 - The `random` module for random number generation.
 - The `ma` module for masked arrays, which supports operations on arrays with missing or invalid values.
 - The `ctypeslib` module includes utility functions for Python `ctypes` module.
 - The `fft` module for Discrete Fourier Transform.

2.1.2 Universal functions (ufunc)

The *universal function* (ufunc) is a fundamental operation for NumPy arrays. Ufunc can perform element-wise operations for elements in NumPy arrays. The basic arithmetic operators (`+`, `-`, `*`, `/`, `**`, `//`, `<<`, `>>`, `|`, and `^`) all operate in the ufunc style. Most arithmetic functions, such as `np.sqrt`, `np.add`, `np.floor`, `np.log`, and `np.cos`, are also ufuncs [1].

Ufuncs allow us to manipulate the entire array objects with a single command, thereby eliminating the need for slow Python loops. The NumPy ufuncs internally vectorize arithmetic operations, which make them more efficient than Python loops for operations on individual elements.

To enhance the performance of ufuncs, there are several strategies to consider:

- It is preferable to use ufuncs on array objects that have a similar contiguous data layout. In such cases, NumPy can work out an optimal iteration plan to traverse the elements in the arrays. Applying ufuncs on arrays with completely opposite data layouts incurs a performance penalty. For example, the operation `a + a.T` is much slower than `a + a` for a square matrix `a`. In Section 2.1.7, we will further explore the data layout of NumPy array and its impact on ufuncs.
- Ufuncs offer a keyword argument `out`, which allows specifying an output array to store the result. This keyword argument can enhance ufuncs performance as it eliminates the need to repeatedly allocate memory for the output.

Let's examine the following example to illustrate the advantages of using the `out` keyword for output storage. Consider the task of finding the points where $P(x) = \epsilon$ along the positive x axis for the distribution function

$$P(x) = x \exp(-ax^2). \quad (2.1)$$

This problem can be solved using the fixed point iteration:

$$x_{i+1} = \sqrt{\frac{1}{a} \log\left(\frac{x_i}{\epsilon}\right)}. \quad (2.2)$$

We can implement this iteration using ufuncs:

```
def search(a, eps=1e-9):
    x = np.full_like(a, 1e3)
    xlast = np.zeros_like(a)
    while np.any(np.abs(x - xlast) > eps):
        xlast = x
        x = np.sqrt(np.log(x/eps)/a)
    return x
```

In each iteration, four additional temporary arrays are created for the intermediate steps, corresponding to the operations `x/eps`, `np.log`, `/a`, and `np.sqrt`. After accounting for the `x` and `xlast` arrays, this function requires more than six times the memory space of the `a` array. It can be a noticeable overhead to allocate and manage the additional arrays, especially for large input arrays. To address this overhead, we can specify the `out` keyword for the output storage, resulting in the following implementation:

```
def search(a, eps=1e-9):
    x = np.full_like(a, 1e3)
    xlast = np.zeros_like(a)
    while np.any(np.abs(x - xlast) > eps):
        xlast, x = x, np.sqrt(np.divide(np.log(np.divide(
            x, eps, out=xlast), out=xlast), a, out=xlast), out=xlast)
    return x
```

This version reuses the memory space of the `xlast` array to store the intermediate results. After calling the `np.sqrt` function, the updated results are stored in `xlast`. We then swap the `x` and `xlast` objects to update the value of `x`. This implementation only requires the memory space for `x` and `xlast`, effectively reducing the memory consumption and the overhead of memory allocation.

Reusing output storage for ufuncs is closely related to inplace operations.

2.1.3 Inplace operations

When applying ufuncs with standard Python operators (`+`, `-`, `*`, `/`, `**`, `//`, `<<`, `>>`), it generates a new object to hold the result. This means that the original array remains

unchanged. In contrast, an *inplace operation* (the standard operators with an = sign, such as +=, >>=) modifies data directly within its own memory space, rather than creating a new object to hold the result.

In general, inplace operations are faster than normal operations due to their improved memory efficiency:

- They reduce memory management overhead by eliminating the need to allocate new memory.
- They reduce the frequency of *page faults* [2] since page faults often occur when accessing newly allocated memory.
- The reused memory space is more likely to be already mapped in the CPU cache, leading to improved *cache hit rates*.

However, using inplace operations requires additional caution and a deeper understanding of the data relationships in the program. When normal operations are replaced by in-place operations, one might encounter bugs that would not occur with normal operations. Performing inplace operations on one array may unintentionally alter the contents of other arrays, as a NumPy array may share data with other arrays. To safely implement in-place operations, it is crucial to ensure whether an array is involved in memory sharing with other arrays. In Section 2.1.8, we will discuss more technical details on data sharing between arrays through the technique of array views.

Inplace operations cannot automatically handle data type conversions of the result. When an operation involves arrays of different data types, this limitation can result in program failures if the operation is expected to yield a different data type than the one provided by the output array. For example, one may receive the following error for incompatible data types:

```
In [1]: a = np.arange(3)
         a /= 2
UFuncTypeError: Cannot cast ufunc 'divide' output from dtype('float64') to
dtype('int64') with casting rule 'same_kind'
```

Fortunately, NumPy can detect most data type incompatibilities caused by inplace operations and raise errors. These errors are generally easy to catch and fix based on the printed error messages.

2.1.4 Broadcasting

If the input arrays for ufuncs have different shapes, ufuncs apply the operation to the input arrays using a method known as broadcasting. Broadcasting follows two rules:

- Padding dimensions. If arrays differ in the number of dimensions, broadcasting automatically expands the dimensions. The array with fewer dimensions is automatically padded with new axes on the left side, each with a dimension size of 1, until arrays have the same number of dimensions. For example, the statement `np.eye(3) * 5` is identical to

```
np.eye(3) * np.array(5)[np.newaxis, np.newaxis]
```

where `np.array(5)[np.newaxis, np.newaxis]` will produce a 1×1 array, equivalent to `array([[5]])`.

- **Compatible dimensions.** Dimensions are compatible if the sizes of corresponding dimensions are identical, or one of them equals to 1. When one array has a dimension of size 1, broadcasting treats the element in that dimension as if it is replicated to match the size of the other array's corresponding dimension. For instance, the previous statement `np.eye(3) * 5` is identical to

```
np.eye(3) * np.array([[5, 5, 5]])
```

and the second array can be further broadcast across the rows, as if performing

```
np.eye(3) * np.array([[5, 5, 5], [5, 5, 5], [5, 5, 5]])
```

In the previous examples, we mentioned the use of `np.newaxis` to add new dimensions to an array. By inserting `np.newaxis` at any position within an array, we can add a new axis with a dimension size of 1 at that position. The Python built-in keyword `None` is synonymous with `np.newaxis`, and it is frequently used in many programs. For example, the statements `np.array(5)[np.newaxis, np.newaxis]` and `np.array(5)[None, None]` will produce the same result.

It should be noted that the array with the augmented `np.newaxis` is simply a view of the original array. This operation is equivalent to using the `reshape` function with an additional dimension of size 1 in the output shape, such as `np.array(5).reshape((1, 1))`. Utilizing `np.newaxis` for dimension expansion is more straightforward than `reshape`, as it eliminates the need to account for the original shape of the array.

By integrating `np.newaxis`, broadcasting, and ufuncs, we can achieve tasks with concise code which would otherwise require the use of nested loops. For instance, consider a scenario where we have a group of Cartesian coordinates for a set of points and need to calculate the pairwise distances. The traditional approach using nested loops would require the following code:

```
points = np.random.rand(10, 3)
dist = np.zeros((10, 10))
for i, ri in enumerate(points):
    for j, rj in enumerate(points):
        rij = ri - rj
        dist[i,j] = sum(rij**2)**.5
```

Using broadcasting, this code is simplified to just one line:

```
dist = np.sum((points[:,None] - points)**2, axis=-1)**.5
```

Broadcasting, in conjunction with ufuncs, is fundamental tool that greatly enhance both the efficiency and expressiveness of NumPy code. They are extensively utilized in various scenarios within NumPy, especially in array indexing.

2.1.5 Fancy indexing

Fancy indexing in NumPy allows us to access multiple elements through a single indexing operation. For a one-dimensional array, it is not difficult to understand how the fancy indexing works, such as the example below:

```
In [1]: a = np.arange(0, 20, 3)
       idx = [4, 1, 3]
       print(a[idx])
Out[1]:
array([12,  3,  9])
```

Some programmers might find it challenging to interpret the fancy indexing code when working with high-dimensional arrays. Here is an approach to understanding fancy indexing, which can be broken down into three steps.

1. Use the indices to construct an address array. The broadcasting rules should be applied during the address construction.
2. Determine the shape and data type of the output array. Specifically, the shape of the address array (from the previous step) determines the shape of the output array. The output array retains the same data type as the input array.
3. Flatten the address array and use it to select elements from the flattened input array. The selected elements are then filled into the output array.

This procedure of elements selection is demonstrated in the following function

```
def fancy_index_decomposed(a, idxs):
    # Step 1, construct the addresses
    addr = 0
    for i, idx in enumerate(idxs):
        stride = np.prod(a.shape[i+1:], dtype=int)
        if isinstance(idx, slice):
            addr = addr + stride * np.arange(idx.start, idx.stop, idx.step)
        else:
            addr = addr + stride * np.array(idx)

    # Step 2, get the data, as 1D array
    data = a.ravel()[addr.ravel()]
    # Step 3, format the data
    output = data.reshape(addr.shape)
    return output
```

This decomposed approach can help us interpret complex fancy indexing code. For instance, consider the following nested loops that select elements from a matrix:

```
output = np.empty((len(idx1), len(idx2)), dtype=a.dtype)
for i, n in enumerate(idx1):
    for j, m in enumerate(idx2):
        output[i,j] = a[n,m]
```

The shape of the output array is `(len(idx1), len(idx2))`. To resemble this shape, the address array should be constructed with broadcasting between `idx1[:,None]` and `idx2`. The rows of the output array are derived from `idx1`, and the columns from `idx2`. Therefore, `idx1` should be positioned in the row dimension of array `a`, and `idx2` in the column dimension. An efficient fancy indexing expression can be derived to replace the nested loops:

```
output = a[idx1[:,None], idx2]
```

Next, let's swap the shape of the indices `idx1` and `idx2` and examine the effects of the following fancy indexing expression:

```
output = a[idx1, idx2[:,None]]
```

Due to the broadcasting between `idx1` and `idx2[:,None]`, we can expect that elements selected by `idx2` will contribute to the rows of the output array, while `idx1` will contribute to the columns. The elements in `idx1` select the rows of array `a` and act as the columns of the address array in the `fancy_index_decomposed` function. When the address array is flattened, it corresponds to the innermost loop. Therefore, the effects of the fancy indexing code are equivalent to the following nested loops:

```
for j, m in enumerate(idx2):
    for i, n in enumerate(idx1):
        output[j,i] = a[n,m]
```

Here is another complex example that utilizes two-dimensional indices.

```
a = np.zeros((n, m))
b = np.random.rand(n, m)
lookup = np.random.rand(n, m).argsort(axis=1)
a[np.arange(n)[:,None], lookup] = b
```

Although the array being indexed, `a`, serves as the output of the assignment, we can still utilize the rules of the fancy indexing operation to select elements within array `a`. These selected elements are then replaced by the corresponding elements from array `b`. The rows of array `a` are indexed using a continuous sequence of integers `np.arange(n)[:,None]`, which means that the row locations in the output are directly mapped from the original array. Column selection in the output array is governed by the two-dimensional array, `lookup`. `lookup` is an $n \times m$ integer matrix, where the n rows can provide different column locations. When applying the address array

```
np.arange(n)[:,None] * m + lookup
```

to `a`, the addresses for the i th row have the same effects as indexing elements of `a` using the statement `a[i,lookup[i,:]]`. Therefore, the fancy-index assignment code performs the elements selection in a manner equivalent to the following nested loops.

```
for i in range(n):
    a[i,lookup[i,:]] = b[i,:]
```

This code assigns each row of array `b` to the corresponding row of array `a` at the positions specified by the `lookup` array.

If we want to obtain an array `at` such that `at = a.T`, in other words, distributing `b` to `a.T` as

```
b -> (at.T)[np.arange(n)[:,None], lookup]
```

which of the following fancy-indexing assignments can achieve this requirement? Additionally, what could be the problems in the other statements? We will leave this question to the reader. You can exercise the same procedure described above to analyze the fancy indices and any errors that may arise during the process.

```
# at = np.zeros((m, n))
at[np.arange(n), lookup] = b
at[np.arange(n), lookup.T] = b.T
at[lookup, np.arange(n)] = b
at[lookup.T, np.arange(n)] = b.T
at[lookup, np.arange(n)[:,None]] = b
at[lookup.T, np.arange(n)[:,None]] = b.T
```

In various scenarios, you might encounter the `np.ix_` method, which is used to select a sub-block of a high-dimensional array, such as

```
a[np.ix_(idx1, idx2, idx3)]
```

The `np.ix_` method can add the appropriate `newaxis` to the indices in the arguments, transforming them into fancy indexing syntax. The previous `np.ix_` indexing code is equivalent to the following fancy indexing:

```
a[idx1[:,None,None], idx2[:,None], idx3]
```

Fancy indexing and slices can be used together, but combining multiple indices and slices can lead to more complicated indexing behavior. Consider the combinations of the indices and slices in the following example:

```
In [2]: a = np.ones((5,5,5,5))
idx = np.arange(2)
print(a[idx, idx,:,:].shape)
print(a[:,idx, idx,:].shape)
```

```
print(a[:, :, idx, idx].shape)
print(a[:, idx, :, idx].shape)
print(a[idx, :, :, idx].shape)
print(a[idx, :, idx, :].shape)

Out[2]:
(2, 5, 5)
(5, 2, 5)
(5, 5, 2)
(2, 5, 5)
(2, 5, 5)
(2, 5, 5)
```

There are two distinct cases regarding the locations of multiple indices, as demonstrated in this example. They are handled using different approaches:

- When indices are placed next to each other, they act like the slicing operations. The dimensions corresponding to the indices are aggregated and inserted into the output array at the same spot they occupied in the original array. The positions of the other dimensions remain unchanged in the output array.
- If indices are separated by slices, they are first processed and placed in the leading dimensions of the output array. Then, the remaining dimensions are arranged in the subsequent positions of the output array.

The code involving multiple indices and slices can often be misleading and difficult to maintain. Particularly, in the second scenario, the dimensions of the output array may be quite different from those in the original array. To simplify code that involves such complex indexing, one strategy is to insert transpose operations to transform the original array into a more organized structure. This will group the indices and slices together in the transposed array. By doing so, the rules for the first scenario can be applied to select elements. For example, for clarity, the statement `a[idx, :, :, idx]` can be rewritten as

```
a.transpose(0,3,1,2)[idx, idx]
```

2.1.6 Mask array

Fancy indexing allows for the selection of elements based on a specific pattern. However, when the objective is to select elements that meet certain conditions, the distribution of the data may not follow any particular patterns. This is the situation where mask arrays become useful.

For instance, a common application of mask array is to identify and filter out `nan` values from an array.

```
a = np.sqrt(np.random.rand(5,5) - .5)
a[np.isnan(a)] = 0.
```

In this example, `np.isnan(a)` produces a mask array. This array can be used to select all elements that are `nan` and replace them with zero.

A mask array is essentially a binary array consisting of `True` and `False` values. It can select elements across multiple dimensions. When using a mask array to “index” another array, the result is a flattened array that contains elements corresponding to the `True` values of the mask array. Indexing elements using a mask array is governed by two rules:

- The shape of the mask array must match the shape of the array that is being indexed.
- For the masked dimensions, the output is always a flattened (one-dimensional) object.

The second rule implies that the mask-array selection cannot preserve the original dimensions of the data, even if the `True` values within the mask array are arranged in a structured pattern. When values are assigned to the elements specified by the mask, the data source must be reshaped to match the shape of the flattened target array. For example,

```
In [3]: a = np.zeros((4, 4, 4))
        mask = np.ones((4, 4), dtype=bool)
        print(a[mask].shape)

Out[3]:
(16, 4)

In [4]: a[mask] = np.arange(16.)[:, np.newaxis]
```

Mask arrays and fancy indexing operate differently. In terms of efficiency, each exhibits its advantages in different scenarios. Let us consider the four scenarios below.

Indexing a dense array

If we want to manipulate the upper triangular part of a square matrix, we could use fancy indexing as shown below:

```
a = np.ones((n, n))
idx = np.triu_indices_from(a)
a[idx] += 1.5
```

The `np.triu_indices_from` function generates the indices of the upper triangular part of the matrix. If we employ a mask array, the implementation changes to:

```
a = np.ones((n, n))
idx = np.arange(n)
a[idx[:, None] <= idx] += 1.5
```

The statement `idx[:,None] <= idx` creates a square matrix via the broadcasting rule. This statement compares the values of the two indices. Only the upper triangular part satisfies the condition that the first index is less (or equal) than the second index, resulting in a mask of `True` values in those positions.

For sufficiently large arrays, the mask array approach demonstrated in this example can outperform the fancy indexing approach, potentially being three times faster. This is because the `True` values are contiguously distributed in the upper triangular part of the mask array. Such an ordered distribution can enable the CPU to achieve a high rate of successful branch prediction. The only overhead is the creation of the $n \times n$ binary array as an intermediate variable. On the other hand, the fancy indexing approach, which employs the `triu_indices` function, results in two integer vectors, each with size $\frac{1}{2}n(n + 1)$. The two index vectors consume more temporary memory, leading to a higher overhead in data transfer between the CPU and memory. Moreover, the indexing code employs indirect addressing mode [3]. This addressing mode is inherently less efficient, as it prevents the CPU from effectively predicting the data access pattern.

Indexing a sparse array

If the data to be selected is relatively sparse, the mask array method may not be as efficient as fancy indexing. For instance, when operations are performed on the diagonal elements of an array,

```
a = np.ones((n, n))
mask = np.eye(n, dtype=bool)
a[mask] *= .5
```

the following code that employs fancy indexing is more efficient.

```
a = np.ones((n, n))
idx = np.diag_indices_from(a)
a[idx] *= .5
```

In this example, the slower performance of the mask array can be attributed to the necessity of traversing the entire binary array, which results in a complexity of $O(n^2)$. In contrast, the fancy indexing method uses an n -size integer vector for indexing, which corresponds to a complexity of $O(n)$.

Randomly distributed mask array

The mask-array approach is inefficient when `True` values are distributed randomly within the mask array. Suppose we have the option to choose between a randomly distributed mask array and its corresponding indices for element selection. In this case, it is more efficient to select data using fancy indices

```
a[indices] *= .5
```

than using a mask array

```
a[mask] *= .5
```

Even if the number of operations required is close to that in the example of selecting an upper triangular array, the execution time for the mask array approach can increase by ten times, while the time required by the fancy indexing method is mostly unchanged. The substantial performance drop with the mask-array approach can be attributed to the high overhead of if-else branching. The CPU cannot make effective branch predictions for the random mask array. The mis-predicted branches result in significant execution penalties.

Indexing a high-dimensional array

Calculating the addresses for elements in high-dimensional arrays, such as the 4-dimensional electron repulsion integral tensor in quantum chemistry applications, is computationally expensive. The process of address calculation requires a series of integer multiplications and additions, which can be a demanding task. This computational cost can greatly affect the performance of the fancy indexing method. Despite the potential drawbacks of branch mis-prediction, the mask array approach can sometimes provide superior performance.

2.1.7 Data structure of NumPy ndarray

NumPy library is designed to be transparent to the underlying data layout in memory. However, understanding the data structure of NumPy arrays can enhance the understanding of certain complex NumPy code. The knowledge of NumPy array structure can influence the code optimization practice, the program stability, and even the algorithm design. Furthermore, when the goal is to optimize NumPy code through C/C++ extensions, it becomes necessary to be familiar with the structure of NumPy arrays.

The memory space in a computer is a linear space. For one-dimensional data, such as a vector, each element can be easily mapped to a specific location in the memory space. However, when dealing with multi-dimensional arrays, how does NumPy organize and represent the array objects using linear memory storage?

Array in NumPy is a block of memory plus some metadata to describe the data layout. The following attributes are introduced by NumPy for the representation of an array:

- The `dtype` attribute indicates the data type of the array elements. It determines the memory size occupied by each element (accessible via the attribute `itemsize`).
- The `ndim` attribute is an integer that represents the number of dimensions.
- The `shape` attribute is a tuple that indicates the size of each dimension. The length of `shape` is identical to `ndim`.
- The `data` attribute represents the buffer that stores the raw data. The memory address of the buffer can be obtained via the attribute `ctypes.data` or the attribute `_array_interface_['data'][0]` [4].

- The `strides` attribute is a tuple indicating the number of bytes that should be skipped in memory when moving from one element to the next along each dimension of the array.

These attributes can be directly accessed, for example:

```
In [1]: import numpy as np
        a = np.random.rand(3,5,2)
        print(a.dtype, a.itemsize)
        print(a.ndim)
        print(a.shape)
        print(a.strides)
        print(a.ctypes.data)

Out[1]:
dtype('float64'), 8
3
(3, 5, 2)
(80, 16, 8)
20823872
```

The attribute `strides` are used by NumPy core library to calculate the memory addresses of individual elements in an array. In this example, the address of the array element `a[i,j,k]` can be calculated as

`20823872 + i * 80 + j * 16 + 8`

where `20823872` is the address of the data buffer for array `a`. How are the `(80, 16, 8)` strides obtained? The strides equal to

`(8 * 5 * 2, 8 * 2, 8)`

where the number `8` is from the `itemsize` of the array, and the numbers `5` and `2` are obtained from the `shape` of the array. The calculation can be characterized using a Python expression

```
[a.itemsize * np.prod(a.shape[i+1:], dtype=int) for i in range(a.ndim)]
```

2.1.7.1 The `strides` attribute for flexible array structure

In the aforementioned example, the values of the `strides` attribute are derived from the other attributes of the array. At first glance, the `strides` attribute may seem redundant. It is actually an elegant design of NumPy that offers a flexible and efficient way of representing multi-dimensional arrays. The `strides` attribute allows us to reuse memory and share data between different arrays, which effectively reduces the memory consumption and the cost of copying data between arrays.

To illustrate this advantage, let's consider the Hankel matrix

$$H = \begin{pmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \\ a_1 & a_2 & a_3 & & a_n \\ a_2 & a_3 & \ddots & \ddots & \vdots \\ \vdots & & \ddots & & a_{2n-3} \\ a_{n-1} & a_n & \cdots & a_{2n-3} & a_{2n-2} \end{pmatrix}. \quad (2.3)$$

The Hankel matrix is a square matrix whose elements are derived from a vector. Each element is defined by the formula $H_{ij} = a_{i+j}$. To construct the Hankel matrix, the code can be implemented with nested loops

```
def hankel_matrix(a):
    n = (a.size + 1) // 2
    out = np.empty((n, n), dtype=a.dtype)
    for i in range(n):
        for j in range(n):
            out[i, j] = a[i+j]
    return out
```

This code can be slightly optimized using the NumPy ufunc-style assignment.

```
def hankel_matrix(a):
    n = (a.size + 1) // 2
    out = np.empty((n, n), dtype=a.dtype)
    for i in range(n):
        out[i] = a[i:n]
    return out
```

By using the `strides` attribute to change the layout of the vector, we can construct the Hankel matrix without the need for allocating new memory or copying data. This approach is much faster than any of the above implementations.

```
def hankel_matrix(a):
    n = (a.size + 1) // 2
    return np.ndarray(shape=(n, n), dtype=a.dtype,
                      strides=(a.itemsize, a.itemsize), buffer=a)
```

Here, we invoke the class instantiation method `np.ndarray` along with a data buffer to initialize an array instance. `ndarray` can accept keywords like `shape`, `dtype`, `buffer`, and `strides` to define a new array. These keywords correspond to the metadata of the array that we discussed previously. Due to the argument `buffer`, a *view* of the original array object is generated.

Let's consider how the *view* represents the Hankel matrix in terms of the original vector. The strides of the output array are `(a.itemsize, a.itemsize)`. Therefore, the matrix element `H[i, j]` corresponds to the data stored at the address

```
a.ctypes.data + a.itemsize * i + a.itemsize * j
```

which simplifies to

```
a.ctypes.data + a.itemsize * (i + j)
```

This is exactly the address of $a[i+j]$. The new array reuses the memory buffer of the vector a , thus occupies only $2n$ words of memory for the n^2 elements.

2.1.7.2 C-contiguous and F-contiguous storage

The `strides` attribute is closely related to the storage layout of an array. High-dimensional arrays are often organized in two types of storage layouts: *C-contiguous* and *F-contiguous*. Taking the two-dimensional array as an example, the C-order and Fortran-order storage layout are illustrated in Fig. 2.1.

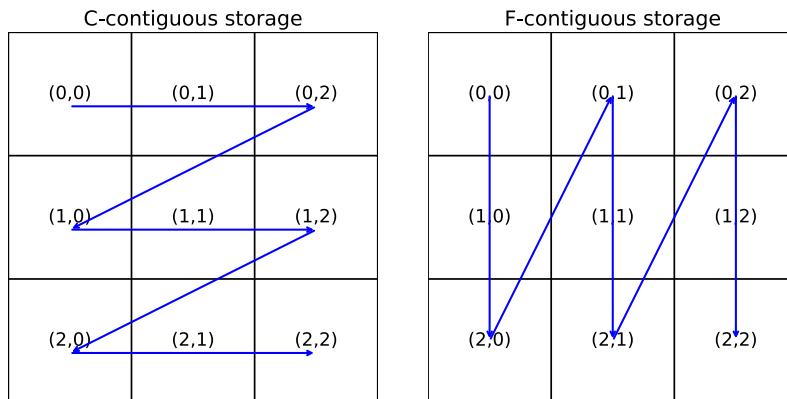


FIGURE 2.1

C-contiguous and F-contiguous storage format.

C-contiguous storage adopts the approach used in C programming language for storing multi-dimensional arrays. It places the elements in the last dimension next to each other in memory. In such a layout, the last item in `strides` is equal to the `itemsize`. Elements within the `strides` attribute are arranged in a descending order.

Multi-dimensional arrays in Fortran programming language employ a different layout, which arranges the elements in the first dimension contiguous in memory. In the F-contiguous layout, the first item in the `strides` tuple is equal to the `itemsize`. Elements within the `strides` attribute are arranged in an ascending order.

When initializing a new array, NumPy employs the C-contiguous layout as the default storage mode. However, due to the existence of array views, arrays can be neither C-contiguous nor F-contiguous. For example, the Hankel matrix we demonstrated previously is subject to this case. To determine the storage layout of an array, we can examine its `flags` attribute:

```
a.flags['C_CONTIGUOUS']
a.flags['F_CONTIGUOUS']
```

These two flags reveal if the array is stored in a C-style or Fortran-style contiguous pattern, respectively.

Understanding the storage layout of a dataset is critical for optimizing the performance of NumPy code. When iterating over the data in an array, it is preferable to iterate over *the smallest stride first*, as this facilitates iterating over *consecutive memory* locations. Accessing consecutive memory locations has several performance advantages:

- When data is read into the CPU, it is loaded in units known as cache lines [5]. The entire cache line has to be loaded, even if only one byte within the cache line is accessed. By iterating over small strides, the next data element is likely already fetched in the same cache line. The CPU does not need to load the data from memory again. In contrast, iterating over large strides can lead to a waste of memory bandwidth.
- CPU can prefetch data for sequential memory access. It can load the next cache line before the current one is fully processed.
- Data accessed in small strides is likely to be resident within the same memory page. In the case of large strides, there is a higher chance of accessing data from different pages, which can lead to page faults. Page faults significantly slow down the data access performance. We will explore this issue in Chapter 9.

The storage layout can greatly influence the performance of ufunc operations. When a ufunc is applied to two array objects with identical layouts, NumPy is able to detect the data layout of the operands and select the optimal iteration schemes to traverse the arrays.

For instance, let us consider two arrays with C-contiguous storage:

```
a = np.ones((100,100,100,100))
b = np.ones((100,100,100,100))
```

The time spent on the two operations below is almost the same.

```
c = a + b
c = a.transpose(2,0,3,1) + b.transpose(2,0,3,1)
```

If the smallest strides of the two arrays are placed on the corresponding dimension, no matter whether it is the first, last, or any intermediate dimension, NumPy can identify the smallest strides and place them in the inner-most iteration, achieving high memory access efficiency. Please note that the output from a ufunc is not guaranteed to be in a C-contiguous layout. To optimize memory access efficiency, NumPy may choose a layout for the output that is close to the layout of the operands. For example, the layout of the output array `c` in the following snippet is the same to the layout of the first operand `a.transpose(2,0,3,1)`

```
c = a.transpose(2,0,3,1) + b.transpose(2,1,3,0)
```

NumPy cannot optimize the memory access pattern if the operands for ufuncs have different data layout, more specifically, if their smallest strides are not located in the corresponding dimensions. In such circumstances, NumPy will use the default C-contiguous storage layout for the output. When processing ufunc operations, NumPy traverses across all dimensions following the C-contiguous memory access pattern. This is optimal only for the layout of the output data. If we encounter arrays whose smallest strides are not located in the corresponding dimensions, we should at least try to optimize the access pattern for one operand. This can be done by aligning the smallest stride of that operand with the smallest stride of the output, i.e., placing it in the last dimension. This approach ensures that only one operand is accessed inefficiently. For instance, in the two types of transpose summation below, the first is more efficient than the second.

```
c = a.transpose(0,3,2,1) + b
c = a.transpose(3,1,0,2) + b.transpose(1,3,0,2)
```

2.1.8 Array views

An array view is a mechanism for interpreting the data of an array in a different manner. It reuses the data buffer of the original array but owns different array attributes such as `strides`, `shape`, and `dtype`. This mechanism enables data to be shared between NumPy arrays, allowing NumPy code to use memory more efficiently. Any modifications made to the array view will also appear in the original array.

When working with high-dimensional arrays, we often need to modify the elements within a small block of the array. In such situations, we can create a low-dimensional array that acts as a view of the specific sub-block. Instead of manipulating the original high-dimensional array, we can modify the low-dimensional array view. This is an effective way to simplify NumPy code. For instance, consider the following code, which modifies an 8-dimensional array with the shape $(2,n,2,n,2,n,2,n)$:

```
for s in range(2):
    for ip, i in enumerate(idx1):
        for jp, j in enumerate(idx2):
            a[s,i,s,j,s,j,s,i] += b[ip,jp]
```

In the context of quantum chemistry, the 8-dimensional array can be used to represent the electron repulsion integrals. This code snippet can be viewed as an adjustment (the `b` matrix) to the Hartree-Fock exchange elements. By introducing a 4-dimensional array view of the 8-dimensional array, we can rewrite the code snippet into a more readable form.

```

for s in range(2):
    # Extract a spin component of array a
    a_view = a[s,:,:s,:,s,:,:]
    a_view[idx1[:,None],idx2, idx2, idx1[:,None]] += b

```

Array views can also be employed to simplify the fancy-indexing code, enhancing its readability. For example, the task in the previous example can be accomplished with the following one-liner using fancy indexing.

```

diag = np.arange(2)[:,None,None]
a[diag, idx1[:,None], diag, idx2, diag, idx2, diag, idx1[:,None]] += b

```

By using an array view to restructure the array and decrease its dimensionality, the fancy-indexing part can be more straightforward to understand.

```

a_view = np.einsum('sisjsjsi->sij', a)
a_view[:,idx1[:,None],idx2] += b

```

Here, we have utilized the `einsum` function to create an array view, a feature described in the `np.einsum` documentation:

When there is only one operand, no axes are summed, and no output parameter is provided, a view into the operand is returned instead of a new array. Thus, taking the diagonal as “`np.einsum('ii->i', a)`” produces a view (changed in version 1.10.0).

Utilizing a low-dimensional array for the sub-blocks of a high-dimensional array offers an additional benefit beyond readability. Accessing elements in a lower-dimensional array has lower overhead than in a high-dimensional array. This performance advantage, due to the efficiency of indexing, is evident in the quantum chemistry integral computation program, which will be addressed in Chapter 12.

Array views do not always contribute positively to the computational performance or memory efficiency.

Array views can influence the life-cycle of the base array. The base array and all its views share the same life-cycle. Even a single element of the base array is referenced by any views, the entire base array cannot be released, potentially leading to increased memory consumption. The solution to this problem is simple. By making an explicit copy of the array view, we can eliminate the reference to the base array. This allows the base array to be immediately freed upon exiting the scope in which it was created.

Array views can result in non-contiguous memory access. For example, the `np.real` function can return the real part of a complex array, which is just a view of the complex array with different `dtype` and `strides` attributes. The elements in the real-valued array are not stored contiguously. As a result, accessing this array does not offer continuous memory read and write operations, which can reduce the CPU cache utilization and memory bandwidth.

Array views can introduce potential bugs in certain NumPy code, more specifically, in the context of inplace operations and the use of the `out` keyword for ufuncs. These operations may accidentally alter the contents of another array. Therefore, it is important to ensure whether these operations are applied to array views.

How can we tell if an array is a view or not? We can use the `base` attribute of an array to determine this property. If `base` is `None`, the array owns the data and it is not a view of another array. Otherwise, the current array is a view of another array object, and the `base` attribute points to the original array or the data buffer. For example:

```
In [1]: a = np.random.rand(3,5,2)
        b = a[:,2]
        c = a[[0,1]]
        print(b.base is None) # b is an array view
        print(c.base is None) # c is not an array view
Out[1]:
False
True
```

To avoid potential bugs caused by array views, it is essential to understand which NumPy operations can produce array views. Typically, array views are created in the following scenarios.

Slicing an array

When using slicing to select a sub-array, the output is a view of the original array with adjusted shapes and strides. The data buffer of the sub-array (accessible via the attribute `.ctypes.data`) points to the same data buffer of the original array with an address offset. For example:

```
In [2]: a = np.random.rand(3,5,2)
        print(a.strides, a.shape, a.ctypes.data)
Out[2]:
(80, 16, 8) (3, 5, 2) 61128864

In [3]: b = a[:,1:5:2]
        print(b.strides, b.shape, b.ctypes.data)
Out[3]:
(80, 32, 8) (3, 2, 2) 61128880

In [4]: c = a[:,::-1]
        print(c.strides, c.shape, c.ctypes.data)
Out[4]:
(80, -16, 8) (3, 5, 2) 61128928
```

Let's examine the addresses of the data buffer for the original array and the sub-arrays:

- For array `b`, the address of its first element is $61128880 = 61128864 + 16$, which matches the address of `a[0,1,0]`.
- For array `c`, the address of its first element is $61128928 = 61128864 + 64$, which matches the address of `a[0,4,0]`.

NumPy functions that return a view of the input array

Functions `np.transpose` and `np.diagonal` are two examples of this category. The output of the `np.transpose` function only swaps the strides of the input array. The `np.diagonal` function generates a view with strides (`itemsize * (n+1)`) to represent the diagonal elements of the $n \times n$ square matrix. Additionally, whenever applicable, functions like `np.reshape`, `np.ravel`, and `np.einsum` also generate array views.

Calling `np.ndarray()` with the keyword argument `buffer`

We have been using this technique in the example of Hankel matrix. This technique allows us to use a pre-allocated data buffer for an array. Please note that the data buffer is not limited to NumPy objects. Several options are available for the data buffer, including `bytearray`, shared memory [6], and memory maps.

2.1.9 The `reshape` function

Most NumPy functions operate in a deterministic manner, either returning a view of an array or a new copy of the data. `np.reshape` and `np.ravel` are exceptions. They may return either a view or a new array depending on the layout of the input array. The `ravel` function can be regarded as a special case of the `reshape` function, which essentially performs `np.reshape(a, -1)` to flatten the input array. In the following, we will focus on the functionality of the `reshape` function.

The `reshape` function returns a view of the input array whenever possible, as demonstrated in the following example.

```
In [1]: a = np.zeros((3,4,3))
        print(a.reshape((3,1,2,2,3)).base is a)
        print(a.reshape((6,6)).base is a)
        print(a.transpose(2,1,0).reshape((6,6)).base is a)
        print(a.transpose(2,1,0).reshape((3,1,2,2,3)).base is a)

Out[1]:
True
True
False
True
```

If the `reshape` function is used to break existing dimensions into smaller ones, it always returns views of the array. However, the task of merging dimensions is more complex. If the dimensions targeted for merging are in C-contiguous order, the `reshape` function can efficiently return a view of the array. Otherwise, if the dimensions are not in C-contiguous order, the `reshape` function will generate a copy

of the array. The second scenario often occurs when `reshape` is used together with the `transpose` function. For example, the operation `a.transpose(2,1,0)` changes the array layout to the F-contiguous storage. In this case, it is impossible to merge dimensions without reordering the data.

2.2 Data types in NumPy

NumPy and NumPy-based libraries, such as Pandas, CuPy, JAX, and PyTorch, handle data types according to various implicit rules. This implicit management may sometimes affect the performance or even the correctness of the program. In this section, we will explore the problems related to data types in Python numerical computation.

2.2.1 Type casting

NumPy offers various data types, covering the integers, floats, complex numbers, strings, datetimes, bytes, and anonymous Python object types. The complete list of built-in data types in NumPy is available in the dictionary `np.sctypeDict`. When NumPy array objects are created from Python lists (or tuples), the Python built-in types are mapped to their corresponding NumPy types:

- Booleans (`bool`): Mapped to `np.bool_`, which is compatible with the one-byte integer type `np.int8`. Please note that in NumPy 2.*, `np.bool_` is replaced by `np.bool`.
- Integers (`int`): Mapped to `np.int_`, which is an alias for the `np.int64` type. However, Python long integers (integers $> 2^{63}$) are not supported by any NumPy integer types and are instead converted to the `np.object_` type.
- Floating-point numbers (`float`): Mapped to `np.float64`.
- Complex numbers (`complex`): Mapped to `np.complex128`.
- Strings (`str`): Mapped to `np.str_` type.
- Objects (`object`): If the array contains Python objects that do not fall into the aforementioned categories, or mixed objects that involve strings, the data type is mapped to `np.object_`. When dealing with the generic type (`np.object_`), NumPy does not have a substantial performance advantage over the standard Python list.

When arithmetic operations are performed between two NumPy arrays, or between a NumPy array and a Python object, type casting may occur implicitly. When operating with arrays of different data types, *type upcasting* is performed to ensure the resulting type has a higher precision or more digits. The resulting type can be determined using the `np.result_type` function:

```
In [1]: print(np.result_type(np.int32([2]), np.float32([3])))
        print(np.result_type(np.float32([2]), np.array([3.5])))
        print(np.result_type(np.int32([2]), np.zeros(3)))

Out[1]:
float64
```

```
float64
float64
```

Please note that when one of the operands is a scalar, type casting might not always produce the type with more digits or higher precision. Additionally, the behavior of type casting varies between NumPy 2.* and NumPy 1.* versions. For instance, in NumPy 1.*, type casting between floating-point numbers results in:

```
In [2]: print(np.result_type(np.float32([2]), 3))
        print(np.result_type(np.float32([2]), np.float64(3)))
        print(np.result_type(np.float32(2), 3))
        print(np.result_type(np.float32(2), np.float64(3)))

Out[2]:
float32
float32
float64
float64
```

The operation between a `float32` array and a `float64` scalar is downcast to the `float32` type. We may accidentally lose computational precision during this conversion. In NumPy 2.*, these type castings yield different outputs as follows:

```
In [3]: print(np.result_type(np.float32([2]), 3))
        print(np.result_type(np.float32([2]), np.float64(3)))
        print(np.result_type(np.float32(2), 3))
        print(np.result_type(np.float32(2), np.float64(3)))

Out[3]:
float32
float64
float32
float64
```

The operation between a `float32` scalar and a Python built-in float number is downcast to the `float32` type. Similar inconsistencies and ambiguities in type casting are also observed between `int32`, `int64`, and Python's built-in integers.

There are many scenarios where a scalar is produced, such as indexing an element of an array or calling a reduction function that computes a single result. When a scalar is returned from a NumPy operation and if the scalar needs to be used in further operations with other arrays or Python scalars, it is crucial to handle the type carefully to prevent precision loss or integer overflow.

Please note that these type conversion rules are specific to NumPy. NumPy-based libraries (such as CuPy, PyTorch, JAX) may adopt different convention rules for type casting. When incorporating NumPy and these libraries in the same program, to guarantee the appropriate data types, it is often necessary to explicitly specify the `dtype` parameter in any array creation functions.

2.2.2 Scalar type and zero-dimensional array

Zero-dimensional arrays can be created by calling `np.array` with a scalar variable.

```
In [1]: type(np.array(3))
Out[1]:
numpy.ndarray
```

However, in NumPy-based programs, you might notice that zero-dimensional arrays rarely appear. When encountering zero-dimensional arrays, such as the output of reduction functions (`np.sum`, `np.mean`, `np.prod`), the NumPy `ndarray` class smartly converts the zero-dimensional arrays into scalar variables.

```
In [2]: print(isinstance(np.zeros(3).sum(), np.ndarray))
        print(isinstance(np.array(3) + 5, np.ndarray))
        print(isinstance(np.max(np.zeros(3)), np.ndarray))
Out[2]:
False
False
False
```

When working with subclasses of the NumPy `ndarray`, zero-dimensional arrays are produced by NumPy reduction functions and ufuncs.

```
In [3]: class Array(np.ndarray):
    pass
    print(isinstance(Array(3).sum(), np.ndarray))
    print(isinstance(Array(3) + 5, np.ndarray))
    print(isinstance(np.max(Array(3)), np.ndarray))
Out[3]:
True
True
True
```

This is also found in some NumPy-based libraries, which may universally return the array-like objects rather than scalar variables for reduction functions and ufuncs.

```
In [4]: print(type(jax.numpy.arange(3).sum()))
        print(type(jax.numpy.array(3) + 5))
        print(type(jax.numpy.count_nonzero(np.zeros(3))))
Out[4]:
<class 'jaxlib.xla_extension.ArrayImpl'>
<class 'jaxlib.xla_extension.ArrayImpl'>
<class 'jaxlib.xla_extension.ArrayImpl'>
```

In many scenarios, scalars and zero-dimensional arrays behave similarly and appear to be interchangeable. For example, they follow the same broadcasting rules in

ufuncs. Also, both can be used as regular integers when provided as arguments to NumPy functions.

```
In [5]: i3 = np.int64(3)
        a = np.zeros((i3, i3))
        a += i3
        a.max(axis=np.int64(1))

Out[5]:
array([3., 3., 3.])

In [6]: a3 = np.array(3)
        a = np.zeros((a3, a3))
        a += a3
        a.max(axis=np.array(1))

Out[6]:
array([3., 3., 3.])
```

You might be wondering: Are zero-dimensional arrays identical to scalar variables? Mathematically, they are equivalent. However, they function differently in programming. NumPy scalar variables behave similarly to regular Python built-in scalars, whereas zero-dimensional arrays are more close to NumPy arrays. Their differences can be observed in the following scenarios.

- *Views:* In the case of zero-dimensional arrays, variables can be shared with other objects through array views. When performing an inplace operation on zero-dimensional arrays, the content of the associated object also changes. In contrast, for scalar variables, assignments will create new objects. Memory is not shared. For example:

```
In [7]: a = np.array(3)
        x = a
        a *= 2
        print(a, x)

Out[7]:
(6, 6)

In [8]: a = np.int64(3)
        x = a
        a *= 2
        print(a, x)

Out[8]:
(6, 3)
```

- *Hashability:* Zero-dimensional arrays are not hashable, like any NumPy array objects. This means that they cannot be used as keys in dictionaries, nor can the `set`

function be employed to eliminate duplicates. In contrast, scalar variables can be used in dictionaries and sets.

In certain scenarios, it is necessary to ensure that we are working with scalar variables rather than zero-dimensional arrays. There are two methods to ensure the scalar type:

- Retrieving the scalar value through the indexing syntax `a[()]`.
- Explicitly converting the array to a specific data type.

```
In [9]: print(type(np.array(2)[()]))      # Accessing the scalar value
        print(type(int(np.array(2))))     # Explicit type conversion to int
        print(type(float(np.array(2))))   # Explicit type conversion to float
Out[9]:
<class 'numpy.int64'>
<class 'int'>
<class 'float'>
```

2.2.3 Infinity (`inf`) and not-a-number (`nan`)

Occasionally, in numerical computations, we may encounter `nan` (not a number) and `inf` (infinity). What situations might cause these values to appear in our programs? Apart from intentionally assigning `nan` or `inf`, these numbers can arise in the following circumstances:

- Division by zero can result in `inf` or `-inf` depending on the sign of the numerator. For example

```
In [1]: np.ones(3) / np.zeros(3)
RuntimeWarning: divide by zero encountered in divide
Out[1]:
array([inf, inf, inf])
```

- Invalid operations, such as taking the square root of a negative number or the logarithm of a negative number, can yield `nan`.

```
In [2]: sqrt(-np.ones(3))
RuntimeWarning: invalid value encountered in sqrt
Out[2]:
array([nan, nan, nan])
```

- Overflow of floating-point operations, such as `np.exp(1e3)` and `np.log(0)`, can lead to `inf`.
- Indeterminate forms [7] results in `nan`, such as

```
In [3]: np.zeros(3) / np.zeros(3)
```

```
RuntimeWarning: invalid value encountered in divide
Out[3]:
array([nan, nan, nan])

In [4]: np.inf * 0
Out[4]:
nan
```

Due to the indeterminate forms, The statement `a *= 0` should be avoided as it can lead to unexpected results. One should use either `a[:] = 0` or `a.fill(0)` to zero out an array.

- Missing data in Pandas: When using Pandas to read data, missing entries may be filled with `nan`. When Pandas objects are converted to NumPy arrays, `nan` values are introduced into the resulting array.

When working with the Python built-in floating-point numbers, these problematic arithmetic operations may trigger exceptions such as `ZeroDivisionError`, `OverflowError`, or `ValueError`. These exceptions can help us identify errors and fix problems. However, by default, NumPy only issues a `RuntimeWarning` message without interrupting the computation. The computation continues with `inf` or `nan` retained in the array. Such warning messages can easily be overlooked.

To avoid numerical problems caused by the `inf` and `nan` values, NumPy can be configured to raise errors for floating-point issues, using the error handler:

```
np.seterr(all='raise')
```

This configuration ensures that an error will be raised whenever a floating-point problem occurs, rather than just printing a warning message.

It might be inevitable encountering `inf` and `nan` in the program, especially when handling data input from external sources. These values require careful consideration and special treatment to ensure the correctness of the program. Below are some practical advices:

- The comparison to `nan` always yields `false`, even when `nan` is compared with itself.

```
In [5]: np.nan == np.nan
Out[5]:
False
```

To determine whether a value is `nan`, we should use the `np.isnan` function provided by NumPy.

- To replace `inf` and `nan` with finite values, the `np.nan_to_num` function is an appropriate tool. This function can substitute `nan` with zero and `inf` with large finite numbers or any numbers specified by the user.
- When an array contains `nan` values, reduction functions return `nan` as the result. To handle this, we can employ the corresponding `nan` versions of these functions

(with the prefix `nan` in the function name), such as `np.nansum` and `np.nanmean`, which discard `nan` values when performing the calculation.

- Type conversion problems. It is not feasible to cast floating-point numbers to integers while preserving the `inf` and `nan` values. When using the `ndarray.astype()` method to perform such type conversions, both `inf` and `nan` results in an integer -9223372036854775808 ($= -2^{63}$) and the conversion issues a warning to indicate the presence of these invalid values.

```
In [6]: np.array([np.inf, np.nan]).astype(int)
RuntimeWarning: invalid value encountered in cast
    np.array([np.inf, np.nan]).astype(int)
Out[6]:
array([-9223372036854775808, -9223372036854775808])
```

To avoid such issues, `inf` and `nan` values should be filtered out before type conversion, such as using the `np.nan_to_num` function. Additionally, the `np.seterr` (or `np.errstate`) function can be configured to prevent these inappropriate type conversions.

2.2.4 Data with high precision

NumPy can support arrays with quadruple precision (128-bit float), by specifying the data type as `np.float128`. If 128-bit float does not meet our precision requirements, the `mpmath` library [8] can be used for high precision calculations. `mpmath` can be installed with the command:

```
pip install mpmath[gmpy]
```

The optional dependency `gmpy` can improve the efficiency of `mpmath` in high-precision arithmetic computations.

The `mpmath` library implements an extensive collection of computational routines for basic functions, special functions, and some linear algebra capabilities. When initializing the `mpmath` module, we can set the computational precision and the printing format.

```
import mpmath as mp
mp.mp.dps = 25
mp.mp.pretty = True
```

In the code examples below, we will use the alias `mp` to refer to `mpmath`. To avoid rounding errors, the floating-point constants passed to `mpmath` functions should be quoted as string literals.

```
In [1]: print(mp.mpf('0.0'))
        print(mp.mpf('0.1'))
        print(mp.mpf(0.1))
Out[1]:
```

```
0.0
0.1
0.100000000000000055511151231
```

`mpmath.mpf` objects can be integrated into NumPy arrays [9], granting NumPy the capability to perform arithmetic operations with arbitrary precision. To enable this feature, we can convert a list of `mpf` numbers into a NumPy array and leverage the NumPy code to manage `mpf` objects. For example, when using the operator `*` to perform broadcasting between two arrays of `mpf` objects, NumPy executes the `mpmath` multiplication for each corresponding pair of `mpf` elements.

```
In [2]: def mparange(n):
    return np.array(mp.arange(n))

In [3]: idx = mparange(3)
        idx[:,None] * idx
Out[3]:
array([[mpf('0.0'), mpf('0.0'), mpf('0.0')],
       [mpf('0.0'), mpf('1.0'), mpf('2.0')],
       [mpf('0.0'), mpf('2.0'), mpf('4.0')]], dtype=object)
```

The `dtype` of the NumPy array is shown as `object` in this case. It indicates that the elements are not NumPy built-in data types. Instead, the elements of the array are treated as regular Python objects.

Please note that most NumPy ufuncs are incompatible with `mpf` objects. To integrate `mpmath` functions into the ufuncs, we can use `np.vectorize` to extend `mpmath` functions, enabling them to handle arrays of `mpf` objects. The function `np.vectorize` facilitates the broadcasting of scalar functions, producing ufuncs that behave like native NumPy functions. With this vectorized enhancement, arithmetic operations on `mpf` arrays can now be performed as if they were regular NumPy arrays.

```
In [4]: mpxp = np.vectorize(mp.exp)
        mpsqrt = np.vectorize(mp.sqrt)

In [5]: print(mpxp(mparange(3)))
        print(mpsqrt(mparange(3)))
Out[5]:
array([mpf('1.0'), mpf('2.718281828459045235360287496'),
       mpf('7.389056098930650227230427434')], dtype=object)
array([mpf('0.0'), mpf('1.0'), mpf('1.414213562373095048801688713')], dtype=
=object)
```

By combining `mpf` arrays and `np.vectorize` ufuncs, we can launch more sophisticated NumPy functionalities. For example, NumPy has a module `np.fft` for performing discrete Fourier transform (DFT) operations. Its core functions include

`np.fft.fft` for the DFT formula

$$f_m = \sum_{n=0}^{N-1} x_n e^{-i2\pi k_m n}, \quad (2.4)$$

and `np.fft.ifft` for the inverse DFT

$$x_n = \frac{1}{N} \sum_{m=0}^{N-1} f_m e^{i2\pi k_m n}, \quad (2.5)$$

where the sample frequencies k_m are generated by the `np.fft.fftfreq` function, centered inside a period.

$$k_m \in [0, \frac{1}{n}, \dots, \frac{n/2 - 1}{n}, -\frac{1}{2}, \dots, -1], \quad n \text{ is even}, \quad (2.6)$$

$$k_m \in [0, \frac{1}{n}, \dots, \frac{n-1}{2n}, -\frac{n-1}{2n}, \dots, -1], \quad n \text{ is odd}. \quad (2.7)$$

The arbitrary precision DFT and its inverse can be implemented as:

```
import numpy as np, mpmath as mp
mp.mp.dps = 30

def mpfftfreq(n):
    if n % 2 == 0:
        p1 = mp.arange(0, n//2)
        p2 = mp.arange(-n//2, 0)
    else:
        p1 = mp.arange(0, (n+1)//2)
        p2 = mp.arange(-(n-1)//2, 0)
    return np.append(p1, p2) / n

mpexp = np.vectorize(mp.exp)

def mpfft(a):
    n = a.shape[-1]
    kx = 2j*mp.pi * mpfftfreq(n)
    fac = mpexp(-np.arange(n)[:,None] * kx)
    return a.dot(fac)

def mpifft(a):
    n = a.shape[-1]
    kx = 2j*mp.pi * mpfftfreq(n)
    fac = mpexp(np.arange(n)[:,None] * kx)
    return a.dot(fac) / n
```

Table 2.1 Comparison of linear algebra functions between `mpmath` and NumPy/SciPy.

<code>scipy.linalg</code>	<code>np.linalg</code>	<code>mpmath</code>	comments
<code>solve</code>	<code>solve</code>	<code>lu_solve</code> ^a <code>qr_solve</code> <code>cholesky_solve</code>	Solves the linear equation $ax = b$.
<code>lu</code>		<code>lu</code> ^c	LU decomposition of a matrix.
<code>qr</code>	<code>qr</code>	<code>qr</code> ^c	QR decomposition of a matrix.
<code>cholesky</code>	<code>cholesky</code>	<code>cholesky</code> ^c	The Cholesky decomposition of a matrix.
<code>inv</code>	<code>inv</code>	<code>inverse</code>	The inverse of a matrix.
<code>det</code>	<code>det</code>	<code>det</code>	The determinant of a matrix.
	<code>cond</code>	<code>cond</code> ^b	The condition number of a matrix.
<code>eigh</code>	<code>eigh</code>	<code>eigh</code>	Solve eigenvalue problem for a symmetric matrix.
<code>svd</code>	<code>svd</code>	<code>svd</code>	Singular Value Decomposition.
<code>schur</code>		<code>schur</code>	Schur decomposition of a matrix.
<code>hessenberg</code>		<code>hessenberg</code>	Hessenberg form of a matrix.
<code>eig</code>	<code>eig</code>	<code>eig</code> ^c	Solve eigenvalue problem of a square matrix.
<code>norm</code>	<code>norm</code>	<code>norm</code> ^c	Matrix norm.
<code>expm</code>		<code>expm</code>	Matrix exponential of an array.
<code>logm</code>		<code>logm</code>	Matrix logarithm.
<code>cosm</code>		<code>cosm</code> ^c	Matrix cosine.
<code>sinm</code>		<code>sinm</code> ^c	Matrix sine.
<code>sqrtm</code>		<code>sqrtm</code>	Matrix square root.

^a `mpmath` does not have a direct `solve` function. We can use one of the three functions instead.

^b `cond` in `mpmath` behaves similarly to `np.linalg.cond(a, p=1)` in NumPy.

^c These functions are incompatible with `mparray` class (see discussions in the main text). They can only be used with `mpmath.matrix` objects.

If we need to perform computations involving linear algebra on `mpf` arrays, such as eigenvalue decomposition, we cannot use the built-in linear algebra functions of NumPy. This is because NumPy conducts linear algebra using BLAS and LAPACK routines, which only support single and double precision types. Here, we need to use the linear algebra functions provided by `mpmath`. In Table 2.1, we summarize the commonly used linear algebra functions from the NumPy and SciPy libraries, along with their equivalents in the `mpmath` library.

The linear algebra functions in `mpmath` (as of version 1.3) are developed based on its built-in `mpmath.matrix` class. These functions are not fully compatible with the `ndarray` class from NumPy. One of the main differences is the use of attributes `rows` and `cols` in the `mpmath` matrix class. To accommodate this difference, we subclass the NumPy `ndarray` to introduce these two attributes, thanks to Python's duck typing [10]

```
class mparray(np.ndarray):
    @property
```

```
def rows(self):
    return self.shape[0]

@property
def cols(self):
    return self.shape[1]

def __array_wrap__(self, out, context=None):
    if out.ndim == 0:
        out = out[()]
    return out
```

Please note that, in this custom array class, we override the underscore method `__array_wrap__`. This modification converts zero-dimensional arrays to scalars variables for reduction functions. As discussed in Section 2.2, NumPy reduction functions do not automatically perform this conversion when dealing with the subclasses of `ndarray`. To ensure that our `mparray` behaves as close as possible to standard NumPy arrays, we manually perform the conversion in the `__array_wrap__` method. For a more detailed description of the rules associated with NumPy subclassing, we recommend readers to refer to the NumPy documentation [11].

The `mparray` objects can now be used for several `mpmath` linear algebra functions, such as `expm`, `svd`, and `eigh`.

```
In [4]: idx = mparange(3).view(mparray)
      a = idx[:,None] * idx
      array([[0.0, 0.0, 0.0],
             [0.0, 1.0, 2.0],
             [0.0, 2.0, 4.0]], dtype=object)
      print(mp.eigh(a))

Out[4]:
(matrix(
[[0.0],
 [0.0],
 [5.0]]),
mparray([[mpf('1.0'), mpf('0.0'), mpf('0.0')],
[mpf('0.0'), mpf('-0.8944271909999158785636694691'), mpf('
0.4472135954999579392818347346')],
[0, mpf('0.4472135954999579392818347346'), mpf('
0.8944271909999158785636694691')]], dtype=object))
```

However, using `mparray` objects with `lu`, `qr`, `cholesky`, `norm`, and other functions results in errors. These functions contain type checks and operations that are tailored for the `mpmath.matrix` class. As a result, it is necessary to explicitly convert NumPy arrays into `mpmath` matrices before invoking these functions. In these cases, we can develop a wrapper to streamline this process.

```

from functools import wraps
def adapt_to_numpy(f):
    @wraps(f)
    def np_wrapper(a):
        if isinstance(a, np.ndarray):
            a = mp.matrix(a)
        return f(a)
    return np_wrapper

mplu = adapt_to_numpy(mp.lu)
mpcholesky = adapt_to_numpy(mp.cholesky)
mpnorm = adapt_to_numpy(mp.norm)

```

In addition, to simplify the integration of `mpmath` arrays within NumPy code, we can implement some NumPy-style functions to accommodate the `mparray` class, such as

```

def mpzeros(shape):
    return np.full(shape, mp.mpf(0)).view(mparray)

def mpempty(shape):
    return np.empty(shape, dtype=object).view(mparray)

def mpeye(n):
    out = mpzeros((n, n))
    idx = np.arange(n)
    out[idx, idx] = mp.mpf(1)
    return out.view(mparray)

```

More practical examples of using NumPy with the `mpmath` library for the computation of integral quadratures will be presented in Chapter 12.

2.2.5 Structured array

In addition to the standard numerical data types, NumPy also supports structured array with custom types for labeled data. For instance, the following structured array utilizes the `atom_type` to represent the atom information in a molecule:

```

In [1]: atom_type = np.dtype([
    ('symbol', 'U2'),
    ('x', float),
    ('y', float),
    ('z', float)
])
mole = np.array([('H', 0.5, 0.5, 0.5), ('H', 0, 0, 0)], 

```

```
dtype=atom_type)
```

- The `dtype` of the structured array is similar to the type definitions of C/C++ structures. This similarity is useful when implementing interface with C/C++ programs. Therefore, the structured array can represent an array of C/C++ structures. Further details on this topic can be found in Chapter 8.
- Structured arrays work similar to the `DataFrame` class provided by the Pandas library. Compared to `DataFrame`, structured arrays offer fewer functionalities. For the day-to-day use case, Pandas is generally the more adequate choice for handling labeled data.
- When functionalities are available in both structured arrays and Pandas `DataFrame`, structured arrays can be more efficient in performance due to their better data alignment and the simpler data indexing methods. When performance is a concern, structured arrays can be used to optimize Pandas `DataFrame` code.

There are two approaches to indexing a structured array. The first approach treats it as a conventional array, allowing the use of integer and slice indices to access a single element or a sub-block of the structured array. All indexing methods we previously discussed, such as fancy indexing and mask arrays, can be used to index the structured arrays. The resultant element is the array or an element of the compound data type.

```
In [2]: mole[0]
Out[2]:
('H', 0.5, 0.5, 0.5)

In [3]: mole[0].dtype
Out[3]:
dtype([('symbol', 'U2'), ('x', 'f8'), ('y', 'f8'), ('z', 'f8')])
```

The second method for accessing elements in a structured array is through the fields of the compound type, for instance

```
In [4]: m['x']
Out[4]:
array([0.5, 0.])
```

2.3 Data with labels: Pandas

In data analysis tasks, we may frequently find labeled data. Although NumPy structured arrays provide certain functionalities for handling data with labels, Pandas is recognized as the premier tool for such tasks. If we consider NumPy as a tool for

vectorized operations on lists, Pandas can be viewed as a vectorized toolkit designed for dictionaries.

In this section, we will explore the fundamental usage of Pandas. Given that data analysis tasks in chemistry are typically not complicated, we will not delve into advanced or extraordinary Pandas tricks. For readers interested in in-depth materials, the official Pandas documentation is an excellent resource. Additionally, there are several high quality books which offer comprehensive and detailed examples for mastering the basics of the Pandas library, such as *Python Data Science Handbook: Essential Tools for Working with Data*, 2Ed [12] by Jake Vanderplas and *Python for Data Analysis*, 3Ed [13] by William McKinney.

In many data analysis programs, the abbreviation `pd` is used as a substitute for `pandas` for convenience. We will use this abbreviation throughout this section.

2.3.1 Pandas data objects

2.3.1.1 Series

A Pandas `Series` is essentially a one-dimensional array characterized by labels for its elements. It can be constructed from a dictionary, a list, or a NumPy array. For instance, we can create a `Series` object to represent an atom in a molecule:

```
In [1]: an_atom = pd.Series({'symbol': 'O', 'x': 0.5009, 'y': 2.7238, 'z': 0.8464})
```

In this example, the keys of the dictionary act as labels for the elements in the `Series` object. Labels can be accessed via the `index` attribute of a `Series` object (somewhat like the `.keys()` method of a dictionary).

```
In [2]: an_atom.index
Out[2]:
Index(['symbol', 'x', 'y', 'z', 'Znuc'], dtype='object')
```

A `Series` object can be constructed from a list or a NumPy array:

```
In [3]: coordinates = pd.Series(np.random.rand(2, 3))

In [4]: an_atom = pd.Series([0, 0.5009, 2.7238, 0.8464],
                           index=['symbol', 'x', 'y', 'z'])
```

The labels are specified in a list and passed to the `Series` constructor using the keyword argument `index`. The number of index labels must match the size of the list. If the keyword `index` is not specified, the `Series` object automatically generates a continuous `RangeIndex`, which functions similar to the `range(n)` index, where `n` is the size of the `Series`.

When treating a `Series` object as a single entity, it functions similarly to a NumPy array. It supports indexing with an integer, a slice, a list of integers, a NumPy array, or a Pandas `Index` object. Furthermore, `Series` objects can be seamlessly integrated with

NumPy. A `Series` object can be passed as the argument to most NumPy universal functions (ufuncs), which then perform element-wise operations and return a new `Series` object. For instance:

```
In [5]: np.sqrt(pd.Series([0, 1, 4]))  
Out[5]:  
0    0.0  
1    1.0  
2    2.0  
dtype: float64
```

A `Series` object can be considered as a fixed-size dictionary. We can index a `Series` object using strings or a list of strings:

```
In [6]: an_atom[['z', 'y', 'x']]  
Out[6]:  
z    0.8464  
y    2.7238  
x    0.5009  
dtype: object  
  
In [7]: an_atom['x'] += 0.5
```

We can use a `Series` object to update the value of another `Series` object, which is analogous to the `.update()` method of a dictionary:

```
In [8]: an_atom.update({'x': 1.5, 'z': 2.5})
```

Additionally, the dictionary comprehension syntax also works for `Series` objects:

```
In [9]: {**an_atom}  
Out[9]: {'symbol': 'O', 'x': 1.5, 'y': 2.7238, 'z': 2.5}
```

2.3.1.2 DataFrame

Pandas `DataFrame` is another fundamental data structure for storing tabular data, which functions like a two-dimensional array. A `DataFrame` is essentially a collection of `Series` objects, stacked along columns. For example, the geometry of a molecule can be stored in a `DataFrame` table:

```
In [1]: mole = pd.DataFrame({  
    'symbol': ['O', 'H', 'H'],  
    'x': [0.5009, 1.4049, -0.0934],  
    'y': [2.7238, 2.4390, 1.9786],  
    'z': [0.8464, 0.6938, 0.7321],  
})
```

```

        print(mole)
Out[1]:
symbol      x      y      z
0      0  0.5009  2.7238  0.8464
1      H  1.4049  2.4390  0.6938
2      H -0.0934  1.9786  0.7321

```

A DataFrame has two sets of labels to index its elements. The labels can be accessed through the `index` attribute (corresponding to the rows of the table) and the `columns` attribute. Due to the dual labeling system, a DataFrame can be viewed as a nested dictionary, where both rows and columns can be indexed using labels. An element within a DataFrame can be accessed by two labels, first by column then by row:

```

In [2]: mole.index
Out[2]:
RangeIndex(start=0, stop=3, step=1)

In [3]: mole.columns
Out[3]:
Index(['symbol', 'x', 'y', 'z'], dtype='object')

In [4]: mole['symbol'][0]
Out[4]:
'0'

```

Pandas provides various methods to access the elements within a DataFrame. We will explore more details of Pandas indexing methods in Section 2.3.3.

DataFrame objects can be constructed from iterable objects such as dictionaries, lists of Series, NumPy arrays, etc. When constructing a DataFrame from a dictionary, as illustrated in the previous example, the keys of the dictionary become the column labels of the DataFrame. Each item of the dictionary is processed as if constructing a Series object, following the previously mentioned rules for Series object construction. The Series' labels are imported to the row labels of the DataFrame, which can be accessed through the `DataFrame.index` attribute.

When constructing a DataFrame from a list of dictionaries, each dictionary in the list is converted into a Series object. The labels of these Series objects are then used as the column labels of the DataFrame. The row labels of the DataFrame form a RangeIndex, which corresponds to the position of each dictionary within the list.

```

In [5]: mole = pd.DataFrame([
    {'symbol': 'O', 'x': 0.5009, 'y': 2.7238, 'z': 0.8464},
    {'symbol': 'H', 'x': 1.4049, 'y': 2.4390, 'z': 0.6938},
    {'symbol': 'H', 'x': -0.0934, 'y': 1.9786, 'z': 0.7321},
])

```

```
    print(mole)
Out[5]:
  symbol      x      y      z
0     O  0.5009  2.7238  0.8464
1     H  1.4049  2.4390  0.6938
2     H -0.0934  1.9786  0.7321
```

This approach significantly differs from constructing a `DataFrame` from a dictionary, where row labels are derived from `Series.index`. To remove any ambiguity regarding the index and column labels, we can specify the index and columns explicitly when constructing the `DataFrame`.

```
In [6]: mole = pd.DataFrame(
    [('O', 0.5009, 2.7238, 0.8464),
     ('H', 1.4049, 2.4390, 0.6938),
     ('H', -0.0934, 1.9786, 0.7321)],
    index=range(3), columns=['symbol', 'x', 'y', 'z']
)
```

When constructing a `DataFrame` from other iterable objects, the process can be conceptually divided into two steps. First, each element of the iterables is converted into a `Series`. Next, depending on whether the iterable is a mapping or a list-like container, the construction process is classified into one of the two scenarios mentioned previously. For instance, a `DataFrame` can be constructed from a structured array:

```
In [7]: atom_type = np.dtype([
    ('symbol', 'U2'),
    ('x', float),
    ('y', float),
    ('z', float)])
mole_array = np.array([
    ('O', 0.5009, 2.7238, 0.8464),
    ('H', 1.4049, 2.4390, 0.6938),
    ('H', -0.0934, 1.9786, 0.7321]), dtype=atom_type)
pd.DataFrame(mole_array)

Out[7]:
  symbol      x      y      z
0     O  0.5009  2.7238  0.8464
1     H  1.4049  2.4390  0.6938
2     H -0.0934  1.9786  0.7321
```

The column labels of the `DataFrame` are derived from the `dtype` of the structured array, since each element of the structured array can be manipulated like a dictionary.

`DataFrames` can also be created from various file types and strings, such as from CSV, JSON, and databases. For instance, we can initialize a `DataFrame` from a CSV string as shown below:

```
In [8]: !cat water.csv
Out[8]:
symbol,x,y,z
0,0.5009,2.7238,0.8464
H,1.4049,2.4390,0.6938
H,-0.0934,1.9786,0.7321

In [9]: mole = pd.read_csv('water.csv')
```

The `DataFrame` structure is suitable for storing two-dimensional data. In certain scenarios, we may want more dimensions for data storage. For instance, suppose that we have a series of geometry configurations for a molecular motion trajectory. It could be more convenient to manage the data with an additional dimension for the time tick. To incorporate the additional information in a `DataFrame`, the nested indexing approach with `MultiIndex` [14] can be utilized to “group” data along one dimension.

```
In [10]: pd.concat({0: mole, 1: mole}, names=['tick', 'Id'])
Out[10]:
      symbol      x      y      z
tick Id
0   0     0  0.5009  2.7238  0.8464
    1     H  1.4049  2.4390  0.6938
    2     H -0.0934  1.9786  0.7321
1   0     0  0.5009  2.7238  0.8464
    1     H  1.4049  2.4390  0.6938
    2     H -0.0934  1.9786  0.7321
```

However, working with `MultiIndex` is complex in terms of interpretation and manipulation. For scenarios involving higher-dimensional labeled data, the `xarray` library [15] is a more suitable alternative, offering a more straightforward approach.

2.3.2 Broadcasting

While Pandas data objects share many similarities with NumPy arrays, the rules for broadcasting are slightly different. Pandas supports broadcasting among its data objects, NumPy arrays, and scalar variables. When applying regular arithmetic operations, or NumPy ufunc operations with a `Series` or a `DataFrame`, the broadcasting mechanism follows the same principles observed in NumPy. In such scenarios, the Pandas object is treated as an array and the shape of the other operand must be compatible with the shape of the Pandas object.

When operations are carried out between two Pandas `Series` objects, Pandas aligns their labels. This implies that the size of the two `Series` objects can be different. For any labels that do not match between the two `Series`, Pandas assigns `NaN` (not a number) to the corresponding fields. Therefore, operations can be carried out

even when the `Series` objects do not have identical labels. For instance, adding a `Series` with labeled data to another one with `RangeIndex` labels results in:

```
In [11]: s1 = pd.Series([0.1, 0.2, 0.3])
         s2 = an_atom
         s1 + s2
Out[11]:
0      NaN
1      NaN
2      NaN
symbol    NaN
x      NaN
y      NaN
z      NaN
dtype: object
```

This process can be understood in terms of the iteration code for two dictionaries:

```
out = {}
for key in {*s1.keys(), *s2.keys()}:
    if key in s1 and key in s2:
        out[key] = s1[key] + s2[key]
    else:
        out[key] = np.nan
```

Here, the expression `{*s1.keys(), *s2.keys()}` yields the union keys from the two `Series` objects.

When ufuncs are performed between two `DataFrame` tables, Pandas aligns both the index and column labels. If the `DataFrame` tables have unmatched labels, similar to the treatment of `Series`, Pandas will include the corresponding rows or columns in the result and assign `NaN` to these fields.

```
In [12]: mole = pd.read_csv('water.csv')
         mole1 = pd.DataFrame([
             {'x': 1.0},
             {'x': -1.0},])
         mole = mole1
Out[12]:
symbol      x     y     z
0      NaN -0.4991  NaN  NaN
1      NaN  2.4049  NaN  NaN
2      NaN      NaN  NaN  NaN
```

When operating on a `DataFrame` with a `Series`, the `Series` is applied along the columns by default. The column labels of the `DataFrame` and the `Series`'s index will be aligned. If the goal is to apply the `Series` along the rows, there are a couple of

approaches available. One option is to convert the `Series` object into a NumPy array using the `to_numpy()` method. We can then reshape the NumPy array (by adding a new axis as a column) to enable broadcasting with `DataFrame`. Alternatively, we can directly utilize ufunc methods, specifying the direction of application with the `axis` argument. The direction can be set to `axis='index'` (or equivalently `axis=0`) for the operation across the rows, or to `axis='columns'` (`axis=1`) for the columns. The following example illustrates both methods for the task of calculating the center of mass of a molecule.

```
mass = pd.Series([15.999, 1.008, 1.008])
# option 1
charge_center = np.sum(mole[['x', 'y', 'z']] * mass.to_numpy()[:,None],
                       axis=0)
# option 2
charge_center = np.sum(mole[['x', 'y', 'z']].mul(mass, axis=0), axis=0)
```

When broadcasting between Pandas objects, we need to treat the label alignment feature with caution. Sometimes, this feature can lead to results contrary to our intuition. The data in Pandas objects may not be processed in the order they are displayed. *Even if the labels of two Pandas objects are integer indices, Pandas still treats them as keys for broadcasting.* For example, when using the indexing `[::-1]` to reverse the data in a `Series`, it also alters the labels of the `Series`. As a result, executing `ser + ser[::-1]` as shown below produces an output identical to `ser * 2`.

```
In [13]: ser = pd.Series([2, 3, 4])
        ser + ser[::-1]
Out[13]:
0    4
1    6
2    8
dtype: int64
```

If we wish to perform broadcasting for Pandas objects in the order as they appear, one effective approach is to strip the labels from one of the two objects using `to_numpy()`. This transforms the Pandas broadcasting into an operation between a Pandas object and a NumPy array, which then adheres to the broadcasting rules of NumPy arrays.

```
In [14]: ser + ser[::-1].to_numpy()
Out[14]:
0    6
1    6
2    6
dtype: int64
```

2.3.3 Indexing

Indexing elements in Pandas objects is similar to indexing elements in NumPy arrays. It is frequently used for selecting elements and modifying specific contents of Pandas data objects. Pandas indexing is more complicated than the indexing methods for NumPy arrays. Pandas developed several conventions for indexing elements in Series and DataFrame objects, including:

- Location-based indexing methods.
- Label-based indexing methods.
- Attribute-based indexing methods.

2.3.3.1 Location-based indexing

In the location-based indexing methods, a Series is treated as a one-dimensional array, and a DataFrame is like a two-dimensional array. Elements can be indexed based on their position, similar to how elements are indexed in a NumPy array. Pandas allows direct use of slicing within [] (such as [2:4]) to index data objects. The output is a subset of the Series or a range of rows from the DataFrame.

When using a single integer as the indexer, the behavior varies between Series and DataFrame objects. For a Series object, it retrieves an element at the specified position. In the case of a DataFrame, an integer indexer does not select a row as it would in a two-dimensional array. Instead, it is interpreted as a column label. To select a row at the required location, the .iloc attribute of DataFrame can be utilized. The .iloc indexing approach supports several NumPy fancy indexing options, including integer, index list, slice, and mask array.

```
mole.iloc[:2]
mole.iloc[2,2]
mole.iloc[1:,1:]
mole.iloc[[0,2,1]]
mole.iloc[1:,[False, True, False, True]]
```

The .iloc attribute does not support broadcasting rules in indexing, such as using the two-dimensional indices np.arange(3)[:,None]. Consequently, many of the techniques in NumPy fancy indexing are not directly applicable to pandas data structures. There is no need to consider how indices are coordinated through broadcasting rules like in NumPy. To select a sub-table from a Pandas DataFrame, we can simply place the row indices and column indices of the sub-table within the .iloc attribute. For example,

```
In [15]: mole.iloc[[1,2], [False,True,False,True]]
Out[15]:
      x      z
1  1.4049  0.6938
2 -0.0934  0.7321
```

2.3.3.2 Label-based indexing

Label-based indexing is facilitated by the `.loc` attribute of Pandas data objects. In label-based indexing methods, indexers — whether they are integers, strings, or lists of strings — are all treated as keys. Accessing an indexer is like retrieving a value using a key in a conventional dictionary. A pair of keys can be used to index a specific element within the `DataFrame` table. We can use the `.loc` attribute with a tuple of indexers to access elements.

```
mole.loc[:, 'x']
mole.loc[[0, 2], ['x', 'z']]
mole.loc[:2, 'x':'z']
```

Please note that, in the `.loc` label-indexing mode, an integer is treated as a label, rather than its numeric value. Integer labels in the Pandas object might not be sorted in an ascending order. For instance, consider the following code snippet where the `index` is reversed. The output of the indexing expression `.loc[2]` will differ from the output of the indexing `.iloc[2]`.

```
In [16]: mole_inv = mole.iloc[::-1]
          mole_inv.loc[2]

Out[16]:
symbol      H
x       -0.0934
y        1.9786
z        0.7321
Name: 2, dtype: object

In [17]: mole_inv.iloc[2]
Out[17]:
symbol      0
x         0.5009
y         2.7238
z         0.8464
Name: 0, dtype: object
```

The label-indexing method allows us to use labels as boundaries of a slice, such as `mole.loc[:, 'x':'z']`. Unlike slicing with location-indexing, where the ending position is excluded, the ending position is included when using the label-slicing method. Please note that when integers are specified in a slice for the `.loc` attribute, these integers are also interpreted as labels, rather than positional indicators. Therefore, the ending integer is included in the output. This is very different from the convention used in `.iloc` indexing mode. For example, the following integer slices, although look similar, produce different outputs for `.loc` and `.iloc`.

```
In [18]: mole.loc[:2, 'x':'z']
Out[18]:
```

```
x      y      z
0  0.5009  2.7238  0.8464
1  1.4049  2.4390  0.6938
2 -0.0934  1.9786  0.7321

In [19]: mole.iloc[:2]
Out[19]:
symbol      x      y      z
0          O  0.5009  2.7238  0.8464
1          H  1.4049  2.4390  0.6938

In [20]: mole_inv.loc[2:1]
Out[20]:
symbol      x      y      z
2          H -0.0934  1.9786  0.7321
1          H  1.4049  2.4390  0.6938

In [21]: mole_inv.iloc[2:1]
Out[21]:
Empty DataFrame
Columns: [symbol, x, y, z]
Index: []
```

If the specified integer for slicing is not present within the axis labels, the output will be an empty data object. Consequently, the wraparound notation `:-1`, which is commonly used in Python indexing, is not applicable in the `.loc` indexing method.

```
In [22]: mole.loc[0:-1]
Out[22]:
Empty DataFrame
Columns: [symbol, x, y, z, Z]
Index: []
```

2.3.3.3 Attribute-based indexing

In the attribute indexing method, accessing an attribute of a Series or a DataFrame is translated into label indexing. If a label of a Series or DataFrame is a string, it can be treated as an attribute of the object. For example

```
In [23]: print(mole.symbol)
Out[23]:
0    O
1    H
2    H
Name: symbol, dtype: object
```

Although this indexing approach does not offer the same level of flexibility as the previous two methods, it is also frequently used in Pandas code thanks to its simplicity.

2.3.3.4 Indexing elements in DataFrame

The indexing notation `[]` (i.e., the `__getitem__` method) of a `DataFrame` supports both location and label indexing conventions. Pandas needs to determine which indexing mode to use based on the type or the structure of the indexer. The outcome could be some rows of the table (returned by the location-based indexing) or certain columns (as a result of the label-based indexing). This flexibility might make the program difficult to understand. For clarity and ease of understanding, it is generally recommended to use the `.loc` and `.iloc` attributes for element selection in a `DataFrame`.

To select elements from a `DataFrame`, one would need two indexers, one for the row and the other for the column. Although it is technically possible to select elements from a `DataFrame` using chained indexing, such as `df[v1][v2]`, this approach is discouraged for several reasons:

- It is ambiguous since the indexing syntax `[]` might access either rows or columns, depending on the type of the indexer.
- An intermediate object will be created for the first indexer, which is inefficient.
- Modifying elements of the `DataFrame` is prohibited when using chained indexing [16].

The `.iloc` and `.loc` methods are specifically designed for location-based and label-based indexing methods, respectively. Mixing the two types of indexing methods in either mode is not supported. Doing so can result in incorrect outputs. If we aim to use different indexing methods for the rows and columns of a `DataFrame`, what approach can we take?

One solution is to convert one type of the indexer into the other type before employing `.loc` or `.iloc` to select elements. Converting a location index to a label index is straightforward. By selecting labels from the `Index` object using the location index, we can obtain a list of labels. These labels can then be utilized in the `.loc` indexing mode. The reversed conversion, mapping a label index to a location index, requires the `get_indexer` method from the `Index` class. This method can produce the location of the labels within the `Index`, for example

```
In [24]: num_idx = np.array([0, 2])
          key_idx = ['x', 'y', 'z']
          mole.loc[mole.index[num_idx], key_idx]

Out[24]:
      x      y      z
0  0.5009  2.7238  0.8464
2 -0.0934  1.9786  0.7321

In [25]: mole.iloc[num_idx, mole.columns.get_indexer(key_idx)]
```

```
Out[25]:
```

	x	y	z
0	0.5009	2.7238	0.8464
2	-0.0934	1.9786	0.7321

Another viable option is the use of a mask array, which can be freely combined with both location index and label index. This approach is supported by both `.loc` and `.iloc` indexing modes in Pandas. For instance, we can utilize the mask array to select the hydrogen atoms in the molecule using three different indexing syntaxes:

```
In [26]: h_mask = (mole.symbol == 'H').to_numpy()
mole[h_mask]
```

```
Out[26]:
```

	symbol	x	y	z
1	H	1.4049	2.4390	0.6938
2	H	-0.0934	1.9786	0.7321


```
In [27]: mole.loc[h_mask]
```

```
Out[27]:
```

	symbol	x	y	z
1	H	1.4049	2.4390	0.6938
2	H	-0.0934	1.9786	0.7321


```
In [28]: mole.iloc[h_mask]
```

```
Out[28]:
```

	symbol	x	y	z
1	H	1.4049	2.4390	0.6938
2	H	-0.0934	1.9786	0.7321

In this example, the `to_numpy` method is explicitly invoked to convert the boolean-type Series object `mole.symbol == 'H'` into a NumPy mask array. Please notice that although a boolean-type Series looks similarly to a NumPy boolean array, they are different when used as a mask array indexer. We should view the boolean-type Series indexer as a label-based indexer, and it cannot be applied to the `.iloc` method.

```
In [29]: out = mole[mole.symbol == 'H']
```



```
In [30]: out = mole.loc[mole.symbol == 'H']
```



```
In [31]: out = mole.iloc[mole.symbol == 'H']
```

```
...
```

```
NotImplementedError: iLocation based boolean indexing on an integer type is
not available
```

In other words, the `index` attribute of the boolean-type Series must match the labels of the target object. This matching process is identical to the loop code with a dictionary

```
# out = mole.loc[mole.symbol == 'H']
bool_series = {0: False, 1: True, 2: True}
for key in df.index:
    if key in bool_series and bool_series[key]:
        out.loc[key] = mole.loc[key]
```

2.3.4 query and eval methods

When selecting rows from a `DataFrame`, filtering rows based on specific columns is a common task. For example, to select hydrogen atoms that are located inside a small cubic box, we can use the following code:

```
In [32]: mole[(mole['symbol'] == 'H') &
            (mole['x'] > 0 & mole['x'] < 3.0) &
            (mole['y'] > 0 & mole['y'] < 3.0) &
            (mole['z'] > 0 & mole['z'] < 3.0)]
Out[32]:
      symbol      x      y      z
1      H  1.4049  2.439  0.6938
```

Although this selection code is not incorrect, it appears rather cumbersome. `DataFrame` class offers a `query()` method that allows for selection through a string expression. By using `query()`, the above selection code can be simplified to:

```
mole.query('symbol == "H" & x > 0 & x < 3.0 & y > 0 & y < 3.0 & z > 0 & z < 3.0')
```

or written in a more Pythonic fashion:

```
mole.query('symbol == "H" & 0 < x < 3.0 & 0 < y < 3.0 & 0 < z < 3.0')
```

In fact, string expressions are not limited to selecting elements. By using the `eval` method of the `DataFrame` class, the string expression can facilitate simple numerical computations as well. For example, using the `eval` method to evaluate the previous string-valued filter can create a mask array. By using this mask array to select elements, we achieve the same output as that obtained using the `query` method:

```
In [33]: mask = mole.eval('symbol == "H" & 0 < x < 3.0 & 0 < y < 3.0 & 0 < z < 3.0')
          print(mole[mask])
Out[33]:
      symbol      x      y      z
1      H  1.4049  2.439  0.6938
```

The `eval` method can be used for more general computations. For instance, the code snippet below demonstrates how to compute the distances from the atoms in the

molecule to a reference point at R:

```
In [34]: R = np.array([-1., -1., -1.])
expr = '(x - @R[0])**2 + (y - @R[1])**2 + (z - @R[2])**2'
dist = mole.eval(expr)**.5
```

The notation `@R` refers to a predefined variable. Pandas supports a variety of syntax for string expression evaluation, as documented in the official Pandas documentation [\[17\]](#).

What are the advantages of using the `query` and `eval` methods in Pandas? Besides simplicity and readability, these methods often perform faster than the traditional code. In these methods, the `numexpr` engine [\[18\]](#) is utilized to boost performance. In the traditional code, Python needs to continuously create temporary arrays or `Series` objects for the intermediate steps. This overhead becomes significant, especially for large-size `DataFrame` tables. By fusing multiple operations into a single loop, `numexpr` avoids the creation for intermediate arrays, thus significantly improving the speed of expression evaluation.

2.3.5 Altering structure of DataFrame

There are several typical scenarios that involve altering the structure of a `DataFrame` object, such as

- Changing the axis labels;
- Adding or deleting rows or columns;
- Rearranging rows or columns.

2.3.5.1 Changing axes labels

To modify the row or column labels of a `DataFrame`, `set_axis()` and `rename()` are two frequently used methods. The `set_axis` method can take the list or Pandas `Index` to replace the row or column labels.

```
In [35]: mole = pd.read_csv('water.csv')
mole = mole.set_axis(['O0', 'H1', 'H2'], axis=0)
print(mole)

Out[35]:
   symbol      x      y      z
O0      O  0.5009  2.7238  0.8464
H1      H  1.4049  2.4390  0.6938
H2      H -0.0934  1.9786  0.7321
```

If the task is to modify only a few labels in the axes, the `rename` method is a more appropriate choice. This method requires a dictionary to specify the replacement rules.

```
In [36]: mole = mole.rename({'O': 'O', 'H1': 'H0', 'H2': 'H1'}, axis=0)
        mole = mole.rename({'symbol': 'name'}, axis=1)
        print(mole)
Out[36]:
      name      x      y      z
0    O  0.5009  2.7238  0.8464
H0   H  1.4049  2.4390  0.6938
H1   H -0.0934  1.9786  0.7321
```

2.3.5.2 Inserting or removing rows and columns

To insert a few rows or columns into a DataFrame, we can assign data to new labels:

```
mole = pd.read_csv('water.csv')
In [37]: mole.loc[:, ['mass', 'Znuc']] = \
            np.array([[15.999, 1.008, 1.008], [8, 1, 1]).T
        mole.loc['O1'] = mole.iloc[0]
        print(mole)
Out[37]:
      name      x      y      z      mass  Znuc
0    O  0.5009  2.7238  0.8464  15.999     8
H0   H  1.4049  2.4390  0.6938   1.008     1
H1   H -0.0934  1.9786  0.7321   1.008     1
O1   O  0.5009  2.7238  0.8464  15.999     8
```

Alternatively, we can put new data in a separated table, and then use the `pd.concat` function to combine the new table with the original one.

```
In [38]: mole = pd.read_csv('water.csv')
        info = pd.DataFrame([[15.999, 8],
                             [1.008, 1],
                             [1.008, 1]], columns=['mass', 'Znuc'])
        pd.concat([mole, info], axis=1)
Out[38]:
      symbol      x      y      z      mass  Znuc
0      O  0.5009  2.7238  0.8464  15.999     8
1      H  1.4049  2.4390  0.6938   1.008     1
2      H -0.0934  1.9786  0.7321   1.008     1
```

The `pd.concat` function automatically aligns labels. If the labels of two DataFrame are not completely identical, the extra labels will be placed in new columns (or rows)

```
In [39]: mole1 = pd.DataFrame([
        {'symbol': 'H', 'z': 0.5843, 'x': 0.2245, 'y': 3.6050,
         'Zeff': 0.5}])
```

```
pd.concat([mole, mole1], axis=1)
Out[39]:
   symbol      x      y      z  Zeff
0        O  0.5009  2.7238  0.8464    NaN
1        H  1.4049  2.4390  0.6938    NaN
2        H -0.0934  1.9786  0.7321    NaN
0        H  0.2245  3.6050  0.5843  0.5
```

When adding new columns to a table, we might want to merge data according to the category information in another column. This process is analogous to the `join` operation in relational databases. The `join` method in `DataFrame` offers the functionality of merging one table into another. For instance, consider a table that stores constants for various atoms, such as the atom masses and nuclear charges. We can insert this information to new columns in the `mole` table according to the type of atom. Instead of iterating over each atom in `mole` and assigning extra properties one by one, the `join` method can efficiently extract the properties from the second table with a single command.

```
In [40]: mole = pd.read_csv('water.csv')
atom_prop = pd.DataFrame({
    'mass': {'H': 1.008, 'C': 12.01, 'O': 15.999},
    'Znuc': {'H': 1, 'C': 6, 'O': 8})
mole.join(atom_prop, on='symbol', how='left')
Out[40]:
   symbol      x      y      z    mass  Znuc
0        O  0.5009  2.7238  0.8464  15.999      1
1        H  1.4049  2.4390  0.6938   1.008      1
2        H -0.0934  1.9786  0.7321   1.008      1
```

In this example, the keyword `on='symbol'` specifies that the elements in the `'symbol'` column are used as keys to select corresponding elements from the second table. This command can be viewed as the result of the series of operations listed below.

```
In [41]: on = mole.loc[:, 'symbol']
df1 = atom_prop.loc[on]
df2 = df1.reset_index(drop=True)
pd.concat([mole, df2], axis=1)
```

Besides the `join` method, one can also use `pd.merge` to join two `DataFrames`. The `pd.merge` function supports joining two tables with a broader range of data merging control options. Nevertheless, its fundamental concept is not significantly different from the `join` method. When setting the merging options, the `pd.merge` function requires specifying options for each table, while in the case of `DataFrame.join`, we only need the settings for one table. For instance, we can use `pd.merge` to achieve the same results as the `join` command in the previous example.

```
In [42]: pd.merge(mole, atom_prop,
                 left_on='symbol', right_index=True, how='left')
Out[42]:
   symbol      x      y      z    mass  Znuc
0      O  0.5009  2.7238  0.8464  15.999     1
1      H  1.4049  2.4390  0.6938   1.008     8
2      H -0.0934  1.9786  0.7321   1.008     8
```

The merging keys of the first table are specified by the keywords `left_on` and `left_index`, whereas the keys for the second table are determined by `right_on` and `right_index`. The `left_on` keyword specified that the column from the first table is to be used as the join keys. If `left_index=True` is set, the index of the first table should serve as the join keys. Similarly, the keywords `right_on` and `right_index` apply the same logic to the second table. Roughly, the output of the `pd.merge` function can be viewed as the result of the following four steps.

1. Extract keys from both tables.
2. Merge these keys to create a set of union keys for indexing.
3. Use the union keys to index the elements in both tables.
4. Concatenate the output, similar to the process demonstrated previously for the `join` method.

By setting different values for the keyword `how`, the `DataFrame.join` method and the `pd.merge` function can merge tables and expand the rows or columns of a table in various ways. Specifying `how='left'` means that the output table will exclusively use the keys from the first table. In other words, no additional rows will be extended in the output table.

`JOIN` is the fundamental operation in relational databases (SQL DB). The `join` and `merge` methods in Pandas are closely related to the design of `JOIN` operation in SQL. Therefore, in addition to the Pandas documentation, one can also consult the SQL documentation for the rules and examples on table joining. A comprehensive comparison for the `JOIN` operation between SQL and Pandas is available in the official Pandas document [19].

2.3.5.3 Reorganizing rows and columns

In many cases, reordering rows and columns just requires indexing operation. For instance, to change the column order in a `DataFrame`, we can index the columns by a list of labels in the new order:

```
In [43]: mole.loc[:, ['x', 'y', 'z', 'symbol']]
Out[43]:
      x      y      z  symbol
0  0.5009  2.7238  0.8464      0
1  1.4049  2.4390  0.6938      H
2 -0.0934  1.9786  0.7321      H
```

Other methods, such as `pd.reindex` and `pd.sort_index`, also provide the functionality to reorder rows or columns. `pd.reindex` is quite similar to the `.loc` indexing method. The enhancement in this function is the capability to accommodate the absent labels. `pd.reindex` can create new rows or columns for the absent labels, thereby avoiding the `KeyError` which would otherwise be raised by the `.loc` indexing method.

```
In [44]: mole.reindex([2, 3, 1, 3, 0], axis=0)
Out[44]:
   symbol      x      y      z
2      H -0.0934  1.9786  0.7321
3     NaN      NaN      NaN      NaN
1      H  1.4049  2.4390  0.6938
3     NaN      NaN      NaN      NaN
0      O  0.5009  2.7238  0.8464

In [45]: mole.reindex([2, 3, 1, 3, 0], axis=0).sort_index(axis=0)
Out[45]:
   symbol      x      y      z
0      O  0.5009  2.7238  0.8464
1      H  1.4049  2.4390  0.6938
2      H -0.0934  1.9786  0.7321
3     NaN      NaN      NaN      NaN
3     NaN      NaN      NaN      NaN
```

In some scenarios, we might want to use the contents of specific columns as the new labels for rows and columns, and reorganize the table according to these new rows and columns. The functions `pd.pivot` and `pd.pivot_table` are specifically designed for this kind of rearrangement. For example, let's consider a cluster of water molecules.

```
In [46]: import io
cluster = pd.read_csv(io.StringIO('''
symbol,x,y,z,monomer,Q
O0,  2.8340, 0.1692,-1.5178, 1,-0.65
H1,  3.0891,-0.0804,-2.4090, 1, 0.33
H2,  2.5136, 1.0741,-1.5178, 1, 0.33
O0,  0.5009,  2.7238, 0.8464, 2,-0.64
H1,  1.4049,  2.4390, 0.6938, 2, 0.31
H2, -0.0934,  1.9786, 0.7321, 2, 0.32
'''))
```

In this table, unique identifiers are assigned to different water molecules under the label `monomer`. The `Q` column records the effective charges of the atoms within the cluster. To study the effective charges of the atoms in the cluster, we can extract them into a separate table using the `pd.pivot` function:

```
In [47]: cluster.pivot(columns='symbol', index='monomer', values='Q')
Out[47]:
symbol      H1      H2      O
monomer
1           0.33   0.33 -0.65
2           0.31   0.32 -0.64
```

In the output table, the columns `monomer` and `symbol` from the original table are reorganized into rows and columns, each with unique labels. The data from the `Q` column is positioned accordingly in the new table.

When using the `pd.pivot` function to reorganize a table, it is subject to certain constraints regarding the content of the `DataFrame`. For instance, it requires that the combination of `columns` and `index` must not contain duplicated label pairs. This restriction can be overcome by the more powerful tool `pd.pivot_table` [20]. The principle of `pd.pivot_table` is similar to that of the `pd.pivot` function. However, `pd.pivot_table` introduces the capability to apply aggregation methods to data with duplicate labels.

2.3.6 Data types

A Pandas `Series` object is restricted to a single data type across all fields. To check the data type of a `Series`, we can inspect its `.dtype` attribute. In the case of `DataFrame`, a `DataFrame` object can encompass multiple `Series` as its columns, accommodating different data types for different columns. The `.dtypes` attribute can be used to examine the data type of each column.

When creating a `DataFrame`, if data types are not explicitly defined, Pandas will automatically infer the data types for each column. This automatic inference is generally accurate. Nonetheless, it may become inaccurate when missing data are present in certain fields. Pandas infers the data types based on the available data, which might not always match the expected data types.

Let's consider the following example. The `Znuc` column is designed to record the number of protons for each atom using an integer type. This information is missing in two rows. These missing entries are labeled as `NaN` in Pandas. Because integer types cannot represent `NaN`, Pandas automatically infers the `dtype` of this column as `float` to accommodate `NaN` values.

```
In [48]: mole = pd.DataFrame([
    {'symbol': 'O', 'x': 0.5009, 'y': 2.7238, 'z': 0.8464, 'Znuc': 8},
    {'symbol': 'H', 'x': 1.4049, 'y': 2.4390, 'z': 0.6938, },
    {'symbol': 'H', 'x': -0.0934, 'y': 1.9786, 'z': 0.7321, }, ])
print(mole.dtypes)
Out[48]:
symbol      object
```

```
x      float64  
y      float64  
z      float64  
Znuc    float64  
dtype: object
```

Can we force Pandas to follow our specified data type for this column when constructing `DataFrame`? Unfortunately, as of version 2.2, Pandas does not support specifying data types for specific columns when instantiating `DataFrame`.

Some functions, such as `pd.read_csv`, provide the capability to specify data types for columns when creating `DataFrame`. If `DataFrame` is created from a CSV file, as shown in the following example, we can use a dictionary to enforce data type for each column. For columns where data types are not explicitly specified, Pandas automatically infers their data types.

```
In [49]: pd.read_csv(io.StringIO(''  
        symbol,x,y,z,Znuc  
        0, 0.5009,2.7238,0.8464,8  
        H, 1.4049,2.4390,0.6938,  
        H,-0.0934,1.9786,0.7321,  
        '''), dtype={'Znuc': pd.Int64Dtype()})  
Out[49]:  
      symbol      x      y      z   Znuc  
0       0  0.5009  2.7238  0.8464      8  
1       H  1.4049  2.4390  0.6938  <NA>  
2       H -0.0934  1.9786  0.7321  <NA>
```

You might have noticed that in this example, the data type for the `Znuc` column is defined as `pd.Int64Dtype()`, instead of the Python `int` or the NumPy `np.integer` type. `pd.Int64Dtype` is an extended integer type introduced by Pandas to overcome the limitation of the conventional integer types, which are unable to represent missing values or NaN data. The extended integer type uses `pd.NA` to denote missing values. In a similar fashion, to address the issue of missing value representation for other Python and NumPy built-in data types, Pandas also introduces extended types, such as '`Int32`', '`Int16`', '`boolean`', '`Float64`', '`Float32`', etc. The constant variable `pd.NA` is utilized to represent missing values for these extended types [21].

2.3.7 Missing data

Unlike the computation code with NumPy, missing values (or NaN) are quite common in Pandas data objects and operations. Let's list some typical scenarios where missing values can be introduced:

- When adding two `Series` (or `DataFrame`) objects, NaN values are generated for fields that exist only in one of the objects.

- Missing values can occur when merging or concatenating two `DataFrame` objects if there are unmatched labels.
- Missing values are often encountered when reading data from external sources, such as a CSV file.

It is crucial to handle `Nan` values carefully in Pandas applications, just as in NumPy. The following functions (and methods of `DataFrame` and `Series`) are commonly used to filter or process `Nan` values:

- `isna` generates a boolean mask to indicate the presence of missing values.
- `notna` identifies non-missing values, which is the opposite of `isna`.
- `dropna` removes any rows or columns that contain `Nan` values.
- `fillna` replaces `Nan` values with specified data.

2.3.8 Grouping and aggregation

The `groupby` method encapsulates the *split-apply-combine* operations into a single process. For instance, consider the example we used for the `pd.pivot` function. We can use the split-apply-combine approach to implement the functionality of `pd.pivot` as follows:

```
In [50]: out = []
index = []
for monomer in set(cluster['monomer']):
    # split
    df = cluster[cluster['monomer'] == monomer]
    # apply
    data = df.set_index('symbol').Q
    index.append(monomer)
    out.append(df)
    # combine
    out = pd.DataFrame(out, index=index)
print(out)

Out[50]:
symbol      H1      H2      00
1          0.33   0.33 -0.65
2          0.31   0.32 -0.64
```

The same functionality can be achieved using `groupby` in a single line of code.

```
out = cluster.groupby('monomer').apply(lambda df: df.set_index('symbol').Q)
```

In the *split* step, `groupby` categorizes the table based on the values of the specified column. For each category, it generates a `DataFrame`, which is then passed as an argument to the function specified by the `apply` method.

```
In [51]: for monomer, df in cluster.groupby('monomer'):
    print(monomer)
    print(df)

Out[51]:
1
  symbol      x      y      z  monomer      Q
0     00  2.8340  0.1692 -1.5178      1 -0.65
1     H1  3.0891 -0.0804 -2.4090      1  0.33
2     H2  2.5136  1.0741 -1.5178      1  0.33

2
  symbol      x      y      z  monomer      Q
3     00  0.5009  2.7238  0.8464      2 -0.64
4     H1  1.4049  2.4390  0.6938      2  0.31
5     H2 -0.0934  1.9786  0.7321      2  0.32
```

For efficiency, Pandas does not generate an actual intermediate DataFrame. Instead, the `groupby` method returns a `GroupBy` object, representing the outcome of the splitting process.

Besides using a column within the `DataFrame`, the `groupby` method can also split a `DataFrame` (or a `Series`) based on custom labels or any functions that can categorize the `index` of the Pandas object. For instance, custom labels are utilized for `groupby` in the following code:

```
In [52]: cluster['Q'].groupby(['A', 'B', 'A', 'B', 'A', 'B']).max()
Out[52]:
A    0.33
B    0.33
Name: Q, dtype: float64
```

`groupby` also supports a function to generate labels:

```
In [53]: cluster['Q'].groupby(lambda index: index % 2).max()
Out[53]:
0    0.33
1    0.33
Name: Q, dtype: float64
```

In the `apply` step, we can customize a Python function and pass it to the `GroupBy.apply` method. The output of the custom function for each group is then aggregated into a single Pandas object. The aggregation can result in different types of Pandas objects, depending on the data type of output from the custom function. If the output is not a Pandas data object, such as a scalar variable, it is aggregated into a `Series`. If the outputs are `DataFrame` objects, they are merged into a single `DataFrame`. If the custom function returns a `Series`, the `GroupBy.apply` method tries to infer the format of the aggregated result. The inference is based on the `index` of the `Series`. If

all outputs have the consistent `index`, the `apply` method returns a `DataFrame`. This is the reason why we invoke the `set_index('symbol')` method in the custom function in the previous example, which leads to a consistent index.

While the `apply` method offers a general solution for the `GroupBy` object, Pandas specializes several `groupby-apply` patterns for common use cases [22]. These functionalities include:

- *Aggregation.* The `GroupBy.agg` method is employed to apply aggregation functions to each column in each group. The aggregation function can reduce a column to a scalar. For instance,

```
cluster.groupby(cluster['monomer']).apply(lambda x: x.sum(axis=0))
```

can be rewritten using the column-wise `agg` method as

```
cluster.groupby(cluster['monomer']).agg(lambda x: x.sum())
```

or even more concisely as

```
cluster.groupby(cluster['monomer']).sum()
```

Certain aggregation functions, such as `max`, `sum`, `mean`, and `std`, are highly optimized and implemented as methods of the `GroupBy` class.

- *Transformation.* The `GroupBy.transform` method can replace the `DataFrame` of each group with a `DataFrame` that employs the same `index`.
- *Filtering.* The `GroupBy.filter` method can exclude certain groups according to the specified filtering rules.

2.3.9 View and copy

Pandas incorporates a memory-sharing design among its data objects, closely related to the concepts of view and copy in NumPy arrays. This characteristic raises an important question: how can we determine whether a Pandas object exclusively owns its data or shares it with others? Unfortunately, the data sharing rules in Pandas are not as clear as those in NumPy. Pandas employs different data sharing mechanisms for different data objects.

Memory management for `Series` objects is similar to that of regular NumPy arrays. When elements are selected from a `Series` object, data is shared between the `Series` objects whenever possible. In other words, a slice of a `Series` object shares memory with the original `Series`. Other indexing methods, such as location indexing or label indexing, will result in the data being copied to the new `Series`.

```
In [54]: an_atom = pd.Series({'symbol': 'O', 'x': 0.5009, 'y': 2.7238, 'z': 0.8464})
         ser = an_atom[3::-1]
         ser[1] = 2.5
         print(an_atom)
```

```

Out[54]:
symbol      0
x          0.5009
y          2.5
z          0.8464
dtype: object

In [55]: an_atom = pd.Series({'symbol': 'O', 'x': 0.5009, 'y': 2.7238, 'z': 0.8464})
          ser = an_atom[[3,2,1]]
          ser[1] = 2.5
          print(an_atom)

Out[55]:
symbol      0
x          0.5009
y          2.7238
z          0.8464
dtype: object

```

If a `Series` is transformed into a NumPy array using the `to_numpy()` method, the resulting NumPy array and the original `Series` object share the same data buffer.

```

In [56]: atom_array = an_atom.to_numpy()
          atom_array[1:] = 0.
          print(an_atom)

Out[56]:
symbol      0
x          0.0
y          0.0
z          0.0
dtype: object

```

A Pandas `DataFrame` is a collection of `Series` objects. If we select a column from a `DataFrame`, we are essentially accessing a reference to the corresponding `Series` object. This reference acts as a view of the original `DataFrame`. Modifications made to these `Series` objects will be reflected in the associated `DataFrame`.

```

In [57]: mole = pd.read_csv('water.csv')
          coord_y = mole.loc[:, 'y']
          mole.loc[mole.symbol=='O', 'x':'z'] = 1.0
          print(coord_y)

Out[57]:
0    1.0000
1    2.4390
2    1.9786

```

```
Name: y, dtype: float64
```

This is the only scenario in `DataFrame` that we can ensure the data sharing behavior. Any other operations, such as selecting a single row, slicing multiple rows or columns, or transposing a `DataFrame`, may result in either a view or a copy of the original table. The outcome is influenced by multiple factors, including the data types, the structure of the table, the operations performed on the table, and so forth. Therefore, it is difficult to assure whether modification to a `DataFrame` will impact the contents of its sub-tables. In fact, modifying Pandas data objects after their creation is generally discouraged. If modifications are necessary, whenever in doubt, it is advisable to explicitly invoke the `.copy()` method to create an independent copy of the data.

NumPy eagerly returns array views for better memory efficiency. Unlike NumPy, Pandas does not prioritize performance in the same way when dealing with data views and copies. Even when it is possible to reuse data in the output, functions and methods in Pandas, such as `set_axis` and `fillna`, return a new copy of the data object. Such a design choice can lead to performance loss, especially when working with larger datasets. To address this issue, many Pandas functions provide the `inplace=True` keyword argument. This argument, when set, instructs the functions to modify the input `DataFrame` directly, rather than creating a new copy. By default, the `inplace` option is always disabled.

Summary

In this chapter, we introduced some basic usage methods for NumPy and Pandas, including ufuncs, broadcasting, indexing, and other features for both libraries. We discussed in detail the data structure of NumPy arrays and the characteristics of array views. Based on this knowledge, we proposed numerous usage tips for both NumPy and Pandas. However, these only represent a fraction of what is possible with NumPy and Pandas. These libraries are particularly important tools, as they are extensively used in data analysis works. Carefully reading their documentation is always beneficial.

In the next chapter, we will shift our focus to data visualization, which is another crucial aspect of data processing and analysis.

References

- [1] NumPy Developers, Numpy documentation - universal functions (ufuncs), <https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs>, 2024.
- [2] Wikipedia Contributors, Page fault, https://en.wikipedia.org/wiki/Page_fault, 2024.
- [3] Wikipedia Contributors, Addressing mode, https://en.wikipedia.org/wiki/Addressing_mode, 2024.
- [4] NumPy Developers, Numpy documentation - the array interface protocol, https://numpy.org/doc/stable/reference/arrays.interface.html#object.__array_interface__, 2024.

- [5] Wikipedia Contributors, Cpu cache, https://en.wikipedia.org/wiki/CPU_cache, 2024.
- [6] Python Software Foundation, multiprocessing.shared_memory – shared memory for direct access across processes, https://docs.python.org/3/library/multiprocessing.shared_memory.html, 2024.
- [7] Wikipedia Contributors, Indeterminate form, https://en.wikipedia.org/wiki/Indeterminate_form, 2024.
- [8] The mpmath development team, mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.3.0), <http://mpmath.org/>, 2023.
- [9] S. Berg, Nep 41 — first step towards a new datatype system, <https://numpy.org/neps/nep-0041-improved-dtype-support.html>, 2024.
- [10] L.P. Ramos, Duck typing in python: writing flexible and decoupled code, <https://realpython.com/duck-typing-python/>, 2024.
- [11] NumPy Developers, Numpy documentation - subclassing ndarray, <https://numpy.org/doc/stable/user/basics.subclassing.html>, 2024.
- [12] J. VanderPlas, Python Data Science Handbook: Essential Tools for Working with Data, O'Reilly Media, Inc, Sebastopol, CA, 2016.
- [13] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, O'Reilly Media, 2015, <https://books.google.co.in/books?id=bIZHCgAAQBAJ>.
- [14] Pandas Development Team, Pandas user guide - multiindex / advanced indexing, https://pandas.pydata.org/docs/user_guide/advanced.html, 2024.
- [15] S. Hoyer, J. Hamman, xarray: N-D labeled arrays and datasets in Python, Journal of Open Research Software 5 (1) (2017), <https://doi.org/10.5334/jors.148>.
- [16] Pandas Development Team, Pandas user guide - why does assignment fail when using chained indexing?, https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#why-does-assignment-fail-when-using-chained-indexing, 2024.
- [17] Pandas Development Team, Pandas user guide - expression evaluation via eval(), https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html#expression-evaluation-via-eval, 2024.
- [18] D.M. Cooke, F. Alted, et al., Numexpr: fast numerical expression evaluation, <https://numexpr.readthedocs.io/en/latest/>, 2024.
- [19] Pandas Development Team, Pandas user guide - comparison with sql, https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_sql.html#compare-with-sql-join, 2024.
- [20] Pandas Development Team, Pandas user guide - reshaping and pivot tables, https://pandas.pydata.org/pandas-docs/stable/user_guide/reshaping.html, 2024.
- [21] Pandas Development Team, Pandas user guide - working with missing data, https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html, 2024.
- [22] Pandas Development Team, Pandas user guide - group by: split-apply-combine, https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html, 2024.

Visualization

3

Visualization plays a crucial role in data processing and analysis. It can provide a more intuitive understanding of problems or ideas compared to formulas and theoretical analysis.

A wide range of open-source tools is available for creating graphics in Python. In the realm of scientific data, Matplotlib emerges as the primary tool for handling data in NumPy arrays and Pandas data objects.

Although Matplotlib is excellent for creating 2D images, its capabilities for 3D visualization are quite limited and its performance is poor. So, what options do we have for 3D visualization? One option is the Mayavi library. It is specifically designed for scientific data visualization.

Visualizing quantum chemistry data presents unique challenges. Besides using tools like Matplotlib and Mayavi for data analysis, we often need to use external visualization software to display 3D visualizations of objects like molecules, crystals, wavefunctions, etc. The external visualization software packages cannot be directly operated in Python. Instead, we have to interact with them using specific data formats, such as Molden and Cube files. How can we programmatically generate these files? The Ninja templating technique in Python provides some insights.

In this chapter, we will discuss how to use these Python tools for the tasks of data visualization.

3.1 Matplotlib

The following import convention for Matplotlib is applied in this section.

```
import matplotlib.pyplot as plt
```

A typical workflow for Matplotlib plotting can be summarized as follows:

1. Create a `plt.Figure` object and `plt.Axes` objects.
2. Invoke the plotting function provided by the `Axes` object.
3. Adjust axes properties as needed.
4. Display the plot.

In the first step, we can use the `plt.figure()` function to create a `Figure` instance. Within this `Figure`, there are several methods available to create single or multiple plots.

- `Figure.subplots()`: This function creates a set of plots. These plots are `Axes` objects.
- `Figure.add_subplot()`: This adds a single plot (i.e., `Axes` object) inside a figure.
- `Figure.subfigures()`: This creates a set of logical figures inside a figure. Within each sub-figure, creating plots (`Axes`) is still required before processing the actual plotting functionalities.
- `Figure.add_subfigure()`: This adds a single logical figure within a figure.
- `plt.subplot()`: A shortcut to create a plot. This command is essentially equivalent to `plt.figure().add_subplot()`.
- `plt.GridSpec()`: This class introduces grids, which can divide a figure into subplots of arbitrary shapes.
- `Figure.add_axes()`: This function inserts a single plot inside a figure.

Here, we briefly demonstrate how to use the `Figure.add_subplot` method. If you are interested in exploring other methods, we recommend consulting the Matplotlib official documentation or dedicated books such as *Python Data Science Handbook: Essential Tools for Working with Data* [1] by Jake Vanderplas. The function call

```
add_subplot(rows, cols, index)
```

divides the figure into `rows × cols` cells. Based on the argument `index`, one of these cells, from upper left to bottom right, will be selected and an `Axes` object will be created there. Fig. 3.1 illustrates the effects of the `add_subplot` command.

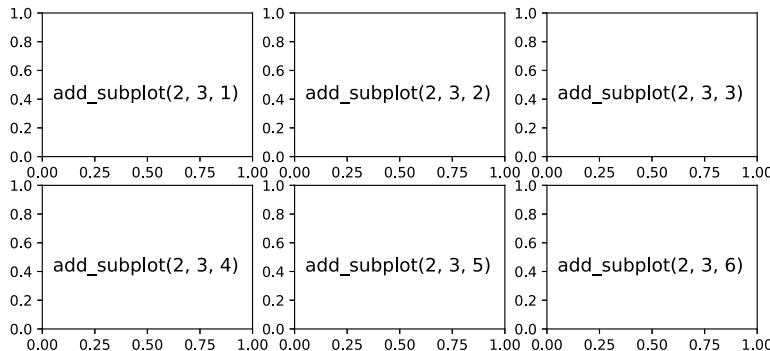
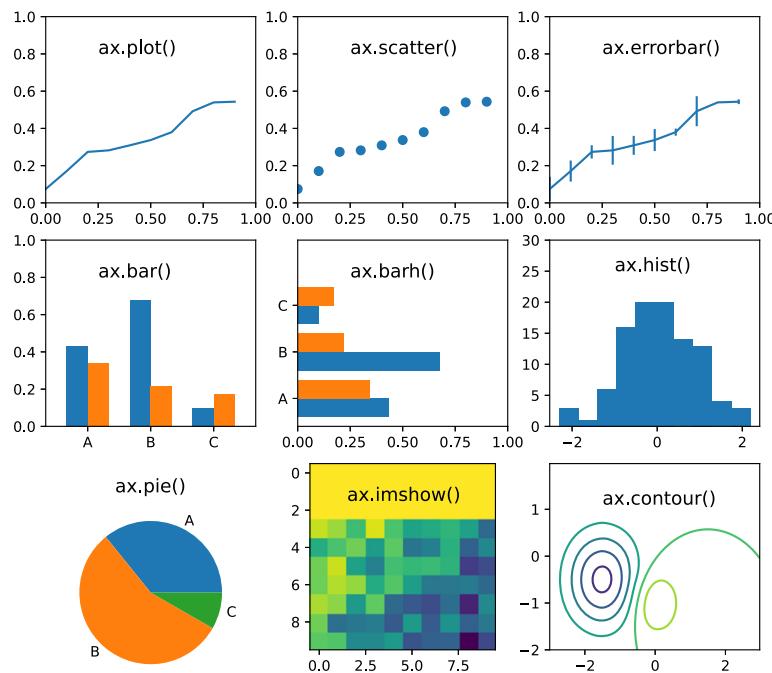


FIGURE 3.1

The output of `add_subplot`.

Next, we can create basic plots for the obtained `Axes` object. Several common visualization tasks are demonstrated below. The outcomes of these tasks are illustrated in Fig. 3.2.

**FIGURE 3.2**

The output of various plotting functions.

```
fig, ax = plt.subplots()

# Line plots
x = np.arange(10)
y = np.random.rand(10).cumsum()
ax.plot(x, y)

# Scatter plots
ax.scatter(x, y)

# Error bars
yerr = .1 * np.random.rand(len(x))
ax.errorbar(x, y, yerr=yerr)

# Vertical bars
np.random.seed(1)
cat = ('A', 'B', 'C')
```

```

weights = np.random.rand(2, 3)
ax.bar(np.arange(.15, .85, .3), weights[0], width=.1)
ax.bar(np.arange(.15, .85, .3)+.1, weights[1], width=.1)
ax.set_xticks([.2, .5, .8], labels=cat)

# Horizontal bars
ax.bart(np.arange(.1, .7, .25), weights[0], height=.1)
ax.bart(np.arange(.1, .7, .25)+.1, weights[1], height=.1)
ax.set_yticks([.15, .4, .65], labels=cat)

# Histogram
ax.hist(np.random.standard_normal(100), bins=10)

# Pie plots
ax.pie(weights, labels=cat)

# The value of a matrix
xy = np.random.rand(10,10)
ax.imshow(xy)

# Contour
X, Y = np.meshgrid(np.arange(-3.0, 3.0, .05), np.arange(-2.0, 2.0, .05))
r0 = X**2 + (Y+1.)**2
r1 = (X+1.5)**2 + (Y+.5)**2
Z = .32*np.exp(-2*r0) - .65*np.exp(-r1)
ax.contour(X, Y, Z)

```

After obtaining basic plots, we can improve the visual appearance by adjusting various properties of `Axes` objects, including color, line style, and axes styles, etc. In the following, we outline some of the properties that are frequently adjusted in plotting:

- *Data marker.* In both `ax.scatter` and `ax.plot` plots, we can customize the visualization effects of the data pointer marker, including their shape, size, color, and transparency. Common shape markers include:
 - '`o`' for a circle marker,
 - '`s`' for a square marker,
 - '`v`' for a downward-facing triangle marker,
 - '`^`' for an upward-facing triangle marker,
 - '`+`' for a plus-shaped marker,
 - '`x`' for a X-shaped marker.

The marker size can be modified with the `markersize` keyword. To specify colors, single letters like '`r`', '`g`', '`b`', '`k`' (black), '`c`' (cyan) can be assigned to the `color` keyword. Transparency can be controlled using the `alpha` keyword, which accepts a number between 0 and 1.

- *Line styles* for the `ax.plot` method. Options for the keyword `linestyle` (or `ls` in short) include:
 - solid line `ls='-'`,
 - dash line `ls='--'`,
 - dot line `ls=':'`,
 - dash-dot line `ls='-.'`.

The appearance of the line can be improved by using the `marker` keyword to highlight data points, the `color` keyword to set the color, and the `linewidth` keyword to adjust the line width. Besides these keywords, Matplotlib supports a shorthand string notation to simultaneously specify the data marker, line style, and line color. For instance, the shorthand notation '`s--r`' indicates a red dash line with square markers.

- *Color representation*. In addition to the single letter representation mentioned previously, color can also be specified through predefined names, such as `color='darkgreen'` and `color='tab:purple'`. Furthermore, Matplotlib supports the color definition in terms of RGB fractions, either by a hex code '`#RRGGBB`', or a three-float tuple `(r, g, b)`. Detailed descriptions of Matplotlib color schemes can be found in the Matplotlib documentation [2–4].
- *Appearance of axes*. The axes can be adjusted using the `set_xlim`, `set_ylim`, `set_xticks`, and `set_yticks` functions. As their names suggest, they can be used to adjust the range of the axes and the display of ticks on the axes, respectively. Axis labels can be added using the `ax.set_xlabel` and `ax.set_ylabel` methods. To enable logarithmic coordinates,

```
ax.set_xscale('log')
ax.set_yscale('log')
```

can be called.

- *Legend*. If the `label` keyword has been used in any plotting functions, the `ax.legend` method can generate a legend for these labels. The `loc` keyword in the `ax.legend` method can set the position of the legend. For instance,

```
ax.legend(loc='upper left')
```

places the legend in the upper left corner of the plot. Other common positions include 'upper center', 'upper right', 'center left', 'center right', etc.

- *Text and Annotation*. The `ax.text` method can insert an explanatory text to a plot. The command

```
ax.text(x, y, 'message')
```

positions the 'message' at the coordinates `(x, y)` on the plot. Additionally, the `ax.annotate` method offers a way to insert more informative elements along with the text, such as arrows and other annotation symbols. This method necessitates the specification of the locations for both the text and the arrow. For instance, the following statement positions the text 'message' at the coordinates `(2, 2)` and draws an arrow pointing from `(2, 2)` to `(1, 3)`.

```
ax.annotate('message', xy=(1, 3), xytext=(2, 2),
           arrowprops=dict(arrowstyle='->'))
```

- *Text formatting.* In text-related methods, such as `ax.text`, `ax.annotate`, and the previously mentioned `ax.set_xlabel`, `ax.set_xticks`, etc., parameters like `fontfamily`, `fontsize`, and `rotation` are available to adjust the appearance of text. Moreover, Matplotlib supports the LaTeX math notations within text messages. For instance, inputting the Tex-style notation `r'μ_0'` can display μ_0 . Please note the prefix `r` for defining a raw Python string. This ensures that the raw string is passed to Matplotlib functions without translating the backslash character.

The final step in the plotting workflow is to display or save the image. Normally, the `plt.show()` method is called to display images. However, explicitly calling the `plt.show()` function is not necessary if the `%matplotlib` magic is enabled in a Jupyter Notebook. Images are displayed automatically.

There are two common methods to save images. In the interactive window that displays the image, you can click the *Save* button and choose a format to save the plot. To save images in a non-interactive environment, the `Figure.savefig` function can be used, such as

```
fig.savefig('result.png')
fig.savefig('result.jpg')
fig.savefig('result.svg')
fig.savefig('result.pdf')
```

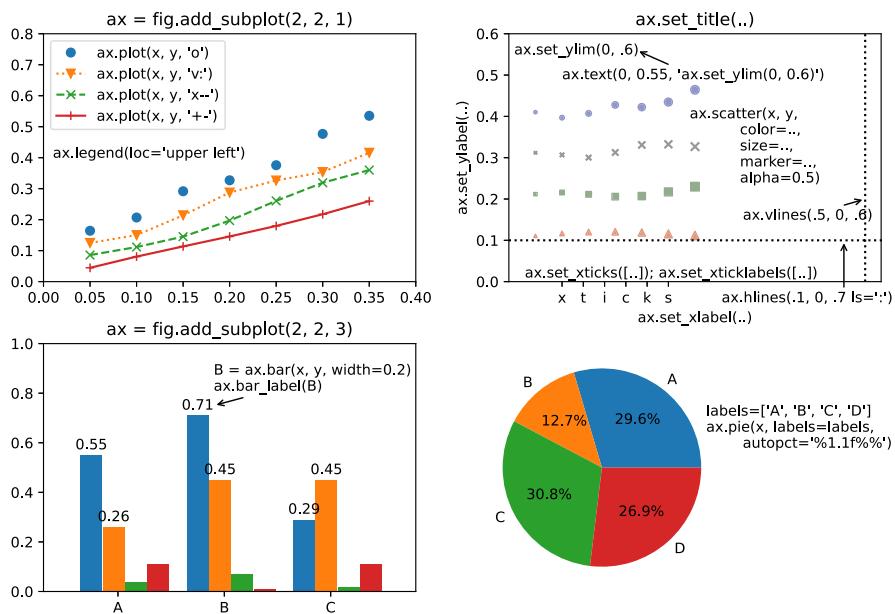
Based on the suffix of the filename, Matplotlib can automatically select an appropriate backend to render the plots in the required image format.

To help you quickly locating the basic configurations of a Matplotlib plot, we summarize the common functionalities of Matplotlib in Fig. 3.3

So far, we have provided an overview of how to use Matplotlib. If you are seeking a comprehensive guide to learning Matplotlib, *Python Data Science Handbook: Essential Tools for Working with Data, 2nd Edition* by Jake Vanderplas is an excellent reference. This book offers detailed instructions and examples on how to use Matplotlib. Additionally, exploring the gallery on Matplotlib official website is another effective method for mastering Matplotlib.

3.2 Pandas visualization

Several plotting functions in Matplotlib also support the visualization of Pandas data objects. For example, the `ax.plot` function can recognize `Series` and `DataFrame`, automatically selecting the index and data for visualization. Nevertheless, a more common method for visualizing Pandas data is the `.plot()` method of the `Series` and `DataFrame` objects. This Pandas built-in plotting feature extends the functionality of

**FIGURE 3.3**

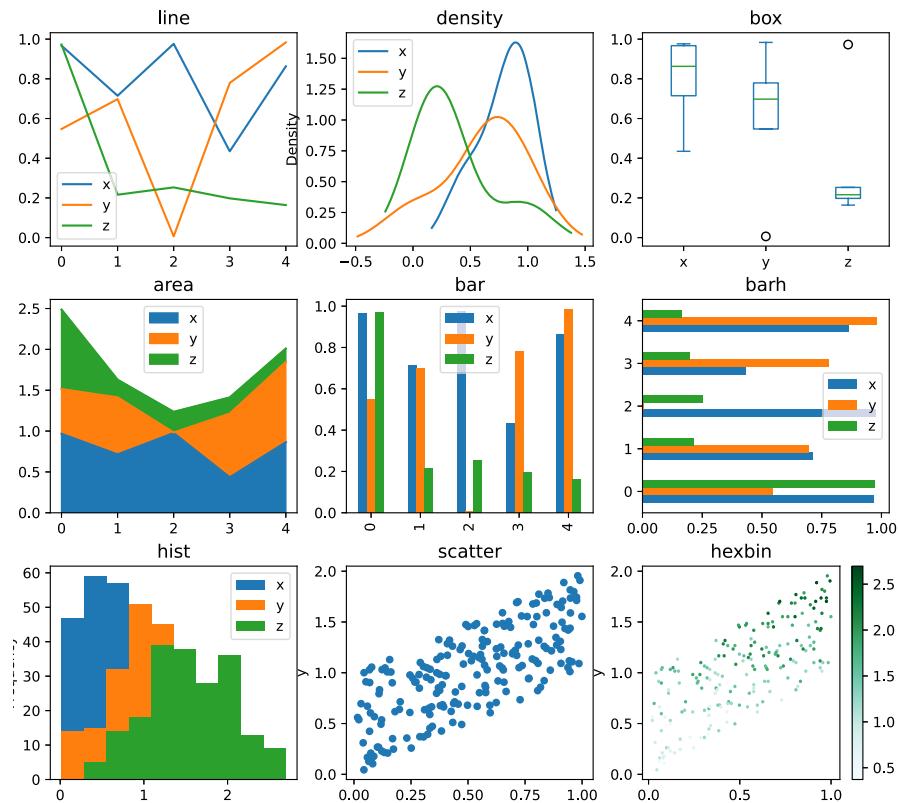
A cheat sheet for Matplotlib plotting functionalities.

Matplotlib APIs. Specifically, for `DataFrame` objects, it can automatically select data columns, and configure legends, tick labels, and other visualization elements.

The Pandas `.plot` method supports most of the plotting types provided by Matplotlib, such as `area`, `barh`, `density`, `hist`, `line`, `scatter`, `bar`, `box`, `hexbin`, `kde`, and `pie`. The usage of these plotting functions is straightforward. It just requires to access the specific plotting function `.plot.{function_name}` for a `DataFrame` object. For example, by implementing the following code through this API, we generate the plots shown in Fig. 3.4.

```
fig, axs = plt.subplots(3, 3, figsize=(9, 8))
df = pd.DataFrame(np.random.rand(5,3), columns=['x', 'y', 'z'])
for i, key in enumerate(['line', 'density', 'box', 'area', 'bar', 'barh']):
    getattr(df.plot, key)(ax=axs[i//3,i%3], title=key)

data = np.random.rand(200,3).cumsum(axis=1)
df = pd.DataFrame(data, columns=['x', 'y', 'z'])
df.plot.hist(ax=axs[2,0], title='hist')
df.plot.scatter('x', 'y', ax=axs[2,1], title='scatter')
df.plot.hexbin('x', 'y', 'z', ax=axs[2,2], title='scatter')
```

**FIGURE 3.4**

Pandas plotting methods.

To customize a Pandas plot, some keyword arguments can be specified for the Pandas `.plot` method, including `label`, `style`, `xticks`, `yticks`, `xlim`, `ylim`, `title`, `alpha`, `kind`, `figsize`, etc. Additionally, to direct the output to a specific `Axes` object, the `ax` keyword argument can be utilized. This option is useful for adding further enhancement and customization to the Pandas plots. For instance,

```
In [3]: ax = plt.subplot(111)
        df.plot.bar(ax=ax)
        ax.legend(loc='upper left')
```

Besides the built-in plotting function in Pandas, there are several excellent open-source packages available for visualizing Pandas data. For instance, Seaborn [5] is a Matplotlib-based tool, tailored for Pandas data visualization. This library is specialized in post-processing Pandas data and creating informative statistical graphics.

3.3 Mayavi for 3D plotting

Matplotlib provides 3D plotting capabilities. However, its functionalities for 3D plotting are somewhat limited. When dealing with large 3D plots, such as a protein molecule, you might experience significant slowdowns in Matplotlib. To effectively visualize 3D objects, we should consider alternative tools that are specifically designed for 3D plotting.

To customize 3D visualizations in Python, popular toolkits include Mayavi [6], PyVista [7], VisPy [8], and Scikit-image [9]. Among these toolkits, Mayavi is particularly suited for visualizing scientific data. It offers a user experience close to the `pylab` module of Matplotlib. If you are already familiar with Matplotlib, transitioning to Mayavi for 3D visualization is not difficult.

The installation process for Mayavi is more involved than that for typical Python packages. Mayavi relies on the Python QT libraries. According to the official installation instructions, the following command can be executed to install Mayavi and PyQt5:

```
$ pip install mayavi PyQt5
```

However, this command installs only the Python wrapper for the QT libraries. To enable the full visualization functionality, it is necessary to separately install dependent libraries, including the Visualization Toolkit (VTK) and the GUI (graphical user interface) libraries QtCore, QtGui, QtSvg, and QtWidgets. These libraries can be conveniently installed using the package managers provided by the operating systems. For example, on Ubuntu distributions, you can use the following command:

```
$ sudo apt install -y libqt5core5a libqt5gui5 libqt5svg5 libqt5widgets5 \
    vtk9
```

If you are running a Conda environment, Mayavi, PyQt5, and the dependent QT libraries should be installed through Conda packages.

To utilize Mayavi within IPython or Jupyter Notebooks, additional configuration steps may be required [10]. Within the IPython shell, you can use the IPython magic `%gui qt` to change the GUI backend to QT.¹ To leverage Mayavi within Jupyter Notebook, it may be necessary to install the latest version of `node.js`, as well as the following IPython extensions for Jupyter Notebook.

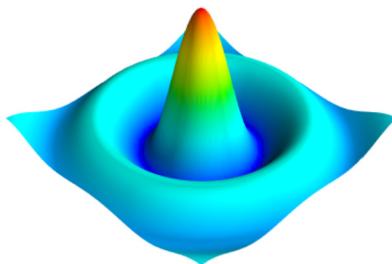
```
$ jupyter-labextension enable widgetsnbextension
$ jupyter-labextension enable ipyevents
```

To verify the installation of Mayavi, we can run the built-in tests provided by the library:

¹ QT seems working more stable than other GUI backends on Ubuntu machine.

```
In [1]: %gui qt

In [2]: from mayavi import mlab
        mlab.test_mesh()
```

**FIGURE 3.5**

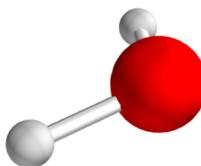
An example of surface plotting with `mlab.surf`.

`mayavi.mlab` can render various 3D plots, including surfaces, contours, points, lines, bar charts, and vector fields. Let's first examine a simple surface visualization using the `mlab.surf` function. The `mlab.surf` function is designed to visualize the values of a 2D array in a carpet plot, where each value on the z-axis corresponds to a data point on the 2D plane. The script below illustrates the steps for data preparation and 3D image plotting. The output is visualized in Fig. 3.5.

```
from mayavi import mlab
x, y = np.mgrid[-10:10:100j, -10:10:100j]          # (1)
r = np.sqrt(x**2 + y**2)
z = np.sin(r)/r

mlab.clf() # clear the current figure
mlab.surf(x, y, z, warp_scale='auto')
mlab.show()
```

The `np.mgrid` function is frequently used in Mayavi 3D visualization. It can create dense, multi-dimensional meshgrids. For instance, the statement in line (1) generates a uniform 100×100 mesh of two-dimensional points. These points are readily applicable in Mayavi functions. `np.mgrid` employs the indexing syntax `([])` along with slicing to specify the uniform grids. The slicing `start:stop:step` defines the start point and the end point. The `step` parameter can be either a real number, which indicates the spacing between points, or an imaginary number, which denotes the number of intervals between the start and end points. For real-valued `step`, the slicing works like `np.arange(start, stop, step)`. For an imaginary `step`, it is equivalent to `np.linspace(start, stop, step.imag)`.

**FIGURE 3.6**

Molecule visualized by Mayavi.

It is also feasible to use Mayavi to visualize complex molecular properties, such as electron density distributions and electrostatic potential surfaces. Let's examine the molecule represented in a Pandas DataFrame, where 'Q' denotes the effective charge associated with each atom:

```
mole = pd.DataFrame({
    'symbol': ['O', 'H', 'H'],
    'x': [0.5009, 1.4049, -0.0934],
    'y': [2.7238, 2.4390, 1.9786],
    'z': [0.8464, 0.6938, 0.7321],
    'Q': [-0.64, 0.32, 0.32],
})
```

We can start with the `mlab.points3d` and `mlab.plot3d` functions to draw the atoms and bonds in the molecule. The script below generates a view of the molecule as shown in Fig. 3.6:

```
R = (1, 0, 0)
G = (0, 1, 0)
B = (0, 0, 1)
W = (1, 1, 1)
K = (0, 0, 0)

color_map = pd.Series({
    'H': (.8, .8, .8),
    'C': (.6, .6, .6),
    'N': B,
    'O': R,
})

covalent_radius = pd.Series({
    'H': 0.31,
    'C': 0.73,
    'N': 0.71,
    'O': 0.66,
```

```

        )

def plot_mole(mole):
    geom = mole.loc[:, 'x':'z']
    for symbol, xyz in geom.groupby(mole.symbol):
        x, y, z = xyz.to_numpy().T
        mlab.points3d(x, y, z, scale_factor=covalent_radius[symbol],
                      resolution=30, color=color_map[symbol])
    # Bonds between atoms
    geom = mole.loc[:, 'x':'z'].to_numpy()
    radii = covalent_radius[mole.symbol].to_numpy()
    dist = np.linalg.norm(geom[:, None] - geom, axis=2)
    conn = dist < radii[:, None] + radii
    for i, j in np.argwhere(conn):
        if i < j:
            xyz = geom[[i, j]]
            mlab.plot3d(xyz[:, 0], xyz[:, 1], xyz[:, 2], tube_radius=0.06)

plot_mole(mole)

```

In this plotting script, we have employed the following techniques:

- The `mlab.points3d` function is used to plot spheres as atoms. This function can handle the coordinates of multiple points and treat them as a single object for rendering.
- We use different colors and radii to distinguish different types of atoms. Unlike Matplotlib which offers various predefined color schemes, there is no color name alias in Mayavi. Colors are specified by a three-element tuple, with each represents the fraction of RGB in each color channel.
- We utilize `mlab.plot3d` to draw bonds between atoms. To determine the covalent bonds, we consider the distance between atoms and compare it to the sum of their corresponding covalent radii.

Let's proceed to visualize the contours of the electrostatic potential in a molecule. Accurate evaluation of the electrostatic potential requires the program developed in Chapter 16. For simplicity, we employed a simplified model to represent the electrostatic potential in 3D space. In this model, the effective charge of an atom is approximated by a Gaussian distribution, which decays with respect to the distance s from the center of the atom:

$$\rho(s) = QN(\xi)e^{-(s/\xi)^2}. \quad (3.1)$$

Q is the effective charge and $N(\xi)$ is a normalization factor for the Gaussian distribution $e^{-(s/\xi)^2}$. The parameter ξ controls the spread of the Gaussian distribution, which we characterize with covalent radius. We can derive the electrostatic potential

$V(r)$ of the Gaussian-distributed charge by evaluating the integral:

$$V(r) = \int \frac{\rho(s)}{|\mathbf{s} - \mathbf{r}|} ds^2 \sin(\phi) d\phi d\theta = \frac{Q}{r} \operatorname{erf}\left(\frac{r}{\zeta}\right). \quad (3.2)$$

By summing up the contributions from all atoms in a molecule, we can derive a Python function to calculate the potential on the specified grids.

```

def eval_potential(mole, grids):
    assert grids.shape[0] == 3 # the x, y, z components
    xyz = mole.loc[:, 'x':'z'].to_numpy()
    r = np.linalg.norm(grids.reshape(3,-1) - xyz[:, :, None], axis=1)
    radii = covalent_radius[mole.symbol].to_numpy()
    v = np.einsum('q,qv->v', mole.Q, erf(r/radii[:, None])) / r
    return v.reshape(grids.shape[1:])

```

We then use the `mlab.contour3d` function to visualize the contour of the electrostatic potential. This function requires to sample the potential using three-dimensional mesh in space. We place the molecule at the center of a $4 \times 4 \times 4$ box and invoke the `np.mgrid` function to generate uniform mesh grids inside this box. This uniform sampling scheme is essential for the `mlab.contour3d` function to generate smooth contours. The grids are passed into the `eval_potential` function to evaluate the potential values. It is important to ensure that the output shape matches the shape of the grids. Additionally, both the coordinates of the sampling grids and the potential values should be structured as three-dimensional arrays. By default, the `mlab.contour3d` function generates five contour surfaces. To control which surfaces to plot, we can supply a list of values to the `contour` keyword argument to select surfaces. The contours for electrostatic potential are shown in Fig. 3.7.

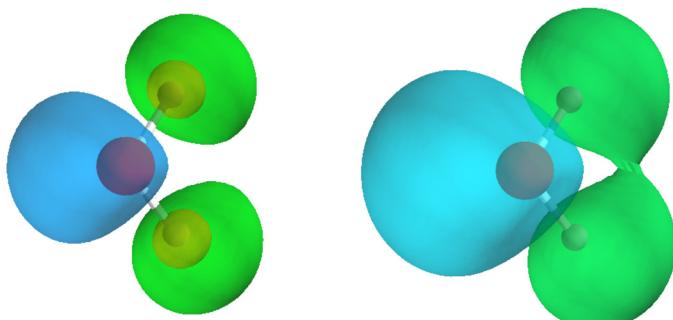
```

v = fn(mole, grids)
mlab.contour3d(grids[0], grids[1], grids[2], v,
               contours=contours, transparent=True)

mlab.view(75, 0, 7)
mlab.show()

view_contour(mole, eval_potential)
view_contour(mole, eval_potential, [-.1, .1])

```

**FIGURE 3.7**

Visualizing electrostatic potential in contours.

Mayavi supports to visualize the potential value on a given surface. This visualization task can be achieved using the `mlab.mesh` function. Unlike `mlab.surf`, which is limited to 2D meshes on a plane, `mlab.mesh` allows us to use a 2D mesh in 3D space to represent the surface. Each point on this mesh is defined by its x, y, and z Cartesian coordinates. For instance, here we create a spherical surface centered at the geometric center of the molecule. The radius of the sphere is set slightly larger than the distance from the center to the furthest atom.

We use `np.mgrid` to generate a uniform mesh for the sphere's polar and azimuthal angles. These mesh grids are then transformed into Cartesian coordinates. The potential values on these grids, along with their coordinates, are input into the `mlab.mesh` function.

```

def view_sph_surface(mole, fn, contours=5):
    mlab.figure(1, bgcolor=W, fgcolor=K, size=(500, 500))
    mlab.clf() # Clear figure
    plot_mole(mole)

    xyz = mole.loc[:, 'x':'z'].to_numpy()
    xyz_center = xyz.mean(axis=0)

```

```

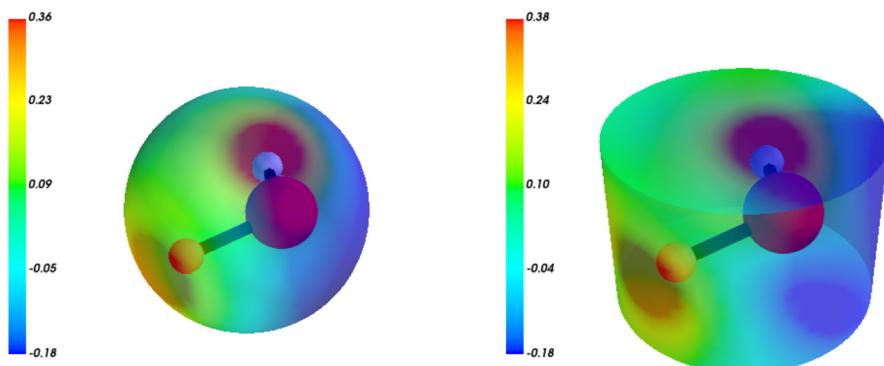
# Create a sphere surface
radii = covalent_radius[mole.symbol].to_numpy()
r = (np.linalg.norm(xyz - xyz_center, axis=1) + radii).max()
phi, theta = np.mgrid[0:np.pi:80j, 0:2*np.pi:80j]
x = r * np.sin(phi) * np.cos(theta)
y = r * np.sin(phi) * np.sin(theta)
z = r * np.cos(phi)
# The grids of the surface in Cartesian coordinates representation.
grids = np.array([x,y,z]) + xyz_center[:,None,None]

v = fn(mole, grids)
surface = mlab.mesh(grids[0], grids[1], grids[2],
                     scalars=v, transparent=True)
mlab.scalarbar(surface, orientation='vertical', label_fmt='%.2f',
                nb_labels=5)

mlab.view(75, 0, 7)
mlab.show()

view_sph_surface(mole, eval_potential)

```

**FIGURE 3.8**

Strength of electrostatic potential on spherical and cylindrical surface.

As an additional example, we construct a 2D mesh for the surface of a cylinder. This 2D mesh consists of three parts: a bottom plate, a side wall, and a top plate. For the two plates, we generate meshes in terms of the radial distance and azimuthal angle. For the side wall, the meshes are created by considering the azimuthal angle and the height. The mesh coordinates are demonstrated in the code below.

```

nx, ny = resolution
r, z = 1.1, 0.7
h, theta = np.mgrid[-z:z:complex(0, nx), 0:2*np.pi:complex(0, ny)]
u, theta = np.mgrid[0:r:complex(0, nx), 0:2*np.pi:complex(0, ny)]
grids = np.concatenate([
    # top plate
    [u * np.cos(theta), u * np.sin(theta), np.full((nx, ny), -z)],
    # side wall
    [r * np.cos(theta), r * np.sin(theta), h],
    # bottom plate
    [u * np.cos(theta), u * np.sin(theta), np.full((nx, ny), z)],
], axis=1) + xyz_center[:,None,None]

```

The spherical and cylindrical surfaces, along with the potential values on them, are displayed in Fig. 3.8. These visualizations can provide information such as the directions from which another molecule may approach the current molecule.

For a general surface in 3D space, creating a 2D surface mesh can be a difficult task. In such situations, we may have to utilize triangle meshes to define the surface, then invoke the `mlab.triangular_mesh` function to render the surface. The process of generating a suitable triangle mesh for a 3D object is beyond the scope of this book. We will not discuss in detail here.

From these examples, you may notice the different coding style between Mayavi and matplotlib. Matplotlib maps the plotting process to object-oriented tasks, where various instances are used to manage different elements in a plot. We can accomplish the plotting task by creating and combining these instances. To modify the appearance of a plot, we just need to access the instances and adjust their properties. The `mlab` module is not entirely object-oriented. It works on a default canvas to which the visualization elements are added. Visualizing options are configured in the global namespace. Finally, `mlab` invokes the underlying visualization engines to perform rendering. Its usage is more imperative, similar to the legacy `pylab` interface of the Matplotlib library.

In the gallery provided by Mayavi documentation, you may find some examples that utilize the `mlab.pipeline` functions to create visualization elements. A pipeline in Mayavi refers to a series of data processing and visualization components that collaborate to produce a 3D visualization [11]. A pipeline typically consists of the following components:

- *Data Source*, which loads raw data, and stores and validates the raw data.
- *Data Filters*, which perform operations such smoothing, shaping, and screening on the raw data.
- *Module*, which provides the specific visualization model, like surface plots, volume rendering, contours, etc.
- *Scene*, which is the stage where the visualization is rendered.

In the interactive window as shown in Fig. 3.9, the pipeline components are displayed in a hierarchical tree structure. To modify the visualization effects, we can inspect the properties of each component and adjust their configurations in the interactive window.

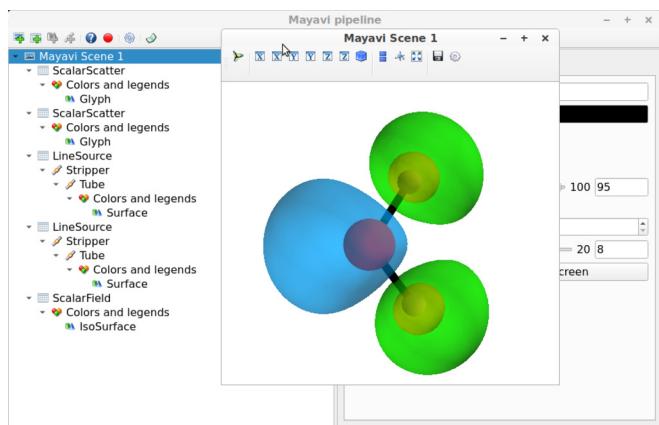


FIGURE 3.9

The layout and pipeline of a Mayavi scene.

The `mlab` 3D plotting functions, such as `mlab.plot3d`, `mlab.surf`, `mlab.mesh`, and `mlab.contour3d`, are pipeline classes that incorporate all the necessary intermediate components for particular visualization tasks. When you need a more detailed customization of the visualization effects, the `mlab.pipeline` functions offer the ability to tailor or adjust the pipeline components.

3.4 Quantum chemistry visualization

In the context of chemical systems, there are several dedicated tools for molecule visualization. Given a molecule structure file, such as XYZ, PDB (Protein Data Bank), MOL (Molecular Design Limited), or SDF (Structure Data File), many tools (such as PyMOL, Avogadro, py3Dmol) are readily available for visualizing the 3D structure of a molecule. When it comes to the realm of quantum chemistry, the focus of visualization moves to the electrons within a molecule, including their orbitals, density, and other quantum chemistry outcomes. Unfortunately, there are no Python libraries specialized for visualizing quantum chemistry results in 3D. One viable option is the Mayavi library. However, functionalities in Mayavi are relatively basic and might require significant configuration and coding effort to achieve a satisfactory display effect.

A more practical approach to visualizing quantum chemistry results is to save them in specialized file formats, such as Molden and Cube. Quantum chemistry vi-

sualization software, like VMD or Jmol, can then read these files to generate 3D displays. Therefore, the key step in this approach is the generation of Molden or Cube files. In this context, the Jinja templating technique is very useful to efficiently generate Molden and Cube files.

3.4.1 Jinja template

The Jinja templating process can be carried out in two primary steps:

1. Loading the template with the `jinja2.Template` function. Jinja offers various template loaders for importing templates from files. As a simple alternative, we can utilize the Python `open` function together with the `jinja2.Template` function to load template files.
2. Calling the `.render()` method with keyword arguments to render the template.

```
In [1]: import jinja2
        tpl = jinja2.Template('template_string {{ key }}')
        tpl.render(key=value)
Out[1]:
'template_string value'
```

Let's next examine the syntax of Jinja template [12]. There are various syntaxes one can use in the template string. However, in our use cases for Molden and Cube format, we only need fundamental syntaxes.

Expression

Expressions are denoted by the notation `{{ }}`. During the template rendering process, variables within these expressions are substituted by the Python objects supplied to the template. These expressions can be basic Python variables such as numbers, strings, lists, and dictionaries, as well as more complex expressions that require runtime evaluation. Examples of such complex expressions include mathematical expressions, accessing an attribute of an object, invoking a function, and utilizing the one-line if-else statement.

Control statements

Control Statements are denoted by the notation `% %`. Jinja supports several common control statements, such as if-else, for-loop, assignments and so forth. The if-else statement can be utilized to manage missing values. For example,

```
In [2]: jinja2.Template('''\
        {% if key %}{{ key }}{% endif %}
        {% if absent %}{{ absent }}{% endif %}''').render(key='a')
Out[2]:
'a\n'
```

The assignment statement, indicated by `set`, allows us to define temporary variables. These variables can be utilized as regular variables within Jinja expressions. For instance,

```
In [3]: jinja2.Template(''')
    {% set x, y, z = item.split() %}
    {{ z }} {{ y }} {{ x }}''').render(items='a b c')
Out[3]:
'\nc b a'
```

Whitespace control

Whitespace control is integrated with the expressions or control statements, such as `{%-`, `-%}`, `{-`, and `-}`). The minus sign in these template tags is used to eliminate leading and trailing whitespace, and blank lines that are introduced by control statements. For instance,

```
In [4]: jinja2.Template('''
    {% set x, y, z = item.split() %}
    {{ - z }} {{ y }} {{ - x }}''').render(items='a b c')
Out[4]:
'c b a'
```

3.4.2 Molden format

The Molden format is a standard file format for storing quantum chemistry information. This includes the molecular geometry, basis sets, electronic orbitals, and so forth. A common application of the Molden format is for the visualization of molecular orbitals. With appropriate configurations, the Molden format can be utilized to visualize custom orbitals.

The structure of the Molden format is well-documented and can be referenced in the online documentation https://www.theochem.ru.nl/molden/molden_format.html. Below, we will use the Jinja template to demonstrate the structure of the Molden format.

The Molden format comprises a series of *sections*, denoted by brackets, such as `[Molden Format]`, `[Atoms]`, `[GTO]`, etc. These sections organize the Molden file into distinct segments.

```
molden_tpl = jinja2.Template('''
[Molden Format]
Comment lines
[Atoms] ({{ unit or 'Ang' }})
{%- for element, atom_number, x, y, z in atoms %}
{{ element }} {{ loop.index }} {{ atom_number }} {{ x }} {{ y }} {{ z }}
{%- endfor %}
```

```
[GTO]
{%- for gto_shells in gtos %}
{{ loop.index }} 0
{{ gto_shells }}
{%- endfor %}

[5d]
[7f]
[M0]
{%- for mo in mos %}
{{ mo }}
{%- endfor %}
...)
```

The molecular geometry is detailed in the [Atoms] section of the Molden format. In this section, each line corresponds to an atom within the molecule.

To represent electronic orbitals, both the basis set information and the orbital coefficients should be supplied. The details of the basis sets are contained within the [GTO] section. Here, Molden format requires to specify the basis set information for each atom individually. Please note the {%- endfor %} notation in the template above. According to the Molden format specifications, an empty line must be inserted between the basis sets of different atoms. Therefore, the trailing whitespace in the Jinja code {%- for loop %} is intentionally preserved to fulfill this requirement.

The configuration of the basis sets is specified by a series of *shells*. The structure of each shell is outlined in the following `gto_shell_tpl` template. For regular Gaussian-type orbital (GTO) basis sets, their information can be obtained from the Basis Set Exchange (BSE) online database, which is accessible via the Python package `basis-set-exchange` [13,14].

```
from basis_set_exchange import get_basis
from basis_set_exchange.manip import uncontract_general

gto_shell_tpl = jinja2.Template(''\
{{ spdf_shell }} {{ len(exponents) }} 1.00
{%- for e_cs in zip(exponents, *coefficients) %}
{{ ' '.join(e_cs) }}
{%- endfor %}'')

def bse_to_gto_shell(shell):
    spdfg = 'spdfg'
    exponents = shell['exponents']
    coefficients = shell['coefficients']
    spdf_shell = ''.join(spdfg[l] for l in shell['angular_momentum'])
    return gto_shell_tpl.render(
        spdf_shell=spdf_shell, exponents=exponents,
        coefficients=coefficients, len=len, zip=zip)
```

```
def gto_session(basis_name, element):
    bse_data = get_basis(basis_name, element)
    bse_data = uncontract_general(bse_data)
    _, basis = bse_data['elements'].popitem()
    return '\n'.join([bse_to_gto_shell(shell)
                     for shell in basis['electron_shells']])
```

Following the [GTO] section, the [5d] and [7f] sections are optional. These sections specify the use of spherical D and F functions for the atomic basis functions.

Next, the electronic orbitals can be included in the [MO] section. As shown in the mo_tpl template below, for each orbital, the MO section allows for the specification of various properties, including symmetry, orbital energy, and occupancy, etc, then a list of orbital coefficients. When handling the orbital coefficients, please note the ordering of the D, F, and G functions. For instance, the spherical D functions are arranged as D0, D+1, D-1, D+2, D-2. For simplicity, we have omitted the reordering process for these orbitals.

```
mo_tpl = jinja2.Template('''\
Sym= {{ symmetry or 'A' }}
Ene= {{ energy or 0.0 }}
Spin= {{ spin or 'Alpha' }}
Occup= {{ occupancy or 0.0 }}
{%- for c in coefficients %}
{{ loop.index }} {{ c }}
{%- endfor %}

def mo_session(orbitals):
    return [mo_tpl.render(coefficients=c) for c in orbitals]
```

Putting all components together, we can create the render_molden function:

```
from basis_set_exchange.lut import element_Z_from_sym

def render_molden(elements, coordinates, basis_name, orbitals):
    x, y, z = coordinates.T
    atom_numbers = [element_Z_from_sym(ele) for ele in elements]
    atoms = list(zip(elements, atom_numbers, x, y, z))
    return molden_tpl.render(
        Z=element_Z_from_sym, atoms=atoms,
        gtos=[gto_session(basis_name, atom[0]) for atom in atoms],
        mos=mo_session(orbitals))
```

For instance, we can output all atomic orbitals of a molecule in a Molden file using the following code.

```

import pandas as pd
import pyscf
mole = pd.DataFrame({
    'symbol': ['O', 'H', 'H'],
    'x': [0.5009, 1.4049, -0.0934],
    'y': [2.7238, 2.4390, 1.9786],
    'z': [0.8464, 0.6938, 0.7321],
})
basis = 'sto-3g'
AOs = np.identity(5)
elements = mole.symbol
coordinates = mole.loc[:, 'x':'z'].to_numpy()
with open('demo.molden', 'w') as f:
    f.write(render_molden(elements, coordinates, basis, AOs))

```

3.4.3 Cube format

The Cube format originates from the Gaussian computational chemistry software. It has become the de facto standard for storing volumetric quantum chemistry data. It is well-suited for the detailed representation of the molecular orbital, electron density, and potential surface within a three-dimensional grid.

The structure of the Cube format is briefly described in the Gaussian manual. A more detailed explanation can be found in some online documentation [15,16]. Despite being a text-based format, the Cube format must follow a strict, predetermined format for data representation. A Cube file must include the origin of the volumetric region, the voxel information, the atoms in the molecule, and the details of the volumetric data. For the information on atoms and voxels, integers must be formatted as 5d, and floating-point numbers must be formatted as 12.6f. When storing the volumetric data, the numbers are formatted as 13.5E, with six numbers per line.

The following Jinja template demonstrates the structure of the Cube format.

```

cube_tpl = jinja2.Template('''

Comment line 1
Comment line 2
{{ '%5d' % len(atoms) }} {{ '%12.6f' % origin[0] }} {{ '%12.6f' % origin[1]
}} {{ '%12.6f' % origin[2] }}
%- set vx = voxel[0] %
%- set vy = voxel[1] %
%- set vz = voxel[2] %
{{ '%5d' % n_voxels[0] }} {{ '%12.6f' % vx[0] }} {{ '%12.6f' % vx[1] }} {{ '%12.6f' % vx[2] }}
{{ '%5d' % n_voxels[1] }} {{ '%12.6f' % vy[0] }} {{ '%12.6f' % vy[1] }} {{ '%12.6f' % vy[2] }}

```

```

{{ '%5d' % n_voxels[2] }} {{ '%12.6f' % vz[0] }} {{ '%12.6f' % vz[1] }} {{ '%12.6f' % vz[2] }}
% set charge = 0 %
% for atom_number, x, y, z in atoms -%
{{ '%5d' % atom_number }} {{ '%12.6f' % charge }} {{ '%12.6f' % x }} {{ '%12.6f' % y }} {{ '%12.6f' % z }}
% endfor -%
% for data in volumetric_data -%
{{ data }}
% endfor -%
''')

def render_cube(elements, coordinates, voxel, origin, data):
    atom_numbers = [element_Z_from_sym(ele) for ele in elements]
    x, y, z = coordinates.T
    atoms = list(zip(atom_numbers, x, y, z))
    n_voxels = data.shape
    formatted_data = []
    for ix in range(n_voxels[0]):
        for iy in range(n_voxels[1]):
            for iz in range(0, n_voxels[2], 6):
                # In each line, writes up to 6 floats in the 13.5E format
                formatted_data.append(
                    ''.join(f'{v:13.5E}' for v in data[ix,iy,iz:iz+6]))
    return cube_tpl.render(
        atoms=atoms, len=len,
        n_voxels=n_voxels, voxel=voxel, origin=origin,
        volumetric_data=formatted_data)

```

To visualize an electronic property, one can create a grid, calculate its values on the grids, and use this template to render the Cube file.

For example, we can create a Cube file to visualize a *p*-type atomic orbital as follows. The voxel grid is a cubic volume of $0.3 \times 0.3 \times 0.3$ Bohrs. 60 voxels are allocated along each of the *x*, *y*, and *z* axes. For simplicity, we have employed the quantum chemistry program PySCF [17,18] to evaluate the values of the *p* orbital on these grids. Please note the unit conversion applied to the molecular geometry. All distance data in the Cube file must be presented in Bohr.²

```

# NOTE: all units are in Bohr
boundary = [[-9., 9.],
            [-9., 9.],
            [-9., 9.]]

```

² The Gaussian package allows the use of Angstrom as a unit by setting some parameters to negative values. However, this can easily lead to errors and is strongly discouraged.

```

        [-9., 9.]]
mesh = [60, 60, 60]
mgrids = np.mgrid[[slice(r[0], r[1], m*1j)
                    for r, m in zip(boundary, mesh)]]
grids = mgrids.reshape(3, -1).T
origin = mgrids[:,0,0,0]
voxel = np.array([mgrids[:,1,0,0] - origin,
                  mgrids[:,0,1,0] - origin,
                  mgrids[:,0,0,1] - origin])

# AO values on given grids
basis = 'sto-3g'
mol = pyscf.M(atom=list(mole.to_numpy()), basis=basis, verbose=0)
ao = mol.eval_gto('GT0val', grids)
ao_2pz = ao[:,4].reshape(mesh)

elements = mole.symbol
coordinates = mole.loc[:, 'x':'z'].to_numpy() / 0.529177249
with open('demo.cub', 'w') as f:
    f.write(render_cube(elements, coordinates, voxel, origin, ao_2pz))

```

Summary

In this chapter, we have explored 2D and 3D visualization techniques for scientific data visualization, including the use of Matplotlib for 2D visualization and Mayavi for 3D visualization. For visualizing quantum chemistry data, we turn to external software and utilize Molden and Cube files to exchange data. We discussed the use of Jinja template technology for generating Molden and Cube files.

Our discussions are limited to the technical aspects of data visualization. Beyond the visualization techniques, a more significant challenge lies in creating informative visualizations that effectively convey the story behind the data. On this topic, the book *Fundamentals of Data Visualization* [19] by Claus O. Wilke is an excellent resource.

References

- [1] J. VanderPlas, Python Data Science Handbook: Essential Tools for Working with Data, O'Reilly Media, Inc, Sebastopol, CA, 2016.
- [2] J. Hunter, D. Dale, E. Firing, M. Droettboom, the Matplotlib development team, Matplotlib examples - list of named colors, https://matplotlib.org/stable/gallery/color/named_colors.html, 2024.

- [3] J. Hunter, D. Dale, E. Firing, M. Droettboom, the Matplotlib development team, Matplotlib users guide - specifying colors, <https://matplotlib.org/stable/users/explain/colors/colors.html>, 2024.
- [4] J. Hunter, D. Dale, E. Firing, M. Droettboom, the Matplotlib development team, Matplotlib examples - color demo, https://matplotlib.org/stable/gallery/color/color_demo.html, 2024.
- [5] M.L. Waskom, seaborn: statistical data visualization, *The Journal of Open Source Software* 6 (60) (2021) 3021, <https://doi.org/10.21105/joss.03021>.
- [6] P. Ramachandran, G. Varoquaux, Mayavi: 3d visualization of scientific data, *Computing in Science & Engineering* 13 (2) (2011) 40–51, <https://doi.org/10.1109/MCSE.2011.35>.
- [7] C.B. Sullivan, A. Kaszynski, PyVista: 3d plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK), *The Journal of Open Source Software* 4 (37) (2019) 1450, <https://doi.org/10.21105/joss.01450>.
- [8] Vispy developers, Vispy: interactive scientific visualization in Python, <https://vispy.org/>, 2024.
- [9] S. van der Walt, J.L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J.D. Warner, N. Yager, E. Gouillart, T. Yu, the scikit-image contributors, scikit-image: image processing in Python, *PeerJ* 2 (2014) e453, <https://doi.org/10.7717/peerj.453>.
- [10] Enthought, Mayavi on Jupyter, https://github.com/enthought/mayavi/blob/master/examples/mayavi/mayavi_jupyter.ipynb, 2024.
- [11] Enthought, Mayavi documentation - objects populating the mayavi pipeline, https://docs.enthought.com/mayavi/mayavi/mayavi_objects.html, 2024.
- [12] Pallets, Jinja2 documentation - template designer documentation, <https://jinja.palletsprojects.com/en/latest/templates/>, 2024.
- [13] K.L. Schuchardt, B.T. Didier, T. Elsethagen, L. Sun, V. Gurumoorthi, J. Chase, J. Li, T.L. Windus, Basis set exchange: a community database for computational sciences, *Journal of Chemical Information and Modeling* 47 (3) (2007) 1045–1052, <https://doi.org/10.1021/ci600510j>, pMID: 17428029.
- [14] B.P. Pritchard, D. Altarawy, B. Didier, T.D. Gibson, T.L. Windus, New basis set exchange: an open, up-to-date resource for the molecular sciences community, *Journal of Chemical Information and Modeling* 59 (11) (2019) 4814–4820, <https://doi.org/10.1021/acs.jcim.9b00725>, pMID: 31600445.
- [15] P. Bourke, Gaussian cube files, <https://paulbourke.net/dataformats/cube/>, 2024.
- [16] B. Skinn, Gaussian cube file format, <https://h5cube-spec.readthedocs.io/en/latest/cubeformat.html>, 2024.
- [17] The PySCF Developers, Quantum chemistry with Python, <https://pyscf.org/>, 2024.
- [18] Q. Sun, X. Zhang, S. Banerjee, P. Bao, M. Barbry, N.S. Blunt, N.A. Bogdanov, G.H. Booth, J. Chen, Z.-H. Cui, J.J. Eriksen, Y. Gao, S. Guo, J. Hermann, M.R. Hermes, K. Koh, P. Koval, S. Lehtola, Z. Li, J. Liu, N. Mardirossian, J.D. McClain, M. Motta, B. Mussard, H.Q. Pham, A. Pulkin, W. Purwanto, P.J. Robinson, E. Ronca, E.R. Say-futyarova, M. Scheurer, H.F. Schurkus, J.E.T. Smith, C. Sun, S.-N. Sun, S. Upadhyay, L.K. Wagner, X. Wang, A. White, J.D. Whitfield, M.J. Williamson, S. Wouters, J. Yang, J.M. Yu, T. Zhu, T.C. Berkelbach, S. Sharma, A.Y. Sokolov, G.K.-L. Chan, Recent developments in the PySCF program package, *Journal of Chemical Physics* 153 (2) (2020) 024109, <https://doi.org/10.1063/5.0006074>, https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0006074/16722275/024109_1_online.pdf.
- [19] C. Wilke, Fundamentals of data visualization, <https://clauswilke.com/dataviz/>, 2024.

Scientific computing tools

4

Scientific computing is a broad field, with various tools being employed by different areas. Even when the same tools are employed, their applications and focuses can vary significantly across different fields. We do not intend to discuss the fundamentals of general scientific computing tools, such as SciPy. The fundamentals of the scientific computing tools is already well-covered in various literatures, such as the book *Numerical Python - Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*, by Robert Johansson.

In this chapter, we will explore the scientific computing tools specialized for quantum chemistry programs and applications, covering the following topics:

- Linear algebra tools.
- Tools for sparse matrices since the Hamiltonian in quantum chemistry are essentially large and sparse matrices.
- Tensor contraction. This is the most frequently used tool in many-body methods. It strongly impacts the performance of quantum chemistry applications.
- Fourier transform. When dealing with problems involving periodic boundary conditions, a necessary computation tool is the Fourier transform.

Computational performance is always a critical concern in scientific computing. When evaluating these tools, we will consider them from multiple perspectives in terms of performance, including:

- What are the factors that influence their efficiency?
- How to use them properly to maximize the performance?
- Which candidate to use if multiple options are available?

4.1 Linear algebra

When linear algebra operations are required in a scientific computing application, both the `numpy.linalg` and `scipy.linalg` modules can be considered. What are the differences between the two modules? When should we choose one over the other? We will explore these questions in this section.

The `numpy.linalg` module provides the basic linear algebra operations, including the linear equation solver (`solve`, `lstsq`), matrix factorization (`qr`, `cholesky`, `eig`, `eigh`, `svd`), matrix inversion (`inv`, `pinv`), functions for matrix determinant (`det`), matrix or

vector norm (`norm`), matrix condition number (`cond`), an integer power of a square matrix (`matrix_power`), and so forth. The SciPy linear algebra module, `scipy.linalg`, incorporates most of the `numpy.linalg` functionalities. Additionally, it offers extra features and improved performance optimization.

When developing a scientific computing program, `numpy.linalg` can be a starting point to achieve basic functionalities. If better performance and advanced features are desired, we can replace the `numpy.linalg` functions with the corresponding ones in `scipy.linalg`.

Enhancements in `scipy.linalg` involve the following categories:

- *Additional linear equation solvers.* Unlike NumPy, which provides only a general solver `np.linalg.solve` for all kinds of matrices, `scipy.linalg` allows us to choose different solvers based on the characteristics of the matrix, so as to improve performance or numerical stability. For examples, `scipy.linalg` offers `cho_solve()` for solving linear equations using Cholesky factorization, the `lu_solve` solver with LU factorization, `solve_triangular()` specialized for triangular matrix, `solve_sylvester()` for the equation $AX + XB = Q$, `solve_banded()` for non-square matrices. The tailored solvers are prefixed with `solve_`, which can be easily identified.
- *Additional functions for matrix factorization and decomposition.* Examples include `lu()` for LU decomposition, `rq()` for the $A = RQ$ decomposition, `hessenberg()` for Hessenberg decomposition, and `schur()` for Schur decomposition. Certain factorization functions can utilize the special characteristics of matrices, such as `cholesky_banded()` for Cholesky decomposition of banded matrices, `eig_banded()` for eigenvalue decomposition of banded matrices, and `eigh_tridiagonal()` for the eigenvalue problem for a real symmetric tri-diagonal matrix.
- *Mathematical functions for matrices*, such as `expm()`, `logm()`, `sqrtm()`, `cosm()`, `sinm()`, `tanm()`.
- *Special matrices and more array constructors*, such as `block_diag()` for creating a block diagonal matrix, `companion()` for creating a companion matrix, `dft()` for creating a discrete Fourier transform (DFT) matrix, defined as $A_{mn} = \exp\left(-\frac{2i\pi mn}{N}\right)$.
- *Low-level interfaces to BLAS and LAPACK libraries.* BLAS and LAPACK functions, such as `dsyrk()` and `dsyev()`, can be accessed and utilized with NumPy arrays. These interfaces are available through the `scipy.linalg.blas` and `scipy.linalg.lapack` modules.

In each category, we only list some of the representative functions. For a comprehensive list of functions for each category, please refer to the SciPy documentation [1].

For functions available in `numpy.linalg`, SciPy inherits their APIs and introduces extra parameters to provide additional functionalities. These additional parameters are primarily related to memory management strategies. For instance, the `overwrite_a` option, appearing in most functions, enables the overwriting of the input

matrix to reduce memory footprint. Also, there are parameters designed to enhance performance. For example, the `check_finite` option allows for bypassing the value validation checks on the input matrix. Besides these common parameters, in the following, we briefly summarize other specialized improvements that SciPy has made to the `numpy.linalg` functions:

- For the linear equation $Ax = b$, the `scipy.linalg.solve` function introduces an option `assume_a` to specify whether the matrix is symmetric, hermitian, or positive-definite. Depending on the matrix type, different LAPACK routines will be invoked.
- The `qr` function in SciPy offers an option `pivoting=True` to activate pivoting with rank-revealing QR decomposition [2]. This option allows the library to use a numerically more stable routine for processing ill-conditioned matrices. Please note that the default mode for `qr` is different between SciPy and NumPy. In SciPy, the `Q` matrix is a square matrix with extended dimensions, whereas in NumPy, `Q` is a matrix with reduced columns.
- The `cholesky` function of SciPy executes the upper-triangular Cholesky factorization by default. To align it with the behavior in NumPy, the keyword argument `lower=False` should be specified.
- The `eigh` function in SciPy can solve generalized eigenvalue problems, including

$$\begin{aligned} Av &= \lambda Bv, \\ ABv &= \lambda v, \text{ or} \\ BA v &= \lambda v. \end{aligned}$$

In addition, it offers the `subset_by_value` and `subset_by_index` options to compute only a selected subset of eigenvalues.

- For the eigenvalue problem of general matrices, the `eig` function in SciPy can compute both left and right eigenvectors, while the NumPy version only computes right eigenvectors. Additionally, the SciPy version supports the generalized eigenvalue problem $Av = \lambda Bv$.

4.2 Sparse matrices

Sparse matrices, i.e., matrices dominated by elements that are zero, are common in quantum chemistry programs and many scientific computing applications. To reduce memory usage for sparse matrices, one can store only the non-zero values along with the necessary details about their locations. However, the discontinuous storage of sparse matrices makes them more challenging to handle than dense matrices.

The primary challenges associated with sparse matrices are the storage format and the required operations to manipulate them. To address these challenges, the `scipy.sparse` module provides a comprehensive suite of functionalities for managing sparse matrices. Regarding the linear algebra algorithms, `numpy.linalg` and

`scipy.linalg` are designed for dense matrices. They are not applicable to sparse matrices. Linear algebra operations on sparse matrices can be performed using the functions in `scipy.sparse.linalg`.

4.2.1 Storage formats

SciPy supports various formats for storing sparse matrices, each with its suitable use cases. Formats like CSR, CSC, and BSR, are optimized for efficient access to matrix elements and executing matrix-vector operations. Other formats are more appropriate for constructing matrices.

- CSR (Compressed Sparse Row) format stores the indices of non-zero elements for each row. This format utilizes a NumPy array (accessible via the `.data` attribute) to store non-zero elements, another array (associated with the `.indices` attribute) for the indices, and a third array (via the `.indptr` attribute) to store the row pointer for the indices. Specifically, the column indices for row i are stored in `indices[indptr[i]:indptr[i+1]]`, and their corresponding values are stored in `data[indptr[i]:indptr[i+1]]`. This format is convenient for accessing elements by row and efficient for matrix-vector multiplications. However, inserting new elements into the matrix is highly inefficient. Each time a new element is added, the index and data arrays are reconstructed, typically through the intermediate in COO format.
- CSC (Compressed Sparse Column) format is similar to CSR but is designed for column-wise storage.
- BSR (Block Sparse Row) format is similar to CSR, but it accommodates blocks of arrays rather than scalar variables. This format imposes several constraints on the array blocks: All array blocks within BSR storage have the same block size. Each dimension of the array should have a size that is a multiple of the block size. These constraints restrict the applicability of BSR. However, when a sparse matrix is compatible with this format, BSR is considered more efficient than other formats for arithmetic operations and matrix-vector multiplications.
- COO (Coordinate) format stores the row index, column index, and value of each non-zero element separately in lists. This format does not support direct access to matrix elements using the array indexing syntax (e.g., `a[i,j]`). In `scipy.sparse`, this format is often used as an intermediate format for converting between sparse storage formats.
- DOK (Dictionary of Keys) format stores non-zero elements in a dictionary, using (row, column) tuples as keys. This format is similar to the COO format, but it supports accessing an individual matrix element. If we need to construct sparse matrices incrementally, this format is a suitable option.
- LIL (List of Lists) format stores data and column indices in separate lists for each row. This format is closely related to the CSR format. Conversions between LIL and CSR formats are efficient. The LIL format supports accessing individual elements. When inserting new elements to the sparse matrix, only the data and index lists for the relevant rows need to be updated.

- DIA (Diagonal) format stores the diagonal elements and their offsets from the main diagonal. It does not support the access or modification of individual elements. This format is not suitable for representing general sparse matrices. It is primarily used in specialized scenarios, such as storing band matrices.

`scipy.sparse` provides dedicated classes for each storage format, all inherited from the `scipy.sparse.spmatrix` class. Each format can be instantiated through its class, using either `scipy.sparse.{format}_array` or `scipy.sparse.{format}_matrix`, where the placeholder `{format}` can be replaced with any of the formats mentioned above. For example, to instantiate a CSR matrix, we can construct it from a dense matrix,

```
scipy.sparse.csr_array(np.eye(4))
```

or from another sparse matrix,

```
scipy.sparse.csr_array(scipy.sparse.eye(4, format='coo'))
```

Given a list of data along with their row and column indices, we can create a sparse matrix as follows:

```
row_indices = np.arange(4)
col_indices = np.arange(4)
scipy.sparse.csr_array(np.ones(4), (row_indices, col_indices))
```

Alternatively, by directly specifying the CSR indices and `indptr` along with the data array, we can directly create a CSR sparse array:

```
data = np.ones(4)
indices = np.array([0,1,2,3])
indptr = np.array([0,1,2,3,4])
scipy.sparse.csr_array((data, indices, indptr), shape=(4, 4))
```

Conversion between different storage formats can be accomplished in two methods. One is to use the `.to{format}()` method of each sparse matrix class. The other is the class instantiation functions `{format}_array` or `{format}_matrix`, as previously discussed. For example, the code snippet below illustrates the conversion from COO format to other formats.

```
a_coo = scipy.sparse.coo_array(np.eye(4))
# convert to dense numpy array
a_coo.toarray()
# convert to CSC format
a_coo.tocsc()
# convert to BSR format
a_coo.tobsr(blocksize=(2,2))
```

Table 4.1 Intermediate formats required for conversion between two sparse matrices. The notation “-” indicates that one format can be directly converted into another without the need for intermediate formats.

To \ From	CSR	CSC	BSR	COO	DOK	LIL	DIA	ndarray
CSR	-	-	-	-	COO	-	COO	-
CSC	-	-	CSR	-	COO	CSR	COO	-
BSR	-	CSR	-	-	COO	CSR	COO	COO
COO	-	-	CSR	-	-	CSR	-	-
DOK	COO	COO	CSR	-	-	CSR	COO	COO
LIL	-	CSR	CSR	CSR	COO	-	COO	-
DIA	COO	-	CSR	-	COO	CSR	-	COO

Some conversions can be performed directly, while others may rely on an intermediate format. This is an important factor to consider when selecting a sparse format, since format conversion can implicitly introduce overhead. Table 4.1 provides a summary of the intermediate storage format required for conversion between sparse matrix formats. The need for an intermediate format means additional overheads in terms of memory usage and computational effort. However, it should be noted that the capability of direct conversion does not necessarily lead to fast conversion performance. It simply indicates that there is no additional overhead from generating intermediate formats. Certain conversions, especially those between CSR, CSC, BSR, and COO formats, are much faster than others.

The API for sparse matrices is similar to that of NumPy arrays. This allows us to use similar code to manipulate both the sparse matrices and the NumPy arrays. Attributes like `dtype`, `shape`, and `ndim` in NumPy arrays describe the structure of an array. These attributes are also present in sparse matrices. However, a notable difference exists in the `.size` attribute of a sparse matrix. This attribute indicates the number of non-zero elements, corresponding to the `.nnz` attribute, rather than the total number of elements in the matrix. This is a key difference in the convention of attributes between dense arrays and sparse matrices.

Several attributes and methods used for manipulating NumPy arrays are also applicable in sparse matrices, such as `.T`, `.real`, `.imag`, `.reshape()`, `.diagonal()`, and `.dot()`, etc. Their usage is similar to that in dense arrays. Please note that some methods may perform a format conversion implicitly, which can lead to additional computational costs. Furthermore, this may result in outputs being provided in a different format. For instance, invoking the `reshape` method on a CSR matrix will produce a COO matrix. If you need to work with the matrix in a CSR format, you might have to convert the output back to a CSR matrix to ensure compatibility and performance.

In the `.dot()` method for matrix-vector or matrix-matrix operations, efficient implementations are available in the CSR, CSC, BSR, and COO formats. Among these, the `dot` operations for CSR and BSR formats are more efficient than the other two.

Other formats are converted to CSR format before applying the actual `dot` operation. This format conversion can sometimes impact performance. If the `.dot()` method needs to be executed multiple times, to avoid the repeated format conversion, it is advisable to manually convert to the CSR format once.

Sparse arrays require some additional attributes to effectively manage their non-zero elements. The names and meanings of these additional attributes can be found in the doc-string of each `{format}_array` class. By utilizing these attributes, we can efficiently iterate over the non-zero elements of a sparse array. For example, the non-zero elements of COO arrays can be accessed through its `.data`, `.row`, and `.col` attributes:

```
a = scipy.sparse.coo_array(np.eye(3))
for row, col, value in zip(a.row, a.col, a.data):
    print(f'({row}, {col}): {value}'')
```

The non-zero elements of CSR can be accessed via the attributes `.indptr`, `.indices`, and `.data`:

```
for row in range(csr_matrix.shape[0]):
    for ptr in range(csr_matrix.indptr[row], csr_matrix.indptr[row+1]):
        col = csr_matrix.indices[ptr]
        value = csr_matrix.data[ptr]
        print(f'({row}, {col}): {value}'')
```

In the case of CSC, similar code can be applied:

```
for col in range(csc_matrix.shape[1]):
    for ptr in range(csc_matrix.indptr[col], csc_matrix.indptr[col + 1]):
        row = csc_matrix.indices[ptr]
        value = csc_matrix.data[ptr]
        print(f'({row}, {col}): {value}'')
```

The non-zero elements of LIL are managed by the attributes `.rows` and `.data`, which can be accessed:

```
for row in range(lil_matrix.shape[0]):
    for col, value in zip(lil_matrix.rows[row], lil_matrix.data[row]):
        print(f'({row}, {col}): {value}'')
```

For DOK, traversing through non-zero elements is similar to the iteration over a dictionary:

```
for (row, col), value in dok_matrix.items():
    print(f'({row}, {col}): {value}'')
```

4.2.2 Linear algebra for sparse matrix

One primary use of sparse matrices is to perform the linear algebra operations on large matrices which would be impractical or inefficient with dense matrix representations. The linear algebra module `scipy.linalg` is not a suitable choice for sparse matrices since it is designed for dense matrices. Here, the linear algebra functions in the `scipy.sparse.linalg` module can be employed. However, `scipy.sparse.linalg` does not offer as many features as `scipy.linalg`. Within `scipy.sparse.linalg`, we can find basic linear algebra functions, including:

- Linear equation solvers, such as the `spsolve` and `gmres` (Generalized Minimal RESidual) functions.
- Eigenvalue functions, such as `eigs` and `eigsh` for general eigenvalue problems, and `svds` for singular value decomposition (SVD).
- Matrix factorization function `splu`, which computes the LU decomposition of a sparse, square matrix.
- Some basic matrix operations, including `expm` for matrix exponentiation, `inv` for matrix inversion, and `norm` for computing the matrix norm.

The basic usage of `scipy.sparse.linalg` functions is very intuitive. We will not detail their usage here, since the SciPy official documentation already provides good references [3,4].

When using `scipy.sparse.linalg` functions, we cannot treat them entirely as a black box as we might for dense matrices. Understanding the algorithms used by these functions and their limitations is very helpful for solving linear algebra problems involving sparse matrices.

The eigenvalue function `eigsh` employs iterative methods (Arnoldi iteration) to compute eigenvalues. Given that the application of sparse matrices typically involves large-sized matrices, it is often impractical and unnecessary to compute the full spectrum of eigenvalues as for dense matrices. The iterative method can target only a subset of eigenvalues and eigenvectors. To compute the ground state of a quantum system, we can configure the parameter `which='SA'` for the lowest eigenvalues. An alternative option is the `lobpcg` function (Locally Optimal Block Preconditioned Conjugate Gradient), which allows us to specify a preconditioner to accelerate the convergence of the ground state eigenvalue problem. Another choice for determining the ground state of a quantum system is the Davidson diagonalization algorithm, which we will explore in Chapter 14.

In the case of linear equations, `scipy.sparse.linalg` offers a variety of solvers. These solvers can be categorized into direct solvers and iterative solvers. For instance, the `spsolve` function is a direct solver, whereas `gmres` is an iterative solver. In addition to `gmres`, there are several iterative solvers in the `scipy.sparse.linalg` module, including `cg` (Conjugate Gradient) which is suitable only for symmetric positive definite matrices, `bicg` (BIConjugate Gradient), `minres` (MINimum RESidual), and `qmr` (Quasi-Minimal Residual). For non-square matrices, the `lsqr` solver is available to iteratively solve the linear equation in the least squares sense.

A direct solver (like `lpsolve`) must factorize the sparse matrix A before it can solve the equation $Ax = b$. This factorization process is notably expensive. However, direct solvers can compute the solution precisely, with guaranteed accuracy. Moreover, when the `b` array includes multiple vectors, direct solvers can efficiently solve for multiple `b` vectors with very small additional computational costs.

On the other hand, the iterative solver, such as `gmres`, employs the matrix-vector production Ax to construct a Krylov subspace. The linear equation is then transformed and represented within this subspace, and subsequently solved there. This approach is particularly efficient when solving for one or a few `b` vectors. However, as an iterative method, it may encounter convergence and precision issues.

One important consideration for the sparse-matrix linear algebra functions is their memory consumption. Although the input matrix is sparse, the outputs of linear algebra operations may not necessarily retain this sparsity. For instance, the inverse of a sparse matrix often results in a dense matrix (even if the output of the `inv` function is stored in a sparse matrix format). The `inv` function, therefore, breaks the sparsity and should be avoided whenever possible. Similarly, the `splu` factorization disrupts sparsity, as the outputs of LU factorization are generally dense arrays. Please note that the direct linear equation solver `spsolve` requires the LU factorization of the matrix, and the LU intermediates can consume a significant amount of memory. Therefore, `spsolve` is not suitable for larger sparse matrices, as it may exhaust the available memory.

4.2.3 Linear operator

In some circumstances, it may be impractical or even impossible to store a matrix in a sparse format. Often, these matrices are defined through their product with another vector. The full configuration interaction (FCI) problem in quantum chemistry, as discussed in Chapter 14, is a typical example of this.

We can utilize the `scipy.sparse.linalg.LinearOperator` class to represent sparse matrices that are defined through their matrix-vector product. In fact, all iterative methods within the `scipy.sparse.linalg` module rely on the `matvec` interface provided by the `LinearOperator` class, regardless of whether the underlying matrix is dense, sparse, or a linear operator.

To demonstrate the use of the `LinearOperator` class, let's consider a simple example: the Hamiltonian of a one-dimensional harmonic oscillator:

$$\hat{H} = -\frac{1}{2m} \frac{d^2}{dx^2} + \frac{k}{2} x^2, \quad (4.1)$$

where m represents the particle mass, and k is the force constant. To solve the time-independent Schrödinger equation

$$\hat{H}\psi = E\psi, \quad (4.2)$$

we discretize both the Hamiltonian and the wavefunction on 1D grids with a spacing of a . The kinetic energy term can be calculated using the finite difference formula

(with the open boundary condition):

$$\frac{d^2}{dx^2}\psi(x_n) \approx \frac{1}{a^2}(\psi(x_{n-1}) - 2\psi(x_n) + \psi(x_{n+1})). \quad (4.3)$$

Using this formula, it is straightforward to derive a `matvec` function that computes the product $H\psi$. We then utilize the `LinearOperator` class to register the `matvec` function, which provides an object that is compatible with the `scipy.sparse.linalg` functions. The energy and the corresponding states of the quantum system can then be calculated using the iterative sparse-matrix eigenvalue solvers.

```
def harmonic_oscillator(m, k, ngrids, box_size):
    '''H = 1/2m p^2 + 1/2 k x^2'''
    x = np.linspace(-box_size, box_size, ngrids)
    dx = x[1] - x[0]
    def matvec(psi):
        d2 = psi * -2
        d2[:-1] += psi[1:]
        d2[1:] += psi[:-1]
        return -1/(2*m*dx**2) * d2 + k/2 * x**2 * psi

    return scipy.sparse.linalg.LinearOperator((ngrids, ngrids), matvec=matvec)

if __name__ == '__main__':
    m = 0.5
    k = 0.5
    ngrids = 1000
    box_size = 10.
    A = harmonic_oscillator(m, k, ngrids, box_size)
    e, psi = scipy.sparse.linalg.eigsh(A, which='SM')
    print(e)
```

4.3 Tensor contractions

Many scientific computation tasks involve summing over specific indices of the products of multiple tensors. These tasks can often be transformed into tensor contraction operations. For example, to compute the covariance matrix for a set of vectors

$$cov(X_i, X_j) = \frac{1}{n-1} \sum_{k=1}^n (X_{ik} - \bar{X}_i)(X_{jk} - \bar{X}_j), \quad (4.4)$$

we can implement a program with for-loops:

```
def cov(x):
    m, n = x.shape
    c = np.zeros((m, m))
    for i, xi in enumerate(x):
        for j, xj in enumerate(x):
            c[i,j] = np.dot(xi - xi.mean(), xj - xj.mean()) / (n - 1)
    return c
```

Alternatively, the same functionality can be achieved with the tensor contraction code:

```
def cov(x):
    m, n = x.shape
    dx = x - x.mean(axis=1)[:,None]
    return np.einsum('ik,jk->ij', dx, dx) / (n - 1)
```

In this example, we have utilized the NumPy `np.einsum` function to perform the tensor contraction. Within `np.einsum`, the indices of the input operands and the output tensor are labeled by letters. The contraction pattern is described by a string composed of these letters. In this description string, the inputs and output are separated by the notation ' \rightarrow '. The repeated letters that appear in the inputs but not in the outputs indicate summation over those indices, following the Einstein summation convention. When there are multiple input operands, their indices are separated by the comma notation.

Why do we prefer using tensor contractions in scientific computing problems? In addition to the advantage of code readability, tensor contractions provide superior performance because they can leverage libraries specifically optimized for tensor contraction operations. The covariance matrix computation is one such example. The `einsum` version is an order of magnitude faster than the version that uses for-loops.

Please note that the default implementation of `np.einsum` is not optimal for performance. For instance, when executing the matrix multiplication

```
c = np.einsum('ij,jk->ik', a, b)
```

`np.einsum` is significantly slower than the `np.dot` function. The latter achieves faster performance by utilizing matrix multiplication routines provided by the underlying BLAS library.

The performance of `np.einsum` can be improved by specifying the keyword argument `optimize=True`. With this option, `einsum` leverages the `tensordot` function, which translates the `einsum` pattern into matrix multiplications. Additionally, this option enables the search for the most efficient path for contraction when it involves multiple tensors. For example, consider the tensor contractions below, where both array `a` and array `b` have a shape of (n, n) , and array `c` is a $(n, \frac{n}{2})$ -shaped array.

```
d = np.einsum('ij,jk,kl->il', a, b, c)
```

`np.einsum` by default performs the three-tensor contractions in a single operation, which has a complexity of $O(n^4)$. It is undoubtedly unfavorable. If we break down the contraction process into

```
d = np.einsum('ik,kl->il',
              np.einsum('ij,jk->ik', a, b), c)
```

these two steps of tensor contractions will require $2n^3 + n^3$ floating point operations. If changing the contraction order to

```
d = np.einsum('ij,jl->il', a,
              np.einsum('jk,kl->jl', b, c))
```

the total number of floating point operations becomes $n^3 + n^3$. Clearly, the swapped contraction order is more efficient. When the `optimize=True` option is enabled, the contraction order is optimized by the `np.einsum_path` function. `np.einsum_path` can analyze the floating-point operations in tensor contractions and suggest the optimal contraction path that minimizes the number of floating-point operations.

Is it recommended to always enable the `optimize` option when using `np.einsum`?

Enabling the `optimize` option can often make `einsum` faster. However, there are certain situations where using `optimize` might not be beneficial and could even decrease performance. Regarding to this question, we can consider three factors:

- Whether the `einsum` pattern can be converted into a matrix-multiplication form. The `optimize` option is beneficial when the `einsum` pattern can be mapped to matrix-multiplication. The conversion is possible only if any repeated indices occur exactly twice. If conversion to matrix-multiplication is not feasible, setting `optimize=True` for `np.einsum` will simply invoke the standard `np.einsum` implementation, thus performance is not improved. It is worthy to note that the simple NumPy broadcasting can sometimes outperform `np.einsum` under these circumstances. For example:

```
In [1]: a = np.random.rand(100, 100, 100)
        b = np.random.rand(100, 100, 100)

In [2]: %time c = np.einsum('kij,ijp->ijkp', a, b)
CPU times: user 175 ms, sys: 128 ms, total: 303 ms
Wall time: 302 ms

In [3]: %time c = np.einsum('kij,ijp->ijkp', a, b, optimize=True)
CPU times: user 175 ms, sys: 113 ms, total: 288 ms
Wall time: 287 ms

In [4]: %time c = a.transpose(1,2,0)[:,:,:,:,None] * b[:,None,:,:]
```

CPU times: user 101 ms, sys: 133 ms, total: 234 ms
 Wall time: 233 ms

- The problem size. Typically, `np.einsum_path` can introduce an overhead around 100 milliseconds. When the problem size is relatively small, this overhead can no longer be ignored. On the other hand, for large tensor contraction problems, it is crucial to determine whether the operation is subject to a CPU-bound or a memory-bound problem (details of these concepts will be explored in Chapter 9). When the tensor contraction is a memory-bounded problem, even if it can be converted to matrix multiplication, the original `np.einsum` without `optimize=True` can achieve better efficiency. For example,

```
In [5]: a = np.random.rand(100, 100)
        b = np.random.rand(100, 100)

In [6]: %time c=np.einsum('ij,kl->ijkl', a, b)
CPU times: user 104 ms, sys: 125 ms, total: 229 ms
Wall time: 228 ms

In [7]: %time c=np.einsum('ij,kl->ijkl', a, b, optimize=True)
CPU times: user 132 ms, sys: 100 ms, total: 232 ms
Wall time: 231 ms
```

- The overhead of generating intermediates. To map the `einsum` pattern to matrix multiplication, `tensordot` carries out a series of tensor transpose and reshape operations. As discussed in Chapter 2, these transpose-reshape operations may lead to the creation of a copy of a NumPy array. When large tensors are involved, the creation of a tensor copy may introduce a significant overhead, which can sometimes dominate the computational costs. For example,

```
In [8]: a = np.random.rand(500, 500, 500)
        b = np.random.rand(500, 3, 500)

In [9]: %time c=np.einsum('jki,ipj->kp', a, b)
CPU times: user 529 ms, sys: 0 ns, total: 529 ms
Wall time: 528 ms

In [10]: %time c=np.einsum('jki,ipj->kp', a, b, optimize=True)
CPU times: user 969 ms, sys: 115 ms, total: 1.08 s
Wall time: 1.08 s
```

4.4 Discrete Fourier transforms

Fourier Transform is utilized in many areas of scientific computing. In quantum chemistry applications, it is employed for calculating Coulomb interactions, evaluating integrals, solving differential equations, and other numerical processes. In

computational programs, the Discrete Fourier Transform (DFT) is often used for FT on sample data x_n that are evenly distributed in space. This process outputs a series of data points y_k represented on evenly distributed frequencies k .

$$y_k = \sum_{n=0}^{N-1} x_n \exp\left(\frac{-2i\pi nk}{N}\right). \quad (4.5)$$

The inverse DFT is given by

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k \exp\left(\frac{2i\pi nk}{N}\right). \quad (4.6)$$

The DFT can be computed very efficiently using the Fast Fourier Transform (FFT) algorithm. Although the DFT appears to have an $O(n^2)$ complexity, the FFT algorithm can reduce the complexity to $O(n \log(n))$, thanks to the simple yet efficient Cooley-Tukey algorithm [5]. The Cooley-Tukey algorithm is effective for the FFT length that can be factorized to a product of small primes (e.g., $2^a 3^b 5^c$). For large prime numbers, other algorithms [6,7] can achieve the $O(n \log n)$ scaling for DFT of arbitrary length. However, the performance is generally much worse compared to the cases when the DFT sizes can be factored into products of small primes.

4.4.1 SciPy FFT

The `scipy.fft` module offers a variety of FFT functions for computing the DFT and its inverse, including:

- The general one-dimensional FFT and its inverse functions `fft` and `ifft`.
- Two-dimensional and multi-dimensional FFT functions `fft2`, `fftn`, and their inverses `ifft2`, `ifftn`.
- The `rfft` function. This function is specialized for the real input data and can approximately reduce the computational costs by half. The `irfft` function computes the inverse of `rfft` and returns a real array.
- The `fftfreq` and `rfftfreq` functions. These functions can be utilized to generate the DFT sample frequencies.

For example, given a vector of sample data, we can execute the following DFT and inverse DFT operations,

```
In [1]: x = np.random.rand(10)
        y = scipy.fft.fft(x)
        x = scipy.fft.ifft(y)
        print(y.shape)

Out[1]:
(10,)
```

```
In [2]: x = np.random.rand(10)
        y = scipy.fft.rfft(x)
        x = scipy.fft.irfft(y)
        print(y.shape)

Out[2]:
(6,)

In [3]: print(scipy.fft.rfftfreq(10))
Out[3]:
[0.  0.1 0.2 0.3 0.4 0.5]
```

Please note that the `fft` function outputs a complex array, with the shape identical to the input array. In contrast, the `rfft` function can leverage the symmetry between positive and negative frequencies, and calculate only the DFT of the positive frequencies. As a result, the output of the `rfft` function is effectively half the size of a standard FFT.

An important feature of the `scipy.fft` module is its ability to execute FFTs on multiple vectors in parallel using multithreading. By setting the argument `workers`, we can allocate multiple threads to execute SciPy FFT functions.

```
x = np.random.rand(2000, 10)
y = scipy.fft.fft(x, workers=4) # using 4 threads
y = scipy.fft.fft(x, workers=-1) # using all available CPU cores
```

Since the parallelization is executed across vectors rather than within the FFT dimension, using multiple workers does not offer benefits if there are not enough vectors to transform.

You might have noticed that the NumPy library also provides an implementation of FFT in the `numpy.fft` module. The functionalities of `scipy.fft` and `numpy.fft` are almost identical. However, generally speaking, the `scipy.fft` module is more favorable than `numpy.fft`, due to the following reasons:

- `scipy.fft` uses the FFTPACK library [8], which is more comprehensively optimized and more efficient than the NumPy built-in FFT library.
- `scipy.fft` allows us to overwrite the input data (with the keyword parameter `overwrite_x`), which can reduce both memory usage and data transfer overhead. In contrast, `numpy.fft` always produces a new array for its output.
- `scipy.fft` can utilize the parallel capabilities of multi-core CPUs, whereas `numpy.fft` does not offer multithreading execution.
- Not just `rfft`, the `scipy.fft.fft` function can identify real input data and leverage the symmetry in the transformation to enhance performance.

In addition to the FFT modules offered by NumPy and SciPy, other popular FFT libraries, such as FFTW and Intel FFT, also provide Python interfaces.

4.4.2 Intel FFT

Intel FFT is a high-performance library for X86 hardware. It leverages vectorization and multi-core parallelism to improve FFT performance.

Intel FFT is a component of the Intel Math Kernel Library (MKL). In a Python program, the Intel FFT functions can be accessed through the Python package `mkl_fft`. Please note that the package `mkl_fft` is just an interface to the MKL library. It does not include the MKL library itself, which must be installed separately. On most X86 platforms, the Intel MKL core library and the FFT wrapper can be installed in the Python environment using the `pip` command

```
$ pip install mkl_fft mkl mkl-service
```

The Intel FFT library is straightforward to use. It does not require complex configurations, such as creating or caching a plan, as in FFTW (see Section 4.4.3). Additionally, memory alignment is not mandatory, though aligning memory can enhance performance. The `mkl_fft.interfaces.scipy_fft` module follows the APIs of `scipy.fft` and `numpy.fft`. The code example below illustrates how to achieve a seamless transition from SciPy and NumPy to `mkl_fft` using the compatible interface.

```
y = scipy.fft.rfft(x, workers=4)
y = mkl_fft.interfaces.scipy_fft.rfft(x, workers=4)

y = numpy.fft.rfft(x)
y = mkl_fft.interfaces.numpy_fft.rfft(x)
```

We can use the `mkl_fft` library as a drop-in replacement for `scipy.fft`. For example, `mkl_fft` can be configured as the backend for `scipy.fft`.

```
In [4]: with scipy.fft.set_backend(mkl_fft.interfaces.scipy_fft):
    y = scipy.fft.fft(x, workers=8)
```

Within this context, when functions from the `scipy.fft` module are called, they actually execute the FFT functions from the `mkl_fft` library.

4.4.3 PyFFTW

PyFFTW [9] provides Python bindings for the FFTW [10] project. FFTW is known for its comprehensive and deep optimizations for FFT, making it one of the most efficient FFT libraries available.

To enhance FFT performance, FFTW accommodates a broad range of optimizations in the library. It factors the FFT length using prime number radices up to 13. It also incorporates multiple $O(n \log(n))$ algorithms for the larger prime numbers. Optimizations for both hardware and specific data types are considered. Furthermore, FFTW provides specialized implementations for different types of transformations, such as real-to-real, real-to-complex, and complex-to-real transformations, and so forth. Given the complexity of choosing the optimal FFT scheme due to the diverse

algorithms and optimizations, FFTW introduces an adaptive *planner*. The planner evaluates the performance of various FFT algorithms on the user's hardware to determine the most efficient algorithm before performing the actual transformation.

The PyFFTW package offers two primary interfaces to simplify the use of FFTW in Python: a standard interface that is compatible with the APIs of `scipy.fft` and `numpy.fft`, and an advanced interface that provides extra configuration options for enhanced performance.

The standard interface is similar to that of `mkl_fft`, which is compatible with the FFT modules of SciPy and NumPy. PyFFTW can be used as a backend for `scipy.fft` as well.

```
In [5]: with scipy.fft.set_backend(pyfftw.interfaces.scipy_fft):
    y = scipy.fft.fft(x, workers=8)
```

However, as we will find in the performance benchmark in the next section, it is unlikely to gain performance benefits through the standard interface. If you intend to use PyFFTW in an application, it is recommended to engage PyFFTW's advanced interface to unlock the full performance potential of FFTW.

The advanced interface provides more detailed customization for FFTW operations, including:

- *Memory alignment.* PyFFTW ensures memory alignment before invoking the underlying FFTW functions. If the input array does not meet the memory alignment requirements (which is a system-specific setting), PyFFTW allocates new memory with the appropriate alignment and copies the input array into this newly allocated array, thereby introducing some overhead. To address this issue, it is advisable to allocate NumPy arrays with aligned memory for FFT data storage whenever feasible. PyFFTW offers the `empty_aligned` function to create an empty NumPy array with the required memory alignment.

```
# Preallocate array with aligned memory
x = pyfftw.empty_aligned(shape, dtype=np.complex128)
x[:] = np.random.random(shape)
```

Alternatively, you can use

```
x = pyfftw.byte_align(x)
```

to ensure memory alignment for an array. This statement creates a new array only if the data in the input array is not properly aligned.

- *Configuring an FFT planner.* The default planner in PyFFTW is `FFTW_ESTIMATE`, which creates a plan that is likely sub-optimal. An optimal FFTW runtime plan depends on various factors, such as the problem size, FFT length, data type, transformation type, memory alignment, and memory access patterns. Other planners, such as `FFTW_MEASURE` and `FFTW_PATIENT` [10,11], can conduct FFTW to invest more effort in identifying the fastest possible method. To configure the planner, we can change the global configuration to:

```
pyfftw.config.PLANNER_EFFORT = 'FFTW_MEASURE'
```

or set the keyword argument `planner_effort` of the FFT functions:

```
pyfftw.interfaces.scipy_fft.fft(x, planner_effort='FFTW_MEASURE')
```

- *Caching the execution plan.* The FFTW planners `FFTW_MEASURE` and `FFTW_PATIENT` can introduce significant overhead during setup. If the FFT length is large, the overhead may far exceed the FFT execution time. Therefore, these planners are suitable for programs that repeatedly execute the same type of FFT. To ensure the planner is initialized only once, the generated plan should be cached. This cache can be enabled in PyFFTW by executing:

```
pyfftw.interfaces.cache.enable()
```

When FFTW is configured through the advanced interfaces, it is recommended to use the helper function `pyfftw.builders.fft` to create an instance of the `pyfftw.FFTW` class. This instance contains all the FFTW parameters required for the computation. We can invoke this instance to carry out the actual transformation.

```
In [6]: fft_obj = pyfftw.builders.fft(x, threads=nproc)
        out = fft_obj()
```

4.4.4 FFT performance benchmark

So far, we have introduced four FFT libraries: `numpy.fft`, `scipy.fft`, PyFFTW, and `mkl_fft`. Which FFT library should we use in real-world applications? In this section, we will explore this question through performance benchmarks.

For PyFFTW, the benchmark tests include both its standard interface and the advanced customization. When calling the standard interface, the FFTW planner is `FFTW_ESTIMATE`, which is the default setting in PyFFTW. The customization is aimed at enhancing FFTW performance, which incorporates the explicit memory alignment and the use of the `FFTW_MEASURE` planner. Additionally, before timing the calculation, each test case is executed once to warm up the planner cache. These configurations represent the most ideal usage scenario of the PyFFTW package in real applications.

One focus of the benchmark test is the efficiency of various FFT transformation types, include the complex-to-complex FFT using the regular `fft` function (`c2c-fft`), the real-to-complex FFT using the `fft` function with real input data (`r2c-fft`), the real-to-complex FFT using `rfft` (`r2c-rfft`), and the complex-to-complex two-dimensional FFT using `fft2` (`c2c-fft2`).

FFT problem sizes are a crucial factor for FFT performance. Here, we examine two contrasting scenarios. One involves larger FFT lengths, which reflect the efficiency of the $O(n \log(n))$ algorithm in each FFT library. Additionally, given that quantum chemistry calculations for crystal systems often employ FFT mesh in 3D

space within a moderate box, the FFT length in each axis is usually not large. Therefore, the second set of test cases focuses on scenarios where FFT lengths are less than 100. This test primarily reflects the memory efficiency in each FFT library.

Scientific computations are often executed in parallel using multi-core CPUs. The efficiency of multithreading parallelism is another factor to consider. As such, the performance of each FFT library is benchmarked in both single-thread and 8-thread parallelization scenarios. The multithreading benchmark is performed on the tests of small FFT lengths. To minimize the impact of initialization overhead, the FFTs are applied to an array with a sufficient number of vectors (ranging from 10^5 to 10^7).

The benchmark tests were performed on an AWS c5a.4xlarge EC2 instance. This instance is a 16-core virtual machine running on the AWS cloud, equipped with an AMD EPYC CPU running at 3.3 GHz and 32 GB of memory. On this machine, we installed NumPy v1.24, scipy v1.13, PyFFTW v0.13, mkl_fft v1.3 with MKL 2024.1. These libraries were installed via the pre-compiled wheels provided by PyPI. It is worth noting that the wheel for PyFFTW v0.13 does not support SIMD (Single Instruction Multiple Data) vectorization. To enable FFTW SIMD level optimization, it is necessary to manually compile the FFTW and PyFFTW libraries. However, possibly due to the problem sizes and the overhead of the PyFFTW interface, we did not observe advantages of the manually compiled SIMD-FFTW over the wheel-packaged FFTW in these tests. Therefore, we have excluded the results of the manually compiled FFTW from the benchmark results.

At the large FFT length end, we tested FFT lengths that can be factorized into small prime number radices as well as those that cannot. The performance of each FFT library is summarized in Table 4.2. SciPy emerges as the fastest across multiple scenarios. Intel FFT is slightly slower than `scipy.fft`, but their speeds are generally comparable. In scenarios involving large prime numbers, PyFFTW shows a clear performance advantage. This advantage can be attributed to its extensive choices of $O(n \log(n))$ algorithms available for general FFT lengths. In other cases, surprisingly, PyFFTW exhibits slow performance when the FFT length is divisible into small prime factors. Even after PyFFTW is optimized with the `FFTW_MEASURE` planner and adequately warmed up, it remains approximately twice slower than `scipy.fft`.

In the tests for small FFT lengths, we implemented additional code to assist the benchmark tests.

`numpy.fft` lacks multithreading parallelism capabilities. To enable parallel execution, we utilize the Python multithreading module to execute `numpy.fft` on independent vectors in parallel. Below is a code sample that demonstrates how the `numpy.fft.fft` function is parallelized. Other types of FFT transformations are parallelized in a similar manner.

```
import threading
def np_fft(fn, x, nproc=8):
    y = np.empty(x.shape, dtype=np.complex128)
    block = (x.shape[0]+nproc-1) // nproc

    def _fft_priv(i):
```

Table 4.2 Performance benchmarks of the standard one-dimensional `fft` for large FFT lengths. The performance is measured in milliseconds.

FFT length	NumPy	SciPy	PyFFTW		Intel FFT
			standard	custom	
$129591 = 3^2 7^1 11^2 17^1$	7.4	4.6	11.7	9.5	7.6
$156816 = 2^4 3^4 11^2$	6.5	5.6	10.3	4.8	3.9
$199584 = 2^5 3^4 7^1 11^1$	10.1	6.4	8.5	14.6	6.3
$243432 = 2^3 3^3 7^2 23^1$	14.2	8.7	14.9	19.2	6.2
$425799 = 3^2 11^2 17^1 23^1$	23.6	11.2	31.2	31.6	29.5
$494802 = 2^1 3^3 7^2 11^1 17^1$	29.5	18.6	85.6	31.3	14.5
$94763 = 193 \cdot 491$	22.5	15.1	13.0	11.7	17.9
$133141 = 211 \cdot 631$	30.0	23.9	15.4	16.2	30.9
$225851 = 71 \cdot 3181$	48.3	43.6	25.7	22.7	29.5
$359573 = 103 \cdot 3491$	81.3	73.8	47.9	44.1	72.6
Avg. speed up wrt NumPy		0.71	1.15	0.93	0.77

```
i0, i1 = i * block, (i+1) * block
y[i0:i1] = np.fft.fft(x[i0:i1])

threads = []
for i in range(nproc):
    t = threading.Thread(target=_fft_priv, args=(i,))
    t.start()
    threads.append(t)
[t.join() for t in threads]
return y
```

To ensure optimal performance of FFTW, we implemented the following `fftw_warmup` function. They are executed before calling the PyFFTW functions to initialize the FFT plan.

```
import pyfftw
pyfftw.config.PLANNER_EFFORT = 'FFTW_MEASURE'
pyfftw.interfaces.cache.enable()

def fftw_warmup(x, nproc=8):
    x = pyfftw.byte_align(x)
    fftw_obj = pyfftw.builders.fft(x, threads=nproc)
    fftw_obj()
    return fftw_obj
```

Additionally, based on the DFT equations (4.5) and (4.6), we implemented the naive DFT functions using matrix multiplication. The matrix multiplication is pow-

ered by the OpenBLAS backend from NumPy [12]. The naive DFT implementations for `fft`, `rfft`, and `fft2` are demonstrated below.

```
def naive_fft(x):
    n = x.shape[-1]
    m = np.exp(-2j*np.pi*np.arange(n)[:,None] * np.fft.fftfreq(n))
    return x.dot(m)

def naive_rfft(x):
    assert x.dtype.kind != 'c'
    n = x.shape[-1]
    m = np.exp(-2j*np.pi/n*np.arange(n)[:,None] * np.fft.rfftfreq(n))
    return x.dot(m.T)

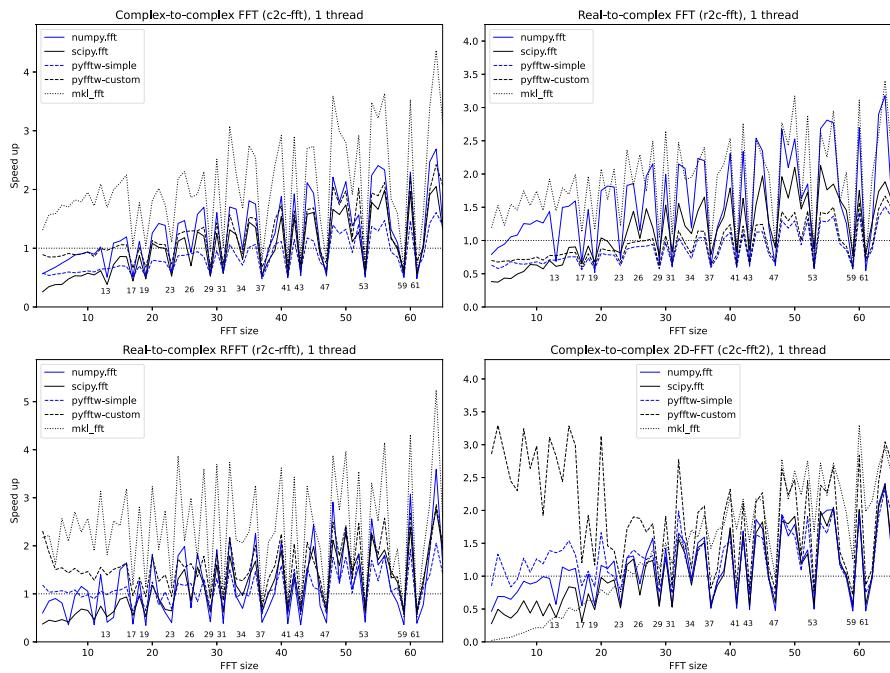
def naive_fft2(x):
    n1, n2 = x.shape[-2:]
    m1 = np.exp(-2j*np.pi/n1*np.arange(n1)[:,None] * np.fft.fftfreq(n1))
    m2 = np.exp(-2j*np.pi/n2*np.arange(n2)[:,None] * np.fft.fftfreq(n2))
    x1 = m1.T.dot(x.reshape(-1,n2).T)
    y = m2.T.dot(x1.reshape(-1,n1).T)
    return y.reshape(n1, n2, -1).transpose(2,0,1)
```

In the benchmark tests, the computational time of naive DFT scales perfectly as N^2 with respect to the FFT length N . This serves as our baseline to measure the performance achieved by other FFT libraries. The performance benchmarks are summarized in Fig. 4.1 and Fig. 4.2. The efficiency of multithreading parallelization are displayed in Fig. 4.3.

We can observe that the speedups with respect to the naive DFT vary significantly across these FFT libraries. The performance for certain FFT lengths is noticeably poorer compared to the adjacent data points. These FFT lengths mainly correspond to prime numbers. FFT libraries typically do not have special optimizations for these moderate-sized prime numbers. On these data points, FFT libraries are even slower than the naive DFT.

For FFT lengths that are not prime numbers, both `numpy.fft` and `scipy.fft` show significant performance enhancements over the naive DFT. Notably, `numpy.fft` outperforms `scipy.fft` in several tests, particularly in the r2c-FFT cases. When multithreading is enabled, `scipy.fft` is faster than `numpy.fft` in c2c-fft, r2c-rfft, and c2c-fft2 jobs, thanks to its better built-in parallel implementation.

The performance advantage of `numpy.fft` can primarily be attributed to the reduced overhead associated with data type conversion and data movement. The NumPy FFT code is slightly more efficient when performing type conversion and data copying than the SciPy FFT functions, which results in superior performance over SciPy in some tests. By default, the FFT functions provided by NumPy and Scipy do not modify the input data. This leads to a significant overhead due to the extra step of data copying in these FFT libraries. If the `overwrite_x` keyword is enabled

**FIGURE 4.1**

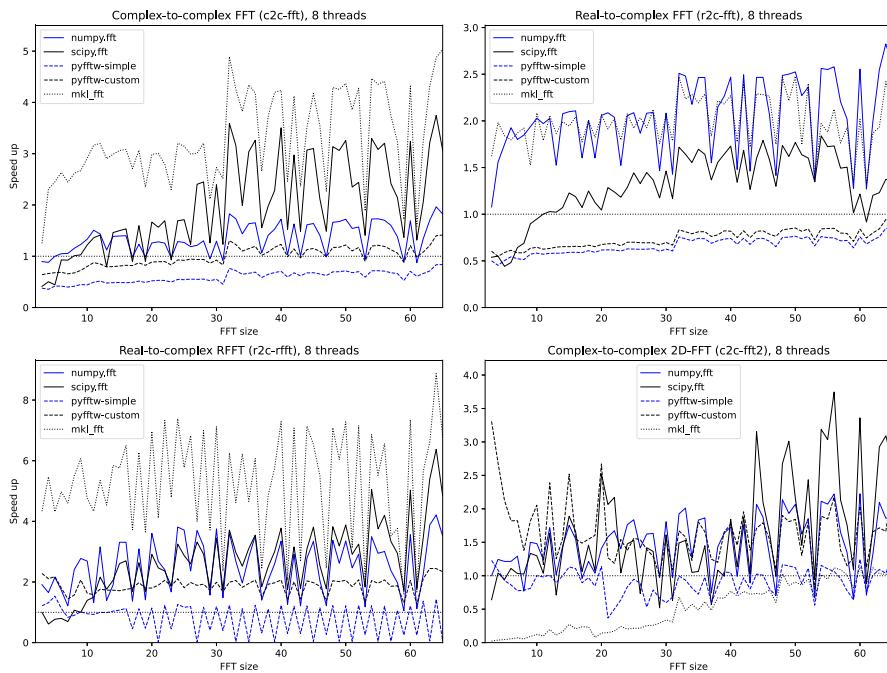
Performance comparison of various FFT libraries for different transformation types with single thread.

for `scipy.fft` functions, the performance of `scipy.fft` can be significantly improved, surpassing `numpy.fft`.

Interestingly, the performance of NumPy r2c-fft stands out significantly, being 30% or faster than `scipy.fft` in many tests. As mentioned earlier, `scipy.fft` includes special code to handle r2c-fft, which utilizes the symmetry of the real-valued FFT. Then, why would this optimization actually lead to reduced performance?

For small FFT problem sizes, the impact of this symmetry on computational cost is minimal. To exploit this symmetry, `scipy.fft` incorporates complex data moving operations. The overhead of these data moving operations is close to or exceeds the computational cost of the FFT itself. What's worse, the data moving operations in SciPy r2c-fft are not processed in parallel, further amplifying the overhead in multithreading calculations. As an evidence, in Fig. 4.3, we can find the multithreading speedups for SciPy r2c-fft are exceptionally low.

As the FFT length increases, the special optimization in SciPy's r2c-fft becomes more effective. When dealing with larger FFTs, SciPy can outperform NumPy in single-threaded execution, although the results here do not show this. However, when

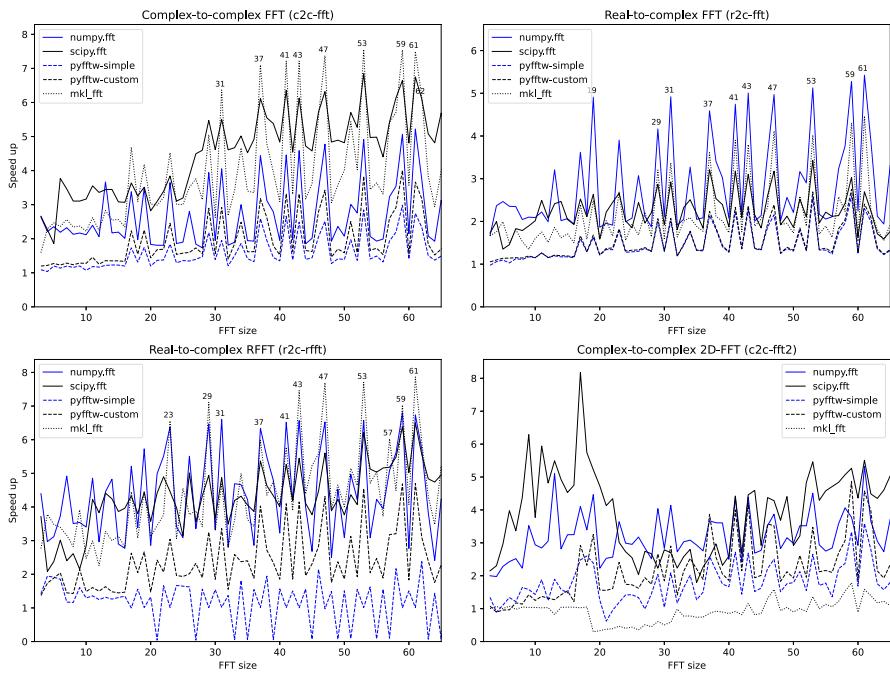
**FIGURE 4.2**

Performance comparison of various FFT libraries for different transformation types with eight threads.

multithreading is enabled, SciPy r2c-fft is still likely slower than that in NumPy, even for FFT lengths near 500.

Overall, the performance of the Intel `mkl_fft` library is the best among these FFT libraries. It stands out particularly in c2c-fft and r2c-rfft tests. In these FFT types, `mkl_fft` is approximately 50% faster than the other FFT libraries regardless of whether it is running in the single-threaded or multi-threaded modes. However, `mkl_fft` shows weak performance in c2c-fft2. In the single-threaded mode, `mkl_fft` performs comparably to other FFT libraries for c2c-fft2. When multithreading is enabled, `mkl_fft` significantly falls behind, even slower than the naive DFT.

PyFFTW exhibits the worst performance in these tests. When using the default `FFTW_ESTIMATE` planner, its performance lags behind the naive DFT, and it is significantly slower than `numpy.fft` and `scipy.fft`. The performance of PyFFTW only approaches that of `scipy.fft` after the `FFTW_MEASURE` planner is employed, data is explicitly aligned, and the planner is warmed up. With these setups, PyFFTW shows good performance in single-threaded c2c-fft2 tests. However, in the multi-threaded c2c-fft2 tests, PyFFTW again falls behind the other libraries. One reason for the performance issue can be attributed to the data moving overhead. Before invoking the

**FIGURE 4.3**

Multithreading efficiency for FFT libraries.

underlying FFTW functions, the PyFFTW wrapper always performs data alignment, leading to additional data movement. For the small FFT problem sizes considered in these tests, this data movement overhead is a crucial factor that impacts PyFFTW’s performance.

In the context of multithreading parallelism, both `scipy.fft` and `mkl_fft` exhibit good parallel efficiency. With 8-thread parallelism, they achieve an efficiency exceeding 50%, leading to a speedup of more than 4 times in most scenarios. Additionally, it is surprising that parallelization of `numpy.fft` using the Python standard `threading` shows notable efficiency. Especially for r2c-fft, its parallel efficiency surpasses that of any other FFT libraries. In r2c-fft jobs, the real array must be converted into a complex array before invoking the `fft` function. In the case of `numpy.fft`, this type conversion is executed across multiple threads. However, in other FFT libraries, the type conversion step is not parallelized, which becomes the main bottleneck for parallel efficiency.

PyFFTW generally exhibits poor parallel efficiency. Regardless of whether the advanced interface is utilized, multi-threaded PyFFTW achieves only a minimal performance improvement over the single-threaded execution. In most tests, the speedups of 8-thread parallelization do not even exceed 2. It can be anticipated that, as more CPU

Table 4.3 The highlight and unfavorable scenarios of each FFT library.

	Highlight Scenarios	Unfavorable Scenarios
Naive DFT	<ul style="list-style-type: none"> Small-prime FFT lengths 	<ul style="list-style-type: none"> All other scenarios
numpy.fft	<ul style="list-style-type: none"> Single-threaded r2c-fft for small FFT lengths Multi-threaded r2c-fft 	
scipy.fft	<ul style="list-style-type: none"> Single-threaded r2c-fft for large FFT lengths Multi-threaded c2c-fft, c2c-fft2 c2c-fft for factorizable FFT lengths 	
PyFFTW-standard		<ul style="list-style-type: none"> All scenarios
PyFFTW-custom	<ul style="list-style-type: none"> Single-threaded c2c-fft2 	<ul style="list-style-type: none"> All multi-threaded FFTs
mkl_fft	<ul style="list-style-type: none"> All scenarios except multi-threaded c2c-fft2 	<ul style="list-style-type: none"> Multi-threaded c2c-fft2

cores are utilized for parallel computation, PyFFTW may exhibit even worse parallel efficiency. Consequently, we may consider using the Python built-in `threading` module instead of relying on the PyFFTW built-in threading parallelism.

Based on this benchmark test, we summarize the highlights and unfavorable scenarios of each FFT library in Table 4.3.

4.4.5 How to choose FFT libraries?

When applying the DFT in a program, which FFT library should we choose? If the DFT does not dominate the cost, then both `scipy.fft` and `numpy.fft` are sufficiently good options. However, if the goal is to maximize FFT performance in scenarios that require intensive DFT computations, various factors should be evaluated to select the most suitable FFT libraries.

- Dimension of FFT.* `mkl_fft` is the preferred choice for one-dimensional FFT. In the case of multi-dimensional FFT, `scipy.fft` or PyFFTW can be considered.
- FFT length.* `mkl_fft` is an appropriate choice for small FFT lengths. When the FFT length is sufficiently large, the selection depends on the characteristics of the FFT length. If the FFT length is a random number, which likely involves large prime numbers, PyFFTW may be more efficient. Otherwise, `mkl_fft` and `scipy.fft` are better suited for most scenarios. When dealing with relatively small prime numbers (for instance, those smaller than 50), the naive DFT can be employed. For primes of moderate size, the choice can be `mkl_fft` for one-dimensional FFT and `scipy.fft` for multi-dimensional FFT.

If an application allows for choosing the size of data samples, to maximize the FFT efficiency, it is generally recommended to choose sizes that can be factorized

into powers of two or the products of small primes. However, in certain quantum chemistry applications, it is often unavoidable to use FFT lengths that are prime numbers. For instance, to achieve better numerical precision, we might need to use DFT frequencies that are symmetric with respect to the origin, potentially resulting in $2n + 1$ data samples on each axis. These samples often turn out to be prime numbers.

- *Parallelism.* If multiple CPU cores are feasible in the computation, `scipy.fft` and `mkl_fft` offer efficient and suitable options. The multi-threaded PyFFTW is found to be inefficient and is not recommended for use in parallel computation environments. For `numpy.fft`, the Python standard library `threading` should be utilized.
- *Data type of the input array.* If the input is guaranteed to be of a specific data type, either real or complex, `rfft` for real inputs and `fft` for complex inputs, can be applied accordingly. In such cases, `mkl_fft` is the most efficient library. On the other hand, if the input data requires a type conversion from real to complex, `mkl_fft` and `numpy.fft` have a relatively lower overhead in terms of type conversion and data moving, making them a suitable choice. Here, FFTs with large lengths are an exception. `scipy.fft` can intelligently choose the optimal transform function based on the data type, which, in some instances, can offset the overhead of type conversion.
- *Whether the input data can be overwritten.* If overwriting the input data is acceptable, the option `overwrite_x=True` can significantly enhance the performance of the `fft` function. However, this option is not applicable to the `rfft` function. The overhead associated with data movement is so substantial that it can outweigh the cost savings offered by `rfft`. Especially when combined with multithreading parallelism, allowing overwriting input data can effectively improve the parallel efficiency of FFT computations. Both the `scipy.fft` and `mkl_fft` libraries benefit from this option.
- Hardware, operating system, and so forth.

These factors are not isolated. They can interact with each other and affect the final FFT performance. In a real application, it is recommended to benchmark the performance based on your specific problem setups to determine the most suitable FFT library.

Summary

In this chapter, we introduce numerical computation tools for scientific computing tasks. To perform linear algebra operations, the `scipy.linalg` module is recommended. It is often more advanced than the `numpy.linalg` module in various aspects. To handle sparse matrices, `scipy.sparse` offers several storage formats and a comprehensive set of linear algebra tools. If mathematical equations can be formulated as tensor contractions, the `numpy.einsum` function is a powerful tool to enhance both readability and computational efficiency. For the discrete Fourier transform, the

`scipy.fft` module is usually sufficient. The Intel FFT, accessible via the `mkl_fft` Python package, is a viable alternative that can potentially offer enhanced computational performance.

Beyond numerical computation, scientific applications frequently involve non-numerical computation tasks. In the next chapter, we will explore techniques for non-numerical computing challenges.

References

- [1] The SciPy community, Scipy api - linear algebra, <https://docs.scipy.org/doc/scipy/reference/linalg.html>, 2024.
- [2] T.F. Chan, Rank revealing qr factorizations, Linear Algebra and Its Applications 88–89 (1987) 67–82, [https://doi.org/10.1016/0024-3795\(87\)90103-0](https://doi.org/10.1016/0024-3795(87)90103-0), <https://www.sciencedirect.com/science/article/pii/0024379587901030>.
- [3] The SciPy community, Scipy api - sparse linear algebra (`scipy.sparse.linalg`), <https://docs.scipy.org/doc/scipy/reference/sparse.linalg.html>, 2024.
- [4] The SciPy community, Scipy user guide - sparse eigenvalue problems with arpack, <https://docs.scipy.org/doc/scipy/tutorial/arpack.html>, 2024.
- [5] J.W. Cooley, J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, Mathematics of Computation 19 (90) (1965) 297–301, <http://www.jstor.org/stable/2003354>.
- [6] C.M. Rader, Discrete Fourier transforms when the number of data samples is prime, Proceedings of the IEEE 56 (6) (1968) 1107–1108, <https://doi.org/10.1109/PROC.1968.6477>.
- [7] L. Bluestein, A linear filtering approach to the computation of discrete Fourier transform, IEEE Transactions on Audio and Electroacoustics 18 (4) (1970) 451–455, <https://doi.org/10.1109/TAU.1970.1162132>.
- [8] SciPy Development Team, Scipy api - legacy discrete Fourier transforms (`scipy.fftpack`), <https://docs.scipy.org/doc/scipy/reference/fftpack.html>, 2024.
- [9] H. Gomersall, pyfftw: Python wrapper around fftw, Astrophysics Source Code Library, record ascl:2109.009 (Sep. 2021).
- [10] M. Frigo, A fast Fourier transform compiler, SIGPLAN Notices 34 (5) (1999) 169–180, <https://doi.org/10.1145/301631.301661>.
- [11] FFTW Development Team, Fftw documentation - planner flags, https://www.fftw.org/fftw3_doc/Planner-Flags.html, 2024.
- [12] NumPy Team, Numpy packages & accelerated linear algebra libraries, <https://numpy.org/install/#numpy-packages--accelerated-linear-algebra-libraries>, 2024.

Meta-programming and non-numerical computation

5

Meta-programming refers to the technique of using program to read, generate, or transform other programs. From its name, this technology appears to be a fascinating technology. Meta-programming is not only fascinating but also a very practical technique in scientific computing applications.

Scientific computing applications encompass more than just numerical computations. In this chapter, we will introduce three programming techniques beyond numerical computation:

- Code generation,
- Symbolic computation,
- Automatic differentiation.

In scientific applications, we might need to derive and implement programs for complex mathematical equations. This process can easily introduce bugs. To reduce the workload and minimize the risk of programming bugs, an efficient approach is to utilize symbolic computation for formula derivation, complemented by code generation and automatic differentiation for the implementation of the computation program.

5.1 Code generation

In C/C++ programs, meta-programming typically refers to code generation and automatic code deduction during compilation stage using techniques like macros or template programming. These techniques are considered as advanced features, which often cause confusion in C/C++ programs.

Due to the complexity of macro and template, Python has not adopted the similar designs in its language specifications. However, this does not mean that meta-programming is unavailable in Python. In fact, as a dynamic programming language, Python can treat code as data and manipulate it at runtime. This enables the application of more flexible and powerful meta-programming techniques, such as:

- Modifying functions at runtime.

- Composing functions programmatically.
- Dynamically generating and registering classes.

5.1.1 eval and exec

The Python built-in functions `eval` and `exec` can parse and evaluate code from a string. This feature allows us to dynamically compose code strings and activate them at runtime.

The function signatures of `eval` and `exec` are similar:

```
eval(expr, globals=None, locals=None)
exec(code, globals=None, locals=None)
```

The first argument of `eval` is a code expression. It is subject to certain restrictions:

- It must be a single expression. Assignment statements are not permitted.
- List comprehensions, dictionary comprehensions, and other comprehension code, are allowed.
- Control structures, such as if-clauses or for loops, are not allowed.
- Lambda expressions can be used for defining functions. However, the `def` or `class` keywords cannot be directly used for defining functions or classes.

The code string for `exec` can be any Python raw code. We can put the code of assignments, creating functions and classes in the code string. Using `exec` to evaluate a code string has the same effect as directly inserting that code at the point of execution.

`eval` and `exec` evaluate a code string within the context provided by `globals` and `locals` dictionaries. By default, these dictionaries correspond to the current global and local namespaces. This feature allows us to control the execution environment for a code string. For instance, by customizing a local environment, we can use the `eval` function to mimic the functionality of Pandas `DataFrame.eval` method, as demonstrated in the `simple_eval` function below.

```
In [1]: def simple_eval(df, expr):
    return eval(expr, {}, {**df})

In [2]: mole = pd.read_csv('water.csv')
        simple_eval(mole, '(x + y + z)/3')
Out[2]:
0    1.357033
1    1.512567
2    0.872433
dtype: float64
```

Here, the Pandas `DataFrame` is treated as a dictionary, which is a feature we have discussed in Chapter 2.

The functionality of `exec` and `eval` complement each other. Typically, they are used in different scenarios. The `eval` function allows us to incorporate programming capabilities when designing configurations for a program. The elements within a configuration can be designed as Python expressions. When parsing and decoding the configuration, `eval` can be employed to dynamically evaluate these expressions. On the other hand, `exec` is frequently used for generating functions and classes, which are then integrated into Python programs to expand functionalities.

5.1.2 Composing code programmatically

The Jinja templating technique, as demonstrated in Section 3.4.1 of Chapter 3, along with f-strings, is often employed for composing program code. Composed functions can be dynamically registered to the Python runtime using `exec` or `eval`. Additionally, the string manipulation techniques can be used to generate program code for other programming languages. In Chapter 12, we will demonstrate a comprehensive example that utilizes Jinja templating techniques to generate C code for efficient computation of the quantum chemistry analytical integrals.

5.1.3 Manipulating classes and functions at runtime

Besides the code generation in terms of strings, we can also use the meta-programming capabilities provided by Python to manipulate a Python function or a class. The use of decorators in Python code is a common example of such meta-programming techniques. The syntax of decorators is explained in many basic Python programming books, such as *Fluent Python: Clear, Concise, and Effective Programming* [1], so we will not discuss it here.

For classes, Python additionally provides the `metaclass` keyword, which allows for the dynamic modification of the functionality of a class.

```
class Foo(metaclass=Meta):
    ...
```

By specifying the `metaclass` keyword, the class factory `Meta` is invoked during the creation of class `Foo`. The `Meta` class can access and modify all attributes and methods of the `Foo` class, regardless of whether they are public or private. However, the use of `metaclass` is relatively rare. Readers who are interested in metaclasses can refer to the book *Fluent Python: Clear, Concise, and Effective Programming* for more details. Here, we will study an alternative method that utilizes the `type` function to dynamically generate class.

The primary functionality of the `type` function in Python is to determine the data type of an object. The effect is the same to accessing the `__class__` attribute of an object: `object.__class__`. The second functionality of `type` is to dynamically create classes:

```
new_cls = type(cls_name, (bases,), cls_attrs)
```

Here, `cls_name` is a string representing the name of the class, similar to accessing `cls.__name__` in a regular class. `bases` is a tuple of classes that serve as parent classes, which can be found in a regular class as `cls.__bases__`. `cls_attrs` is a dictionary that records mappings of attributes and methods within a class, which is accessible via `cls.__dict__` in a regular class. For instance,

```
dyn_cls = type(
    'DynamicClass',
    (BaseClass, ClassMixin),
    {'a': 42,
     '__repr__': lambda self: f'DynamicClass({self.a})',
     'inc': lambda self, n=1: self.a + n})
```

This statement is equivalent to creating a new class with the code:

```
class DynamicClass(BaseClass, ClassMixin):
    a = 42

    def __repr__(self):
        return f'DynamicClass({self.a})'

    def inc(self, n=1):
        return self.a + n

dyn_cls = DynamicClass
```

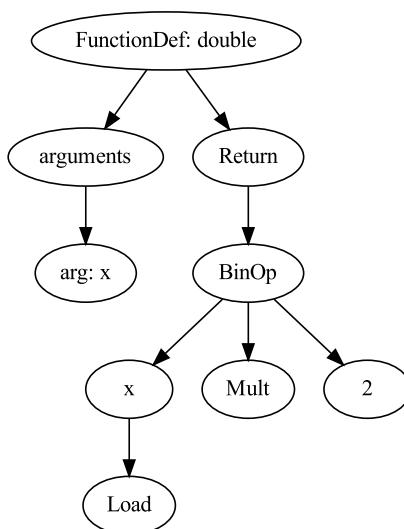
The `type` function, when used to dynamically create classes, is particularly useful for programmatically generating classes that have similar functionalities. The mean-field models in quantum chemistry are a such example, which we will demonstrate in Chapter 13. When Python classes are utilized to abstract the mean-field models, the differences between classes are often subject to minor features in certain attributes or methods. By combining attributes and methods, a new class for a specific mean-field model can be dynamically generated using the `type` function.

5.1.4 Python AST

The Python Abstract Syntax Tree (AST) is a tree representation of the structure of a Python program. Each node in this tree, such as functions, statements, assignments, function calls, and more, corresponds to a specific syntactic element of the program. For instance, the AST for the code

```
def double(x):
    return x * 2
```

is shown in Fig. 5.1. In this figure, the `FunctionDef` node denotes a function definition, and the `Return` node represents a return statement. You can find a comprehensive list of available node types in the Python AST documentation [2].

**FIGURE 5.1**

A simple Python AST.

The Python standard module `ast` offers a suite of tools for inspecting and modifying the AST of a Python program. Although it is possible to build an AST directly using the `Node` classes from the `ast` module, the more common approach is to modify the AST of existing functions and then compile the modified AST back into Python code. The `ast.parse` function can parse the source code of a function to generate an AST. The `ast.unparse` function does the reverse of `ast.parse()`, generating code string from an AST tree.

Analyzing or modifying an AST requires operations to traverse the tree. A simple approach is to use the `ast.walk` function, which is a general function that can traverse through all child nodes of a given node. However, if the task involves specific nodes or elements within the AST tree, the recommended approach is to use the `ast.NodeVisitor` class to traverse the tree, and the `ast.NodeTransformer` class for modification. By defining `visit_X` methods for these two classes, the custom functions will be invoked for nodes of type `X`. Here, `X` represents any node types in AST, such as `Assign`, `Call`, `Expr`, `Constant`, etc. Let's explore how to use `NodeVisitor` and `NodeTransformer` classes with a more realistic example.

Assuming that we want to develop a function that optimizes a Python program by removing unreachable code, the task can be implemented using the following algorithm:

1. Analyze the AST of the function to identify all dependencies between variables.
We can assume that the dependencies are introduced through the assignment statements.

2. The dependency mappings form a dependency tree, where the root is the return value of the function. The reachable variables are those leaves connected to the root node.
3. Identify all unconnected elements in the mappings. These are the unused variables.
4. Traverse the AST and remove any statements that involve the unused variables.
5. Compile the resulting AST into Python code using the `ast.unparse` and `exec` functions.

First, we use the `ast.NodeVisitor` class to search for dependency mappings associated with the assignment statements. To achieve this, we subclass `ast.NodeVisitor` with a `visit_Assign` method to inspect the `Assign` nodes. To collect all variables within each `Assign` node, we invoke the general tree traversing function `ast.walk()` and filter for the `ast.Name` node, as variable names are represented by the `ast.Name` nodes. The dependencies, which are mappings from one string to multiple strings, are stored in a dictionary.

```
import itertools, inspect, ast, textwrap

class DependenciesVisitor(ast.NodeVisitor):
    def __init__(self):
        selfdeps = {}
        selfreturns = []

    def visit_Assign(self, node):
        args = [x.id for x in ast.walk(node.value)
                if isinstance(x, ast.Name)]
        for t in node.targets:
            if isinstance(t, ast.Name):
                if t.id in selfdeps:                                     # (1)
                    selfdeps[t.id].extend(args)
                else:
                    selfdeps[t.id] = args

    def visit_Return(self, node):
        selfreturns.extend([
            [x.id for x in ast.walk(node.value) if isinstance(x, ast.Name)]
        ])
```

It should be noted that some variables may be assigned multiple times in the function, leading to the generation of dependency mappings on multiple occasions. To prevent new dependencies from overwriting existing ones, we have to verify whether the dependencies already exist in the dependency tree, as shown in line (1).

Dependencies can also arise from function calls, inplace operations, and exceptions. To address these scenarios, one can customize additional methods, such as

`visit_AugAssign` for inplace operations, `visit_Call` to manage function calls in case the subroutines modify the input variables, and `visit_Assert` and `visit_Raise` for try-exception statements. Handling these situations is not complicated. For simplicity, we will not delve into each of these cases in detail. You can refer to the AST documentation or seek assistance from tools like GPT or other AI coding assistants to generate the necessary `visit_` code.

After examining the entire AST, the dependencies of all variables are captured in the dictionary `DependenciesVisitor.deps`. Please note that some variables depend on others indirectly. For example, the mappings like `{'b': 'a', 'c': 'b'}` implies that the variable `c` depends on the variable `a`. To address this, we can trace the dependency chain to identify all indirect dependencies. During this tracing process, we may encounter circular dependencies among variables. To break the circular dependencies, as we proceed with the tracing, we continuously remove the accessed items from the dependency dictionary. The process of tracing dependencies is implemented in the following function `flatten_deps`, which identifies all necessary variables based on the provided dependency dictionary. Its output can be used to determine the unreachable variables. This function utilizes the technique of dynamic programming, which will be explained in Section 9.7.3 of Chapter 9.

```
def flatten_deps(keys, deps):
    if not keys or not deps:
        return []
    # Remove keys from deps to avoid potential circular dependency
    sub_deps = {k: v for k, v in deps.items() if k not in keys}
    flattened = [flatten_deps(deps[k], sub_deps)
                for k in keys if k in deps]
    return set(itertools.chain(keys, *flattened))
```

After obtaining the list of unreachable variables, we subclass `ast.NodeTransformer` to modify the AST. Within this class, we implement the `visit_Assign` method to eliminate the assignment statements that involve the unreachable variables. If a `visit_` method returns `None`, the corresponding node will be removed from AST.

```
class RemoveUnreachable(ast.NodeTransformer):
    def __init__(self, to_remove):
        self.to_remove = set(to_remove)

    def visit_Assign(self, node):
        if any(isinstance(t, ast.Name) and t.id in self.to_remove
               for t in node.targets):
            return None # Remove the node
        return node
```

Putting all components together, we can create the following function to optimize a given function.

```
def remove_unreachable_statements(fn):
    if callable(fn):
        fn = textwrap.dedent(inspect.getsource(fn))
        orig_code = ast.parse(fn)
        visitor = DependenciesVisitor()
        visitor.visit(orig_code)
        # Search for variables that are accessible from returns
        referenced = flatten_deps(visitor.returns, visitor.deps)
        unreachable = set(visitor.deps).difference(referenced)

        filtered = RemoveUnreachable(unreachable).visit(orig_code)
    return ast.unparse(filtered)
```

Please note that the `ast.parse` function can only parse the code string of a function, rather than the function object itself. To address this problem, we use the `inspect.getsource` function to retrieve the source code of the input function.

The Python standard library `inspect` is a powerful tool for code generation tasks. Although it cannot directly modify or generate code, `inspect` provides tools for accessing the source code, the signature, argument annotations, local variables, and many other details of a function.

Let's exam the outcome of the `remove_unreachable_statements` function

```
In [1]: def fun1(x, y):
    a = x + 2
    b = a * x
    c = y * 9 - b
    d = f(b)
    b = d
    return b * a
print(remove_unreachable_statements(fun1))

Out[1]:
def fun1(x, y):
    a = x + 2
    b = a * x
    d = f(b)
    b = d
    return b * a

In [2]: def fun2(x, y):
    a = x + 2
    b = a * x
    a, b = a + b, a - b
    c = f(y)
    d = c
```

```
        return a
    print(remove_unreachable_statements(fun2))
Out[2]:
def fun2(x, y):
    a = x + 2
    b = a * x
    a, b = a + b, a - b
    return a
```

Based on the `remove_unreachable_statements` function, we can develop a decorator to transform function objects. This involves using `exec` to compile the output of `remove_unreachable_statements` within a local namespace.

```
def optimize(fn):
    '''A decorator, which can remove unreachable statements of a function
    remove_unreachable_statements'''
    new_fn = remove_unreachable_statements(fn)
    new_code = ast.parse(new_fn)
    # Remove the line containing the decorator from the AST
    new_code.body[0].decorator_list = []

    local_ns = {}
    exec(ast.unparse(new_code), None, local_ns)
    return local_ns[fn.__name__]
```

There is no direct method to convert an AST back into a function object. Using `exec` to compile the AST output is a common practice. This action generates a new function with the same name as the input function but is registered in the local namespace. However, when the AST code is invoked within the decorator, the decorator itself is also included in the generated function, which can lead to a circular dependency. To avoid this, we clear the `decorator_list` of the function before compiling the new code.

5.2 Symbolic computation

Symbolic computation is frequently utilized for deriving formulas. The Python package SymPy [3] facilitates symbolic mathematics for tasks such as differentiation, integration, algebraic manipulation, solving equations, and handling complex functions, etc.

Symbols are the core elements of symbolic computation. In SymPy, class `sympy.Symbol` is the fundamental data structure to represent symbolic variables. A symbolic variable is labeled by a string name. The naming rules for symbols follow those of Python variable names, which can consist of alphabet letters, numbers, and underscores. Besides instantiating symbolic variables with the `sympy.Symbol` class,

the `sympy.symbols` function is often used for creating multiple symbolic variables at once. For instance, the following statements are all valid methods for defining symbolic variables:

```
x, y, z = sympy.symbols('x y z')
x, y, z = sympy.symbols('x,y,z')
x, y, z = sympy.symbols('x y,z')
x, y, z = sympy.symbols(['x', 'y', 'z'])
x0, x1, x2, x3, var1, var2 = sympy.symbols('x(0:2)(2:4) y z')
```

When calling `sympy.symbols`, names for different variables are separated by commas or whitespaces. To reduce typing effort, `sympy.symbols` also supports the range syntax for generating a list of symbols. In the example above, the slices `0:2` and `2:4` within `x(0:2)(2:4)` are expanded individually, resulting in the creation of six variables.

Besides the `Symbol` class, another core data structure in SymPy programs is symbolic expressions, provided by the `sympy.Expr` class. Symbolic expressions are composed of symbols, numbers, and mathematical functions defined on symbols.

To represent a collection of symbols or expressions, the `sympy.Matrix` class is used for representing matrices, and the `sympy.Array` class is available for multi-dimensional arrays. Some methods provided by the `sympy.Matrix` and `sympy.Array` classes, such as transposing, reshaping, and indexing, are somewhat similar to those operations for NumPy arrays. However, most of them differ from those in NumPy. For example, `sympy.Matrix` does not support ufuncs for element-wise operations. When calling `a * b` between two matrices, it performs the standard matrix multiplication.

```
In [1]: a = sympy.Matrix(sympy.symbols('x(:2)(:3)').reshape(2,3))
         a.T * a
Out[1]:
Matrix([
[x00**2 + x10**2, x00*x01 + x10*x11, x00*x02 + x10*x12],
[x00*x01 + x10*x11, x01**2 + x11**2, x01*x02 + x11*x12],
[x00*x02 + x10*x12, x01*x02 + x11*x12, x02**2 + x12*x12]])
```

The `.subs()` method of a symbolic expression can be used to substitute, evaluate, transform or simplify symbolic expressions. It works by replacing symbols based on the rules defined in a dictionary. For instance, the following example demonstrates how to use the `.subs()` method to transform polynomial expressions according to the Affine transformation.

```
In [2]: x, y, z = sympy.symbols('x y z')
P = .5 * x**2 + .5 * y**2
T = sympy.Matrix([[.7, .1, 0., 0.],
                 [0., .9, -.2, .5],
                 [.3, 0., .8, 0.]])
```

```
xprime, yprime, zprime = T * Matrix([x, y, z, 1])

Ptransformed = P.subs({x: xprime, y: yprime, z: zprime},
                      simultaneous=True)
print(Ptransformed.expand())
print(Ptransformed.subs({x: 1.0, y: 2.0, z: 0.5}))

Out[2]:
0.245*x**2 + 0.07*x*y + 0.41*y**2 - 0.18*y*z + 0.45*y + 0.02*z**2 - 0.1*z +
0.125
2.825000000000000
```

When an Affine transformation is applied to the axes, the polynomials are transformed accordingly. Please note that some symbols in the substitution, such as x , y , and z , also appear in the substitution targets. The default substitution process applies the replacement rules sequentially. It leads to symbols from earlier replacements being substituted again in later steps. To avoid this, we use the option `simultaneous=True`, which ensures that all substitutions are carried out simultaneously.

On top of symbol variables and expressions, SymPy provides a comprehensive suite of features for symbolic mathematics. While a detailed discussion of each feature is beyond the scope of this book, we will provide an overview of the key features in SymPy. Readers are encouraged to consult the SymPy documentation for more detailed information on specific features.

- *Simplifying and expanding expressions.* This functionality is offered by the general `sympy.simplify` function. For polynomials, additional features include:
 - `sympy.factor`
 - `sympy.expand`
 - `sympy.collect` (which collects common powers of a term in an expression).Moreover, SymPy also provides simplification and expansion capabilities for trigonometric operations, exponentials and logarithms, and certain special functions, including:
 - `trigsimp`
 - `besselsimp`
 - `gammasimp`
 - `hypersimp`
- *Differentiation of expressions.* Differentiation can generally be performed using the `sympy.diff` function and the `sympy.Derivative` class. For symbolic expressions represented in matrices, SymPy can also compute the Jacobian matrix. This functionality is only applicable to `Matrix` objects and is provided by the `Matrix.jacobian` method.
- *Definite and indefinite integrals.* Integration is facilitated by the `sympy.Integral` class or the `sympy.integrate` function. Indefinite integrals are computed with

```
sympy.integrate(expr, variables)
```

and definite integrals are computed with

```
sympy.integrate(expr, (variable, lower_limit, upper_limit))
```

When specifying the integral limits, the variable `sympy.oo` in SymPy represents ∞ . For the integration of multiple variables, multiple limits can be passed to the `sympy.integrate` function. SymPy also offers common integral transforms. These functionalities can be found in the `sympy.integrals` module, including:

- `fourier_transform`
- `hankel_transform`
- `laplace_transform`
- `mellin_transform`

and their corresponding inverse transforms

- `inverse_fourier_transform`
- `inverse_hankel_transform`
- `inverse_laplace_transform`
- `inverse_mellin_transform`

- *Computing the limit of an expression or a sequence.* These features are associated with the `sympy.limit` and `sympy.limit_seq` functions.
- *Series expansion for a function near an expansion point.* This functionality is provided by the `sympy.series` function.
- *Solving equations.* The general-purpose function `sympy.solve` is useful for solving a variety of equations, including polynomial, transcendental, and linear equations. SymPy also offers specialized solvers for each type of equation. The `sympy.solveset` function is designed to find the finite set or intervals for inequalities or equations. For linear equations, `sympy.linsolve` is the dedicated function for solving them in matrix form. For non-linear equations, `sympy.nonlinsolve` is available to solve them in matrix form. Additionally, `sympy.nsolve` can be used for numerically solving non-linear equations.
- *Ordinary differential equations (ODE).* To solve ODEs, the `sympy.dsolve` function is used in conjunction with the `sympy.Eq` and `sympy.Function` classes. For instance, the differential equation

$$xf'(x) = \cos(x) \quad (5.1)$$

can be translated into the SymPy code

```
f = sympy.Function('f')
eq = sympy.Eq(x*f(x).diff(x), sympy.cos(x))
sympy.dsolve(eq, f(x))
```

To indicate the differentiation of a function, both the `.diff` method and the `sympy.Derivative` class are available for use.

- *Partial differential equations (PDE).* The `sympy.pdsolve` function can be the starting point for PDEs. However, solving PDEs might not be as straightforward as solving ODEs. Often, it is necessary to use functions from the `sympy.solvers.pde`

module, such as `classify_pde`, `pde_separate`, `pde_separate_add`, `pde_separate_mul` to provide hints for the PDE solving process.

- *Functionalities for specific mathematical domains.* SymPy includes modules specifically designed for various mathematical subjects, such as category theory, cryptography, differential geometry, geometry, holonomic functions, Lie algebra, polynomial manipulation, and statistics. The SymPy online documentation includes dedicated sections for the functionalities in these fields.

- A wide range of *special functions and polynomials*. Special functions such as

- `sympy.erf`
- `sympy.hyper`
- `sympy.bessel`
- `sympy.gamma`
- `sympy.functions.special.spherical_harmonics`

and many others can be found in the `sympy.functions` and `sympy.functions.special` modules. Special polynomials, such as

- `hermite_poly`
- `legendre_poly`
- `chebyshev_poly`

are provided by the `sympy.polys` module.

- *Function plotting.* Function plotting is facilitated by the `sympy.plot` function and the `sympy.plotting` module. The `sympy.plot` function is built on Matplotlib. It offers a user experience comparable to that of using the Matplotlib library.
- *Code generation.* The `sympy.printing` module provides functions to generate various types of programming code, such as `ccode()`, `fcode()`, and `pycode()`, and `julia_code()`. Additionally, SymPy can convert expressions into LaTeX equations using `sympy.latex()`.

Besides the functionalities for mathematics, we can find various symbolic features for quantum physics in SymPy:

- *Solutions for regular quantum systems*, such as
 - The `sympy.physics.sho` and `sympy.physics.qho_1d` modules for the harmonic oscillator.
 - The `sympy.physics.hydrogen` module for the hydrogen atom.
- *Clebsch-Gordan and Wigner coefficients.* The `sympy.physics.wigner` module collects functions, such as `clebsch_gordan`, `racah`, `wigner_3j`, `wigner_6j`, `wigner_9j`, and `wigner_d`.
- The `sympy.physics.quantum.spin` module for *angular momentum coupling and manipulation*. The available functionalities include:
 - The `WignerD` class for the rotation matrices.
 - The `J2` class for the \hat{J}^2 operator.
 - The `Jx`, `Jy`, and `Jz` classes for angular momentum along the x , y , and z axes.
 - The `Jplus` and `Jminus` classes for the \hat{J}_+ and \hat{J}_- operators.
- The `sympy.physics.secondquant` module for *second quantization*. Common second quantized operators and operations can be found in this module:

- The `BosonicOperator` and `FermionicOperator` classes for bosonic and fermionic operators, respectively.
- The `Annihilator` and `Creator` classes for annihilation and creation operators.
- The `NO` class for many-body operations expressed in normal order.
- The `wicks` function for deriving the normal order representation for an expression.
- *Quaternion algebra* via the `sympy.Quaternion` class.

The SymPy package can handle most scenarios that require the derivation of symbolic formulas. To more precisely manipulate symbolic expressions for domain-specific problems, occasionally, we may want to develop symbolic programs with custom rules from scratch. As an example, in Chapter 15, we will demonstrate how to customize symbolic programs for Coupled Cluster theory.

5.3 Automatic differentiation

Automatic differentiation (AD) might seem magical at first glance. However, the principle behind AD is not complicated. When applying AD to a function, the input is wrapped in a way that every operation performed on the input is overloaded. As the function is evaluated, the overloaded operations create a computational graph to track how the inputs are transformed. Differentiation rules are then applied to each node within this graph. By traversing the entire graph, one can obtain the derivative of the function. This procedure is closely related to the technique of lazy evaluation, which will be described in more details in Section 9.8 of Chapter 9.

The AD technique is a cornerstone in the development of machine learning and neural networks. Certain terminologies from the neural network domain have become common concepts in AD, including:

- *Forward Propagation*. This process executes the operations of a function in the order they are defined, from the input to the output.
- *Backpropagation*. This is the process of computing the derivatives of a function with respect to its inputs. Starting from the function's output, chain rules are recursively applied to compute the derivatives of each intermediate operation with respect to its inputs.
- *Forward-mode differentiation*. This mode defines the response of the output for perturbations of input.
- *Reverse-Mode differentiation*. This method is employed in backpropagation. It traverses the computational graph in reverse order, computing the derivatives for each intermediate node.

Several neural network tools are available for performing automatic differentiation, including PyTorch [4], JAX [5], TensorFlow [6], and autograd, etc. PyTorch and JAX, in particular, are widely used for AD applications.

5.3.1 PyTorch

PyTorch has a low learning barrier, as it operates much like the standard Python object-oriented programming style. Let's consider a simple problem of computing Coulomb energy to demonstrate the AD capabilities of PyTorch. Given a group of points with their coordinates \mathbf{r} and charges Z , the Coulomb energy can be calculated as

$$E = \sum_{i>j} \frac{Z_i Z_j}{|\mathbf{r}_i - \mathbf{r}_j|}. \quad (5.2)$$

This function can be implemented using NumPy broadcasting rules.

```
def coulomb_energy(coords: torch.Tensor, z: torch.Tensor):
    zz = z[:,None] * z
    rr = coords[:,None,:,:] - coords
    d = torch.linalg.norm(rr, axis=2)
    tril = np.tril_indices(len(z), -1)
    return (zz[tril] / d[tril]).sum()
```

To enable PyTorch AD, we replaced `np.linalg.norm` with `torch.linalg.norm`. Most NumPy functions have their equivalents in PyTorch.

PyTorch offers several methods to access the derivatives of a function with respect to its input variables. One method is to use JAX-like functions provided by the `torch.func` module. For example, the `torch.func.grad` and `torch.func.hessian` functions are available for computing gradients and Hessians, respectively.

```
In [1]: coords = torch.rand(2, 3)
        Z = torch.rand(2, 3)
        torch.func.grad(coulomb_energy, argnums=0)(coords, Z)

Out[1]:
tensor([[ 0.1631,  2.6309, -1.6224],
        [-0.1631, -2.6309,  1.6224]], grad_fn=<AddBackward0>)

In [2]: torch.func.hessian(coulomb_energy, argnums=1)(coords, Z)
Out[2]:
tensor([[0.0000, 2.3221],
        [2.3221, 0.0000]], grad_fn=<ViewBackward0>)
```

The `argnums` parameter specifies the position of the arguments to be differentiated with AD. Please note that the input variables for AD must be `torch.Tensor` objects.

The second method employs back-propagation to generate gradients for each individual variable. In this method, it is necessary to specify the keyword `requires_grad=True` when creating input variables. It indicates that the gradients of these variables should be computed during back-propagation. To calculate the gradients, we start with a forward propagation to compute the result. We then call the `.backward()` method on the result to perform back-propagation. After the two steps, the gradients can be accessed through the `.grad` attribute of the input variables. For example,

```
In [3]: coords = torch.rand(2, 3, requires_grad=True)
        Z = torch.rand(2, 3, requires_grad=True)
        e = coulomb_energy(coords, Z)
        e.backward()
        print(coords.grad)
        print(Z.grad)

Out[3]:
tensor([[ 0.1631,  2.6309, -1.6224],
        [-0.1631, -2.6309,  1.6224]])
tensor([[ 0.0000,  2.3221],
        [ 2.3221,  0.0000]])
```

Please note that the `.backward()` method introduces side effects to the differentiable parameters. The back-propagation process executes AD and updates the gradients of input variables inplace. When the same variables are accessed by the outputs of multiple functions, invoking the `.backward()` method for each output leads to the gradients being accumulated in the same variable, rather than retaining only the most recent one.

```
In [4]: x = torch.tensor(1., requires_grad=True)
        y = x**2
        y.backward()
        print(x.grad)

Out[4]: tensor(2.)

In [5]: y = x * 3
        y.backward()
        print(x.grad)

Out[5]: tensor(5.)
```

To avoid the accumulation of gradients, we manually clear the gradients by setting the `.grad` attribute to `None` or a zero vector.

```
In [6]: x.grad = None
        y = x * 3
        y.backward()
        print(x.grad)

Out[6]: tensor(3.)
```

If we need to customize the gradients of certain functions, we can subclass the `torch.autograd.Function` class to modify the derivative rules. Within the subclass, the `forward` and `backward` methods are supplied to define the forward and backward passes. For example, assume we aim to implement the following differentiation for the Coulomb energy function,

$$\frac{\partial E}{\partial x_k} = \sum_{j \neq k} -\frac{Z_k Z_j (\mathbf{r}_k - \mathbf{r}_j)}{|\mathbf{r}_k - \mathbf{r}_j|^3} \quad (5.3)$$

$$\frac{\partial E}{\partial Z_k} = \sum_{j \neq k} \frac{Z_j}{|\mathbf{r}_k - \mathbf{r}_j|}, \quad (5.4)$$

we can implement the subclass as follows.

```
class CoulombEnergy(torch.autograd.Function):
    @staticmethod
    def forward(ctx, coords, z):
        zz = z[:,None] * z
        rr = coords[:,None,:,:] - coords
        d = np.linalg.norm(coords[:,None,:,:] - coords, axis=2)
        tril = np.tril_indices(len(z), -1)
        res = (zz[tril] / d[tril]).sum()
        # ctx.save_for_backward can cache info in ctx for backward
        # computation. The saved info can be accessed in the
        # .saved_tensors attribute.
        ctx.save_for_backward(coords, z)
        return res

    @staticmethod
    def backward(ctx, grad_output):
        coords, z = ctx.saved_tensors
        zz = z[:,None] * z
        rr = coords[:,None,:,:] - coords
        d = np.linalg.norm(coords[:,None,:,:] - coords, axis=2)
        with np.errstate(all='ignore'):
            rinv = 1. / d
            rinv[np.diag_indices(len(z))] = 0.
        r_grad = -np.einsum('i,j,ijx,ij->ix', z, z, rr, rinv**3)
        z_grad = np.einsum('j,ij->i', z, rinv)
        # grad_output represents the factor associated with the current
        # function.
        return grad_output * r_grad, grad_output * z_grad

    if __name__ == '__main__':
        coords = torch.rand(2, 3, requires_grad=True)
        Z = torch.rand(2, 3, requires_grad=True)
        e = CoulombEnergy.apply(coords, Z)
        e.backward()
        print(coords.grad)
        print(Z.grad)
```

To utilize this subclass, we simply need to call the `apply` method according to the function signature of the `forward` method to execute forward propagation. Next, similar to handling regular PyTorch operations, we invoke the `.backward()` method and access the `.grad` attribute of the output of the forward propagation to retrieve the results of custom gradients.

5.3.2 JAX

The JAX library consists of two main components: the `jax` package which focuses on higher-level APIs for numerical computation and AD operations, and the `jaxlib` package which provides the low-level bindings for hardware-specific computational acceleration. The two components are distributed as separate libraries. Both should be installed to enable JAX’s computational capabilities. If the goal is to explore the functionality of JAX AD, you can install only the CPU backend for simplicity.

```
$ pip install jax[cpu] jaxlib
```

If you wish to enable the backends for Nvidia or AMD GPUs, or use a TPU in a Google Colab notebook, you should install the corresponding JAX backends. More details can be found in JAX documentation [7].

The JAX library is designed with the functional programming paradigm. JAX’s AD is achieved through the use of high-order functions. Compared to PyTorch, new users may need more efforts to become familiar with JAX programs.

To align with the functional programming style, it is generally recommended to implement functions that do not produce side effects. When transitioning from a NumPy program to a JAX function, you may need to update the NumPy code to incorporate the following practices:

- Avoiding modifications to individual elements within an array.
- Using regular assignment statements instead of in-place operations.
- Ensuring that input variables are treated as immutable.

When applying JAX’s AD to a function, it is crucial to assess whether the function is compatible with *JAX transformations*. The fundamental of JAX’s AD system is the transformation between the forward and reverse differentiation modes [8]. To enable JAX’s AD, functions or operations must be JAX-transformable. Most standard arithmetic operations in Python are JAX transformable. However, NumPy functions do not comply with this JAX-transformable property. Given the significant role of NumPy in scientific computations, JAX introduces the `jax.numpy` module,

```
import jax.numpy as jnp
```

to serve as a substitute for the NumPy library in JAX’s AD applications. When a function is entirely written using `jnp`, it is considered JAX-transformable. In most situations, we can begin with a NumPy program and simply replace the `np` module with `jnp`. This might produce a JAX-compatible program.

For the Coulomb energy problem, to ensure JAX-transformability, we have to use an alternative implementation for the `coulomb_energy` function, where excludes the diagonal terms of the `rr` array before calling the `norm` function:

```
def coulomb_energy(coords, z):
    zz = z[:,None] * z
    rr = coords[:,None,:,:] - coords
    # Note zeros are not differentiable for norm.
    # They are discarded before calling norm.
    tril = np.tril_indices(len(z), -1)
    d = jnp.linalg.norm(rr[tril], axis=1)
    return (zz[tril] / d).sum()
```

If we reuse the implementation of the `coulomb_energy` function from the previous section, JAX will return `nan` for gradients. This problem arises from the zeros in the diagonal terms of the `rr` matrix, which causes illegal values when processing the gradients of `jnp.linalg.norm` function. We will not investigate the differences between PyTorch and JAX in the handling of the `norm` function here. Readers just need to be aware of the issue with `jnp.linalg.norm` when encountering `nan` values in gradients.

Now, we can use the `jax.grad` function to compute gradients.

```
In [7]: coords = jnp.array(np.random.rand(2, 3))
Z = jnp.array(np.random.rand(2))
jax.grad(coulomb_energy, argnums=(0, 1))(coords, Z)
Out[7]:
(Array([[-0.03555296, -0.17780247,  0.02858889],
       [ 0.03555296,  0.17780247, -0.02858889]], dtype=float32), Array
([0.58357096, 0.31454977], dtype=float32))
```

The `jax.grad` function is designed to compute gradients for scalar-valued functions. If a function outputs arrays, JAX offers the `jax.jacobian` function to calculate the Jacobian matrix.

```
In [8]: jax.jacobian(coulomb_energy, argnums=0)(coords, Z)
out[8]:
[[-0.03555296 -0.17780247  0.02858889]
 [ 0.03555296  0.17780247 -0.02858889]]
```

Additionally, JAX provides the `jax.jacfwd` and `jax.jacrev` functions for computing the Jacobian matrix. These two functions offer more refined control over the calculation of the Jacobian.

- `jax.jacfwd` calculates the Jacobian using forward-mode AD. This method is efficient for functions that have many inputs and fewer outputs.
- `jax.jacrev` employs reverse-mode AD. It is more suitable for functions with many outputs and fewer inputs.

The `jax.jacobian` function, by defaults, invokes the `jac.jacrev` version. In the case of higher-order derivatives, the `jax.hessian` function is available for second-order derivatives. It essentially acts as a wrapper for the nested Jacobian operations, equivalent to `jax.jacfwd(jax.jacrev(f))`.

It is worth mentioning that in many scenarios, it is not necessary to construct an explicit gradients vector (or Jacobian matrix). Often, we only need to compute the product between the Jacobian and certain displacements. For instance, in gradient descent optimization for a loss function, the objective is to evaluate the change of the loss function in response to a small displacement in the parameter space. This procedure is essentially equivalent to calculating the product of the Jacobian and the displacement vector. To facilitate these product operations, JAX provides the JVP (Jacobian-vector product) function `jax.jvp` and VJP (vector-Jacobian product) function `jax.vjp`.

Given a Python function that computes the value of $f(x)$, `jax.jvp` can evaluate both $f(x)$ and the product $\partial f(x) \cdot v$, whereas `jax.vjp` produces $f(x)$ and $v \cdot \partial f(x)$. Taking the `coulomb_energy` function as an example, we can use `jax.jvp` to model the change in energy caused by the changes in charges. The inputs of the `jax.jvp` function include the function $f(x)$, the primal values at which $\partial f(x)$ is evaluated, and the tangent vector v for the Jacobian-vector product. The *primals* and *tangents* are both tuples that correspond to the positional parameters of $f(x)$. For instance, the code below shows how the `jvp` operation for the Coulomb energy function is invoked.

```
In [9]: coords = jnp.array(np.random.rand(2, 3))
Z = jnp.array(np.random.rand(2))
disp = jnp.array(np.random.rand(2))
fun = lambda z: coulomb_energy(coords, z)
primals = (Z,)
tangents = (disp,)
e, e_tangents = jax.jvp(fun, primals, tangents)
```

Customizing JAX derivatives involves the use of the JAX functions `custom_jvp` or `custom_vjp` to define the forward-mode or reverse-mode derivative rules, respectively. The JAX derivative customization utilizes a different programming model than the direct implementation of `.forward()` and `.backward()` methods in PyTorch. In Chapter 16, we will employ the Coupled Cluster method as an example to illustrate how JVP and VJP can be customized for gradient computation.

Summary

In this chapter, we reviewed the use of `eval` and `exec` for code generation. We discussed the application of the Abstract Syntax Tree (AST) for analyzing and modifying Python code. For symbolic computation tasks, we provided a brief overview of the functionalities of the SymPy package. Furthermore, we demonstrated the basics of the automatic differentiation technique, powered by PyTorch and JAX.

Meta-programming code can be difficult to understand. Given its flexibility and the capability for runtime modification, improper or incorrect use can lead to unexpected errors that are difficult to debug. While it is beneficial to learn how to use meta-programming, one should not abuse this technique.

References

- [1] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, O'Reilly Media, 2015, <https://books.google.co.in/books?id=bIZHCgAAQBAJ>.
- [2] Python Software Foundation, Python documentation - abstract syntax trees (ast), <https://docs.python.org/3/library/ast.html#abstract-grammar>, 2024.
- [3] A. Meurer, C.P. Smith, M. Paprocki, O. Čertík, S.B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J.K. Moore, S. Singh, T. Rathnayake, S. Vig, B.E. Granger, R.P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M.J. Curry, A.R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, Sympy: symbolic computing in python, PeerJ Computer Science 3 (2017) e103, <https://doi.org/10.7717/peerj-cs.103>.
- [4] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Beillard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C.K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M.Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, S. Chintala, Pytorch 2: faster machine learning through dynamic python bytecode transformation and graph compilation, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 929–947, <https://doi.org/10.1145/3620665.3640366>.
- [5] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python+NumPy programs, <http://github.com/google/jax>, 2018.
- [6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: large-scale machine learning on heterogeneous systems, software available from <https://www.tensorflow.org/>, 2015.
- [7] The JAX authors, Installing jax, <https://jax.readthedocs.io/en/latest/installation.html>, 2024.

- [8] The JAX authors, Custom derivative rules for jax-transformable python functions, https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html, 2024.

Input and output

6

The terminology I/O (Input and Output) often refers to the communication between the core components of a computer (CPU and memory) and the supporting devices, such as storage disks and networks.

I/O involve a broad spectrum of technical challenges in computational programs. Significant software engineering efforts have been investigated to address the issue of I/O performance and data availability. Although I/O technology may not appear to be at the core of quantum chemistry applications, we actually frequently encounter challenges related to I/O operations, such as file reading and writing, data exchange, data format conversion, etc. For example, you might have come across the following situations and questions:

- There is a large NumPy array that I need to save to a file. What storage formats are available? How should I choose among them?
- I use pickle to save Python objects. Occasionally, this process results in a `PicklingError`. What could be the causes of this issue?
- I would like to use JSON format to store a NumPy array but JSON does not support this. What can I do?
- I redirected the output of my program to a file. However, I cannot see any contents in the file when the program is running. Why does this happen?
- My program crashed. I tried to continue the computation using the intermediate files it generated. However, sometimes the file is corrupted. What could be the reason?

After reading this chapter, you shall get some ideas for these questions. In addition to the technical issues related to files and data formats, this chapter will also explore technologies associated with network I/O. In the context of quantum chemistry programs, network I/O may be involved when developing applications that run in parallel. Network I/O is also a crucial foundation for cloud computing technologies, which will be discussed in Chapter 7. If network I/O seems complex to you, you may choose to focus only on its concepts. Regarding the tools involved, if they are essential in the work, the AI coding assistant can be our support. In fact, we will extensively leverage AI in this chapter. We will demonstrate how AI tools participate in the design and development of network I/O programs.

6.1 Serialization

Data serialization and I/O are closely related. Serialization is the process that converts complex data objects in memory into a stream representation. This process is usually performed before data can be saved to a file or sent over a network. Deserialization is the reverse process, where objects are reconstructed from the formatted data stream. This process is necessary when data is retrieved from a file or received from a network.

When addressing I/O scenarios, several technical aspects should be considered in data serialization, including:

- The choice between a binary format or a human-readable format. Human-readable data is easier to understand but tends to consume more storage space and require a higher processing overhead.
- Cross-platform compatibility. Errors may occur when data is deserialized on platforms of different architectures, or software of different versions.
- Whether data is serializable. Attempting to serialize an arbitrary object in Python may result in a serialization failure. JSON serialization is a typical example that comes with many serialization limitations.
- Whether serialization can faithfully represent the data. Some serialization schemes might alter the data format or precision in order to reduce the data size or simplify the process.
- The overhead of serialization and deserialization. Certain methods, such as Pickle, require additional CPU resources and memory space for serialization and deserialization, which can lead to non-negligible overhead in high-performance applications.

There are a variety of data serialization schemes available in Python. Among them, Pickle is undoubtedly the best option for general Python objects. It is highly compatible with the Python language and supports most Python objects. However, when dealing with arrays or network I/O, Pickle may not be the best option. Other serialization schemes can offer better performance in these scenarios. They will be explored in Sections 6.2 and 6.5.

Pickle offers a binary serialization solution. In some scenarios, data is better suited to a human-readable serialization approach. Among the available options, JSON (JavaScript Object Notation) is the most popular choice. Despite having originated from JavaScript, JSON is language-independent and extensively used for data exchange between different programming languages. YAML is another popular format, especially for storing the configuration data. Other options include CSV, which is ideal for serializing tables or matrices, and XML, which is more complex but offers more customization capabilities.

When data is serialized in a human-readable format, the processing overhead is typically much higher than that of binary serialization. Although high-performance JSON libraries are available, their functionality is generally more limited compared to the built-in JSON module in Python.

6.1.1 Pickle

In Python, Pickle serialization is provided by the built-in `pickle` module. Several functions of the `pickle` module are frequently used, including `pickle.dump` and `pickle.dumps` for serialization, along with `pickle.load` and `pickle.loads` for deserialization. The `pickle.dumps` function can serialize a Python object into a `bytes` object, which can then be deserialized using `pickle.loads`. On the other hand, `pickle.dump` and `pickle.load` work directly with file objects. They do not need additional memory to hold the serialized `bytes` object.

```
In [1]: import pickle
        with open('file.pkl', 'wb') as f:
            pickle.dump(obj, f)
        with open('file.pkl', 'rb') as f:
            obj = pickle.load(f)

In [2]: bytes_stream = pickle.dumps(obj)
        obj = pickle.loads(bytes_stream)
```

The basic usage of Pickle is generally straightforward, as demonstrated by the examples above. Next, let's examine potential issues in pickle serialization.

What can be pickled?

According to the official documentation [1], objects that can be pickled include:

- Python built-in constants, such as `None`, `True`, `False`;
- Numbers, strings, `bytes`, `bytearray` objects;
- Tuples, lists, sets, and dictionaries (they must only contain picklable objects);
- Built-in functions;
- Functions and classes defined at module level.

Besides, array and array-like objects provided by NumPy, as well as NumPy-like libraries such as Pandas, PyTorch, Jax, and Cupy, are picklable. Additionally, an instance of a class is picklable if all of its attributes are picklable. In most scenarios, extra code is not required to make such an object picklable. When serializing an object, `pickle` can automatically handle the pickling of its attributes in a recursive manner.

Ensuring objects are picklable is crucial during I/O processes, especially in the context of parallel computation, where all input arguments and outputs must be pickled for data exchange. In practice, we may encounter many objects in a parallel program that are *not picklable*, such as:

- Closures and `lambda` functions.
- A file object, such as the one returned by the `open` function.
- A generator object, produced by a generator expression [2] or the `yield` statement.
- A future object, created by the `concurrent.futures` module.
- An iterator object.
- ...

What can we do if an I/O process involves these unpickleable objects?

One strategy is to replace them with objects that are pickleable. For example, if we want to distribute a closure for remote execution, we can use `functools.partial` to produce a replacement for the closure. `functools.partial` can create a normal function with bounded parameters, making it a pickleable object.

```
In [3]: def pow_n(n):
    def partial_pow(m):
        return pow(m, n)
    return partial_pow
pickle.dumps(pow_n(3))
AttributeError: Can't pickle local object 'pow_n.<locals>.partial_pow'

In [4]: def pow_n(n):
    return functools.partial(pow, exp=n)
pickle.dumps(pow_n(3))
Out[4]:
b'\x80\x04\x95\x00\x00\x00\x00\x00\x00\x00\x00\x8...'
```

Given the widespread use of closures and lambda functions in distributed computing, several third-party serialization schemes have been developed to support them, such as `cloudpickle` [3] and `dill` [4]. Unlike Python's built-in `pickle` module which serializes objects by reference, these libraries can serialize a function or a class by value. *Serialization by reference* means that functions and classes are treated as attributes of modules, which must be available or importable at load time. In contrast, *serialization by value* also encodes the body of a closure during the serialization process, thus compromising somewhat on serialization efficiency.

Another strategy to handle the unpickleable instances is to customize serialization methods [5] for the class. If an instance cannot be pickled due to certain unpickleable attributes, we can implement the `__setstate__` and `__getstate__` methods to customize serialization and deserialization, excluding them from the I/O process.

Cross-platform compatibility

Is pickle serialization compatible across different platforms, different Python versions?

Pickle is cross-platform compatible. However, pickle files created with different protocol versions are not compatible with each other. Python introduced several versions of Pickle protocols in its early versions. Transferring pickle files between different Python versions could potentially cause compatibility issues on old platforms. Fortunately, since Python 3.8, the default pickle protocol (accessible by `pickle.DEFAULT_PROTOCOL`) has been standardized to version 4. Given the fact that Python 3.7 has reached its end-of-life, the pickle compatibility across different Python versions is generally not a concern.

Software version dependency

Although the Pickle protocol has been stabilized in recent Python releases, it does not solve the compatibility issue of pickle within third-party packages. Objects from the external libraries may define unique pickle serialization rules. These rules may vary between versions, and potentially cause compatibility issues. For example, the serialization method for Pandas DataFrame has been adjusted in versions 1.1, 1.3, and 2.1. DataFrame objects pickled by a new version of Pandas might not be deserializable in an older Pandas version. To address the issue of serialization compatibility for third-party libraries, we can either stick with a specific version of the third-party library or avoid using pickle for I/O. Without relying on pickle, we can customize serialization methods using more stable formats, such as JSON or HDF5.

Security problem

Python pickle is known to be vulnerable, so it is recommended to only unpickle data that comes from trusted sources.

6.1.2 JSON

The JSON document is very much like a Python object with nested dictionaries and lists. We will not detail the specifications for the JSON format here, as they can be easily found elsewhere.

Python has a built-in module for processing JSON document, which provides serialization functions including `json.dump` and `json.dumps`, along with deserialization functions `json.load` and `json.loads`. Their usage is similar to the corresponding functions in the `pickle` module:

```
In [5]: import json
        with open('file.json', 'w') as f:
            json.dump(obj, f)
        with open('file.json', 'r') as f:
            obj = json.load(f)

In [6]: json.dumps({'a': [2, 3]})
Out[6]:
'{"a": [2, 3]}'

In [7]: json.loads('{"a": [2, 3]}')
Out[7]:
{'a': [2, 3]}
```

JSON is not a native data format for Python. Consequently, there are certain limitations when serializing Python objects into JSON format.

JSON format only supports a subset of Python built-in types, limited to `None`, `True`, `False`, numbers (integers and real numbers), strings, tuples, lists, and dictionaries. It is not possible to directly serialize Python functions or classes using JSON format.

To enable serialization of a wider array of Python objects in JSON, the `json` module provides APIs to customize the JSON encoder and decoder.

For instance, JSON does not directly support the serialization of NumPy arrays. To work around this, we can convert a NumPy array into a nested list and add a unique tag to mark the converted array. This process can be implemented by extending the `JSONEncoder` class with a custom `default` method. When decoding a custom node in a JSON document, we can identify the tag and then use NumPy functions to reconstruct the array.

```
class NpArrayEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.ndarray):
            return {'python/ndarray': obj.tolist()}
        return super().default(obj)

    def np_decoder(obj):
        if 'python/ndarray' in obj:
            return np.array(obj['python/ndarray'])
        return obj
```

The custom encoder and decoder can be executed as:

```
In [6]: arr = np.arange(4.).reshape(2, 2)
        dat_encoded = json.dumps({'a': arr, 'b': [3, 4]},
                                  cls=NpArrayEncoder)
        dat_decoded = json.loads(dat_encoded, object_hook=np_decoder)
        print(dat_encoded)
        print(dat_decoded)

Out[6]:
{"a": {"python/ndarray": [[0.0, 1.0], [2.0, 3.0]]}, "b": [3, 4]}
{'a': array([[0., 1.],
             [2., 3.]]), 'b': [3, 4]}
```

In certain scenarios, we can only convert a Python object into binary data (in `bytes` or `bytearray` types). However, the JSON format mandates the use of UTF-8 encoding. The `json` module in Python cannot process binary data in a JSON document. In such cases, we can use `base64` encoding to translate binary data before storing them in JSON. `Base64` encoding is a method to transform data into ASCII characters.

To illustrate how binary data can be managed in JSON documents, let's still consider the NumPy array as an example. In the JSON encoder, we can extract the raw contents of a NumPy array using the `.tobytes()` method. This binary data is then encoded into a `base64` string using the `b64encode` function from the Python `base64` library. During JSON decoding stage, we first decode the `base64` string back into binary data, and then reconstruct the array using NumPy functions. The program for encoder and decoder is demonstrated in the following.

```

from base64 import b64encode, b64decode

class NpArrayEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.ndarray):
            return {'python/ndarray': b64encode(obj.tobytes()).decode(),
                    'shape': obj.shape}
        return super().default(obj)

    def np_decoder(obj):
        if 'python/ndarray' in obj:
            return np.frombuffer(
                b64decode(obj['python/ndarray'])).reshape(obj['shape'])
        return obj

```

6.1.3 YAML

YAML is a clear, intuitive, and expressive data format. The YAML format is a superset of the JSON format. YAML document can be entirely written in JSON format. YAML extends the JSON format by introducing indentation to represent the data structure and allowing comments within the document. The specifications for the YAML format can be found online [6].

If human readability is a priority in Python object serialization, YAML is often a more favorable data format than JSON. This format is particularly suitable for configuration files. The docker-compose file we mentioned in Chapter 1 is a typical example. In Chapter 7, you will find more examples of YAML documents for managing cloud configurations.

The Python standard libraries do not support reading and writing documents in YAML format. We have to install external libraries to manipulate YAML data. Popular choices include the `pyyaml` and `ruamel.yaml` libraries. The usage of `pyyaml` and `ruamel.yaml` is similar. Taking the `ruamel.yaml` library as an example, we can use its `dump()` function to serialize a Python object and the `load()` function for deserialization. Unlike the `json` and `pickle` modules, the YAML library does not offer `dumps()` and `loads()` functions. Instead, `dump()` and `load()` are used for serialization to and deserialization from string representations. This is different to the API conventions employed in the `json` and `pickle` modules.

```

In [8]: from ruamel import yaml
        with open('file.yaml', 'w') as f:
            yaml.dump(obj, f)
        with open('file.yaml', 'r') as f:
            obj = yaml.load(f)

```

```
In [9]: yaml.dump({'a': [2, 3]})  
Out[9]:  
'a: [2, 3]\n'  
  
In [10]: yaml.load('a: [2, 3]\n')  
Out[10]:  
{'a': [2, 3]}
```

When running this code example, you will receive a `UnsafeLoaderWarning` due to the use of the `yaml.load()` function. This warning arises because the decoders in YAML libraries can execute Python functions on certain nodes. This feature grants YAML broader support for Python objects than JSON. It allows YAML to encode a variety of data types, including sets, timestamps, complex numbers, and even NumPy arrays. However, allowing the YAML decoder to execute Python functions introduces a security risk, similar to the vulnerability associated with pickle deserialization. We can use the `yaml.safe_load()` function to prevent the decoder executing Python object construction functions, though this will limit the supported data types in YAML. Nevertheless, in most scenarios, the `yaml.safe_load()` function should be sufficient to deserialize the YAML document.

How does YAML manage the binary data? As a text-based format, YAML cannot natively represent binary data. The encoder in `ruamel.yaml` library can automatically detect binary data within Python objects and employ a base64 encoder to convert the binary data. When decoding a YAML document, these base64-encoded strings can be automatically identified and transformed back into binary objects. This capability also makes YAML a more convenient option than JSON.

6.2 File I/O

In Python, the basic file I/O operations are typically handled by the `open()` function and the file object this function returns. We will skip the usage of the `open()` function as it has been extensively documented in Python programming textbooks.

In the context of quantum chemistry applications, the term I/O often refers to reading and writing large arrays in files. Regarding this, a new question arises: How can we effectively store and manage the high-dimensional arrays on disk, while ensuring that they are easy and efficient to access? The choice of array storage methods depends on various considerations:

- Is there a need to access individual elements?
- Can the contents of the array in the file be modified?
- Is the data storage intended for temporary use or long-term storage?
- Is there a need to store multiple arrays within a single file?
- Will the files be accessed by different programming languages?

Several high-performance data formats are available for storing arrays, such as NumPy's `npy` format, memory mapping, and HDF5 format storage. Each of these formats has its unique characteristics and specific use cases. A concise comparison of these formats is presented in Table 6.1. In this section, we will explore the functionalities of these data formats. The performance considerations will be deferred to Chapter 9.

Table 6.1 Comparison of `npy`, HDF5 and memmap formats for storing NumPy arrays.

Format	Pros	Cons
.npy file	<ul style="list-style-type: none"> Simple to use. 	<ul style="list-style-type: none"> Inconvenient to access elements or sub-arrays^a. Limited multiple datasets management in a single file.
HDF5	<ul style="list-style-type: none"> Supports slicing sub-arrays and accessing elements. Allows in-place editing of arrays in the file. Multiple datasets management in a single file. 	<ul style="list-style-type: none"> Steep learning curve to acquire its data layout and data types. Requires expertise for optimal performance.
memmap	<ul style="list-style-type: none"> Transparent <code>ndarray</code> replacement. Allows direct element access. Efficient memory use. No serialization overhead. 	<ul style="list-style-type: none"> Requires extra metadata (like shape and <code>dtype</code>) for array reconstruction. Not support managing multiple datasets; limited to one array per file. Needs support from OS. Poor cross-platform compatibility.

^a The memmap mode allows to conditionally access individual elements or sub-arrays of an .npy file.

6.2.1 The npy format

NumPy offers a function `np.save` to serialize arrays to a file in the `.npy` format. A `.npy` file can be loaded into memory using the `np.load` function.

```
In [11]: np.save(np.random.rand(2, 2, 3), 'rand.npy')
         a = np.load('rand.npy')
```

If you want to save multiple arrays in a single file, NumPy provides the `np.savez` function. This function archives multiple arrays into one `.npz` file. To access the contents of an `.npz` archive, the `np.load` function can still be used. `np.load` returns a `NpzFile` object, which behaves similarly to a dictionary. We can access a specific array saved in the `npz` file by accessing the corresponding key of the `NpzFile` object.

```
In [12]: np.savez('multi.npz', arr1=np.random.rand(2, 2, 3),
                  arr2=np.eye(4), index=np.arange(5))
         with np.load('rand.npy') as npz_ar:
```

```

        print(npz_ar['arr1'].shape)
        print(npz_ar['arr2'].shape)
        print(npz_ar['index'].shape)
Out[12]:
(2, 2, 3)
(4, 4)
(5,)

```

Please note that NumPy adopts a lazy loading strategy when handling `.npz` files. Individual arrays in the `.npz` file are only loaded into memory when they are referenced. In other words, if we just read a specific array from a `.npz` file, there's no need to worry about the CPU or memory overhead for reading other arrays in the file.

The `.npz` format is essentially a zipped archive of multiple `.npy` files. By unzipping this `.npz` file, we can clearly see how the `npz` format is structured. The keys of the `NpzFile` object correspond to the filenames.

```

$ unzip multi.npz
Archive:  multi.npz
extracting: arr1.npy
extracting: arr2.npy
extracting: index.npy

```

In fact, if we have multiple `.npy` files, we can manually package them into a single file, simulating the `.npz` file. The `np.load` function is able to identify and load the custom `.npz` file.

By default, the `np.load` function loads the entire array from an `.npy` file into memory. If we only need to access some elements of the `.npy` array, loading the entire array file is inefficient. To address this issue, the `np.load` function provides a memmap mode, which allows us to directly map the `.npy` file to memory (see Section 6.2.3). Then, we can access this memmap array just like a regular NumPy `ndarray`.

```

In [13]: np.save(np.random.rand(2, 2, 3), 'rand.npy')
          a = np.load('rand.npy', mmap_mode='r')
          print(a[1,0,:2])
Out[13]:
[0.79594989, 0.4396052]

```

The memmap mode of `np.load` comes with certain limitations. It does not allow for direct modifications to the contents of the file. Additionally, when accessing arrays in an `.npz` file, the memmap mode is not applicable.

The `.npy` format is a specialized serialization format for NumPy arrays. It is not designed for interoperability across different programming languages. However, NumPy ensures that the `.npy` format maintains cross-platform compatibility [7]. The `.npy` format stores all necessary information (such as the `shape`, and `dtype`) that are required to accurately reconstruct the array. An `.npy` file can be reliably loaded on any operating systems and architectures.

In a pure Python program, the `.npy` format is a practical option for array storage. It is very easy to use, requiring minimal effort to consider the low-level storage details. However, the `.npy` format does not support editing or rearranging arrays directly within the file. Regarding multiple-array management, the `.npz` format essentially performs simple file stacking, without offering any features for managing or organizing arrays.

6.2.2 HDF5 storage

The HDF5 (Hierarchical Data Formats) is a great option to store NumPy arrays as it offers efficient data storage that is accessible across different platforms and programming languages. The missing functionalities in `.npy` storage, such as accessing individual elements, slicing arrays, compressing arrays, and editing arrays directly in the file, can be easily implemented using HDF5 storage.

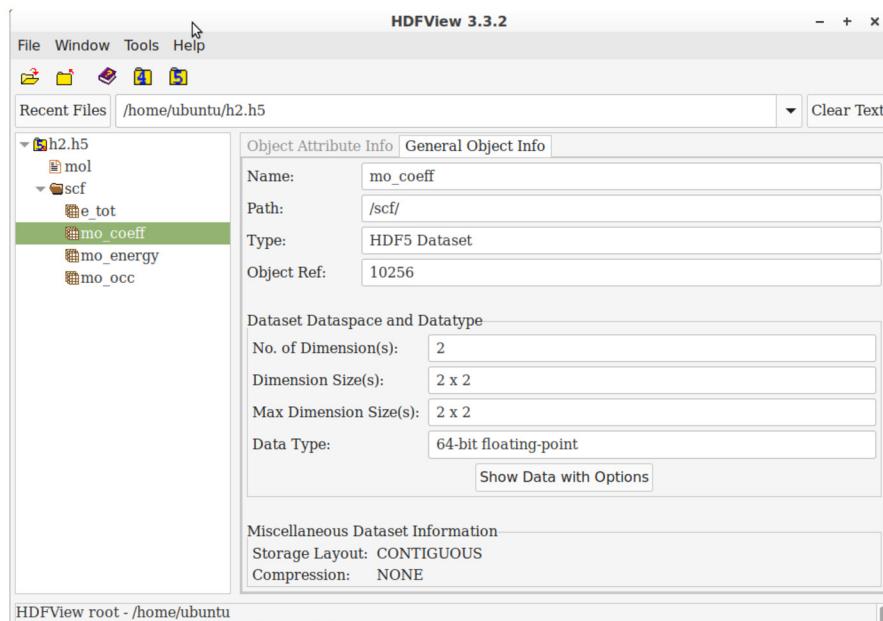


FIGURE 6.1

HDF5 File Structure, visualized in HDFview.

HDF5, as indicated by its name, organizes data within a file using a hierarchical structure, like a directory-based file system (Fig. 6.1). In an HDF5 file, the actual data of an array is stored in a *Dataset*. *Groups* contain datasets or other groups, functioning similarly to directories.

Accessing an HDF5 file in Python is straightforward, thanks to the `h5py` library. This library maps the HDF5 file to dictionary-style objects. Datasets or data groups

can be accessed via keys. Elements within the dataset can be sliced or indexed in the same way as NumPy arrays. Basic functionalities of `h5py` are illustrated in the following code example:

```
# Store array. File is opened in writable mode
with h5py.File('example.h5', 'w') as f:
    dataset = f.create_dataset('data', shape=(8,), maxshape=(None,))
    dataset[:] = np.arange(8.)

    # let h5py automatically set the dataset name, data types, and shape
    f['data1'] = np.arange(8.)

# Load array
with h5py.File('example.h5', 'r') as f:
    print(f['data'])

# Access individual elements
with h5py.File('example.h5', 'r') as f:
    print(f['data'][4:8])

# Edit contents of a dataset
with h5py.File('example.h5', 'a') as f:
    f['data'][::2] = 0.

# Resize a dataset
with h5py.File('example.h5', 'a') as f:
    f['data'].resize(12)

# Drop a dataset
with h5py.File('example.h5', 'a') as f:
    del f['data']

# Rename a dataset
with h5py.File('example.h5', 'a') as f:
    f.move('data', 'new')
```

Due to its data structure and management design, the space occupied by an HDF5 file can often exceed the total size of the arrays it stores. This phenomenon is particularly noticeable if datasets within an HDF5 file are frequently deleted and inserted.

The performance of HDF5 is heavily influenced by the storage configuration of its Datasets. Optimizing HDF5 performance requires expertise and sometimes a redesign of the data storage structure. We will explore the optimization of HDF5 storage in more detail in Chapter 9.

6.2.3 Memory mapping

Memory mapping (memmap) is a technique that allows files to be accessed as if they were in memory. This technique is particularly useful for handling large datasets that do not fit entirely in memory. NumPy can employ memmap as the data buffer for an array. Manipulating a memmap array is almost identical to working with a standard array. It allows for accessing only a small portion of the elements in the array, without the need to load the entire array into memory. The use of NumPy memmap is straightforward. To create a memmap for storing NumPy arrays, one simply needs to use the following command:

```
arr = np.memmap('filename.mmap', shape=(5,5), dtype='f8', mode='w')
```

Anything written to the memmap array is synchronized to the file, which eliminates the need for an additional data serialization process. To load memmap arrays that are already available on disk, the following command can be used:

```
arr = np.memmap('filename.mmap', shape=(5,5), dtype='f8', mode='r+')
```

6.3 I/O buffering

The performance of I/O can be influenced by the frequency of read and write requests. I/O devices typically have high latency, which means that small and frequent I/O requests can significantly increase overhead. To minimize this overhead, operating systems usually write data to an in-memory buffer before carrying out the actual I/O operations. Data is only written to the I/O device when the buffer is filled with sufficient data. Due to buffered I/O, data is managed in blocks rather than being handled individually.

While I/O buffering can effectively improve performance, it may lead to data loss. For example, if we modified a file but forgot to close it afterwards, it may result in incomplete content because some data may still reside in the buffer. When a file is closed, all data in the buffer is synchronized to the I/O devices, regardless of whether the buffer is full. Therefore, the `.close()` method should be invoked for any opened file objects.

Python recommends using the `with` context to handle file objects, which ensures that files are closed properly. Why is it not advised to manually call the `.close()` method in the code? The reason is that if the program crashes due to any errors/exceptions, the `.close()` statements might not be executed. By using the `with` context for file objects, the file can be closed before the program exits, even encountering Python error-exceptions.

Occasionally, programs may be abruptly terminated by users or system events. Even with the use of the `with` context, the exit handler might not be executed, potentially leading to data loss. If this is a concern, you can regularly call the `.flush()`

method of the file object in the code. The `flush()` method ensures that data is immediately synchronized to the I/O devices. For example, after making any changes to an HDF5 file, we can flush the buffer to guarantee data completeness within the file.

```
with h5py.File('example.h5', 'r') as f:  
    f['data'] = np.arange(8.)  
    f.flush()
```

In certain scenarios, we may want to disable the buffered I/O. You might have experienced a scenario like this: when the output from a Python program is redirected to a log file, the output messages do not immediately appear in the file. This delay occurs because the messages written to a file are buffered. To address this issue, the `-u` option can be used to force Python to *unbuffer stdout redirection*, ensuring the output immediately visible in the log file.

```
$ python -u example.py > log
```

6.4 In-memory I/O

In-memory I/O refers to reading data from and writing data to memory as if performing file I/O operations. While this concept may not seem useful at first glance, it has certain practical applications. In-memory I/O can:

- Mimic the functionality of a file object. For instance, we can transform a CSV-formatted string into a file-like object for the `pandas.read_csv()` function to parse, as this function only supports file objects.
- Serve as an I/O buffer. In-memory I/O is often used as a buffer for receiving data during parallel computation.
- Provide a method for concatenating strings (or bytes). When there is a need to concatenate a large number of strings, streaming them using in-memory I/O can improve performance because it avoids repeated memory allocations in standard string operations.
- Act as a temporary file to reduce the pressure on I/O devices. For example, when using a virtual machine in the cloud, I/O is subject to various limitations, such as bandwidth and the I/O operation counts. Replacing some file I/O operations with in-memory I/O can improve the I/O performance.

In-memory I/O can be implemented using the Python standard library `io`. By utilizing the `BytesIO` and `StringIO` classes from the `io` module, we can create in-memory I/O objects that function similarly to file operations.

```
In [15]: from io import BytesIO, StringIO
```

```
In [16]: buf = io.StringIO()
```

```

buf.write('hello')
buf.write(' world')
buf.seek(0)
print(buf.read())
Out[16]:
'hello world'

In [17]: buf = io.BytesIO()
          buf.write(np.random.rand(10).tobytes())

```

6.5 Network I/O

Network I/O refers to the data exchange between processes or computer nodes over a network. The primary application of network I/O is found in web services. In the context of web services, network I/O primarily focuses on network accessibility and availability, i.e. how to properly access a remote service through the network.

At first glance, network I/O for web services might seem unrelated to quantum chemistry. Historically, quantum chemistry programs were designed for High-Performance Computing (HPC) environments, where network I/O is primarily offered by the MPI (Message Passing Interface) library. MPI is a technique tailored for the HPC networks, which are known by their low-latency, high-bandwidth, and low failure rates. When MPI is employed, there is not much we can do in Python, as the program largely depends on MPI to handle all kinds of network communication (see more discussions in Chapter 10). However, with the rapid evolution of the cloud computing resources, some quantum chemistry calculations no longer require HPC environments and can be performed on the cloud. In cloud environments, it is inevitable to utilize the technologies developed for web-service network I/O.

Compared to the low-latency network infrastructure provided by HPC, the network for cloud computing is more complex. A different skill set is required to utilize the web-service type network communication. Several standard libraries in Python, such as the `socket`, `urllib`, and `http` modules, provide the basic functionalities for web-service network interactions. However, in practice, it is not necessary to directly use these low-level tools. There are various mature Python packages available that offer simplified models for web-service network programming.

The technology behind network I/O encompasses numerous details and challenges. In our discussion, we will omit many technical details, focusing instead on the concepts and application scenarios. The example programs are mostly auto-generated using AI-assistant coding tools (with minor manual modifications). Thanks to the abundance of open-source software in this technical field, AI has been trained to work quite well for these coding tasks.

6.5.1 Network requests over HTTP

HTTP is the most fundamental protocol for web services and the most widely used technology for exchanging information over internet. An HTTP request typically includes the following information:

- Request methods: GET, PUT, POST, DELETE, etc;
- An URI (Uniform Resource Identifiers) [8];
- Header, which records encoding, cookies, user agent, authentication, and various metadata.
- Message body (usually referred to as the payload);

An HTTP response from a remote server comprises:

- Response status code (200, 300, 400, 500, etc.);
- Metadata in the header;
- Response messages.

The `requests` library is the most popular tool for handling HTTP requests and responses. It can automatically construct an HTTP request then invoke the low-level libraries to send the network request. Upon receiving a response from the remote HTTP server, it can parse and post-process the response.

When using the `requests` library, the main focus is on setting up the necessary information required by the remote services, which includes:

- The URL;
- The request method (e.g., GET, POST);
- URL parameters (appearing at the end of a URL after a ? mark), which should be placed in the `params` argument;
- The authorization or authentication methods.

Additionally, due to the potential network issues and problems with the remote servers, it is important to configure a timeout for HTTP requests and to check the status code of the response. If the response is a JSON document, the `response.json()` method can convert JSON data into Python objects. We can collect all the necessary information and supply them to an AI coding assistant to generate programs.

For example, the Basis Set Exchange database provides basis sets for download via HTTP requests. By reading its manual [9], we learn that this service allows us to retrieve multiple basis sets simultaneously and returns them in JSON format. We then use GPT-4 to generate the HTTP request function, with the following prompt:

```
Create a Python function to query basis sets from bse:
- URL is "https://www.basissetexchange.org/api/basis/"
- Query multiple elements with the parameters: ?elements=1&elements=6&elements=
- Let the server return basis set in json format.
- Timeout 10s.
```

```
- Handle error in response.  
- Use the following signature  
...  
def query_basis_sets(basis_name, elements: List[str]):  
    ...  
    return {element1: basis_dat1, element2: basis_dat2}  
...
```

The generated program is shown below, where the `requests` library is correctly configured for this task.

```
import requests  
from typing import List  
  
def query_basis_sets(basis_name, elements: List[str]):  
    # Construct the URL for the API request  
    url = f"https://www.basissetexchange.org/api/basis/{basis_name}/format/  
        json/"  
  
    # Prepare the parameters for the request, including multiple elements  
    params = {'elements': elements}  
  
    try:  
        # Make the request to the BSE API with a timeout of 10 seconds  
        response = requests.get(url, params=params, timeout=10)  
  
        # Check if the response was successful (status code 200)  
        if response.status_code == 200:  
            # Parse the JSON response  
            data = response.json()  
  
            # Extract and return the basis set data for the requested  
            elements  
            result = {element: data['elements'][element]['electron_shells']  
                for element in elements if str(element) in data['elements']}  
            return result  
        else:  
            # Handle responses with error status codes  
            print(f"Error fetching data: {response.status_code}")  
            return {}  
    except requests.exceptions.RequestException as e:  
        # Handle exceptions raised by the requests library  
        print(f"An error occurred: {e}")  
        return {}
```

```
# Example usage
elements = ['1', '6'] # Hydrogen and Carbon
basis_name = 'cc-pVDZ' # Example basis set name
data = query_basis_sets(basis_name, elements)
print(data)
```

6.5.2 REST API

When invoking a remote service, another critical aspect is the API and the data format of the service. Among the various designs for APIs and data formats, the REST (Representational State Transfer) API is commonly adopted in web service interfaces.

You might be wondering: why are REST APIs relevant to the development of quantum chemistry applications? Many datasets are made available and accessible through REST APIs. Therefore, to manipulate these datasets, interacting with REST API services becomes inevitable. The Basis Set Exchange (BSE) service, which we have previously demonstrated, is an example that utilizes the RESTful design and returns data in a REST format. The Materials Project [10] is another example which provides a vast amount of computational results for molecules and materials via REST APIs. Furthermore, to access computational resources hosted on the cloud, we often need to utilize REST APIs. We will explore this topic in Chapter 7.

REST is not a protocol or a standard but rather a design approach. When a service interface is described as RESTful, it generally follows certain conventions:

- REST conceptualizes web services as resources, which can be accessed through URIs (Uniform Resource Identifiers). A URI is a hierarchical path that reflects the structure of the data. For instance, in the BSE service, the basis set data can be accessed via the URI `/api/basis/<basis_name>` and references for each basis set are available at the URI `/api/references/<basis_name>`.
- Operations for resources are associated with HTTP methods: GET for retrieving data, POST for creating data, PUT for updating data, and so on. For example, to download a basis set from the BSE, the GET method is used.
- HTTP status codes are used to indicate the result of REST API requests, such as status 200 for successful requests, 400 for invalid requests or processing failures.
- REST APIs transmit data in a structured text format, typically using JSON for both the requests and the responses.
- Besides the contents of the remote resources, responses from REST APIs typically include status and auxiliary information to help users analyze and manage the data. The auxiliary information often includes the API version, data version, error messages, request timestamps, and URLs for subsequent actions, etc.

While there are more recommendations and guidelines for designing a well-structured REST API, those topics exceed the scope of this book. If you are interested in learning more about REST APIs, further information is available online [11].

6.5.3 RPC over HTTP

In an HPC environment, it is usually not necessary to consider Remote Procedure Call (RPC) over HTTP. This technique is more commonly used in cloud environments, where tasks are distributed across remote computing nodes. RPC is utilized to launch and control computation tasks.

In the Python ecosystem, several frameworks are available for setting up RPC services, such as Flask [12], FastAPI [13], json-rpc [14], etc. Here, we use the Flask framework as an example to illustrate how to develop the RPC programs. Although other frameworks offer different designs and APIs, the fundamental concepts are similar.

Suppose that we aim to develop an RPC service to provide DFT calculation services. An initial version of this service could offer the following functionalities:

- An entry point to accept a job with DFT calculation parameters, and then launch a DFT calculation task that runs asynchronously.
- An entry point to query the status or results of a submitted DFT calculation.
- An entry point to monitor tasks that are queued, in progress, or completed.

We can provide these requirements to an AI coding assistant to generate a sample RPC server, which can serve as a starting point to customize or optimize the RPC service. To guide the AI tool, we can outline our requirements using pseudocode within the prompts. The pseudo code can detail the input and output formats for each RPC function, the URI of each function, and so on. Below is our prompt for GPT-4.

```
Create a HTTP service that serves for DFT calculations online
- Use flask framework.
- Provide REST API
- Implement functionalities below
```
REST API: POST at URI '/dft-energy/'
def create_dft_job(geometry: str, basis_set: str, xc_func: str):
 - launch a dft calculation, return an ID for the job
 return {'id': ..., 'status': ...}

REST API: GET at URI '/dft-energy/<id>'
def query_dft_result(id):
 - get the result or the state of a specific job
 return {'results': ..., 'status': ...}

REST API: GET at URI '/jobs/<status>'
def list_jobs():
 - list jobs for the specified status, such as pending, running,
 finished.
 return {'jobs': [id, id, id], 'status': ...}
```

```

The code generated by GPT-4 is presented below. For simplicity, we have omitted some comments and explanations generated by AI.

```
from flask import Flask, jsonify, request, abort
from uuid import uuid4
from threading import Thread
import time

app = Flask(__name__)

# Simulated database for demonstration purposes
jobs_db = {}

# Job statuses
PENDING = 'pending'
RUNNING = 'running'
FINISHED = 'finished'

def simulate_dft_calculation(job_id, geometry, basis_set, xc_func):
    time.sleep(5) # Simulate computation time
    jobs_db[job_id]['status'] = FINISHED
    jobs_db[job_id]['results'] = ...

@app.route('/dft-energy/', methods=['POST'])
def create_dft_job():
    data = request.json
    geometry = data.get('geometry')
    basis_set = data.get('basis_set')
    xc_func = data.get('xc_func')

    if not geometry or not basis_set or not xc_func:
        abort(400) # Bad request if any field is missing

    job_id = str(uuid4())
    jobs_db[job_id] = {'status': PENDING, 'results': None}

    thread = Thread(target=simulate_dft_calculation,
                    args=(job_id, geometry, basis_set, xc_func))
    thread.start()

    return jsonify({'id': job_id, 'status': PENDING}), 202

@app.route('/dft-energy/<id>', methods=['GET'])
def query_dft_result(id):
```

```

job = jobs_db.get(id)
if job is None:
    abort(404) # Not found if the job ID does not exist

return jsonify({'results': job['results'], 'status': job['status']})

@app.route('/jobs/<status>', methods=['GET'])
def list_jobs(status):
    filtered_jobs = [job_id for job_id, details in jobs_db.items() if
        details['status'] == status]
    return jsonify({'jobs': filtered_jobs, 'status': status})

if __name__ == '__main__':
    app.run()

```

In the generated code, as indicated by the comments, the DFT computation part is simulated. To transform this server into a runnable application, here we integrate the quantum chemistry program PySCF [15] to perform the DFT calculations.

```

def simulate_dft_calculation(job_id, geometry, basis_set, xc_func):
    import pyscf
    mf = pyscf.M(atom=geometry, basis=basis_set).RKS(xc=xc_func).run()
    jobs_db[job_id]['status'] = FINISHED
    jobs_db[job_id]['results'] = mf.e_tot

```

To avoid blocking the RPC service, the DFT computation task is executed asynchronously. Asynchronous execution is a standard approach in an RPC service. As demonstrated in this code example, upon receiving a request, the RPC server assigns the computation tasks to a separate thread and immediately returns a job ID for the task. This job ID can be used for job management in the future. Using threads for the asynchronous job execution is not a scalable solution. To support large scale computations, tasks can be offloaded to certain services or resources in the cloud. Further discussions on cloud computing are available in Chapter 7, and additional details on asynchronous computation can be found in Chapter 10.

Please note that an in-memory dictionary is created in this code example to store computational status and results. To avoid the risk of accidental data loss, we can consider to use a real database, such as Redis and MongoDB, for persistent storage. In Python, the Redis database can be managed using the `redis-py` library [16]. Using Redis as a storage service is a standardized approach to data management, and the interface code can be easily generated using AI tools.

Based on this RPC example, we can identify the fundamental components and structure for an RPC service:

- An HTTP server object (such as the Flask `app`) and various URIs to listen GET and POST requests.

- Adoption of JSON format for input and output.
- Asynchronous execution of computationally intensive tasks.
- A database to track the status of the asynchronous tasks.

By running the following command for the RPC server program, we can start a built-in server [17], which is accessible at <http://127.0.0.1:5000/>.

```
$ python dft_rpc.py
```

However, this server is primarily intended for debugging and testing purpose. To make the service reliable for remote access, additional technical aspects such as deployment and network routing need to be addressed [18]. We will skip these technical details here.

Let's now move our attention to the drawbacks of this RPC service. What are the limitations of developing a REST-RPC service in Python?

- Limitation in data structure. Python objects cannot be directly mapped to JSON data. If the input arguments and function outputs contain Python objects that cannot be serialized to JSON, we need to design and implement extra mechanisms to convert them into a format compatible with JSON representation.
- Error handling. If the calculation encounters any Python errors, these errors cannot be raised via HTTP responses. It is necessary to design certain error handling conventions to convey Python error events.
- Performance overhead. One source of overhead involves parsing HTTP protocol and routing HTTP requests. There is a static cost of 1–10 milliseconds associated with these operations. Another overhead is the cost of serialization and deserialization for JSON documents. Regardless of their actual data types, inputs and outputs have to be converted to or from JSON text. The overhead in REST service can impact the performance, especially for applications that require high efficiency and low latency.
- Larger message size, and the higher bandwidth consumption. Text formats are not as compact as binary data. What's more, REST APIs further increase the message size by including field names, parentheses, and other auxiliary text.
- The challenge of handling binary encoding. Binary format data is inevitable in quantum chemistry calculations. For instance, we may need to access the wavefunction, density matrices, and integrals, which are often very large-sized arrays. JSON documents do not natively support binary data. Transmitting binary data to JSON documents involves several intermediate steps for data conversion.

6.5.4 gRPC

The gRPC framework is known for its high performance and low latency. It can be used to optimize data transmission performance in RPC systems.

gRPC uses Protobuf (Protocol Buffers) [19] to serialize and exchange data. Protobuf is a cross-platform, cross-programming language data format. There exists a standardized procedure for developing an RPC service using gRPC and Protobuf

[20]. Taking the DFT RPC service as an example, the procedure generally involves the following steps:

1. Define the service in a .proto file, where we can specify the service interface and the message structure.

```
$ cat dft_rpc.proto
syntax = "proto3";

service DFTCalculation {
    rpc run_dft(DFTJobRequest) returns (DFTJobResponse) {}
}

message DFTJobRequest {
    string geometry = 1;
    string basis_set = 2;
    string xc_func = 3;
}

message DFTJobResponse {
    string job_id = 1;
}
```

2. Use the grpcio-tools package to generate a protobuf module, named dft_rpc_pb2.py, and a gRPC wrapper module, named dft_rpc_pb2_grpc.py.

```
$ pip install grpcio-tools
$ python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=.
dft_rpc.proto
```

3. Complete the server-side code using the generated Protobuf modules. In the server-side code, the generated Protobuf modules are utilized for data transmission. Additionally, the grpc module (provided by the grpcio package) is used to create and launch the service.

```
from concurrent import futures
import grpc
import dft_rpc_pb2
import dft_rpc_pb2_grpc

class DFTCalculationServicer(object):
    def run_dft(self, request, context):
        import pyscf
        mol = pyscf.M(atom=request.geometry, basis=request.basis_set)
        mf = mol.RKS(xc=request.xc_func).run()
        return dft_rpc_pb2.DFTJobResponse(energy=mf.e_tot)
```

```

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    dft_rpc_pb2_grpc.add_DFTCalculationServicer_to_server(
        DFTCalculationServicer(), server)
    server.add_insecure_port("[::]:50051")
    server.start()

if __name__ == '__main__':
    serve()

```

4. Develop a client. The generated Protobuf modules are also utilized on the client side. To simplify management, it is recommended to implement a function within the client that shares the same name as the corresponding function on the RPC server.

```

import grpc
import dft_rpc_pb2
import dft_rpc_pb2_grpc

def run_dft(geometry, basis_set, xc_func):
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = dft_rpc_pb2_grpc.DFTCalculationStub(channel)
        response = stub.run_dft(dft_rpc_pb2.DFTJobRequest(
            geometry=geometry,
            basis_set=basis_set,
            xc_func=xc_func
        ))
    return response.energy

if __name__ == '__main__':
    e = run_dft('O 0 0 0; H 0.757 0.587 0; H -0.757 0.587 0',
                'sto-3g', 'b3lyp')
    print("DFT job results:", e)

```

6.5.5 Apache arrow

Implementing gRPC for a Python application requires designing data formats and the corresponding interfaces based on the characteristics of Protobuf. If the inputs and outputs of the RPC involve complex data objects, such as various NumPy arrays or Pandas DataFrame objects, one has to specify the representation of each object in the .proto file, which can be quite inconvenient. For Python applications, the Apache Arrow library pyarrow [21] can be used as an alternative to the gRPC framework.

Apache Arrow introduces Flight [22], an RPC framework built on top of gRPC. The Flight framework not only offers a Pythonic style of usage but also benefits from the data communication efficiency of gRPC.

To create an RPC service with Flight, we need to inherit its `FlightServerBase` class. This class provides a hook, `do_action`, for transmitting data and executing pre-defined operations on the RPC server. For instance, by overriding this hook method, we can implement a DFT RPC service that outputs the total energy and an array of orbital energies.

```
import pickle
import json
import numpy as np
import pyarrow.flight as flight

def run_dft(geometry, basis_set, xc_func):
    import pyscf
    mf = pyscf.M(atom=geometry, basis=basis_set).RKS(xc=xc_func).run()
    return mf.e_tot, mf.mo_energy

class DFTflightServer(flight.FlightServerBase):
    def do_action(self, context, action):
        if action.type == 'run_dft':                                     # (1)
            # Decode inputs from the action body
            params = json.loads(action.body.to_pybytes())

            energy, orbital_energies = run_dft(
                params['geometry'],
                params['basis_set'],
                params['xc_func'])

            yield flight.Result(pickle.dumps(energy))                  # (2)
            yield flight.Result(pickle.dumps(
                (orbital_energies.shape,
                 orbital_energies.dtype)))
            yield flight.Result(orbital_energies.data)
        else:
            raise NotImplementedError

if __name__ == '__main__':
    server = DFTflightServer('grpc://localhost:50011')
    server.serve()
```

The `do_action` method can handle all requests (called `action` in this context) and dispatch them according to the value of `action.type` in line (1). Please note that

the `do_action` method is not designed to return a value. Instead, it should work as a generator or iterator, yielding PyArrow `Result` objects, as shown at line (2).

Within the Flight framework, I/O operations are processed with PyArrow buffers. PyArrow buffer works like Python's built-in buffer protocol (`bytes` and `bytearray`) and memoryview objects [23]. It is not applicable to directly broadcast a general Python object. We have to use methods like `pickle.dumps` or `json.dumps` to convert them into `bytes` before communication. However, for NumPy arrays, direct access to their memoryview is possible (via the `.data` attribute), which bypasses the overhead of serialization. Additionally, for Pandas objects, PyArrow specifically provides the `pyarrow.serialize_pandas` and `pyarrow.deserialize_pandas` functions to enable fast serialization and deserialization operations.

When dealing with large-scale data, storing the serialized data entirely in memory can lead to significant overhead. To address this, the Flight framework provides streaming methods for distributing data [24], including `do_get` for downloading and `do_put` for uploading.

On the client side, we use the `FlightClient` class to establish a connection with the server. By calling the `do_action()` method, requests are forwarded to the server, triggered the server-side `do_action()`. The response from `do_action()` is a generator. We can use the Python `next()` function to iteratively access the return values. Similar to the server-side process, both inputs and outputs are represented in the buffer protocol, which requires manual serialization and deserialization.

```
import pickle
import json
import numpy as np
from pyarrow import flight

def run_dft(geometry, basis_set, xc_func):
    request = json.dumps({
        "geometry": geometry,
        "basis_set": basis_set,
        "xc_func": xc_func
    })

    action = flight.Action('run_dft', request.encode())
    with flight.FlightClient('grpc://localhost:50011') as client:
        result = client.do_action(action)
        energy = pickle.loads(next(result).body)
        shape, dtype = pickle.loads(next(result).body)
        orbital_energies = np.frombuffer(
            next(result).body, dtype).reshape(shape)
    return energy, orbital_energies

if __name__ == '__main__':

```

```
e, mo_energy = run_dft('O O O O; H 0.757 0.587 0; H -0.757 0.587 0', 'sto-3g', 'b3lyp')
print("DFT energy:", e)
print("orbital energies:", mo_energy)
```

This is a minimal example of the PyArrow Flight framework. To develop an RPC service in a real product, it is often necessary to implement middleware that incorporates authentication, TLS encrypted transmission, and other functionalities. Details for these components can be found in the Flight documentation [24].

In addition to PyArrow, other libraries such as Dask [25], and Ray [26] also provide solutions for RPC services. Each of these libraries focuses on different aspects of RPC communication. Readers are encouraged to explore their documentation to learn how to utilize these tools effectively.

Summary

This chapter covers a range of technical topics, including data serialization formats, file I/O for array data storage, and network I/O handling. For data serialization, Python's built-in `pickle` module is often a suitable choice. However, for specific scenarios such as managing configuration files and network requests, human-readable formats like JSON and YAML are preferred. In the case of file I/O, options include the HDF5 format, NumPy's native `npy` format, and NumPy `memmap`. Each of these techniques has its own advantages. There is no simple winner in all scenarios. When it comes to network I/O, the requirements on performance largely influence the choice of technology. For simplicity, the JSON-based REST API is a viable option. For scenarios that demand high performance, gRPC and its derived technologies such as PyArrow can be considered.

References

- [1] Python Software Foundation, Python documentation - what can be pickled and unpickled, <https://docs.python.org/3/library/pickle.html#what-can-be-pickled-and-unpickled>, 2024.
- [2] Python Software Foundation, Python documentation - generator expressions, <https://docs.python.org/3/reference/expressions.html#generator-expressions>, 2024.
- [3] Cloudpipe Development Team, Cloudpickle, <https://github.com/cloudpipe/cloudpickle>, 2024.
- [4] The Uncertainty Quantification Foundation, dill: serialize all of Python, <https://dill.readthedocs.io/en/latest/>, 2024.
- [5] Python Software Foundation, Python documentation - pickling class instances, <https://docs.python.org/3/library/pickle.html#pickling-class-instances>, 2024.
- [6] YAML Language Development Team, Yaml ain't markup language (yaml) version 1.2, <https://yaml.org/spec/1.2.2/>, 2024.

- [7] NumPy Developers, NumPy documentation - numpy.lib.format, <https://numpy.org/doc/stable/reference/generated/numpy.lib.format.html>, 2024.
- [8] T. Berners-Lee, R. Fielding, L. Masinter, Uniform resource identifier (uri): generic syntax, <https://datatracker.ietf.org/doc/html/rfc3986>, 2005.
- [9] B. Pritchard, Basis set exchange api documentation - rest api reference, https://molssi-bse.github.io/basis_set_exchange-apidoc/reference.html#, 2024.
- [10] A. Jain, S.P. Ong, G. Hautier, W. Chen, W.D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, K.A. Persson, Commentary: The Materials Project: a materials genome approach to accelerating materials innovation, *APL Materials* 1 (1) (2013) 011002, <https://doi.org/10.1063/1.4812323>, https://pubs.aip.org/aip/apm/article-pdf/doi/10.1063/1.4812323/13163869/011002_1_online.pdf.
- [11] L. Gupta, Rest api tutorial, <https://restfulapi.net/>, Dec. 2023.
- [12] Pallets, Flask documentation, <https://flask.palletsprojects.com/en/latest/>, 2024.
- [13] S. Ramírez, FastAPI, <https://github.com/tiangolo/fastapi>.
- [14] JSON-RPC Working Group, Json-rpc specification, <https://www.jsonrpc.org/specification>, 2024.
- [15] The PySCF Developers, Quantum chemistry with Python, <https://pyscf.org/>, 2024.
- [16] Redis Inc, redis-py - Python client for redis, <https://redis-py.readthedocs.io/en/stable/>, 2024.
- [17] Pallets, Flask documentation - development server, <https://flask.palletsprojects.com/en/latest/server/>, 2024.
- [18] Pallets, Flask document - deploying to production, <https://flask.palletsprojects.com/en/latest/deploying/>, 2024.
- [19] Google LLC, Protocol buffers documentation, <https://protobuf.dev/>, 2024.
- [20] gRPC Authors, Basics tutorial - a basic tutorial introduction to grpc in Python, <https://grpc.io/docs/languages/python/basics/>, 2024.
- [21] N. Richardson, I. Cook, N. Crane, D. Dunnington, R. François, J. Keane, D. Moldovan-Grünfeld, J. Ooms, J. Wuściak-Jens, Apache Arrow, arrow: Integration to ‘Apache’ ‘Arrow’, <https://arrow.apache.org/docs/r/>, 2024.
- [22] W. McKinney, Introducing apache arrow flight: a framework for fast data transport, <https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/>, 2019.
- [23] Apache Software Foundation, Apache arrow document - memory and io interfaces, <https://arrow.apache.org/docs/python/memory.html>, 2024.
- [24] Apache Software Foundation, Apache arrow cookbook - arrow flight, <https://arrow.apache.org/cookbook/py/flight.html>, 2022.
- [25] Dask Development Team, Dask: library for dynamic task scheduling, <http://dask.pydata.org>, 2016.
- [26] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, I. Stoica, Ray: a distributed framework for emerging AI applications, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, 2018, pp. 561–577, <https://www.usenix.org/conference/osdi18/presentation/moritz>.

Working with cloud

7

If we need to perform a large amount of quantum chemistry simulations, where should these computations take place? Some people's first thought might be to use a supercomputer or a high-performance workstation. Aside from these options, cloud computing is an attractive alternative worth considering.

Although quantum chemistry simulations were traditionally executed on supercomputers, it is worth to consider cloud resources as a viable alternative to perform large-scale chemistry simulations. Unlike supercomputers, which are often constrained by quotas or the availability of idle resources, cloud computing generally offers a plentiful supply of resources. This abundance makes cloud computing particularly attractive in high-throughput quantum chemistry simulations. Additionally, the cloud offers a wide range of computational tools and hardware resources that may not be available on supercomputers. Some might consider the uniqueness of supercomputers and how unlikely the cloud computing will replace them. While this is true, the presence and significant impact of cloud computing cannot be ignored. What we can do is to harness it as additional computational resources.

To effectively utilize cloud computing, we need to develop new concepts that differ from those required for supercomputers. In particular, several technical aspects should be considered for cloud computing:

- Where are the computing, storage, and network resources located? How can we access these resources?
- How to configure a service and manage its configuration?
- How to design a workflow and orchestrate the computing resources within the workflow?
- How to schedule parallel jobs and scale them up on the cloud?

Using cloud computing often requires the development of specialized programs. In these programs, large efforts are devoted to configuring cloud services and ensuring their availability. These techniques and skills are more closely related to the work of DevOps. Although DevOps skills do not directly impact algorithm designs or the implementation of scientific programs, they can significantly enhance our productivity. In this chapter, we will explore the programming technologies associated with cloud computing. We will not present a step-by-step tutorial for each cloud service. Instead, we will introduce the general idea and then leverage the AI assistant to complete the remaining works.

7.1 Utilizing cloud computing

Cloud computing offers various technologies and features to harness computational resources. Exploring the details of cloud technology is beyond the scope of this book. In this section, we will discuss the basic principles of cloud computing. We will primarily use Python to deploy computation tasks using cloud computing APIs.

Please be aware that cloud computing charges based on the resources you consumed. This includes not only CPU hours but also storage, data transfer, database, and public IPv4 addresses, all of which are billable resources. In particular, data transferring services can lead to unexpected costs if not managed carefully. It is important to ensure that resources are properly terminated after use. Despite years of development in cloud computing technology, no cloud provider currently offers a solution that is “smart” enough to effectively balance cost and performance. An extraordinary skill in cloud computing is the orchestration of cloud resources in an effective way to minimize expenses. For any given problem, there are likely more than one strategy and combination of resources that can produce the same outcome. If you expect to make substantial use of cloud resources, it is worthwhile to thoroughly review the cloud documentation.

7.1.1 Comparison between cloud and supercomputers

Running simulation tasks on the cloud is very different from executing them on traditional supercomputers with the PBS or SLURM queue systems. Supercomputers offer an interactive environment that is close to the experience of using a single Linux desktop. In this sense, cloud computing is not as intuitive. The different user experience can largely be attributed to the architecture differences between cloud and traditional supercomputers:

- *Data Accessibility.* Traditional supercomputers typically offer a large-scale and high-performance global file system accessible to all computing nodes. Sharing data among different nodes is straightforward. Manipulating file in global file system is similar to accessing data on a local disk. In contrast, cloud typically does not use the architecture of global file systems. Instead, files are stored in cloud object storage, such as AWS S3 or Azure Blob. Accessing files stored in these locations requires the use of specific tools or APIs.
- *Software Stack.* Traditional supercomputers typically offer pre-installed software within a homogeneous environment, operating under the same OS. In the cloud, software is installed in virtual machines (VMs) or Docker images. The runtime environments can be more flexible, isolated, and diverged for each software. For a specific task, users only utilize VMs or images that contain the necessary software.
- *Job Scheduling.* Supercomputers utilize job scheduling systems like PBS or SLURM to manage and prioritize jobs. These systems launch jobs on reusable computing nodes, which have an environment almost identical to the one used in interactive sessions. Cloud computing platforms have their own job schedul-

ing mechanisms. Individual jobs are configured separately and executed within isolated VMs in the cloud.

- *Connectivity in Parallel Execution.* Supercomputers can provide low-latency and high-performance network that enables efficient communication between processors. Exchanging data between different processes via MPI (Message Passing Interface) is relatively simple. This level of connectivity is not as straightforward in the cloud.

7.1.2 Designing a workflow

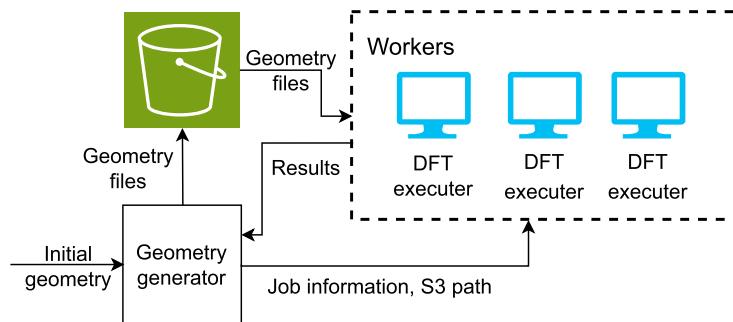
Normally, we need to design a workflow to utilize the cloud computational resources and manage the computation tasks. A workflow is essentially a sequence of pre-defined computational tasks executed in a particular order. We may need to develop a series of Python programs to connect the various components of the workflow.

When designing the workflow, the characteristics of the cloud architecture should be taken into consideration, particularly, involving the following questions:

- How to access the input data? For example, a large volume of input data can be stored in the cloud object storage, and accessed via cloud storage APIs. Input parameters can be stored in GitHub or GitLab, and accessed via HTTP requests or Git operations. Alternatively, essential input data can be cached within the Docker image. Additionally, a database can serve as an option for providing input data.
- Where to save the output? Cloud object storage, and database are two common choices to save results. It is also possible to send results through a remote process call (RPC) service to a receiver.
- How to connect different components of the workflow and how to exchange message between them? Cloud object storage and database can be utilized to exchange data. RPC are often employed to connect different services.
- What resources are required for the computation? Based on the performance and the cost of different devices, we may need to determine the appropriate combination of resources, including the number of CPU cores, GPU cards, memory size, temporary disk space, and network accessibility.
- Which Docker images or virtual machine images to use? Which software or tools should be installed in the image, and how should they be configured?

For example, suppose that we need to design a workflow to generate the potential energy surface (PES) for a molecule using the density functional theory (DFT). This workflow involves two types of executors: the molecule geometry generator and the executor to perform DFT calculations. As illustrated in Fig. 7.1, the two executors coordinate in the following manner:

1. The geometry generator reads the initial geometry of the molecule and generates a coarse-grained grid for the PES.
2. The geometry generator uploads all geometry files to the object storage, such as AWS S3, assuming the cloud provider is AWS.

**FIGURE 7.1**

Workflow for PES using DFT.

3. The geometry generator then invokes cloud APIs to launch several workers to execute DFT calculations. Within each job, metadata, such as the S3 paths of the geometry files, are provided as input to the DFT executor.
4. Each DFT executor downloads a geometry file, and then executes the pre-defined DFT calculation.
5. The DFT executor uploads the results to the S3 storage, and then sends a status signal to the geometry generator via RPC.
6. The geometry generator creates additional grids on PES based on the existing results and returns to step 2 to continue the workflow. Alternatively, it can choose to terminate the workflow if sufficient data points have been generated.

This workflow exhibits the following characteristics regarding the utilization of resources:

- The workflow primarily consists of CPU-intensive tasks.
- The demands for storage space and I/O operations are small.
- If the selected DFT program supports the use of GPU devices, GPU workers can be allocated.

Regarding the software stack, to support the tasks defined in this workflow, we can create a Docker image containing all the necessary components: a geometry generator, a DFT program, cloud management toolkits (such as `awscli`), and certain Python libraries for RPC services.

To design a complete and robust workflow, there are additional matters to consider, such as the fault tolerance for DFT calculations, the authentication scheme, the workflow restart scheme, etc. Developing a workflow for cloud computing is analogous to the process of developing a program. It may require continuous testing and debugging before deploying it on the cloud. Once the workflow is fully operational, it is convenient to repeatedly execute similar jobs in the cloud.

7.1.3 Computing resources

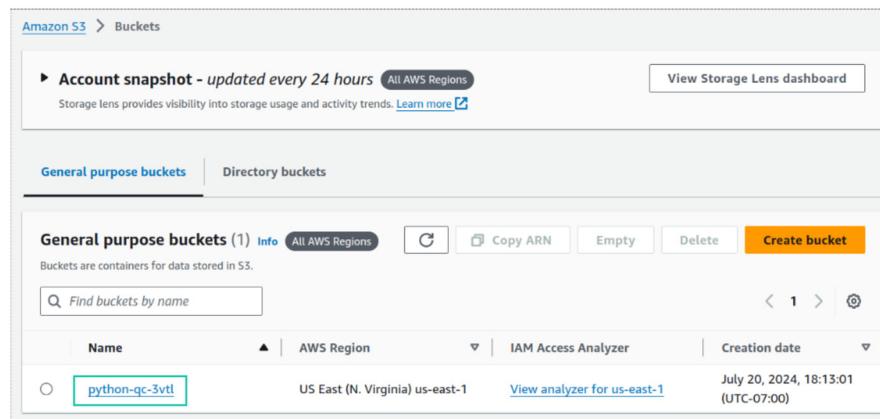
Virtual machines (VM), such as AWS EC2 and Azure VM, are the fundamental computational resources offered by cloud platforms. The simplest method to perform a computation task is to interactively execute the calculation in the cloud virtual machines. To make the runtime environment reproducible across each VM, we can customize the VM images [1] by pre-installing commonly used software stacks. Additionally, we can leverage block storage to persist the configuration files and data [2]. By combining the custom VM images and the persistent storage, we can easily spin up VMs for computation with identical environments.

To efficiently execute a batch of computation tasks, we need a non-interactive scheme to manage the computing resources and environments. There are several options to consider. One option is to use the *Batch services* provided by cloud platforms, such as AWS Batch, Azure Batch, or Google Batch. Another option is to use a *containerized solution* like AWS ECS, Azure ACS, or Google GKE. Although the UIs (user interfaces) of these cloud platforms differ, the design and concepts of the containerized approach are similar. In the following, we will take AWS ECS (Elastic Container Service) [3] as the example to demonstrate how to deploy the containerized computational tasks.

If you just signed up a new account on AWS, you may need several preparation steps to configure the basic cloud computation environment, including:

- AWS VPC (Virtual Private Cloud) [4], the foundational infrastructure where you can allocate virtual networks, virtual machines, and private storages.
- Subnets [5], which provides a range of IP addresses within a VPC.
- S3 (Simple Storage Service) [6], an object storage service that offers persistent data storage.
- ECR (Elastic Container Registry) [7], which provides a repository for hosting private Docker images.
- Security Groups [8], acting as a firewall to control inbound and outbound network traffic for virtual machines.
- AWS credentials [9], providing the necessary keys and authentication to log into virtual machines.
- IAM (Identity and Access Management) roles [10], which manages the permissions for accessing various services in the cloud.

In our following code example for DFT PES computation, we will utilize the ECS, S3 storage, ECR and other relevant services. It is straightforward to configure the S3 services using AWS web console, as shown in the screenshots in Fig. 7.2. We have created an S3 bucket named `python-qc-3vt1` for later use. Please note that AWS S3 bucket names are *unique across all AWS accounts globally*. For instance, the bucket name `python-qc` was already in use by other users. Consequently, we could not use `python-qc` as the bucket name for our project. To ensure uniqueness, we added the suffix `-3vt1`. You would need to choose a different name in your own implementation. The AWS IAM permission system is complicated. For simplicity, we have created an IAM policy with access to a wide range of AWS services as follows:

**FIGURE 7.2**

Creating AWS S3 bucket using AWS web console.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:*",
                "ecs:*",
                "ecr:*",
                "s3:*",
                "lambda:*",
                "sns:*"
            ],
            "Resource": "*"
        }
    ]
}
```

Please note that this AWS IAM configuration is insecure. In real applications, it is recommended to carefully configure the IAM rules and limit access only to the necessary services and resources. Please consult the AWS online documentation for more details.

The containerized approach requires a Docker image specifically designed for the computational task. Assuming the goal is to establish the DFT PES workflow mentioned in Section 7.1.2, the following Dockerfile can be created:

```
FROM nvidia/cuda:12.0.1-runtime-ubuntu22.04

# Install necessary dependencies
RUN apt-get update && apt-get install -y python3 python3-pip \
    && rm -rf /var/lib/apt/lists/*
RUN pip install --no-cache-dir pyscf gpu4pyscf-cuda12x awscli boto3

COPY start.sh /app/start.sh
WORKDIR /app
```

To accommodate the scenarios that require GPU acceleration, the base image is set to the `nvidia/cuda:12.0.1-runtime` image, which provides the CUDA 12 runtime environments. For the DFT computation engine, we utilize the quantum chemistry program PySCF [11] and its GPU acceleration variant `gpu4pyscf-cuda12x` [12,13]. The `awscli` toolkit and `boto3` library are installed for managing and interacting with AWS services. The `start.sh` script is responsible for managing computation and data transfer tasks. This script is implemented as follows:

```
#!/bin/bash
# Set the default time limit to 1 hour
TIMEOUT=$((2:-3600))
S3PATH=$1
aws s3 copy $S3PATH/job.py ./
timeout $TIMEOUT python job.py > job.log
aws s3 copy job.log $S3PATH/
```

The computation task is defined in a Python program, `job.py`, which should be created by the geometry generator service. To prevent the computation from running indefinitely, we use the GNU `timeout` tool to enforce the time limit on the calculation. Computational results are uploaded to the S3 storage. We then build this image and upload it onto the AWS ECR service:

```
$ aws ecr create-repository --repository-name python-qc/dft-gpu
$ docker build -t 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/
    dft-gpu:1.0 .
$ docker push 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/dft-
    gpu:1.0
```

The 12-digit number `987654321000` is just a placeholder for the account ID. You should replace it with your actual account ID.

To deploy the computation workflow on the AWS ECS, the process begins with defining an ECS cluster. An ECS cluster encompasses specifications of the VM hardware resources, operating system, network configuration, and various access permissions. Using the command provided by the `awscli` toolkit, we can establish a basic cluster that operates on dummy VMs, known as FARGATE mode [14], with default settings.

```
$ aws ecs create-cluster --cluster-name ecs-basic-cluster
```

The FARGATE-mode cluster would be sufficient for exploring the basic functionalities of AWS ECS. However, this cluster does not support GPU acceleration. To utilize GPUs, we need to enable ASG (Auto-scaling Groups) [15] for EC2 GPU instances when creating ECS clusters. It is recommended to configure this type of ECS cluster using AWS web console (Fig. 7.3). For the detailed instructions of ECS cluster configurations, please refer to the AWS ECS documentation [3,16].

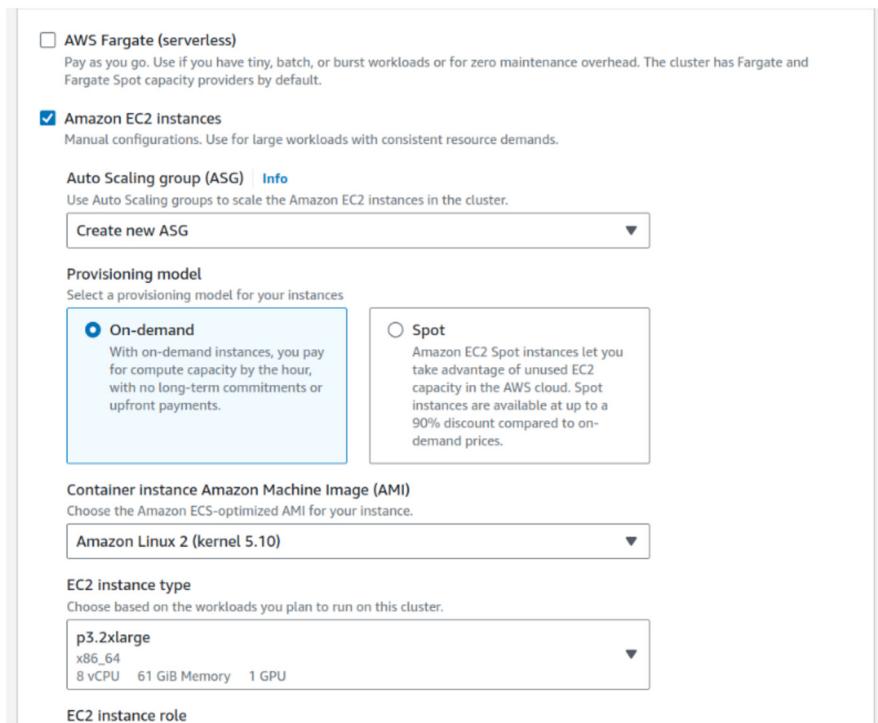


FIGURE 7.3

Creating an ECS cluster with ASG for EC2 GPU instances.

Let's assume that an ECS cluster named `python-qc` with GPU support is properly configured. The next step is to create a *task definition* on this ECS cluster. A task definition acts like a template for a specific computational task. It specifies the computational resources, such as CPU, memory, and the Docker image. The task definitions can be configured on the AWS web console.

To ease the configuration process for future works, we will instead generate and manage task definitions programmatically. The idea is to use code generation techniques along with the AWS Python SDK `boto3` to manage the configuration. To

enhance the readability of the configuration, we store it in the YAML format. The `boto3` library is complex to use as it involves numerous API details of the AWS cloud. Thanks to the progress of AI tools, it is no longer necessary to struggle with the `boto3` API documentation. Configuring cloud computing is a task particularly suitable for AI assistant programming. There is a massive amount of open-source examples available for cloud APIs. AI has been effectively trained on these examples. The output from AI is quite accurate.

Here, we utilize GPT-4 to generate a configuration sample, which can then be modified to meet specific requirements. Different AI coding assistants might produce different code samples. However, the differences should be insignificant for the current task. Our prompt for GPT is:

```
Create a Python script to register an AWS ECS task definition:  
- Use GPU instances on ECS  
- Use boto3 library  
- Follow the template below  
'''  
config = '''  
- ECS configuration in YAML.  
- Include docker image, the CPU, memory and other necessary requirements in  
    configuration.  
...  
py_config = convert YAML config to Python code  
ecs_client = boto3.client('ecs')  
response = ecs_client.appropriate_ecs_function(**py_config)  
'''
```

The output from GPT-4 is as follows:

```
import boto3  
import yaml  
  
# Your YAML configuration for the ECS task  
config = '''  
taskDefinition:  
    family: gpu-task-family  
    networkMode: "awsvpc"  
    containerDefinitions:  
        - name: gpu-container  
          image: your-docker-image  
          cpu: 1024  
          memory: 2048  
    resourceRequirements:  
        - type: GPU  
          value: "1"
```

```
requiresCompatibilities:
  - EC2
memory: "4GB"
cpu: "1024"
...

# Convert YAML config to Python dictionary
py_config = yaml.safe_load(config)

# Initialize a boto3 client for ECS
ecs_client = boto3.client('ecs')

# Register the ECS task definition
response = ecs_client.register_task_definition(
    **py_config['task_definition'])
```

While it is possible to refine the prompt to generate improved code samples, we will not pursue this refinement in the current context. The code sample above is clear enough to illustrate the parameters of the `register_task_definition` function of the `boto3` ECS client. The generated YAML configuration sample is also straightforward to understand. Based on this code sample, we can adjust certain settings and derive a script to register the task definition as follows:

```
config_tpl = '''
task_definition:
  family: dft-demo
  networkMode: "awsvpc"
  containerDefinitions:
    - name: dft-demo-1
      image: 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/dft-gpu
        :1.0
      cpu: 4
      memory: 2048
      resourceRequirements:
        - type: GPU
          value: "1"
  requiresCompatibilities:
    - EC2
  memory: "4GB"
  cpu: "4"
  ...

py_config = yaml.safe_load(config)
ecs_client = boto3.client('ecs')
response = ecs_client.register_task_definition(
    **py_config['task_definition'])
```

```
print(yaml.dump(response))
```

If you have previously created a basic ECS cluster with FARGATE mode, you can remove the setting "resourceRequirements" from the task configurations.

By using a similar methodology, we can define runnable tasks based on the above task definition and submit them to ECS. The runnable task requires the use of run_task function. Here, we omit the intermediate code samples generated by GPT-4 and demonstrate only the revised script.

```
import boto3
import jinja2
import yaml

task_config_tpl = jinja2.Template('''
cluster: python-qc
taskDefinition: dft-demo
count: 1
overrides:
  containerOverrides:
    - name: dft-demo-1
{%- if image %}
    image: {{ image }}
{%- endif %}
    command:
      - /app/start.sh
      - s3://python-qc-3vtl/{{ task }}/{{ job_id }}
      - "{{ timeout or 7200 }}"
    environment:
      - name: JOB_ID
        value: "{{ job_id }}"
      - name: OMP_NUM_THREADS
        value: "{{ threads or 1 }}"
{%- if threads %}
    memory: {{ threads * 2 }}GB
{%- else %}
    memory: 2GB
{%- endif %}
networkConfiguration:
  awsvpcConfiguration:
    subnets:
{%- for v in subnets %}
    - {{ v }}
{%- endfor %}
'''')
ec2_client = boto3.client('ec2')
```

```

resp = ec2_client.describe_subnets()
subnets = [v['SubnetId'] for v in resp['Subnets']]

config = task_config_tpl.render(
    task='dft-demo', job_id='219a3d6c', threads=2, subnets=subnets)
config = yaml.safe_load(config)

ecs_client = boto3.client('ecs')
resp = ecs_client.run_task(**config)
print(yaml.dump(resp))

```

If a task is successfully submitted, the ECS will start a EC2 VM with GPU devices. It then executes the `docker run` command within the VM, based on the parameters defined in the task and the task definition. For example, the task in this case is effectively identical to the following `docker run` command:

```
$ docker run --runtime=nvidia-container-runtime \
-e OMP_NUM_THREADS=2 --cpus 2 --memory 8G \
987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/dft-gpu:1.0 \
/app/start.sh s3://python-qc-3vt1/dft-demo/219a3d6c 7200
```

The ECS task is not limited to just a single run-until-completion job. We can use ECS workers to run Dask, Ray or other job schedulers. These schedulers can execute various types of tasks, simplifying the complex process of configuring individual ECS tasks. We will explore this approach in Section 7.2.

7.1.4 Communications among cloud services

In this PES workflow, we may desire to adjust the geometry grids dynamically based on the progress of the PES calculations. To accomplish this, it is necessary to establish interactions between the geometry generator program and the DFT calculation tasks. Instead of generating all grids on the PES at once, the geometry generator program can generate coarse-grained grids on PES initially and then refine the grids based on the accomplished results. The function `geometry_on_pes` below outlines the prototype of this functionality.

```

def geometry_on_pes(molecule, pes_params, existing_results=None):
    '''Generates molecule geometry for the important grids on PES

    Arguments:
    - molecule: Molecular formula
    - pes_params: Targets to scan, such as bonds, bond angles.
    ...
    # This function should produce molecule configurations
    # based on the results of accomplished calculations.
    # Here is a fake implementation to demonstrate the functionality.

```

```
if len(existing_results) > 1:  
    return []  
h2o_xyz = 'O 0 0 0; H 0.757 0.587 0; H -0.757 0.587 0'  
return [h2o_xyz] * 3
```

Since the DFT tasks are executed in containers or VMs remotely, it is not possible to achieve direct interactions between the geometry generator and DFT tasks within the same memory space. We thus utilize RPC framework to enable the communication between the two programs. The geometry generator can act as a service which generates tasks and launches these tasks in the cloud. Additionally, this geometry service provides an RPC endpoint to accept requests from DFT executors. Each DFT task can send its status or results to the geometry service through this RPC. As shown in Fig. 7.4, the PES workflow requires the collaboration of the following components:

- A geometry service to generate grids on PES.
 - A job launcher to initiate DFT tasks running on ECS.
 - An RPC system for the communication between the geometry service and the DFT task.

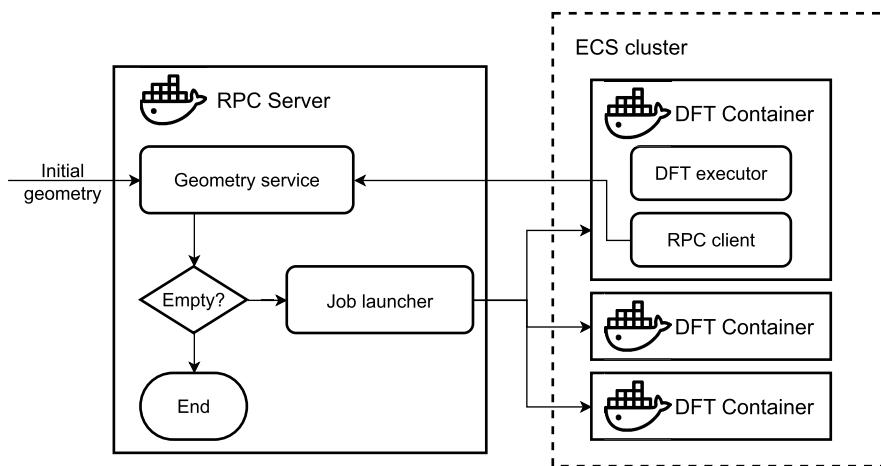


FIGURE 7.4

The design of PES workflow based on RPC service.

The configuration of the DFT task executor

To facilitate the communication with the RPC server, the server address can be provided as a parameter or an environment variable for the Docker container of the DFT task. Based on the ECS task developed in Section 7.1.3, we make slight adjustments to set up DFT tasks:

```

rpc_config_tpl = jinja2.Template('''
cluster: {{ cluster }}
taskDefinition: {{ task }}
count: 1
overrides:
  containerOverrides:
    - name: {{ task }}-1
      command:
        - /app/start.sh
        - {{ job_path }}
        - {{ timeout or 7200 }}
      environment:
        - name: JOB_ID
          value: "{{ job_id }}"
        - name: RPC_SERVER
          value: "{{ rpc_server }}"
        - name: OMP_NUM_THREADS
          value: "{{ threads }}"
      resourceRequirements:
        - type: GPU
          value: "1"
      cpu: {{ threads }} vcpu
      memory: {{ threads * 2 }}GB
  launchType:
    EC2
networkConfiguration:
  awsvpcConfiguration:
    subnets:
      {% for v in subnets %}
        - {{ v }}
      {% endfor %}
    ''')

```

The `start.sh` script for the DFT executor performs the following operations:

1. It retrieves the previously created `job.py` program from S3 storage.
2. It executes the DFT computation by calling `job.py`.
3. It sends the results to the RPC server using the `rpc.py` program.

A possible implementation for `start.sh` is provided below:

```

#!/bin/bash
# Set the default time limit to 1 hour
TIMEOUT=${1:-3600}
S3PATH=$1

```

```
aws s3 copy $S3PATH/job.py ./
if (timeout $TIMEOUT python job.py > job.log); then
    aws s3 copy job.log $S3PATH/
    python /app/rpc.py finished job.log
else
    python /app/rpc.py failed job.log
fi
```

The `rpc.py` program functions as an RPC client. It invokes the RPC endpoint `set_result` of the geometry service to transmit messages, such as the DFT energy or the computation status, back to the geometry service. The code snippet below illustrates the basic functionality of `rpc.py`:

```
from xmlrpclib import ServerProxy
rpc_server = os.getenv('RPC_SERVER')
job_id = os.getenv('JOB_ID')
status = sys.argv[1]
logfile = sys.argv[2]

def parse_log(logfile):
    '''Reads the log file and finds the energy'''
    log = open(logfile, 'r').read()
    ...
    return energy

with ServerProxy(rpc_server) as proxy:
    if status == 'finished':
        proxy.set_result(job_id, parse_log(logfile))
    else:
        proxy.set_result(job_id, status)
```

The RPC communication can be implemented in terms of REST APIs, as demonstrated in Chapter 6. However, for simplicity, we use the Python standard library `xmlrpclib` to create an RPC client here. The client requires the address of the RPC server, which is stored in the environment variable `RPC_SERVER`. This environment variable was configured previously in the ECS task.

The configuration of the RPC server

It is not a trivial task to establish the network communication between the RPC server and the DFT executor containers. One approach is to bind an endpoint to the RPC server that is accessible by other Docker containers. This endpoint can be an IP address, a DNS-resolvable address, or a proxy that forwards messages to the machine hosting the RPC server. In this example, the geometry service and the DFT executors are configured to operate within the AWSVPC network mode [17]. In this network mode, all containers can directly access each other via IP addresses. The IP address

of each container instance can be obtained through the metadata URI provided by AWS [18].

```
import os, requests

def self_ip():
    '''IP address of the current container or EC2 instance'''
    metadata_uri = os.getenv('ECS_CONTAINER_METADATA_URI')
    resp = requests.get(f'{metadata_uri}/task').json()
    return resp['Networks'][0]['IPv4Addresses']
```

To prepare DFT tasks on the RPC server, we use the following template to create the Python script `job.py`:

```
rpc_job_tpl = jinja2.Template('''
import pyscf
from gpu4pyscf.dft import RKS
mol = pyscf.M(atom=""'{ geom }' "", basis='def2-tzvp', verbose=4)
mf = RKS(mol, xc='wb97x').density_fit().run()
''')
```

The `job.py` script, once generated, is uploaded to S3 storage at the location specified by `job_path` for remote access.

These task preparation steps can be integrated into the `launch_tasks` function as shown below:

```
import boto3, hashlib, jinja2, json, yaml
from typing import List, Dict
from concurrent.futures import Future

CLUSTER = 'python-qc'
TASK = 'dft-demo'
RPC_PORT = 5005
s3_client = boto3.client('s3')
ecs_client = boto3.client('ecs')

job_pool: Dict[str, Future] = {}

def launch_tasks(geom_grids, results_path, timeout=7200):
    assert results_path.startswith('s3://')
    ip = self_ip()

    jobs = {}
    for geom in geom_grids:
        job_conf = rpc_job_tpl.render(geom=geom).encode()
        job_id = hashlib.md5(job_conf).hexdigest()
```

```
bucket, key = results_path.replace('s3://', '').split('/', 1)
job_path = f's3://{bucket}/{key}/{job_id}'
s3_client.put_object(Bucket=bucket,
                      Key=f'{key}/{job_id}/job.py',
                      Body=job_conf)

task_config = rpc_config_tpl.render(
    cluster=CLUSTER, task=TASK, job_id=job_id, job_path=job_path,
    rpc_server=f'{ip}:{RPC_PORT}', timeout=timeout, threads=2)
try:
    ecs_client.run_task(**yaml.safe_load(task_config))
except Exception:
    pass
else:
    fut = Future() # (1)
    fut._timeout = timeout
    job_pool[job_id] = fut
    jobs[job_id] = fut
return jobs

def set_result(job_id, result):
    fut = job_pool[job_id]
    fut.set_result(result) # (2)
```

Given a set of molecular geometries, the `launch_tasks` function launches ECS tasks and returns a collection of `Future` objects corresponding to the DFT tasks. These DFT tasks and the geometry generator service operate asynchronously. To track the outcome of the DFT task, we introduce the `Future` object in line (1), which represents the result of the unfinished DFT task. When the `Future.result()` method is invoked, the asynchronous program will be blocked until the `Future.result()` method returns a result. The `Future.set_result` method, at line (2), can be used to set the result, which informs the `Future` object to unblock the state. The `set_result` method is exposed to the RPC service, allowing it to be executed remotely by the RPC client on the DFT workers. The combination of the `Future` object and the `set_result` method is a common technique in asynchronous programming, which we will explore in more detail in Chapter 10.

Deploying RPC server

To prevent the `launch_tasks` function from being blocked by the RPC service, we run the RPC server in the background.

```
from contextlib import contextmanager
from threading import Thread
from xmlrpc.server import SimpleXMLRPCServer
```

```

@contextmanager
def rpc_service(funcs):
    '''Creates an RPC service in background'''
    try:
        rpc_server = SimpleXMLRPCServer("", RPC_PORT)
        for fn in funcs:
            rpc_server.register_function(fn, fn.__name__)
        rpc_service = Thread(target=rpc_server.serve_forever)
        rpc_service.start()
        yield
    finally:
        # Terminate the RPC service
        SimpleXMLRPCServer.shutdown(rpc_server)
        rpc_service.join()

```

We then develop the driver function `pes_app` to manage the RPC server and the workflow. This function is defined in the `pes_scan.py` file. Depending on the results from the completed DFT tasks, the driver can either proceed to generate the next batch of computational tasks or terminate the PES workflow if necessary.

```

def parse_config(config_file):
    assert config_file.startswith('s3://')
    bucket, key = config_file.replace('s3://', '').split('/', 1)
    config = json.loads(s3_client.get_object(Bucket=bucket, Key=key))
    return config

def pes_app(config_file):
    config = parse_config(config_file)
    molecule = config['molecule']
    pes_params = config['params']
    results_path = config_file.rsplit('/', 1)[0]

    with rpc_service([set_result]):
        # Scan geometry until enough data are generated
        results = {}
        geom_grids = geometry_on_pes(molecule, pes_params, results)
        while geom_grids:
            jobs = launch_tasks(geom_grids, results_path)
            for key, fut in jobs.items():
                try:
                    result = fut.result(fut._timeout)
                except TimeoutError:
                    result = 'timeout'
                results[key] = result
            geom_grids = geometry_on_pes(molecule, pes_params, results)

```

```
    return results

if __name__ == '__main__':
    pes_app(sys.argv[1])
```

The RPC server program can be deployed on ECS as a task.

```
rpc_server_config = ...
family: pes-rpc-server
containerDefinitions:
- name: rpc-server
  image: 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/dft-gpu:1.0
  networkBindings:
  - containerPort: 5005
    hostPort: 5005
  command:
  - python
  - pes_scan.py
  - config_file
...
config = yaml.safe_load(rpc_server_config)
ecs_client.register_task_definition(**config)
```

In this configuration, the `networkBindings` must be configured to expose port 5005 (specified by the `RPC_PORT` previously) for RPC communication.

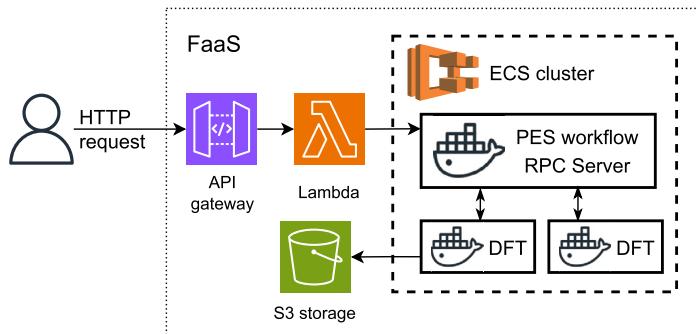
7.1.5 Function-as-a-Service

Developing a workflow on the cloud from scratch requires some coding work and expertise with cloud platforms. Once a workflow is successfully configured, we may want to enhance its accessibility, making it more convenient for future use or available to other users. Here, developing a Function-as-a-Service (FaaS) is one viable option that can simplify the complex deployment procedures associated with cloud platforms. This service can be accessed using RESTful APIs over the HTTP protocol. One only needs to send an HTTP request that specifies the molecular structure and DFT parameters to initiate a PES scan task.

The PES scan service requires the coordination of multiple components:

- An endpoint to receive HTTP requests.
- Middleware to forward HTTP requests to the backend server.
- A backend server to process the request and launch the PES scan job on AWS ECS. It then returns the execution results in an HTTP response.

On the AWS cloud platform, we can leverage various AWS components to implement this service (Fig. 7.5). The AWS API Gateway can provide an HTTP endpoint and act as a proxy to forward HTTP requests [19]. AWS Lambda can serve as the backend to

**FIGURE 7.5**

The architecture of FaaS for the PES scan service.

parse the request and launch the PES scan workflow we developed in Section 7.1.4. The results of the PES scan job can be stored in the S3 storage. The path of the S3 storage is returned in the HTTP response. The configuration of the Lambda handler and the API Gateway trigger are illustrated in Figs. 7.6–7.9.

Let's start the project from the AWS Lambda. AWS Lambda is a serverless computing service that enables us to execute code without the need to provision or manage servers. It automatically allocates computing resources and executes pre-defined functions based on the incoming request. Here, we create a Lambda function using the configurations shown in Fig. 7.6. We name this Lambda function `pes-scan`. This name will be automatically applied to the configurations of other services. You may choose any other name for this function. If you choose a different name, be sure to update all instances of `pes-scan` in the subsequent settings accordingly.

In the AWS Lambda console (Fig. 7.7), we can deploy the following Python function handler [20] in Box 1 of Fig. 7.7.

```

import json, hashlib, boto3
s3_client = boto3.client('s3')
ec2_client = boto3.client('ec2')
ecs_client = boto3.client('ecs')

def lambda_handler(event, context):
    # The data structure of event can be found in
    # https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html
    method = event['httpMethod']
    if method != 'POST':
        return {'statusCode': 400}

    body = json.loads(event['body'])
    job = body['job']

```

```

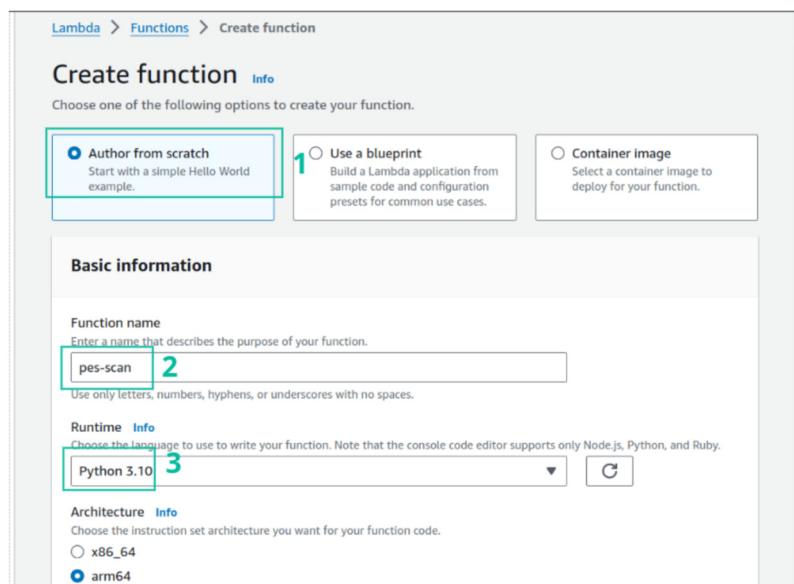
if job.upper() != 'PES':
    msg = f'Unknown job {job}'
    results = 'N/A'

config = json.dumps({
    'molecule': body['molecule'],
    'params': body['params']
}).encode()
job_id = hashlib.md5(config).hexdigest()

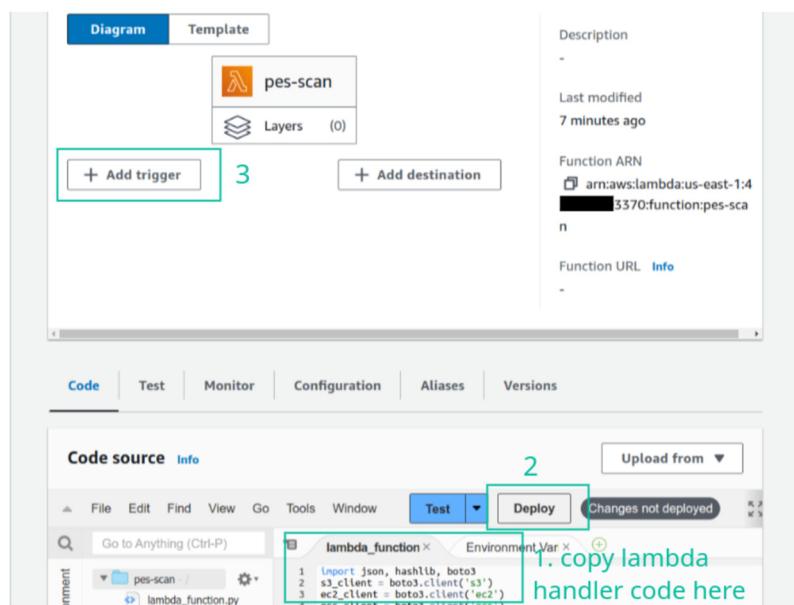
resp = ec2_client.describe_subnets()
subnets = [v['SubnetId'] for v in resp['Subnets']]
bucket, path = 'python-qc', f'pes-faas/{job_id}'
s3_client.put_object(Bucket=bucket, Key=f'{path}/config.json', Body=
    config)
s3path = f's3://{bucket}/{path}/config.json'
resp = ecs_client.run_task(
    cluster='python-qc',
    taskDefinition='pes-rpc-server',
    count=1,
    launchType='FARGATE',
    networkConfiguration={'awsVpcConfiguration': {'subnets': subnets}},
    overrides={
        'containerOverrides': [
            {
                'name': 'rpc-server',
                'command': ['python', 'pes_scan.py', s3path]
            }
        ]
    }
)
msg = json.dumps(resp)
results = f's3://{bucket}/pes-faas/{job_id}/results.log'
return {
    'statusCode': 200,
    'body': {
        'results': results,
        'detail': msg,
    }
}
}

```

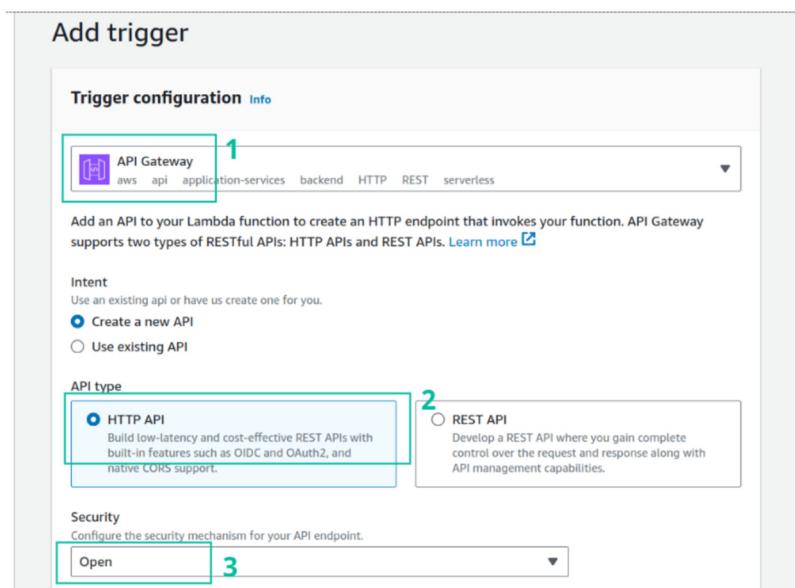
This handler decodes the HTTP requests and launches the PES scan task on ECS. Furthermore, a trigger for the Lambda function can be configured in Box 3 of Fig. 7.7 in the AWS Lambda console. The API Gateway is assigned to the trigger, as shown in the Box 1 of Fig. 7.8. For simplicity, we set the security mechanism of the API Gateway to Open, which allows the gateway to accept all requests and forward them

**FIGURE 7.6**

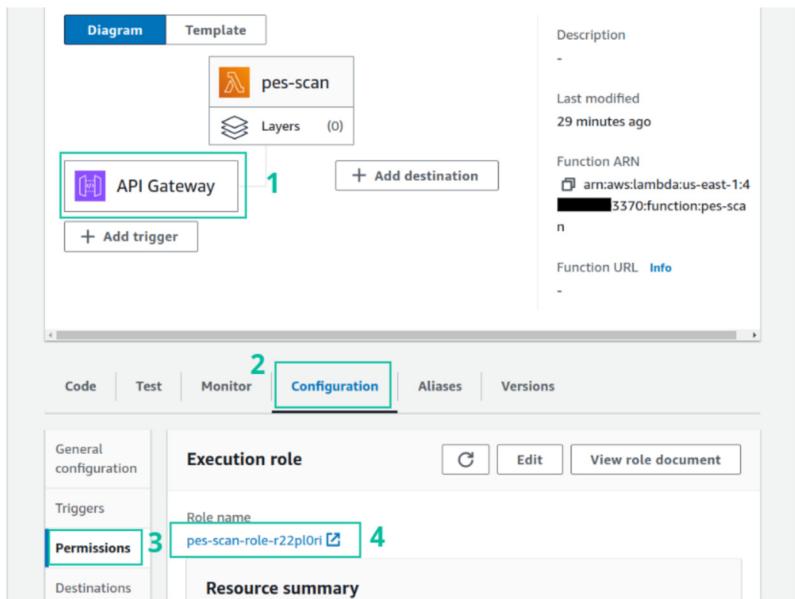
Creating an AWS Lambda function to trigger the PES scan service.

**FIGURE 7.7**

Configuring AWS Lambda.

**FIGURE 7.8**

Adding an AWS API Gateway trigger.

**FIGURE 7.9**

Checking API Gateway endpoint and configuring permissions.

to the Lambda function. Please note that this setting is insecure. It is recommended to configure an authentication method to enhance the security service [21]. By deploying the API Gateway trigger, an HTTP endpoint is generated. This endpoint can be accessed by clicking on Box 1 in Fig. 7.9, which indicates the following URL:

<https://d8iah6mwib.execute-api.us-east-1.amazonaws.com/default/pes-scan>

The Lambda function handler needs access to several AWS services, including the read and write permissions for S3, EC2, and ECS. We can configure the permissions of the Lambda function in the AWS Lambda console, as shown in Fig. 7.9. For the configuration method of the IAM policy, refer to Section 7.1.3. After assigning necessary permissions to the Lambda function, we are ready to provide a PES scan service through the API Gateway endpoint. Here is an example of invoking the HTTP API using a JSON document, which specifies the geometry of a molecule and the PES parameters:

```
import requests
molecule = 'O 0 0 0; H 0.757 0.587 0; H -0.757 0.587 0'
resp = requests.post(
    'https://d8iah6mwib.execute-api.us-east-1.amazonaws.com/default/pes-
    scan',
    json={'job': 'pes', 'molecule': molecule, 'params': 'O,H'}
)
print(resp.json()['detail'])
```

The response will include an S3 path that indicates where the results are stored.

The above example is the most basic framework of an FaaS, in which we have omitted many technical details. In a comprehensive FaaS, there are more engineering and usability issues to consider, such as authorization, authentication, network security, fault tolerance, monitoring, and data persistence, etc. Nevertheless, this basic framework can serve as a starting point, from which you can gradually introduce more features to enhance the usability of the FaaS.

7.2 Distributed job executors

In high-performance computing (HPC) environments, job schedulers such as PBS and SLURM are used to allocate the computational resources for job execution. Although applicable, they are not as commonly used in cloud environments. In cloud settings, resources such as computing nodes and storage are allocated on demand. It is inconvenient to reconfigure the traditional HPC management system each time new computing resources are allocated. Consequently, simple and lightweight tools for job scheduling have become more popular in cloud environments.

To execute Python programs in parallel, Celery, Dask, and Ray are three popular distributed executors well suited for cloud computing. Each of these tools has its unique characteristics, advantages, and suitable scenarios:

- Celery [22] is a distributed computing tool that relies on message queues to distribute tasks across multiple computing nodes. It is easy to scale up workers to process a high volume of similar tasks. Celery is commonly used for handling background tasks in web applications. It can also effectively support quantum chemistry data generation workloads.
- Dask [23] is a library that focuses on large-scale data processing. It is often employed in data analysis and numerical computation jobs that are distributed across multiple computing nodes. Its distribution system is a pure Python application, which does not require any external components (such as message queues or databases). Dask is a flexible framework for distributing workloads. It is not limited to pre-defined functions. Python functions can be created at runtime and sent to workers for execution.
- Ray [24] is a high-performance distributed computing framework. It shares certain similarities with Dask in distributed task execution. The highlight of the Ray project is its native integration with cloud computing environment. It offers convenient cloud resource management and task scheduling capabilities. In addition to distributing workloads, Ray offers a comprehensive solution for memory sharing and parallel computation in a distributed system. If the goal is to develop a distributed application that does more than just workload offloading, the Ray framework is an excellent option.

Distributed job executors typically consist of servers, clients, workers, and other components. Each component requires a different deployment strategy. Servers and workers are usually deployed on the cloud. Clients can be installed in the local Python environment.

Distributed computing is often closely related to parallel computation techniques. In this section, we will focus only on the capabilities and the deployments of distributed job executors in the context of cloud computing. The designs of parallel programs will be explored in Chapter 10.

7.2.1 Celery

The distributed execution in Celery is based on the producer-consumer parallel computing model. In this setup, the Celery client acts as the producer, pushing new tasks into a queue. Celery workers then retrieve and execute tasks from this queue. To deploy Celery, we need to set up a message broker to manage the task queue.

Celery supports a variety of message brokers, such as RabbitMQ, Redis, and Amazon SQS (Simple Queue Service). Each message broker specializes in different functionalities, including security, performance, priority handling, error tolerance, and crash recovery. These considerations are crucial for high-concurrent web applications. However, for tasks such as DFT data generation or other chemistry simulation calculations, the differences among these brokers are minimal, and all are generally sufficient for our use cases.

Let's choose Redis to serve as the broker. It can be deployed locally using the command:

```
$ docker run -d -p 6379:6379 redis:7.2
```

We can accordingly install the Celery library with the Redis extension:

```
$ pip install celery[redis]
```

The first step in applying Celery is to develop a Celery application. A Celery application is essentially a Python module that can be imported both locally and by Celery workers.

```
$ cat dft_app.py
import hashlib
import pyscf
from gpu4pyscf.dft import RKS
from celery import Celery

app = Celery('dft-pes', broker='redis://localhost:6379/0',           # (1)
             backend='redis://localhost:6379/1')                      # (2)

@app.task
def dft_energy(molecule):
    job_id = hashlib.md5('f{molecule=}'.hexdigest())
    mol = pyscf.M(atom=molecule, basis='def2-tzvp', verbose=4)
    mf = RKS(mol, xc='wb97x').density_fit().run()
    return job_id, mf.e_tot
```

The message broker is specified by the keyword `broker` when creating the Celery application, as shown in line (1). It only serves as a task queue for scheduling jobs. To receive output from the remote Celery workers, it is necessary to configure a database backend to store results for the Celery application. There are several options available for result backends in Celery [25]. For simplicity, we continue to use Redis as the results backend. Redis uses numbers in the URL to identify databases. Different numbers correspond to different databases. Results are configured to store in a separate database, as shown in line (2).

In the Celery application code, we should use the decorator `@app.task` to register any remote functions, as required by the Celery library. In our local environment, we can import this application and invoke the `.delay()` method of the remote function to submit tasks.

```
In [1]: from dft_app import dft_energy
results = [
    dft_energy.delay('O 0 0 0; H 0.757 0.5 0; H -0.757 0.5 0'),
    dft_energy.delay('O 0 0 0; H 0.757 0.6 0; H -0.757 0.6 0')]
print(results)
print(results[0].status, results[1].status)
```

```
Out[1]:  
[<AsyncResult: 41100ef3-2fed-4809-b8e7-a599b161b5a8>, <AsyncResult: 17271  
    cc9-6aba-4bef-a8ce-544a512f8e9a>]  
PENDING PENDING
```

Celery returns an `AsyncResult` object to represent the result of the remote function. Initially, the status of the submitted tasks is `PENDING`. This is because we have not yet deployed any workers for this application.

In another terminal, we can start a Celery worker by executing the command:

```
$ celery --app=dft_app worker
```

The application, as specified by the `--app` argument, must be a module that Celery workers can import. In this local setup, the worker successfully loads the Celery application because the command is executed in the same directory where the application program file resides. When deploying workers remotely, the application should be properly installed to ensure that it is importable. The package distribution techniques we discussed in Chapter 1 can be employed in this regard.

The Celery worker retrieves tasks from the message broker and sends the results to the backend database. We can access the running status, error messages, and results via the `AsyncResult` objects.

```
In [2]: print(results[0].get())  
      print(results[1].status)  
Out[2]:  
['2bb88ae9b074ca2c190ac9969b4e7397', -76.43283440169702]  
SUCCESS
```

This example illustrates the basic usage of Celery, which is sufficient for the current data generation tasks. In order to efficiently process a large volume of tasks, we plan to deploy Celery workers on the AWS cloud. We will use AWS ECS to launch Celery workers and AWS SQS to serve as the broker. We choose SQS over deploying a standalone message queue because SQS can be conveniently accessed from a local machine. This eliminates the complex network configurations that would otherwise be necessary for a Celery client to access the message queue.

Despite its accessibility, the use of AWS SQS faces a different restriction. Celery does not support SQS as a backend for result storage. An alternative solution is to use other AWS database services as the result backend. For simplicity, we will just upload the results to AWS S3 storage. The Celery object and the task functions in `dft_app.py` are modified accordingly:

```
import tempfile  
import pyscf  
from gpu4pyscf.dft import RKS  
from celery import Celery  
import boto3
```

```

app = Celery('dft-pes', broker='sns://')
s3 = boto3.client('s3')

@app.task
def dft_energy(molecule, s3_path):
    output = tempfile.mktemp()
    mol = pyscf.M(atom=molecule, basis='def2-tzvp',
                  verbose=4, output=output)
    mf = RKS(mol, xc='wb97x').density_fit().run()
    bucket, key = s3_path.replace('s3://', '').split('/', 1)
    s3.upload_file(output, bucket, key)

```

The next step is to configure AWS ECS for Celery workers. This is similar to the process we discussed in Section 7.1.3. We need to create a Docker image and upload it to AWS ECR. The `Dockerfile` for the Docker image is specified as follows:

```

FROM nvidia/cuda:12.0.1-runtime-ubuntu22.04

RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/*
RUN pip install --no-cache-dir pyscf gpu4pyscf-cuda12x boto3 celery[sqs]

COPY dft_app.py /app/
WORKDIR /app
CMD ["celery", "-A", "dft_app", "worker", "--loglevel=info"]

```

This Docker image is labeled as `python-qc/celery-dft:1.0`.

```

$ aws ecr create-repository --repository-name python-qc/celery-dft
$ docker build -t 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/
    celery-dft:1.0 .
$ docker push 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/celery
    -dft:1.0

```

We then register an ECS task definition using the new Docker image.

```

config = yaml.safe_load('''
family: celery-gpu-worker
networkMode: awsvpc
containerDefinitions:
- name: celery-worker
  image: 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/celery-dft
    :1.0
  environment:

```

```

    - name: OMP_NUM_THREADS
      value: "4"
    runtimePlatform:
      cpuArchitecture: X86_64
      operatingSystemFamily: LINUX
    cpu: 4 vcpu
    memory: 8GB
    ...)

ecs_client = boto3.client('ecs')
resp = ecs_client.register_task_definition(**config)
print(yaml.dump(resp))

```

We then create ECS tasks to launch Celery workers.

```

ec2_client = boto3.client('ec2')
resp = ec2_client.describe_subnets()
subnets = [v['SubnetId'] for v in resp['Subnets']]

config = yaml.safe_load('''
cluster: python-qc
taskDefinition: celery-gpu-worker
count: 2
overrides:
  containerOverrides:
    - name: celery-worker
      environment:
        - name: AWS_ACCESS_KEY_ID
          value: xxxxxxxxxxxxxxxxxxxx
        - name: AWS_SECRET_ACCESS_KEY
          value:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
      resourceRequirements:
        - type: GPU
          value: "1"
networkConfiguration:
  awsvpcConfiguration:
    subnets: {}
'''.format(subnets))

ecs_client = boto3.client('ecs')
resp = ecs_client.run_task(**config)
print(yaml.dump(resp))

```

If more computational resources are required, the `count` field in the configuration can be modified to adjust the number of ECS workers. In the ECS task configurations, the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are con-

figured to provide *AWS access credentials* [26], which are necessary for the Celery workers to access SQS and S3. It is crucial to avoid hardcoding the AWS access credentials directly into the source code or Dockerfile of the application, as they might be publicly accessible.

Now, we are ready to load the modified `dft_app.py` and run the DFT computation tasks in the AWS cloud.

```
In [3]: from dft_app import dft_energy
results = [
    dft_energy.delay('0 0 0 0; H 0.757 0.5 0; H -0.757 0.5 0'),
    dft_energy.delay('0 0 0 0; H 0.757 0.6 0; H -0.757 0.6 0')]
print(results)

Out[3]:
[<AsyncResult: 32f68705-04f8-483c-83eb-45bf2622ced3>, <AsyncResult: 5833
cf35-7fd2-4d7d-9ef2-f9e1acd4579a>]
```

Please note that the Celery workers do not automatically shut down after completing all computational tasks. To terminate Celery workers, we can stop the corresponding ECS tasks using the

```
aws ecs stop-task
```

command via the CLI or by deleting the allocated resources through the web console. Don't forget this cleanup step or you will be continually charged for ECS computing resources. It is possible to implement functions to monitor the status of the task queue and automatically terminate Celery workers upon completion. However, implementing this feature requires the coordination of several AWS cloud services, which is beyond the scope of Python programming. We will not cover this topic in this book.

As shown by this example, using Celery is generally not complicated. If we have a large number of similar computational tasks to execute, Celery is an excellent choice for batch processing.

Let's briefly summarize the strengths and disadvantages of using Celery in distributed computing. Celery has the following advantages:

- Efficient for batch executing a large number of pre-defined jobs, such as the data generation tasks.
- Quick response. As long as the Celery workers are active, they can repeatedly execute tasks. New tasks incur only a small overhead when retrieving input variables from the broker.
- Thanks to the message-queue architecture, Celery provides resilience against worker failures. If any workers crash, the unfinished tasks can be picked up and continued by other workers.

The disadvantages of Celery include:

- Celery uses JSON serialization to encode the task parameters and the results. Consequently, a general Python object may not be used as the input arguments of a

task. If we need to distribute general Python functions or objects, other distributed executors, such as `dask.distributed` can be considered.

- There is no native support to cloud deployment. Deploying Celery requires certain expertise and knowledge of the cloud architecture.
- Scaling workers up or down is not automatic.
- There is no built-in monitor in Celery for conveniently tracking and managing the status of tasks. The package provides only a CLI tool for monitoring tasks. To manage Celery tasks through a GUI, third-party libraries such as Flower [27] are required.

7.2.2 Task

Dask offers the `dask.distributed` module for distributed computing over a Dask cluster. A Dask cluster consists of a scheduler and several workers, as shown in Fig. 7.10. To install the required components for Dask distributed executors, we can execute

```
$ pip install dask distributed bokeh
```

Here, the package `distributed` provides the necessary tools for managing a Dask cluster, and `bokeh` offers a visualization dashboard for monitoring the status of the Dask cluster.

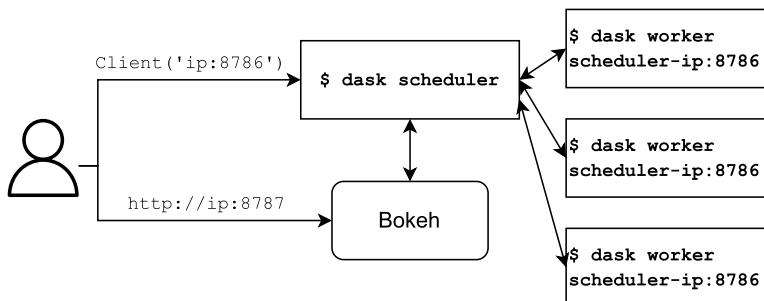


FIGURE 7.10

The architecture of the Dask cluster.

Let's first explore the functionality of `dask.distributed` in a local environment. We can start a Dask cluster by running two commands:

```
$ dask scheduler
$ dask worker localhost:8786
```

The command `dask-scheduler` initializes the scheduler for the Dask cluster. The command `dask-worker` launches workers.

Once the Dask cluster is initialized, the `dask-scheduler` functions as the server. We can then create a `Client` to connect to the scheduler. The `.submit()` method

of the Dask client can be used to offload computation to Dask workers. This operation returns a `Future` object, which represents the pending result of the distributed task. This `Future` object offers the functionality similar to the Python built-in `concurrent.futures.Future` class. Additionally, the Dask client provides a shortcut method, `.gather()`, to quickly retrieve computational results.

```
import pyscf
from dask.distributed import Client

client = Client('localhost:8786')

def dft_energy(molecule):
    mol = pyscf.M(atom=molecule, basis='def2-tzvp', verbose=4)
    mf = mol.RKS(xc='wb97x').density_fit().run()
    return mf.e_tot

molecules = ['O 0 0 0; H 0.757 0.5 0; H -0.757 0.5 0',
             'O 0 0 0; H 0.757 0.6 0; H -0.757 0.6 0']
results = [client.submit(dft_energy, x) for x in molecules]
print(client.gather(results))
```

By default, starting a scheduler automatically serves a bokeh dashboard on port 8787 of the same machine. In a local setup, the dashboard can be accessed at:

`http://localhost:8787/status`

We can use this dashboard to monitor the status of the Dask cluster.

Next, let's examine how to deploy the Dask cluster in the cloud. Deploying the Dask cluster scheduler and the Dask workers may involve different configurations. The Dask scheduler needs to be accessible to the Client running on our local device, either through its public IP address or a proxy. Data transfers via public IP addresses are subject to charges. To avoid data transfers through public IP addresses, communication between workers and the scheduler can occur over the internal network.

If we deploy the Dask scheduler using ECS, additional network configurations will be required to ensure the accessibility of the Dask scheduler. For simplicity, we can launch an EC2 instance to host the Dask scheduler. An EC2 instance can have both public and private IP addresses.¹ After installing the essential packages including `dask`, `distributed`, and `bokeh` in the EC2 instance, we can start the `dask-scheduler` as we have done in the local environment.

On the other hand, for Dask workers, we can still utilize the containerized service. Similar to the ECS deployment method for Celery, we can create a Docker image `python-qc/dask-dft:1.0` using the following `Dockerfile`:

¹ Exposing public IP addresses can introduce security risks. It is recommended to configure the firewall (Security Group) of the EC2 instance to only allow connections from trusted IP addresses.

```
FROM nvidia/cuda:12.0.1-runtime-ubuntu22.04
RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/*
RUN pip install --no-cache-dir gpu4pyscf-cuda12x boto3 dask distributed
```

The ECS task definition for Dask workers is essentially the same as that for Celery workers, which can be reused here.

An ECS worker might be assigned multiple IP addresses. To ensure that the Dask worker attaches to the private address, we should specify the address when starting the `dask-worker` command. This address can be acquired from the AWS web console, or through the `boto3` library. We then override the `image` and `command` in the task definition, leading to the following ECS task configuration to launch Dask workers:

```
ec2_client = boto3.client('ec2')
resp = ec2_client.describe_subnets()
subnets = [v['SubnetId'] for v in resp['Subnets']]
private_ip = ec2_client.describe_instances(
    Filters=[{'Name': 'tag:Name', 'Values': ['dask-server']}])
)['Reservations'][0]['Instances'][0]['PrivateIpAddress']

config = yaml.safe_load('''
cluster: python-qc
taskDefinition: celery-gpu-worker
count: 2
overrides:
    containerOverrides:
        - name: dask-gpu-worker
            image: 987654321000.dkr.ecr.us-east-1.amazonaws.com/python-qc/dask-dft
            :1.0
            command:
                - dask-worker
                - "{0}:port"
            resourceRequirements:
                - type: GPU
                  value: "1"
networkConfiguration:
    awsvpcConfiguration:
        subnets: {1}
'''.format(private_ip, subnets)))
ecs_client = boto3.client('ecs')
resp = ecs_client.run_task(**config)
print(yaml.dump(resp))
```

Compared to Celery, a major advantage of the Dask distributed solution is its support for general Python functions and objects (subject to certain serialization constraints, as discussed in Chapter 6). However, deploying Dask distributed systems in the cloud is complex. Additionally, similar to the autoscaling issues faced with Celery, managing Dask workers and scaling them up or down is relatively inconvenient in cloud environments.

7.2.3 Ray

Ray is more accurately described as a distributed execution framework rather than merely a distributed task executor. It is designed for building and running scalable and distributed applications. It also offers scalable toolkits for developing distributed machine learning applications.

When Ray is used as a distributed task scheduler, its functionality has certain overlaps with Dask. It can offload a general Python function to a remote worker for execution. One of the key advantages in Ray is its native integration with various cloud platforms, such as AWS, GCP, Azure, and Aliyun. Ray allows users to manage a cluster for distributed computation without needing extensive cloud expertise.

A Ray cluster is made of a head node and several worker nodes, as shown in Fig. 7.11. Unlike Dask, which runs a simple service on each node, Ray employs a more complex architecture. On each node, Ray runs multiple services for communication, job scheduling, distributed storage, and other functions. The head node additionally operates an auto-scaling service, which can automatically scale worker nodes according to the workloads. Deploying these complex services does not require a manual, step-by-step setup. By specifying a concise cluster configuration, Ray can automatically configure the virtual cluster on the cloud, including all necessary network connections, message queues, schedulers, storage, autoscaler, and other cloud services required by Ray.

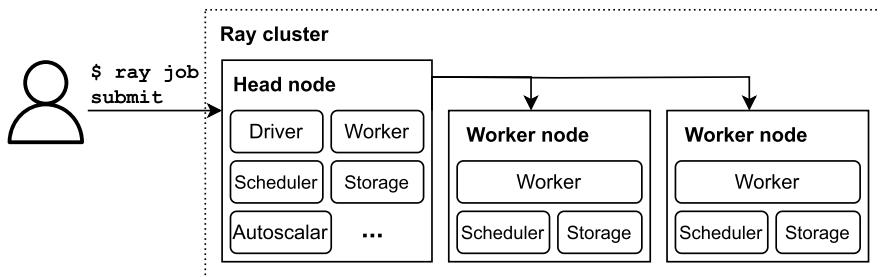


FIGURE 7.11

The architecture of the Ray cluster.

The Ray package integrates both distributed computation functionality and cluster management capabilities. The management of Ray cluster can be operated locally.

Taking the AWS cloud as an example, to deploy a Ray cluster on the AWS cloud, the following packages should be installed:

```
$ pip install -U ray[default,aws] boto3
```

The `boto3` library is required because Ray relies on this library to access AWS APIs.

To offload a computation task to the Ray cluster, we need to prepare and submit a Ray job. A Ray job is typically composed of the following components:

- Functions for distribution. These functions are annotated with the `@ray.remote` decorator. This decorator adds the `.remote()` method to them.
- Calling the `.remote()` method on these functions to generate `Future` objects.
- To ensure completion, retrieving the results of the `Future` objects, using the `ray.get` function.

Below is an example of Ray jobs suitable for the aforementioned DFT calculation task.

```
$ cat dft_job.py
import pyscf
import ray

@ray.remote
def dft_energy(molecule):
    mol = pyscf.M(atom=molecule, basis='def2-tzvp', verbose=4)
    mf = mol.RKS(xc='wb97x').density_fit().run()
    return mf.e_tot

molecules = ['O 0 0 0; H 0.757 0.5 0; H -0.757 0.5 0',
             'O 0 0 0; H 0.757 0.6 0; H -0.757 0.6 0']
futures = [dft_energy.remote(x) for x in molecules]
print(ray.get(futures))
```

To experience the basic usage of Ray jobs, we can start by running the following command to initiate a Ray cluster in the local environment:

```
$ ray start --head
```

According to instructions displayed in the output, we can submit a Ray job using the command:

```
$ export RAY_ADDRESS='http://127.0.0.1:8265'
$ ray job submit --working-dir . -- python3 dft_job.py
```

The double dash `--` notation is a delimiter, which separates the Ray options from the command of the job to be executed remotely. `working-dir` is a local directory where we can put the job scripts and necessary input data.

Please note the `--working-dir` path, which can be a common source of confusion. Cloud environments do not have the shared file systems as that on HPC. Script files, such as `dft_job.py` that we develop locally, are not automatically accessible on the worker nodes. To address the file sharing problem, the command

```
ray job submit
```

will replicate the `working-dir` onto worker nodes. The command for the job (such as `python3 dft_job.py` in our example) is executed within this directory. Therefore, we should *avoid using absolute paths specific to the local machine within the job command*. Additionally, we need to limit the amount of data stored in the `working-dir`. Sending a large volume of data can lead to file transfer errors during the Ray job submission process.

If our job requires large libraries or a significant amount of input data, how to transfer them to the Ray worker nodes?

One approach is to create a script within the `working-dir`, which installs the required libraries and download necessary data from cloud object storage. This script can be executed prior to running the main computation task.

Alternatively, libraries or data can be pre-installed on the worker nodes during the Ray cluster deployment. These data preparation commands can be specified in the cluster configuration, which will be more clear in subsequent discussions.

Let's move on to setting up a Ray cluster on AWS. Below is a sample configuration for a Ray cluster on AWS, which we have saved in a YAML file named `config.yaml`.

```
cluster_name: dft-ray

provider:
  type: aws
  region: us-east-1
  availability_zone: us-east-1a

available_node_types:
  ray.head.default:
    node_config:
      InstanceType: m5.xlarge
      ImageId: ami-080e1f13689e07408 # Ubuntu 22.04
  ray.worker.default:
    max_workers: 3
    node_config:
      InstanceType: p3.2xlarge # GPU instance
      ImageId: ami-0b54855df82eef3a3 # Ubuntu 22.04 with Deep learning tools
  setup_commands:
```

```
- sudo apt install -y python3-pip
- pip install ray[default] boto3 pyscf gpy4pyscf-cuda12x
```

This configuration specifies several key settings:

- The resources are provisioned on the AWS cloud platform, as indicated by the `provider` field.
- An autoscaling scheme is configured, which allows to launch one head node and up to three worker nodes.
- The configuration specifies the EC2 instance type and the operating system for each node. We can search for a public OS image in the AWS marketplace, or a custom image created by ourselves. In this example, `ImageId: ami-0b54855df82eef3a3` is a public deep learning image, which pre-installs the CUDA 12 drivers and runtime tools.
- The `setup_commands` field specifies the initialization commands to run after the nodes boot up. This allows for the installation of new packages and the preparation of data. The software installed in this phase must be compatible with the CUDA toolkit provided by the OS image.

There are more custom options available in the Ray cluster configuration, such as the disk size, the autoscaling strategy, etc. For further configuration options, you can refer to the Ray documentation [28] or the cluster configuration examples available in the Ray source code.

After defining the cluster configuration, we can use the `ray up` command to create or update a cluster.

```
$ ray up -y config.yaml
```

Once all computations are finished, the Ray cluster can be shut down using the command:

```
$ ray down -y config.yaml
```

By executing this command, Ray sends requests to the cloud platform to free and clean up the resources it has allocated. However, this command does not fully track the success of these requests. To prevent potential resource waste and unexpected bills, it is recommended to log into the web console and manually verify the status of each resource.

Summary

In this chapter, we have briefly demonstrated the design and implementation of a quantum chemistry simulation workflow on the cloud. We utilized the containerized cloud service to provide computing resources and the object storage service for data exchange. When configuring cloud services, we chose to manage them by calling

their APIs. This approach was combined with Python templating techniques. Although we have used the AWS cloud to demonstrate this process, similar approaches can be applied to other cloud platforms.

Based on the knowledge of cloud computing, we explored several distributed job executors, including the Celery, Dask, and Ray libraries. We illustrated how to deploy them on the cloud and how to use them to execute computation tasks remotely. Notably, Ray exhibits excellent integration with cloud environments, making it a convenient choice for cloud computing.

During the development of cloud programs, we leveraged an AI coding assistant, specifically the GPT-4 model, to provide prototypes for computation tasks and configuration programs. We then refined the functionalities based on the AI-generated prototypes. The world of cloud computing remains complex. Relevant tools and technologies evolve rapidly. Using AI to assist the development of cloud programs is a very effective method to accommodate this situation.

References

- [1] Amazon Web Services, Amazon machine images (AMIs), <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html>, 2024.
- [2] Amazon Web Services, Amazon EBS volumes, <https://docs.aws.amazon.com/ebs/latest/userguide/ebs-volumes.html>, 2024.
- [3] Amazon Web Services, What is Amazon elastic container service?, <https://docs.aws.amazon.com/AmazonECS/latest/developerguide>Welcome.html>, 2024.
- [4] Amazon Web Services, What is Amazon VPC?, <https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html>, 2024.
- [5] Amazon Web Services, How Amazon VPC works, <https://docs.aws.amazon.com/vpc/latest/userguide/how-it-works.html>, 2024.
- [6] Amazon Web Services, Get started with Amazon S3, <https://docs.aws.amazon.com/AmazonS3/latest/userguide/GetStartedWithS3.html>, 2024.
- [7] Amazon Web Services, Moving an image through its lifecycle in Amazon ECR, <https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-cli.html>, 2024.
- [8] Amazon Web Services, Control traffic to your AWS resources using security groups, <https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html>, 2024.
- [9] Amazon Web Services, Configuration and credential file settings, <https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html>, 2024.
- [10] Amazon Web Services, IAM roles, https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html, 2024.
- [11] The PySCF Developers, Quantum chemistry with Python, <https://pyscf.org/>, 2024.
- [12] R. Li, Q. Sun, X. Zhang, G.K.-L. Chan, Introducing GPU-acceleration into the Python-based simulations of chemistry framework, arXiv:2407.09700, <https://arxiv.org/abs/2407.09700>, 2024.
- [13] X. Wu, Q. Sun, Z. Pu, T. Zheng, W. Ma, W. Yan, X. Yu, Z. Wu, M. Huo, X. Li, W. Ren, S. Gong, Y. Zhang, W. Gao, Enhancing GPU-acceleration in the Python-based simulations of chemistry framework, arXiv:2404.09452, <https://arxiv.org/abs/2404.09452>, 2024.
- [14] Amazon Web Services, Aws fargate for Amazon ECS, https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html, 2024.

- [15] Amazon Web Services, What is Amazon EC2 Auto Scaling?, <https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html>, 2024.
- [16] Amazon Web Services, Amazon ECS task definitions for GPU workloads, <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-gpu.html>, 2024.
- [17] Amazon Web Services, Amazon ECS task networking options for the EC2 launch type, <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-networking.html>, 2024.
- [18] Amazon Web Services, Amazon ECS task metadata endpoint version 3, <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-metadata-endpoint-v3.html>, 2024.
- [19] Amazon Web Services, Tutorial: create a REST API with a Lambda proxy integration, <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-create-api-as-simple-proxy-for-lambda.html>, 2024.
- [20] Amazon Web Services, Define Lambda function handler in Python, <https://docs.aws.amazon.com/lambda/latest/dg/python-handler.html>, 2024.
- [21] Amazon Web Services, Authenticating using an API gateway method, <https://docs.aws.amazon.com/transfer/latest/userguide/authentication-api-gateway.html#authentication-custom-ip>, 2024.
- [22] A. Solem, contributors, Celery - distributed task queue, <https://docs.celeryq.dev/en/stable/>, 2024.
- [23] Dask Development Team, Dask: library for dynamic task scheduling, <http://dask.pydata.org>, 2016.
- [24] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, I. Stoica, Ray: a distributed framework for emerging AI applications, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, 2018, pp. 561–577, <https://www.usenix.org/conference/osdi18/presentation/moritz>.
- [25] A. Solem, contributors, Celery documentation - backends and brokers, <https://docs.celeryq.dev/en/stable/getting-started/backends-and-brokers/index.html>, 2024.
- [26] Amazon Web Services, Environment variables to configure the AWS CLI, <https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-envvars.html>, 2024.
- [27] M. Movsisyan, Flower documentation, <https://flower.readthedocs.io/en/latest/>, 2024.
- [28] The Ray Team, Ray documentation - cluster yaml configuration options, <https://docs.ray.io/en/latest/cluster/vms/references/ray-cluster-configuration.html#cluster-config>, 2024.

PART

High performance
computing with
Python

2

Foreign language interfaces

8

In Python projects, there are various scenarios that require integrating code or libraries written in different programming languages. For computationally intensive tasks, certain functionalities can be developed in programming language like C or C++ to optimize performance. In other cases, projects may need to utilize existing libraries developed in compiled languages, which may be impractical to rewrite in Python. We therefore need an interface to execute functions developed in foreign programming languages.

There are multiple options available for developing such interfaces. Technically, they can be grouped into two categories: Inter-Process Communication (IPC) and function calls within the same memory space.

In the case of IPC, a Python program and a program written in a foreign language can communicate through files, message passing, or shared memory. However, IPC interfaces tend to have relatively high overhead. Function calls within the same memory space can avoid the overhead and complexity associated with IPC. This method, typically implemented through the Python/C API, is the preferred approach.

Several technical aspects need to be considered to develop the foreign language interface. Some of the technical considerations are related to the characteristics of the underlying compiled programming languages and even the operating system. As you read through this chapter, you will find answers to the following questions:

- What are the differences in data types between Python and other languages?
- How can extensions be compiled within a Python package? Additionally, how can CMake be utilized to compile C/C++ libraries within a Python package?
- How to inspect and manage the runtime dependencies of external libraries for Python extensions?
- Are there tools available that can automatically implement the interface between Python and compiled languages, instead of manually using the Python/C API?
- How to call C functions directly from Python without implementing Python/C API code?
- What are the methods for developing interfaces between Python and Fortran?

8.1 Inter-process communication interfaces

When interfacing Python with an external library through IPC, there are essentially two problems to address:

1. How to execute an external library?
2. How to exchange data between Python and the external library?

Most traditional quantum chemistry programs can be executed using shell commands. To interact with these programs, functions `os.system` and `os.popen` in Python provide basic capabilities to execute the shell commands. However, a more robust option is the built-in `subprocess` module, which offers better control and error handling. Particularly, we recommend using the `subprocess.check_call` or `subprocess.check_output` functions from this module. These functions allow for the execution of shell commands while also capturing and managing any exceptions that may occur during the process. The `subprocess` functions can execute commands either in a shell or in direct mode (without starting a shell). It is generally advisable to execute commands in direct mode. In this mode, the command should be provided as a list of separated strings, as demonstrated in the following example:

```
try:  
    cmd = 'pyldd chap01/pyldd'  
    output = subprocess.check_output(cmd.split())  
    print(output.decode())  
except subprocess.CalledProcessError as e:  
    print(e.returncode)  
    print(e.cmd)  
    print(e.output.decode())  
    print(e.stderr)
```

In this IPC interface, communication between Python and an external library is typically file-based. This method involves writing to and reading from temporary files that are accessible to both the Python script and the external library. How can we manage temporary files used for data exchange? Leaving temporary files on the disk after computation is mostly undesirable. To ensure they are cleaned up after use, the `tempfile` module offers functionalities for managing temporary files.

If a single temporary file is used as the input file for an external program, we can use the `tempfile.NamedTemporaryFile` class to create the file. This temporary file is automatically deleted upon closing. When executing the external program, the `subprocess.check_call` function should be placed within the `NamedTemporaryFile` context, as shown in the following example:

```
In [1]: with tempfile.NamedTemporaryFile(mode='w', dir='.') as f:  
    print(f.name)  
    f.write('hello')  
    f.flush()
```

```

    subprocess.check_call(['cat', f.name])
    print(os.path.exists(f.name))
Out[1]:
/home/ubuntu/tmpzpt80qxt
hello
False

```

Please note that writing to `NamedTemporaryFile` is buffered. Its contents might not be immediately visible to the external program. Therefore, the `flush` method is called to ensure all data is synchronized to the file before executing the external shell command.

By default, the temporary files created by `NamedTemporaryFile` are stored in the directory specified by the environment variable `TMPDIR`. To change the location where these files are stored, we can adjust the `dir` keyword argument. The default operation mode of `NamedTemporaryFile` is binary mode, which only allows the writing of `bytes` (or `bytearray`) objects. To enable the writing of text strings, we change the file operation mode to the text mode in this example.

For scenarios that involve multiple temporary files, the `TemporaryDirectory` class provides a convenient solution. This class creates a temporary directory that is automatically deleted upon exiting the `with` context.

```

In [2]: with tempfile.TemporaryDirectory() as tmpdir:
    input_file = f'{tmpdir}/input.dat'
    print(input_file)
    subprocess.call(..., cwd=tmpdir)
    ...
    print(os.path.exists(input_file))

Out[2]:
/tmp/tmpvczd6f5c/input.dat
False

```

8.2 C/C++ interfaces

C and C++ are commonly used to develop extensions for Python. Python has a native C API that allows the integration of C and C++ code into Python projects. This enables Python to invoke C/C++ functions in the same memory space with minimal overhead. An important aspect to consider when developing extensions is the differences in data types between Python and C/C++. Additionally, the compilation settings and build systems must be properly configured.

8.2.1 Data types and type conversion

Basic data types such as integers and floats can be easily mapped between Python and C/C++. However, data structures or classes do not have a straightforward mapping. Python data types have fewer limitations and are generally easier to manipulate than those in C/C++. The constraints imposed by data types in C/C++ can influence the design of the interface and even the algorithm. Managing data types across two languages might require a redesign of the data structure and additional code for memory management in the interface. Below, we will examine the data types that are related to the extensions of Python quantum chemistry programs.

8.2.1.1 Integer

C/C++ have a variety of integer types for representing numbers in different ranges. The `long long` type is a 64-bit integer. It can represent numbers in the range $[-2^{63}, 2^{63} - 1]$. The 64-bit integer is the largest and the default integer type used in the NumPy library. Additionally, NumPy offers `np.int32`, `np.int16`, and `np.int8`, which correspond to the standard 32-bit `int`, `int16_t`, and `int8_t` types in C, respectively.

For big integers, Python can work with arbitrarily large numbers. C/C++ do not have a native data type that supports arbitrarily large numbers. Therefore, extra checks might be necessary to ensure data range before passing Python integers to C functions.

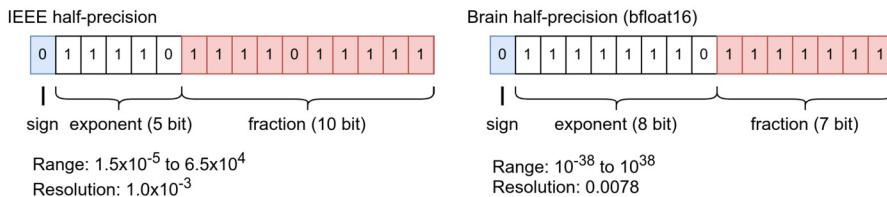
Regarding the Python `bool` type, `bool` can be mapped to any integers in C. `np.bool_` in NumPy is a one-byte data type. It is compatible with `std::boolean` in C++, `Bool_type`, or `int8_t` type in C.

8.2.1.2 Float

Python `float` is a double precision (64-bit) floating-point type, fully compatible with the `double` type in C. It is also the default floating-point type in the NumPy library, known as `np.float64`. NumPy also offers 32-bit and 128-bit floating-point types, `np.float32` and `np.float128`, respectively. They can be mapped to the single precision type `float` and the quadruple precision type `_float128` in C. Please note that the `long double` type in C is guaranteed to be as precise as, or more precise than, the `double` type. However, it may not necessarily meet the precision requirements of `float128`. The precision of `long double` is dependent on the platform. Therefore, `long double` should not be used to represent 128-bit precision data.

The 16-bit floating-point type (half precision) is extensively utilized in machine learning applications due to its performance and memory efficiency advantages. However, the 16-bit float comes in two incompatible formats, the IEEE standard half-precision format and Brain Floating Point format (`bfloat16`), as shown in Fig. 8.1. The `np.float16` type in NumPy adopts the IEEE 754 standard for half-precision floating-point numbers. Machine learning libraries such as PyTorch and Tensorflow, in addition, support the `bfloat16` type.

The data types `std::float16_t` and `std::bfloat16_t` are introduced in the relatively new C++ standards. Unfortunately, these types are not explicitly specified in

**FIGURE 8.1**

IEEE floating point (float16) vs Brain floating point (bfloating16).

the C language standards. Support for half-precision types may not be widely available or consistent across different C and C++ compilers. Extra compiling flags are required to utilize the `float16` type in GCC [1] and Clang [2]. The `bfloat16` type is only available on certain hardware platforms. When incorporating half-precision floating-point numbers in the C/C++ extension for Python, care must be taken as it may involve platform and compiler-dependent issues.

8.2.1.3 String

The specifications for the string type are very different in Python, C, and C++. Python strings are encoded in Unicode, whereas C strings are represented as `char` arrays terminated by a NUL character (`\0`). Consequently, direct manipulation of Python strings within C or C++ functions is not feasible. Only Python `bytes` or `bytearray` objects are compatible with the `char` type in C. To manipulate a Python string in C/C++, the `.encode()` and `.decode()` methods can be used for the conversion between Unicode and `bytes`.

```
a_str = 'py-str'
b_obj = a_str.encode() # converts string to bytes
a_str = b_obj.decode() # converts bytes to string
```

For strings in NumPy arrays, we can use `.astype()` method to convert all strings in the array to bytes, or vice versa [3].

```
str_arr = np.array(['py-str', 'xyz'])
b_arr = str_arr.astype(bytes) # converts all strings in the array to bytes
str_arr = b_arr.astype(str) # converts all bytes to string objects
```

The conversion `.astype(bytes)` results in an array composed of `bytes` objects. The stride of this array, `b_arr`, is equal to the size of the longest string in the `str_arr` array. Shorter strings are padded with 0s at the end to match the length of the longest string. As a result, each element in `b_arr` has the same size, corresponding to `b_arr.itemsize`. Notably, there is no terminating NUL character for the string element within the array. C string functions such as `strcpy` and `strcmp` are unsuitable for individual elements. When working with `b_arr` in C functions, we can iterate over the data with a stride of `b_arr.itemsize` to access each element. The `.dtype` attribute

of `b_arr` reveals a notation, such as `|$6`. It indicates that the array elements are `bytes` objects with 6-byte length in each.

The conversion between Python strings and C or C++ strings can be achieved through the Python/C APIs or the C++ `std::string` constructors:

```
#include <Python.h>
// Create a Python string from C string
PyObject* strObject = PyUnicode_FromString("c-str");
Py_DECREF(strObject);
// Get the pointer to the Python string data
char* strData = PyUnicode_AsUTF8(strObject);
// strData is a null-terminated char array, from which a C++ string can be
// constructed
std::string cpp_str(strData);
```

While manual conversion is possible, it is tedious and error-prone when building the interface module. Tools such as `Cython` and `pybind11` can be utilized to automate these conversions.

8.2.1.4 Class and structure

From the perspective of C/C++ code, an instance of a Python class is essentially a `PyObject`. It cannot be directly mapped to a conventional structure or class in C/C++ code due to the complex data structure of `PyObject`. We must use Python/C APIs to access the attributes in C/C++ code.

To simplify the data conversion between Python and C, one effective method is to serialize the Python object into certain binary format that is compatible with the data structures in C/C++. For example, we can use the Python `struct.pack` function to serialize the instance of a Python class before passing it to C/C++ functions. The serialized binary data can then be interpreted as objects of a C structure or C++ class via a straightforward type casting. Conversely, for the output produced by C/C++ functions, we can deserialize the data using the Python `struct.unpack` function or the `ctypes.Structure` class.

Suppose that an `Atom` class is implemented in C++ that holds the symbol and coordinates of an atom in a molecule.

```
class Atom {
    char symbol[2];
    float x;
    float y;
    float z;
};
```

We can define a serialization protocol accordingly using the `struct.pack` and `struct.unpack` function. The serialization method `dumps` and the deserialization method `loads` can be implemented for this protocol:

```
from dataclasses import dataclass

PROTOCOL = '2sfff'

@dataclass
class Atom:
    symbol: str
    x: float
    y: float
    z: float

    def dumps(self) -> bytearray:
        '''Serializes the Atom object'''
        return bytearray(struct.pack(
            PROTOCOL, self.symbol.encode(), self.x, self.y, self.z))

    @classmethod
    def loads(cls, buffer: Union[bytes, bytearray]) -> Atom:
        '''Deserializes binary data and constructs an Atom object'''
        symbol, x, y, z = struct.unpack(PROTOCOL, buffer)
        return cls(symbol.decode(), x, y, z)
```

The `dataclass` decorator from the standard Python module `dataclasses` [4] can automatically add double underscore methods, such as `__init__` and `__repr__`. The `classmethod` is a special decorator for methods in a class, which enables a method to manipulate the class itself rather than instances of the class. The `classmethod`-decorated method is commonly used to create a new instance as an alternative to the standard class initialization method [5].

Using `ctypes.Structure` is another method to represent data in C structure. For example, we can implement the following serialization and deserialization method with `ctypes.Structure`:

```
class Atom(ctypes.Structure):
    _fields_ = [
        ('symbol', ctypes.c_char*2),
        ('x', ctypes.c_float),
        ('y', ctypes.c_float),
        ('z', ctypes.c_float),
    ]

    def dumps(self) -> bytearray:
        '''Serializes the Atom object'''
        return bytearray(self)
```

```

@classmethod
def loads(cls, buffer: Union[bytes, bytearray]) -> Atom:
    '''Deserializes binary data and constructs an Atom object'''
    if isinstance(buffer, bytearray):
        return cls.from_buffer(buffer)
    else:
        return cls.from_buffer_copy(buffer)

```

More details of `ctypes.Structure` will be explored in Section 8.3.1.4.

Another alternative option is to wrap the data using the `cffi` library [6]:

```

from cffi import FFI
ffi = FFI()
ffi.cdef('''
typedef struct {
    char symbol[2];
    float x;
    float y;
    float z;
} Atom;
''')

@dataclass
class Atom:
    symbol: str
    x: float
    y: float
    z: float

    def dumps(self) -> bytearray:
        '''Serializes the Atom object'''
        c_atom = ffi.new('Atom *')
        c_atom.symbol = self.symbol
        c_atom.x = self.x
        c_atom.y = self.y
        c_atom.z = self.z
        return ffi.buffer(c_atom)

@classmethod
def loads(cls, buffer: Union[bytes, bytearray]) -> Atom:
    '''Deserializes binary data and constructs an Atom object'''
    c_atom = ffi.cast('Atom *', ffi.from_buffer(buffer))[0]
    return cls(c_atom.symbol.decode(), c_atom.x, c_atom.y, c_atom.z)

```

More details of `cffi` will be explored in Section 8.3.2.

If the objective is to manipulate a list of C-structure data, NumPy structured array (see Section 2.2.5 in Chapter 2) is an efficient option to pack or unpack the complex data.

```
atom_type = np.dtype([
    ('symbol', 'S2'),
    ('x', 'f4'),
    ('y', 'f4'),
    ('z', 'f4')
], align=True)

@dataclass
class Atom:
    symbol: str
    x: float
    y: float
    z: float

    def dumps(self) -> bytes:
        '''Serializes the Atom object'''
        arr = np.array([(self.symbol, self.x, self.y, self.z)],
                      dtype=atom_type)
        return bytes(arr.data)

    @classmethod
    def loads(cls, buffer: Union[bytes, bytes]) -> Atom:
        return cls(*np.frombuffer(buffer, dtype=atom_type))
```

Please notice the setting `align=True` when creating the `dtype` for a structured array. This setting ensures the alignment in the memory address for each field. Padding in C structure is automatically applied by C compilers. By default, NumPy structured arrays do not add padding between fields. When the `align` option is enabled, the `dtype` of the structured array becomes equivalent to the one defined using `ctypes.Structure`. We can then initialize the `dtype` of the structured array using `ctypes.Structure`.

```
class Atom(ctypes.Structure):
    _fields_ = [
        ('symbol', ctypes.c_char*2),
        ('x', ctypes.c_float),
        ('y', ctypes.c_float),
        ('z', ctypes.c_float),
    ]
atom_type = np.dtype(Atom)
```

Using NumPy structured arrays, it is convenient to initialize or modify the elements of the structure.

```
arr = np.empty(shape, dtype=atom_type)
arr['symbol'] = ['C', 'H', 'H', ...]
arr['x'] = coordinates[:,0]
arr['y'] = coordinates[:,1]
arr['z'] = coordinates[:,2]
```

8.2.1.5 Array

Multi-dimensional NumPy arrays are quite different from multi-dimensional arrays in C/C++. In C/C++, multi-dimensional arrays are essentially nested pointers. In contrast, a NumPy multi-dimensional array is a contiguous block of data accompanied by shape and strides information. To index elements in a NumPy array, multi-dimensional indices are converted into a single address within this data block. Consequently, the shape and strides information of a NumPy array must be provided to C/C++ functions to perform this index conversion.

In a C/C++ function, it is normal to assume that data in the multi-dimension array are contiguously stored in memory. However, as discussed in Chapter 2, NumPy arrays do not always guarantee data contiguity. The use of array views in NumPy can lead to non-contiguous memory layouts. It is crucial to verify and ensure data contiguity before passing NumPy array data to C/C++ functions.

To ensure contiguous storage, one can create a copy of NumPy array in C-contiguous storage using `np.copy(a, order='C')`. However, copying data adds unnecessary overhead for arrays that are already in C-contiguous order. To circumvent this overhead, functions such as `np.asarray` or `np.require` can be employed.

```
a = np.asarray(a, order='C')
a = np.require(a, requirements=['C'])
```

These functions only trigger a data copy when necessary.

8.2.1.6 Function pointer and callback

Unlike in C/C++, Python does not have a special data type like a function pointer for callback functions. In Python, functions are first-class objects, which can be treated in the same way as any other Python objects. To invoke a Python function callback from C/C++, one method is to use Python/C APIs to decode and execute the callback Python object. Alternatively, a function pointer to a C wrapper can be created, which then calls the Python callback function. This can be achieved using the `ctypes.CFUNCTYPE` function or the `callback` wrapper provided by the `cffi` module. More details on `ctypes` callbacks will be discussed in Section 8.3.1.

Invoking callbacks in a C/C++ function may accidentally disrupt the lifecycle management of Python objects if the callback function returns any Python objects to C. The objects created within the callbacks might be freed by the garbage collector

once program control returns to the C/C++ programs. Accessing the memory block of those Python objects can lead to illegal memory operations.

8.2.2 Cython

Cython is an excellent tool for integrating C and C++ programs in Python. By translating Python code to C code, Cython can significantly improve code efficiency. Furthermore, Cython can generate interfaces to simplify the process of calling C functions from Python. Thus, Cython can serve as a layer to connect Python with external C/C++ libraries. Many scientific computing libraries, including well-known ones like NumPy, SciPy, and scikit-learn, utilize Cython to optimize specific functionalities and embed external libraries into Python.

Cython supports the compilation of pure Python code. However, it is more common to write programs directly in the Cython language. Cython program files are typically denoted with a .pyx suffix. The Cython language is based on the Python language. It introduces extra keywords to help Cython compiler infer data types for each variable. Many Cython language keywords are Python keywords with a prefix of c, such as `cdef`, `cimport`, and `__cinit__`. These keywords indicate that the code block or statements in the Cython source code are treated at C-level operations.

For example, the `scipy.special` module has a function called `comb` that computes the binomial value. This function calls a C function which is implemented with Cython language in the SciPy source code `scipy/special/_comb.pyx`. Apart from a handful of data type declarations using `cdef`, the function `_comb_int_long` looks very much like a Python function.

```
cdef extern from "limits.h":
    unsigned long ULONG_MAX

...
cdef unsigned long _comb_int_long(unsigned long N, unsigned long k):
    """
        Compute binom(N, k) for integers.
        Returns 0 if error/overflow encountered.
    """
    cdef unsigned long val, j, M, nterms
    if k > N or N == ULONG_MAX:
        return 0

    M = N + 1
    nterms = min(k, N - k)

    val = 1
    for j in xrange(1, nterms + 1):
        val *= (M - j + 1) // j
    return val
```

```

# Overflow check
if val > ULONG_MAX // (M - j):
    return 0

val *= M - j
val /= j

return val

```

To structure Cython code effectively, we can distribute the source code across multiple files based on their roles or functions. Function signatures, along with class and data type declarations are usually placed in a Cython header file, denoted by a .pxd suffix. The implementations of the function and class are collected in .pyx files. By using the `cimport` keyword, we can import the declaration code from a .pxd file into a .pyx file. To include regular C/C++ header files, such as `math.h`, the statement

```
cdef extern from "..."
```

can be employed.

When compiling a Cython program, only the .pyx files are processed by the compiler. Similar to the compilation of C/C++ source code, the compiler searches for the dependent .pxd files and C/C++ headers along the paths specified by the `-I` options or the `include_path` option in `cythonize()`, as well as those in the `sys.path`.

There are two common methods to compile Cython code into a Python extension.

The cythonize compiler

We can execute the `cythonize` compiler from command line, for example:

```
$ cythonize --inplace scipy/special/_comb.pyx
```

This command creates a Python importable shared library named

```
_comb.cpython-310-x86_64-linux-gnu.so
```

in the directory where the source code lives.

Cython compiler does not directly generate the shared library. In fact, it first translates the Python code into C code, then invokes C/C++ compilers to generate the shared library. The `cythonize` compiler also generates an intermediate C file for each .pyx file in the source code directory. This file contains details of the generated C functions as well as the corresponding Python/C APIs, which is helpful for tracking performance issues and bugs in the interface.

The setuptools extension

The `setuptools` extension can be configured to compile Cython code. For example, the simplest `setup.py` for compiling `_comb.pyx` would be:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize('scipy/special/_comb.pyx'),
    package_data={'my_pkg': ['scipy/special/_comb.pyx']},
)
```

We can then build the extension from command line:

```
$ python setup.py build_ext --inplace
```

8.2.3 pybind11

Cython is designed to provide a Python-like coding experience. It targets Python programmers, allowing them to quickly develop Python extensions based on Python code. From the C++ side, `pybind11` is designed to use C++ templates to generate Python bindings by C++ compilers. While both tools are capable of creating Python extensions, they employ different approaches and are suited for different use cases. Cython is appropriate for optimizing existing Python code. `pybind11` is better suited for exposing the functionalities of a C++ program to Python.

`pybind11` is used by many Python libraries. For example, PyTorch utilizes `pybind11` to create Python extensions. To simplify the development workflow for PyTorch extensions, it provides a tool, `torch.utils.cpp_extension`, based on `pybind11`. There is a PyTorch tutorial [7] that guides you through the process of building PyTorch extensions using `pybind11`.

`pybind11` has a very intuitive abstract for defining a Python interface. A typical code sample is like:

```
#include <pybind11/pybind11.h>

double square(double x) {
    return x * x;
}

PYBIND11_MODULE(example_module, m) {
    m.def("square", &square);
}
```

There are several methods to compile the `pybind11` extension. Depending on whether the extension is compiled in a Python project or a C++ project, different methods are recommended. In a Python project, one can install the `pybind11` Python package (`pip install pybind11`) and use it in `setup.py`, for example:

```
from setuptools import setup
from pybind11.setup_helpers import Pybind11Extension

setup(ext_modules=[
    Pybind11Extension('example_module', ['example.cpp']),
])
```

Building this `setup.py` with the command

```
$ python setup.py build_ext --inplace
```

we obtain the module Python module `example_module` which can be directly imported into a Python program.

```
In [1]: import example_module
        print(example_module.square(9))
Out[1]: 81.0
```

More examples for using the `pybind11` library in Python projects are available in its GitHub repository [8].

For C++ projects, it's convenient to use the CMake system to compile Python extensions. Assuming that the `pybind11` source code is located in a directory named `pybind11` within the codebase of a C++ project, the following CMake configuration enables the compilation of a Python extension:

```
add_subdirectory(pybind11)
pybind11_add_module(example_module example.cpp)
```

8.2.4 Compiling C++ code into a Python module

In a Python project, utilizing the `setup.py` script with the `setuptools.Extension` function is a standard method for compiling C++ extension. As previously demonstrated, compiling the extension requires executing the command:

```
$ python setup.py build_ext --inplace
```

For example, we can create a simple `setup.py` to handle several C++ files as follows:

```
from setuptools import setup, Extension

setup(ext_modules=[

    Extension(
        'example.simple_cpp',
        sources=['src/cpp/example_A.cpp', 'src/cpp/example_B.cpp'],
        language='c++',
```

```
    )],  
)
```

This configuration example specifies three key aspects:

- The extension is a Python module named `example.simple_cpp`. On Linux, the module file `simple_cpp.cpython-310-x86_64-linux-gnu.so` is generated within the `example` directory.
- The extension is comprised of two C++ files.
- Compilation of the extension requires a C++ compiler.

By adding more settings to the `Extension` configuration, such as `extra_compile_args`, `libraries`, `library_dirs`, `include_dirs`, and `extra_link_args`, developers can build extensions with more sophisticated requirements. Let's examine how the C/C++ extensions can be configured in the following scenarios.

Enabling OpenMP

It can be achieved by adding the compiler flag `-fopenmp` to `extra_compile_args` and `extra_link_args`:

```
Extension(  
    ...  
    extra_compile_args=[ '-fopenmp' ],  
    extra_link_args=[ '-fopenmp' ],  
)
```

Using Intel compilers

One simple way to enable Intel compilers is to set the environment variables `CC=icc` and `CXX=icpc` before running `setup.py`. The `setuptools` package reads environment variables such as `CC`, `CFLAGS`, `LDFLAGS`, and others to configure compilers and compile options universally. Please note that `CXXFLAGS` is an exception, which is not supported by `setuptools`.

Linking to external libraries

External libraries, such as BLAS and LAPACK, are commonly used in quantum chemistry programs. To build the extension using external libraries, we need to configure the `libraries` and `library_dirs` entries. For example, to use the Intel MKL (Math Kernel Library) in the program, we need the following command line options for GCC compilers, as suggested by the Intel MKL link adviser [9],

```
-L${MKLROOT}/lib/intel64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core
```

These options can be translated to the configuration:

```
Extension(  
    ...
```

```

        libraries=['mkl_intel_lp64', 'mkl_sequential', 'mkl_core'],
        library_dirs=[f'{os.getenv("MKLROOT")}/lib/intel64'],
)

```

Most shared libraries on Linux follow the naming convention `lib{name}.so`. When configuring the `libraries` option for `Extension`, we should only include the *name* portion of the shared library. `setuptools` can prepend the notation `-L` to each directory listed in `library_dirs` and `-l` to each name listed in `libraries`. However, some shared libraries may not follow the conventional naming pattern. For instance, on certain Linux distributions, the BLAS library is located at `/usr/lib/libblas.so.3`. To link against such a library, we can write

```
libraries=[':libblas.so.3']
```

This setting generates the command line option `-l:libblas.so.3`, allowing the linker to locate `/usr/lib/libblas.so.3` rather than the one with the standard name `libblas.so`.

If a Python project invokes C/C++ extensions, it is necessary to correctly configure the paths to the dynamic libraries, similar to running standard C/C++ applications. On Linux, one approach is to set the `LD_LIBRARY_PATH` environment variable. Another method is to embed the `rpath` information directly into the extension by using the linker option `-Wl,-rpath`. This option can be added through the `extra_link_args` parameter. For instance, the settings below can add the directory of the Python project to the dynamic library searching path:

```

if platform.system() == "Linux":
    link_args = ['-Wl,-rpath,$ORIGIN']
elif platform.system() == "Darwin": # MacOS
    link_args = ['-Wl,-rpath,@loader_path']
else:
    link_flags = []

Extension(
    ...
    libraries=['mkl_intel_lp64', 'mkl_sequential', 'mkl_core'],
    library_dirs=[f'{os.getenv("MKLROOT")}/lib/intel64'],
    extra_link_flags=link_flags,
)

```

Here, `$ORIGIN` and `@loader_path` refer to the directory where the extension is located, for Linux and macOS, respectively.

In standard C/C++ programs, we might encounter version and path issues in dependent libraries. Similar problems can also arise with C/C++ extensions in Python. On Linux, several tools can help us to identify the runtime problems of a Python extension:

- `readelf`. ELF (Executable and Linkable Format) is a standard format stores variable symbols, runtime paths, dependencies and other information for a shared library. For more details about how ELF operates, you can refer to Chapter 7 of the book *Computer Systems: A Programmer's Perspective* [10]. `readelf` is the tool used for inspecting the contents of ELF files.
- `ldd`. This tool can display the dependencies of a shared library.
- `nm`. This tool can list all global symbols in an shared library [11]. We can use this tool to identify which function symbols in a shared library depend on external libraries.

Extensions with MPI

To enable MPI (Message Passing Interface) in C/C++ extensions, we need to use MPI compilers to compile the source code. Setting the environment variables `CC=mpicc` and `CXX=mpicxx` would apply the MPI compilers to all extensions. However, if MPI is only required for one of the extensions, we can limit the scope of the MPI compiler by customizing the `build_ext` component of the setup configuration. The workload of compilation in `setup.py` is performed by the `build_extension` method of the `build_ext` class. For MPI extensions, we can configure compilers before calling this method. After compiling the MPI extension, compiler settings should be restored since they are shared by all extensions.

```
from setuptools import setup, Extension
from setuptools.command.build_ext import build_ext

class MPIExtension(Extension):
    pass

class MPIBuild(build_ext):
    def build_extension(self, ext: Extension):
        if not isinstance(ext, MPIExtension):
            return super().build_extension(ext)

        # Back up the existing settings.
        compiler_c = self.compiler.compiler_c
        compiler_cxx = self.compiler.compiler_cxx
        compiler_so = self.compiler.compiler_so
        linker_so = self.compiler.linker_so
        self.compiler.set_executables(
            compiler_c='mpicc', compiler_cxx='mpicxx',
            compiler_so='mpicxx', linker_so='mpicxx')

        build_ext.build_extension(self, ext)

    # Restore old compiler settings
    self.compiler.set_executables()
```

```

        compiler_c=compiler_c, compiler_cxx=compiler_cxx,
        compiler_so=compiler_so, linker_so=linker_so)

setup(
    ext_modules=[
        Extension('example.serial_ext', sources=['serial_ext.cpp'],
                  language='c++'),
        MPIExtension('example.mpi_ext', sources=['mpi_ext.cpp'],
                     language='c++'),
    ],
    cmdclass={"build_ext": MPIBuild},
)

```

Invoking CMake

Certain Python packages, such as PyTorch, employ CMake to compile their extensions. In these scenarios, `setup.py` is only used to trigger the `cmake` commands. The approach is to bypass the compilation tasks within the `build_ext.build_extension` method, and invoke the `cmake` command directly. If other preprocessing or post-processing steps are required, they can be integrated within this method as well.

```

from setuptools.command.build_ext import build_ext

class CMakeExtension(Extension):
    pass

class CMakeBuild(build_ext):
    def build_extension(self, ext: Extension):
        if not isinstance(ext, CMakeExtension):
            return super().build_extension(ext)

        # Process cmake configuration
        cmd = ['cmake', f'-B{self.build_temp}']
        self.spawn(cmd)

        # Process building
        cmd = ['cmake', '--build', self.build_temp]
        if hasattr(self, "parallel") and self.parallel:
            cmd += [f"--parallel={self.parallel}"]
        self.spawn(cmd)

setup(
    ...
    ext_modules=[
        CMakeExtension('my_pkg.ext_via_cmake', sources=[]),

```

```
    ],
    cmdclass={"build_ext": CMakeBuild},
    package_data={'my_pkg': ['src/CMakeLists.txt']},
)
```

Additionally, in the `pyproject.toml` file, we include the `cmake` package as a build-time dependency.

```
[build-system]
requires = ['setuptools', 'wheel', 'cmake']
```

NumPy header files

When the statement `cimport numpy` is used in a Cython code, it is necessary for the code to access the NumPy header files. The paths of these header files can be obtained using `numpy.get_include()`. They are then assigned to the `include_dirs` parameter of the `Extension` object in the following manner:

```
setup(
    ...
    ext_modules=cythonize([
        Extension('example.cython_ext', ['src/example/cython_code.pyx'],
                  include_dirs=[numpy.get_include()]),
    ]),
    ...
)
```

8.3 Foreign function interfaces

Although Python offers C APIs to invoke C functions, utilizing these Python/C APIs is inconvenient since it requires an additional layer written in C. Actually, Python can execute C functions through FFI (Foreign Function Interface) without the need to write or compile any Python/C API code. In Python, `ctypes` and `cffi` are popular solutions that leverage FFI to call C functions.

8.3.1 Ctypes

`ctypes` is a Python standard library for interfacing with C functions from Python. However, the use of `ctypes` is limited by certain restrictions:

- This approach has a relatively higher overhead for issuing function calls compared to native Python/C APIs.
- `ctypes` does not support native C++ functions.
- `ctypes` can only load shared libraries that are compiled with the `-fPIC` option.

Generally, developing a Python/C interface with `ctypes` involves two steps:

1. Loading the shared library with the `ctypes.CDLL` function.
2. Declaring the prototype for C functions, which involves specifying `argtypes` for the types of all input arguments and `restype` for the type of the return value.

For example, we can use `ctypes` to interface with the BLAS function `ddot_`, which is then employed to compute the inner product for two NumPy arrays.

```
import numpy as np
import ctypes

libblas = ctypes.CDLL('libblas.so.3')

c_int_p = ctypes.POINTER(ctypes.c_int)
c_double_p = ctypes.POINTER(ctypes.c_double)
libblas.ddot_.argtypes = [c_int_p, c_double_p, c_int_p, c_double_p, c_int_p] # (1)
libblas.ddot_.restype = ctypes.c_double

def blas_ddot(x: np.ndarray, y: np.ndarray) -> float:
    assert x.size == y.size
    assert x.dtype == y.dtype == np.float64
    assert x.ndim == y.ndim == 1
    px = x.ctypes.data_as(c_double_p)
    py = y.ctypes.data_as(c_double_p)
    size = ctypes.c_int(x.size)
    incx = ctypes.c_int(x.strides[0] // x.itemsize)
    incy = ctypes.c_int(y.strides[0] // y.itemsize)
    out = libblas.ddot_(size, px, incx, py, incy)
    return out
```

The function prototype for `libblas.ddot_.argtypes` at line (1) is declared according to the signature of `ddot_`:

```
double ddot_(int *n, double *dx, int *incx, double *dy, int *incy);
```

8.3.1.1 Function prototype

The C function prototype `argtypes` is a list of `ctypes` converters that not only reflect the data type information but also define how to convert each Python input argument to a C variable. When calling a C function, `ctypes` invokes the converter's `from_param` method [12] to validate the data type of the input argument and to perform necessary conversions. `ctypes` built-in data types, such as `ctypes.c_int` and `ctypes.c_double`, have their own `from_param` methods. For classes implemented in our own projects, we can customize `from_param` to integrate with the `ctypes` interface. In the `from_param`

method, we can pack the necessary attributes into a C-structure variable and then return its memory address. This approach eliminates the need to manually write conversion statement for each input argument when calling C functions. For instance, we can implement the `ctypes` interface for the `Atom` class, which we introduced in Section 8.2.1, and utilize it in a C function prototype.

```
import struct

@dataclass
class Atom:
    symbol: str
    x: float
    y: float
    z: float

    def dumps(self) -> bytearray:
        '''Serializes the Atom object'''
        return bytes(struct.pack(PROTOCOL, self.symbol.encode(),
                                 self.x, self.y, self.z))

    @classmethod
    def loads(cls, buffer: Union[bytes, bytearray]) -> Atom:
        '''Deserializes binary data and constructs an Atom object'''
        symbol, x, y, z = struct.unpack(PROTOCOL, buffer)
        return cls(symbol.decode(), x, y, z)

    @classmethod
    def from_param(cls, obj: bytes):                                     # (1)
        # Passing the pointer of struct
        return ctypes.cast(obj, ctypes.c_void_p)

    @property
    def _as_parameter_(self):                                         # (2)
        return self.dumps()

    ...

atom_distance.argtypes = [Atom, Atom]
atom_distance.restype = ctypes.c_float
atom1 = atom2 = Atom('H', 0, 0, 0)
dist = atom_distance(atom1, atom2)
```

The data conversion method `from_param` at line (1) is invoked only if the function prototype (defined by `.argtypes`) for the C function is declared. If the function prototype is not specified, the `_as_parameter_` attribute of each input argument will be utilized

and passed to the C function. Therefore, we additionally develop the `_as_parameter_` attribute, as shown at line (2), for this scenario.

8.3.1.2 NumPy array

NumPy arrays integrate smoothly with `ctypes`. There are several methods to pass a NumPy array, more precisely the pointer of the NumPy array, to C functions. One method is to access the `.ctypes` attribute of a NumPy array to obtain the pointer. Taking the BLAS `ddot_` function as an example, this method simplifies the C function interface to:

```
import numpy as np
import ctypes

libblas.ddot_.restype = ctypes.c_double

def blas_ddot(x: np.ndarray, y: np.ndarray) -> float:
    assert x.size == y.size
    assert x.dtype == y.dtype == np.float64
    assert x.ndim == y.ndim == 1
    size = ctypes.c_int(x.size)
    incx = ctypes.c_int(x.strides[0] // x.itemsize)
    incy = ctypes.c_int(y.strides[0] // y.itemsize)
    out = libblas.ddot_(size, x.ctypes, incx, y.ctypes, incy)
    return out
```

However, this interface still requires multiple checks to verify the data type, array shape, and data contiguity before passing the pointer. To simplify this process, the `numpy.ctypeslib` module provides an elegant wrapper `ndpointer` to automate the checks and data type conversions. As illustrated in the code snippet below, by specifying `np.ctypeslib.ndpointer` with parameters such as `dtype` and `ndim`, we can bypass the data type conversion and relevant checks in the `blas_ddot` function.

```
import numpy as np
import ctypes

c_int_p = ctypes.POINTER(ctypes.c_int)
narray_p = np.ctypeslib.ndpointer(ctypes.float64, ndim=1)
libblas = ctypes.CDLL('libblas.so.3')
libblas.ddot_.argtypes = [c_int_p, narray_p, c_int_p, narray_p, c_int_p]
libblas.ddot_.restype = ctypes.c_double

def blas_ddot(x: np.ndarray, y: np.ndarray) -> float:
    assert x.size == y.size
    size = ctypes.c_int(x.size)
    incx = ctypes.c_int(x.strides[0] // x.itemsize)
```

```
incy = ctypes.c_int(y.strides[0] // y.itemsize)
out = libblas.ddot_(size, x, incx, y, incy)
return out
```

8.3.1.3 Pointers

For standard Python objects, such as integers, floats, and strings, one cannot obtain their address using `ctypes` functions. If a pointer is desired to reference the value of a standard Python object, a temporary `ctypes` object must be created to store the value of the Python object. The pointer to this temporary `ctypes` object can then be passed to C functions. Below is an example demonstrating how to obtain pointers for `bytes`, a NumPy array, and an `int`.

```
import ctypes
liblapack = ctypes.CDLL('liblapack.so')
# work is the workspace buffer for the general eigenvalue solver dgeev_
lwork = 4000
work = np.empty(lwork)
liblapack.dgeev_(
    ctypes.c_char_p(b'N'),
    ctypes.c_char_p(b'V'),
    ...,
    work.ctypes,
    ctypes.byref(ctypes.c_int(lwork)),
    ...)
```

In this example, several temporary objects are created. The `ctypes.c_char_p` function creates a temporary copy for `bytes` and returns a pointer of the copy. To pass the pointer of an integer, we create a `ctypes.c_int` object with the value of the integer. Then `ctypes.byref` is used to obtain a reference to the `ctypes.c_int` object, which can be treated as a pointer.

8.3.1.4 Structure and complex numbers

`ctypes` offers a comprehensive support for C struct types through the `ctypes.Structure` class. One typical use case of `ctypes.Structure` is to represent complex numbers. Although the complex number is defined in C99 standard, `ctypes` does not provide a native data type for complex numbers. A complex number is represented by two floating-point numbers stored sequentially in memory. The data type for representing complex numbers can be defined as follows:

```
import ctypes
class c_complex(ctypes.Structure):
    _fields_ = [
        ('real', ctypes.c_double),
        ('imag', ctypes.c_double),
    ]
```

We can use this custom data type to interact with C/C++ functions that require complex numbers as input or output. For example, the BLAS function `zdotu_` computes the dot product of two complex vectors and returns a complex number.

```
double complex zdotu_(int *n, double complex *zx, int *incx, double complex
*zy, int *incy);
```

The function wrapper for `zdotu_` in Python can be implemented as:

```
import numpy as np
import ctypes

libblas = ctypes.CDLL('libblas.so.3')
c_int_p = ctypes.POINTER(ctypes.c_int)
narray_p = np.ctypeslib.ndpointer(np.complex128, ndim=1)
libblas.zdotu_.argtypes = [c_int_p, narray_p, c_int_p, narray_p, c_int_p]
libblas.zdotu_.restype = c_complex

def blas_zdot(x: np.ndarray, y: np.ndarray) -> complex:
    assert x.size == y.size
    size = ctypes.c_int(x.size)
    incx = ctypes.c_int(x.strides[0] // x.itemsize)
    incy = ctypes.c_int(y.strides[0] // y.itemsize)
    out = libblas.zdotu_(size, x, incx, y, incy)
    return complex(out.real, out.imag)
```

8.3.1.5 Callbacks

In certain scenarios, we might want to invoke Python functions within C/C++ code. One example is the use of the `numpy.load` function in a C/C++ function to read an array from a file. To achieve this, we can use `ctypes.CFUNCTYPE` to transform a Python function into a C/C++ callable function pointer. In the C code, the function pointer is passed in as an input argument.

```
void callback_example(void (*np_load)(char *filename, double *buf))
{
    char filename = "example.npy";
    double *buf = (double *)malloc(sizeof(double) * 100);
    (*np_load)(&filename[0], buf);
    free(buf);
}
```

In the Python side, we implement the callback function `np_load` as follows.

```
import numpy as np
import ctypes
```

```
libcb = ctypes.CDLL(...)

@ctypes.CFUNCTYPE(None, ctypes.c_char_p, ctypes.POINTER(ctypes.c_double))
def np_load(filename: bytes, buf: ctypes.c_void_p):
    arr = np.load(filename.decode())
    out = np.ctypeslib.asarray(buf, dtype=arr.dtype, shape=arr.shape)
    out[:] = arr

libcb.callback_example(
    np_load
)
```

The `ctypes.CFUNCTYPE` decorator is used to specify type hints for the callback function pointer. The first argument specifies the type of the return value, which is `None` in this example. It indicates that no return value is expected. The subsequent arguments specify the types of the inputs for the callback function. The function pointer of the Python callback is then passed to the C function `callback_example`. When the callback is invoked in C, the `CFUNCTYPE` wrapper automatically creates a Python object and maps C variables to each argument based on the type hints.

It should be noted that the Python code executed in callbacks does not support every Python feature. For instance, Python cannot catch errors that are raised inside callback functions. If an error occurs within the callback, the control flow will immediately return to the C caller. Consequently, any information about the error will be lost. The C caller will then continue execution and return to its Python caller as if no error had occurred.

8.3.2 CFFI

The CFFI package [6] is another option for loading external shared libraries and binding them to standard Python code. Similar to `ctypes`, `cffi` also offers an ABI (Application Binary Interface) level interface for executing C/C++ functions. Unlike the `ctypes` function prototypes, which are written in Python style, `cffi` incorporates a built-in C language parser that allows for declaring the C function signature in C-style.

For instance, to execute the BLAS `ddot_` function using `cffi`, we can implement the interface code as follows:

```
import numpy as np
from cffi import FFI

ffi = FFI()
ffi.cdef('''
double ddot_(int *n, double *dx, int *incx, double *dy, int *incy);
''')
# Load the external library, similar to ctypes.CDLL
```

```

libblas = ffi.dlopen('libblas.so.3')

def blas_ddot(x: np.ndarray, y: np.ndarray) -> float:
    assert x.size == y.size
    assert x.dtype == y.dtype == np.float64
    assert x.ndim == y.ndim == 1

    # Get the address of array, similar to x.ctypes.data_as(c_double_p)
    px = ffi.cast('double *', x.ctypes.data)                      # (1)
    py = ffi.cast('double *', y.ctypes.data)
    # Allocate one int, similar to ctypes.byref(ctypes.c_int(x.size))
    size = ffi.new('int *', x.size)
    incx = ffi.new('int *', x.strides[0] // x.itemsize)
    incy = ffi.new('int *', y.strides[0] // y.itemsize)
    out = libblas.ddot_(size, px, incx, py, incy)
    return out

```

NumPy library does not provide a tailored wrapper for `cffi` as it does for `ctypes`. Manipulating NumPy array with `cffi` is not as convenient as that with `ctypes`. We need to write the type conversion code for NumPy arrays before calling C functions. We can use the `ffi.cast` method to obtain the pointer to a NumPy array, as shown in line (1).

There exists another way to handle the pointers to NumPy arrays. We can use `cffi` to allocate memory and then utilize it as the data buffer for a NumPy array. For example:

```

float_p = ffi.new('double []', 100)
buf = ffi.buffer(float_p)
arr = np.ndarray(shape=(100,), dtype=np.float64, buffer=buf)

```

The `cffi` pointer object `float_p` can be directly utilized by C functions. The `ffi.buffer` method converts a `cffi` pointer into a bytes-like object, which is the required data buffer format for the `numpy.ndarray` function.

The `cffi` library also provides APIs for managing pointers, C structures, and callbacks. These functionalities are well-documented in the official `cffi` documentation [13].

One major advantage of `cffi` over `ctypes` is its capability to provide *API-level C function integration*. Using the API-level interface code, `cffi` generates an intermediate C file that incorporates Python/C APIs, which can then be compiled into a C extension. This approach provides a simple interface while ensuring optimal performance. For instance, the interface to the `ddot_` function of the BLAS library can be compiled into a regular C extension by running the script:

```
from cffi import FFI
```

```

sig = '''
double ddot_(int *n, double *dx, int *incx, double *dy, int *incy);
'''

ffibuilder = FFI()
ffibuilder.set_source('_blas_lite', sig, libraries=[':libblas.so.3'])
ffibuilder.cdef(sig)

if __name__ == '__main__':
    ffibuilder.compile(verbose=True)

```

The compiled extension can be imported into Python. The C function is accessible under the namespace of the `lib` module.

```

import numpy as np
from _blas_lite import ffi, lib

def blas_ddot(x: np.ndarray, y: np.ndarray) -> float:
    assert x.size == y.size
    assert x.dtype == y.dtype == np.float64
    assert x.ndim == y.ndim == 1

    # Get the address of array, similar to x.ctypes.data_as(c_double_p)
    px = ffi.cast('double *', x.ctypes.data)
    py = ffi.cast('double *', y.ctypes.data)
    # Allocate one int, similar to ctypes.byref(ctypes.c_int(x.size))
    size = ffi.new('int *', x.size)
    incx = ffi.new('int *', x.strides[0] // x.itemsize)
    incy = ffi.new('int *', y.strides[0] // y.itemsize)
    out = lib.ddot_(size, px, incx, py, incy)
    return out

```

We can add the `cffi` extension in `setup.py`:

```

# setup.py (with automatic dependency tracking)
from setuptools import setup

setup(
    name='blas_lite',
    version='0.1',
    cffi_modules=['blas_lite_builder.py:ffibuilder'],
)

```

and the build-time dependencies in `pyproject.toml`:

```

[build-system]
requires = ['setuptools', 'wheel', 'cffi>=1.0.0']

```

8.3.3 Memory leaks

Python employs an effective lifecycle management system for objects based on reference counting. Generally, developers do not have to concern memory leaks in Python.¹ However, exposing C pointers in Python code can introduce memory leaks.

In C/C++, there are two common types of memory leaks:

- *Unreachable memory*: This occurs when memory is allocated but not freed.
- *Dangling Pointer*: This occurs when memory has been freed, but the program still holds references to the pointer. Accessing dangling pointers can lead to *Segmentation Fault* crash.

Normally, unreachable memory is not a concern in Python due to its garbage collection mechanism. A buffer in memory can be created by Python functions and is automatically referenced. When no objects reference the buffer, it is freed by the garbage collector. This is how Python's reference counting system functions, which guarantees that every allocated memory buffer is referenced.

If pointers are created in a Python program, one might encounter memory leaks due to dangling pointers. Python itself is unable to track the contents of these pointers. Consequently, Python may not properly dispose of pointer objects, even if they reference invalid memory addresses.

In certain scenarios, we might want to use Python to manage memory for C/C++ extensions. This involves allocating memory buffers in Python and then passing their pointers to C/C++ functions. For example, below are some valid methods to allocate and pass a memory buffer to a C function.

```
import numpy as np
import ctypes
from cffi import FFI
ffi = FFI()

# Method 1: allocating buffer with ctypes array.
libc = ctypes.CDLL('')
buf = (ctypes.c_byte * n * 4)()
libc.memset(ctypes.addressof(buf), ctypes.c_int(0), ctypes.c_size_t(200))

# Method 2: allocating buffer with numpy array, then extracting its pointer
# with array.ctypes.data_as
buf = np.empty(n, dtype=np.int32)
libc.memset(buf.ctypes.data_as(ctypes.POINTER(ctypes.c_int)),
            ctypes.c_int(0), ctypes.c_size_t(200))
```

¹ Memory leaks in Python can occur due to circular references, which is different to the C-type memory leaks discussed here.

```
# Method 3: allocating buffer with numpy array, then extracting its pointer
# with ctypes.cast
buf = np.empty(n, dtype=np.int32)
libc.memset(ctypes.cast('double *', buf.ctypes.data), ctypes.c_int(0),
            ctypes.c_size_t(200))

# Method 4: allocating buffer with numpy array, then extracting its pointer
# with ffi.cast
ffi.cdef('void *memset(void *, int c, size_t n);')
libc = ffi.dlopen(None)
buf = np.empty(n, dtype=np.int32)
libc.memset(ffi.cast('double *', buf.ctypes.data), ctypes.c_int(0),
            ctypes.c_size_t(200))
```

However, moving these memory buffer allocation code into Python functions can lead to memory leaks due to dangling pointers.

```
import numpy as np
import ctypes
from cffi import FFI

def alloc_buffer1(n):
    buf = (ctypes.c_byte * n * 4)()
    return ctypes.addressof(buf)

def alloc_buffer2(n):
    buf = np.empty(n, dtype=np.int32)
    return buf.ctypes.data_as(ctypes.POINTER(ctypes.c_int))

def alloc_buffer3(n):
    buf = np.empty(n, dtype=np.int32)
    return ctypes.cast('double *', buf.ctypes.data)

def alloc_buffer4(n):
    buf = np.empty(n, dtype=np.int32)
    return ffi.cast('double *', buf.ctypes.data)
```

In these functions, memory buffers are allocated within a Python function, and only the pointers that reference these memory buffers are returned. The lifetimes of the memory buffer and its pointer are not aligned. The memory buffer only survives within the scope of the Python function and is automatically deallocated when the function returns. However, the pointer can be accessed outside of the function. These pointers are dangling pointers and should not be referenced.

Calling C functions with inline buffer allocation, as shown in the examples below, can also lead to memory leaks. The issue stems from the inconsistency between the lifetimes of the memory buffer and the pointer:

```
ffi.cfunc(ffi.cast('double *', np.empty(100).ctypes.data))
ctypes_lib.cfunc(ctypes.cast(ctypes.c_char_p, np.empty(100).ctypes.data))
```

Here, `np.empty()` creates an anonymous NumPy array to serve as a buffer. This buffer object exists only within the scope of the `ffi.cast` and `ctypes.cast` functions. Once the `cast` function completes, the buffer object is deallocated because it is not referenced by any objects. The pointer returned from the `cast` function still references the buffer, leading to the dangling pointer situation.

In some programs, NumPy arrays are allocated, and their pointers are accessed using the `.ctypes.data_as()` method during the call to a C/C++ function. For example:

```
ctypes_lib.cfunc(np.empty(100).ctypes.data_as(ctypes.c_void_p))
```

You might worry about the risk of memory leaks in this scenario. Thankfully, this concern has been addressed in the NumPy library. The pointer provided by `.ctypes.data_as()` maintains a reference to the NumPy array object. This ensures that the lifetime of the NumPy array buffer is not shorter than the lifetime of the pointer. In other words, as long as the pointer returned by `.ctypes.data_as()` is in use, the NumPy array buffer will remain accessible.

There are other scenarios where dangling pointers can emerge. Covering all scenarios is not our objective. The general rule is that whenever a function returns a pointer, we must be cautious about the lifespan of the pointer and the lifespan of the data buffer it references. To avoid issues with dangling pointers, it is crucial to ensure that the data buffer object lives longer than the pointer object.

8.3.4 Duplicated function names in extensions

In a dynamic library developed with C/C++, if the same function (or symbol) name is present in multiple dependent dynamic libraries, the dynamic linker will select the first available symbol from the dependent libraries, following the library searching order defined in `LD_LIBRARY_PATH` and `ld.so.cache`. Through this mechanism, symbols can be overridden at runtime using the `LD_PRELOAD` environment variable [14,15]. When a Python package contains multiple C/C++ extensions where the same function name appears in several extensions, how does Python manage the name duplication? Does Python use only one version of the function across all extensions, or does each extension use its own version of the function?

A typical scenario of this issue is the use of BLAS libraries in C/C++ extensions. Various BLAS implementations are available from different vendors. Different packages may link to different BLAS implementations through extensions. For example, one extension might be built using the Intel multi-threaded MKL library, while another might utilize OpenBLAS.

On Linux, we can conduct a small test to investigate the symbol resolution problem for this scenario. First, we create a file named `test_blas.c` with the following code:

```
void dcopy_(int *, double *, int *, double *, int *);
void test_copy(double *in, double *out, int n)
{
    int incl = 1;
    dcopy_(&n, in, &incl, out, &incl);
}
```

We then compile this code and link it to the MKL library:

```
$ gcc -g -fPIC -shared -o libtest_ld_mkl.so test_blas.c \
-L${MKLROOT}/lib/intel64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lm
```

We also compile and link this code against OpenBLAS:

```
$ gcc -g -fPIC -shared -o libtest_ld_openblas.so test_blas.c \
-L/usr/lib/x86_64-linux-gnu -lopenblas
```

Next, we write a Python test program `ffi_load_test.y` using `ctypes` to load the two libraries:

```
import ctypes
ld_mkl = ctypes.CDLL('libtest_ld_mkl.so')
print(ld_mkl.test_copy)
ld_openblas = ctypes.CDLL('libtest_ld_openblas.so')
print(ld_openblas.test_copy)
```

To inspect the runtime function/symbol resolution, we can set the environment variable `LD_DEBUG=bindings` and then execute the program.

```
$ LD_DEBUG=bindings python ffi_load_test.py
```

The `LD_DEBUG` setting will prompt the system to display details of function bindings performed by the dynamic linker. For more details on the `LD_DEBUG` setting, please refer to the manual of `ld.so`. By searching for the keyword `dcopy_` in the output, we can find messages such as

```
...
516985:      binding file libtest_ld_mkl.so [0] to /opt/intel/lib/
intel64/libmkl_intel_lp64.so [0]: normal symbol 'dcopy_'
...
516985:      binding file libtest_ld_openblas.so [0] to /usr/lib/x86_64-
linux-gnu/libopenblas.so.0 [0]: normal symbol 'dcopy_'
...
```

It indicates that the two `dcopy_` function calls in the two extensions are bound to different BLAS libraries.

When two libraries (or the same library stored in different directories) are imported, by default, the dynamic linker resolves the symbols independently for each library. Please note that this behavior is affected by the setting of `dlopenflags`, which is accessible via `sys.getdlopenflags()`. Some packages, such as MKL and PyTorch, may change this setting and lead to symbol conflicts [16,17]. Each Python extension can have its own separate definitions for global symbols, which will be resolved to different memory addresses. This is different from running a single C/C++ executable. In a C/C++ executable, if multiple shared libraries include functions with the same name, only one of them will be used, typically the first occurrence in the dependent libraries.

When resolving function symbols for each extension, the dynamic linker reads the `NEEDED` entries in the ELF header (which can be checked by `readelf -d`) to determine the dependent libraries. To locate these libraries, the linker then searches through the paths specified in the `RUNPATH` (or `RPATH`) in the ELF header, `LD_LIBRARY_PATH` and `ld.so.cache`, until it finds a match. In the above example, the two extensions have different BLAS libraries in their `NEEDED` entries. The dynamic linker is able to solve the `dcopy_` symbols in different dependencies without any ambiguity.

When the system contains multiple versions of a library, the linker might select an incorrect version. This issue is encountered in both C/C++ and Python programs. To resolve this, one strategy is to explicitly configure the option `-Wl,-rpath`, as discussed in Section 8.2.4. An alternative approach is to use the `auditwheel` tool, which embeds libraries, along with their version information, into Python packages. This tool was explored in Section 1.3.4 of Chapter 1.

8.4 Fortran interfaces

Fortran is a popular language for scientific applications. Many quantum chemistry programs are entirely written in Fortran, particularly for projects that were initialized decades ago.

There are several methods to interface Fortran with Python. One approach is to use FFI (through the `ctypes` or `cffi` libraries) to call Fortran functions from Python, thanks to the Fortran/C interoperability. Another method involves the use of specialized tools like `f2py`. `f2py` simplifies the process by automatically generating the necessary interface C code.

8.4.1 `ctypes` for Fortran

Fortran uses different ABI conventions than the C language. This leads to different treatments for the interface between Fortran and Python than that between C and Python. Below, we outline the key differences one might encounter. More informa-

tion can be found in Fortran/C interoperability documentation [18] and the compiler manual [19].

- *Fortran function (or subroutine) names.* When compiling Fortran programs, compilers engage in name mangling of Fortran function names. More specifically, this process makes the names of Fortran functions case-insensitive and modifies these names by adding underscores. For instance, GNU and Intel Fortran compilers (`gfortran` and `ifort`) convert function names to lowercase and append a single underscore to them [20]. The name mangling process varies depending on the compiler and operating system. Understanding how the compiler and operating system manage this process is crucial when developing the Fortran-Python interface.
- *How arguments are passed.* Fortran passes all arguments by reference, which means any modifications to an argument inside a function are reflected outside that function. This is different from C, where arguments are passed by value.
- *Treatments of strings.* Fortran and C handle strings differently. C strings are terminated with a `\0` character, whereas Fortran maintains the length information for strings separately. When strings are passed to Fortran functions, their lengths must also be provided. The convention for passing string arguments, including how length information is handled, depends on the compiler and platform. In the case of `gfortran` and `ifort`, strings are passed as pointers with their length information appended to the end of the argument list [21].

Below is an example of Python-Fortran interface based on the `gfortran` compiler and the Python `ctypes` module. Consider a Fortran program that offers multiple subroutines for data copying:

```

subroutine dcopy(len, inArray, outArray)
    integer*4, intent(in) :: len
    real*8, intent(in) :: inArray(*)
    real*8, intent(out) :: outArray(*)
    integer :: i
    do i = 1, len
        outArray(i) = inArray(i)
    end do
end subroutine

subroutine strcpy(inStr, outStr)
    character(*), intent(in) :: inStr
    character(*), intent(out) :: outStr
    print*, inStr
    outStr = inStr
end subroutine

```

Assuming its filename is `fcopy.f90`, we can compile this code to produce a shared library `libfcopy.so`.

```
$ gfortran -fPIC -shared -o libfcopy.so fcopy.f90
```

Below is the code to create the Python interface.

```
import ctypes
import numpy as np

libcopy = ctypes.CDLL('libfcopy.so')
libcopy.dcopy_.argtypes = [           # (1)
    ctypes.POINTER(ctypes.c_int),
    ctypes.c_void_p,
    ctypes.c_void_p,
]
libcopy.strcopy_.argtypes = [           # (2)
    ctypes.c_char_p,
    ctypes.c_char_p,
    ctypes.c_size_t,  # length of inStr
    ctypes.c_size_t,  # length of outStr
]

def dcopy(a: np.ndarray) -> np.ndarray:
    assert a.dtype == np.float64
    assert a.ndim == 1
    assert a.flags.contiguous
    b = np.empty_like(a)
    size = ctypes.c_int(a.size)
    libcopy.dcopy_(ctypes.byref(size), a.ctypes, b.ctypes)
    return b

def strcpy(a: bytes) -> bytes:
    assert isinstance(a, bytes)
    b = bytes(len(a))
    libcopy.strcopy_(a, b, len(a), len(b))
    return b
```

To accommodate the name mangling scheme used by `gfortran`, we use the function names `dcopy_` and `strcpy_` in Python. In the `strcpy` function, the length of each string needs to be appended at the end of the argument list. The length parameter is of the `size_t` type, and this variable is passed by value, as required by the `gfortran` convention [21].

Fortran also supports defining sub-programs through the `function` keyword, which are designed to return a value. The method of passing arguments to these sub-programs is the same as that of subroutines. The return value can be accessed through the `restype` attribute of the function in `ctypes`. For example, below is a Fortran function to compute the Euclidean norm of a vector:

```

function dnorm(len, inArray)
  real*8 :: dnorm
  integer*4, intent(in) :: len
  real*8, intent(in) :: inArray(*)
  integer :: i
  real*8 :: norm
  norm = 0.0
  do i = 1, len
    norm = norm + inArray(i) * inArray(i)
  end do
  dnorm = sqrt(norm)
end function

```

Its Python interface can be developed as follows:

```

libcopy = ctypes.CDLL('libfcopy.so')
libcopy.dnorm_.argtypes = [
    ctypes.POINTER(ctypes.c_int),
    ctypes.POINTER(ctypes.c_double),
]
libcopy.dnorm_.restype = ctypes.c_double

```

8.4.2 f2py compiler

The `ctypes` interface between Python and Fortran requires a detailed understanding of the Fortran ABI conventions, which can be difficult to develop and maintain. The `f2py` compiler presents a solution that translates the signatures of Fortran functions into intermediate C code. This C code acts as a bridge for the communication between Python and Fortran. `f2py` is a component of the NumPy library [22].

For example, to compile the Fortran extension for the previous example, we can employ `f2py` to scan the Fortran source code and generate a preliminary signature file named `fcopy.pyf`.

```
$ f2py -m fcopy fcopy.f90 -h fcopy.pyf
```

The contents of the signature file are:

```

$ cat fcopy.pyf
!      -*- f90 -*-
! Note: the context of this file is case sensitive.

python module fcopy ! in
  interface ! in :fcopy
    subroutine dcopy(len_bn,inarray,outarray) ! in :fcopy:fcopy.f90

```

```

    integer*4, optional,intent(in),check(len(inarray)>=len_bn),
depend(inarray) :: len_bn=len(inarray)
    real*8 dimension(len_bn),intent(in) :: inarray
    real*8 dimension(len_bn),intent(out),depend(len_bn) :: outarray
end subroutine dcopy
subroutine strcpy(instr,outstr) ! in :fcopy:fcopy.f90
    character(*) intent(in) :: instr
    character(*) intent(out) :: outstr
end subroutine strcpy
end interface
end python module fcopy

! This file was auto-generated with f2py (version:1.21.6).
! See http://cens.ioc.ee/projects/f2py2e/

```

This signature file is intended for generating the intermediate C code. However, an issue in this signature needs to be addressed before proceeding with the code generation. Within the `strcpy` block, the length of the output argument `outStr` is not defined. Given that the length of the input argument `inStr` can be determined from the actual string arguments at runtime, we can let the length of `outStr` match that of `inStr` using the statement `len=slen(inStr)`. After refining the signature file, we obtain a revised signature file.

```

python module fcopy
interface
    subroutine dcopy(len, inArray, outArray)
        integer*4,intent(in,hide),depend(inArray) :: len_bn=len(inArray)
        real*8,dimension(len_bn),intent(in) :: inArray
        real*8,dimension(len_bn),intent(out),depend(len_bn) :: outArray
    end subroutine

    subroutine strcpy(inStr, outStr)
        character(*),intent(in) :: instr
        character(len=slen(inStr)),intent(out) :: outstr
    end subroutine
end interface
end python module fcopy

```

Next, we can use `gfortran` to compile the Fortran source code and `f2py` to compile the signature file:

```
$ gfortran -fPIC -shared -o libcopy.so fcopy.f90
$ f2py -c -m fcopy fcopy.pyf -L. -lcopy
```

The output is a shared library `fcopy.cpython-310-x86_64-linux-gnu.so`, which can be imported as a Python module.

```

import fcopy
import numpy as np

def dcopy(a: np.ndarray) -> np.ndarray:
    assert a.dtype == np.float64
    assert a.ndim == 1
    assert a.flags.contiguous
    return fcopy.dcopy(a)

def str_copy(a: bytes) -> bytes:
    assert isinstance(a, bytes)
    return fcopy.strcopy(a)

```

Finally, let us discuss how to integrate the `f2py` interface into the Python build system. You may have noticed the `numpy.distutils` module, which offers the `Extension` class to compile Fortran code and the `f2py` interface in `setup.py` [23]. This tool was developed based on the Python standard library `distutils`, which is not compatible with the `Extension` class from the `setuptools` package. In other words, one should avoid using `numpy.distutils` and `setuptools` in the same `setup.py` script. If we opt to use `setuptools` for packaging, how can we compile the Fortran extension and the `f2py` interface with `setuptools`? To address this issue, we can manually generate the intermediate C file using the `f2py` compiler:

```
$ f2py -m fcopy fcopy.pyf
```

This command generates two additional intermediate files `fcopy-f2pywrappers.f` and `fcopymodule.c`. We can then treat `fcopymodule.c` as regular C code and use only the `Extension` class from `setuptools` to compile this intermediate C file.

8.5 Rust interfaces

The Rust language has gained popularity in recent years, thanks to its modern design and impressive features. One standout feature of Rust is the performance, which is comparable to that of C/C++. Moreover, Rust's ownership system enforces strict compile-time memory safety checks, which effectively eliminate many of the memory leak issues in languages such as C and C++. These characteristics make Rust a compelling choice for scientific applications that require both high performance and security.

There are several approaches to integrating Rust into Python projects. The Rust community provides the `PyO3` package, which generates Python bindings through Python/C APIs. Rust libraries can also be utilized through its FFI due to its interoperability with C. By leveraging tools such as `ctypes` and `cffi`, one can dynamically

load Rust libraries in Python, similarly to how one would with a standard C library.

To access Rust extensions using FFI, the first step is to compile Rust code to a dynamically loadable library. Taking the previous `dcopy` extension as an example, we can create a Rust project `rust_copy` using the command:

```
$ cargo new --lib rust_copy
```

Then, edit the `Cargo.toml` file to enable the compilation of dynamic library.

```
[package]
name = "rcopy"
version = "0.1.0"

[lib]
crate-type = ["cdylib"]
```

The function `dcopy` can be defined in `src/lib.rs`.

```
#[no_mangle]
pub extern "C" fn dcopy(len: i32, input: *mut f64, output: *mut f64) {
    let size = len as usize;
    let i_vec = unsafe { std::slice::from_raw_parts(input, size) };
    let o_vec = unsafe { std::slice::from_raw_parts_mut(output, size) };

    for i in 0..size {
        o_vec[i] = i_vec[i];
    }
}
```

To ensure compatibility with C ABIs, the notation `pub extern "C"` is specified to disable function name mangling in the library. Additionally, the arguments of the Rust function must be compatible with C built-in data types.

Next, we can execute the command `cargo build` to compile the project, which generates a shared library named `librcopy.so` in the `target/debug` directory. This library can be loaded by `cffi` or `ctypes` in Python, just like a regular C library.

```
import ctypes

c_double_p = ctypes.POINTER(ctypes.c_double)
narray_p = np.ctypeslib.ndpointer(ctypes.float64, ndim=1)
librcopy = ctypes.CDLL('target/debug/librcopy.so')
librcopy.dcopy.argtypes = [ctypes.c_int, narray_p, narray_p]

def dcopy(a: np.ndarray) -> np.ndarray:
    assert a.dtype == np.float64
    assert a.ndim == 1
```

```

assert a.flags.contiguous
b = np.empty_like(a)
librcopy.dcopy(a.size, a, b)
return b

```

Summary

In this chapter, we have explored several methods for interfacing Python with foreign programming languages. For software that utilizes command line input, the interface can be developed using the Python `subprocess` module. If an external program provides accessible functions in memory, the interface can be developed using the Python/C API or the Foreign Function Interface (FFI). By employing these interfacing techniques, we demonstrated how to integrate programs written in C, C++, Fortran, and Rust into Python.

The Python/C API is a standard approach for developing interfaces as importable modules. Although Python/C API has low overhead and better portability, writing Python/C API code is generally complex and involved. For C and C++ projects, tools like Cython and `pybind11` can automate the generation of interfaces, simplifying the process of developing Python/C API interfaces. FFI offers a simpler development process for integrating dynamic libraries. Tools such as `ctypes` and `cffi` can be used to connect Python with libraries compiled in other languages. However, FFI generally incurs more overhead and its interfaces are not as robust as those of the Python/C API approach.

When integrating external libraries in Python, we may encounter several technical challenges that are not present in pure Python programs. We need to understand the differences in data types between Python and other languages to avoid bugs caused by data type incompatibilities. The interface to foreign languages introduces pointers, which can potentially lead to memory leaks in Python. The lifecycle of pointers and the associated memory buffers should be carefully managed to avoid memory leaks. External libraries generally have runtime dependencies. The runtime paths and duplicated symbols in external libraries can be managed similarly to those in typical C/C++ programs.

References

- [1] GNU Compiler Collection Team, Gcc documentation - half-precision floating point, <https://gcc.gnu.org/onlinedocs/gcc/Half-Precision.html>, 2024.
- [2] The Clang Team, Clang documentation - half-precision floating point, <https://clang.llvm.org/docs/LanguageExtensions.html#half-precision-floating-point>, 2024.
- [3] NumPy Developers, Numpy reference - data types, <https://numpy.org/doc/stable/reference/arrays.dtypes.html>, 2024.

- [4] E.V. Smith, Pep 557 – data classes, <https://peps.python.org/pep-0557/>, Jun. 2017.
- [5] D. Bader, Python’s instance, class, and static methods demystified, <https://realpython.com/instance-class-and-static-methods-demystified>, 2024.
- [6] A. Rigo, M. Fijalkowski, Cffi documentation, <https://cffi.readthedocs.io/en/latest/>, 2024.
- [7] P. Goldsborough, Custom C++ and CUDA extensions, https://pytorch.org/tutorials/advanced/cpp_extension.html, 2024.
- [8] W. Jakob, pybind11 — seamless operability between C++11 and Python, <https://pybind11.readthedocs.io/en/stable/index.html>, 2024.
- [9] Intel Corporation, Intel oneapi math kernel library link line advisor, <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html>, 2024.
- [10] R.E. Bryant, D.R. O’Hallaron, Computer Systems: A Programmer’s Perspective, 2nd edition, Addison-Wesley Publishing Company, USA, 2010.
- [11] H. Arora, Linux nm command tutorial for beginners, <https://www.howtoforge.com/linux-nm-command/>, 2024.
- [12] Python Software Foundation, Python documentation - ctypes: specifying the required argument types, <https://docs.python.org/3/library/ctypes.html#specifying-the-required-argument-types-function-prototypes>, 2024.
- [13] A. Rigo, M. Fijalkowski, Cffi documentation - using the ffi/lib objects, <https://cffi.readthedocs.io/en/latest/using.html>, 2024.
- [14] M. Kerrisk, Id.so(8) — Linux manual page, <https://man7.org/linux/man-pages/man8/ld.so.8.html>, 2024.
- [15] M. Kerrisk, The Linux Programming Interface: A Linux and UNIX System Programming Handbook, 1st edition, No Starch Press, USA, 2010.
- [16] PyTorch developers, Pytorch github issue - avoid loading with rtld_global, <https://github.com/pytorch/pytorch/issues/3059>, 2024.
- [17] NumPy Developers, Numpy github issue - how we want to link with mkl, and a missing symbol issue, <https://github.com/numpy/numpy/issues/24824>, 2024.
- [18] FortranWiki, Fortran wiki - C interoperability, <https://fortranwiki.org/fortran/show/C+interoperability>, 2024.
- [19] GNU Compiler Collection Team, Gfortran documentation - naming and argument-passing conventions, <https://gcc.gnu.org/onlinedocs/gfortran/Naming-and-argument-passing-conventions.html>, 2024.
- [20] GNU Compiler Collection Team, Gfortran documentation - naming conventions, <https://gcc.gnu.org/onlinedocs/gfortran/Naming-conventions.html>, 2024.
- [21] GNU Compiler Collection Team, Gfortran documentation - argument passing conventions, <https://gcc.gnu.org/onlinedocs/gfortran/Argument-passing-conventions.html>, 2024.
- [22] P. Peterson, F2py: a tool for connecting Fortran and Python programs, International Journal of Computational Science and Engineering 4 (4) (2009) 296–305, <https://www.inderscience.com/info/inarticle.php?artid=29165>.
- [23] NumPy Developers, F2py and build systems - using via numpy.distutils, <https://numpy.org/doc/stable/f2py/buildtools/distutils.html>, 2024.

Program performance optimization

9

Performance is an essential concern in quantum chemistry programs. Although huge efforts are being made to improve the efficiency of Python implementation [1], its performance for quantum chemistry applications remains insufficient. Libraries such as NumPy and SciPy significantly accelerate the performance of numerical computations in Python, making quantum chemistry applications more feasible. However, there is still a noticeable performance gap compared to the well-optimized C/C++ programs.

How can a Python quantum chemistry program be made to run faster? Some might say, rewriting the program in C++. While rewriting Python code in C++ is one option, the same performance improvement can be achieved through various approaches without resorting to C++ programming. How can we optimize Python programs with minimal cost, preferably without writing C++? This is the main focus of this chapter.

Performance optimization involves various types of knowledge. Blindly rewriting Python code in C/C++ will not magically improve the performance. C/C++ primarily reduces the cost of dynamic dispatching performed by the Python interpreter. To achieve optimal performance, additional efforts are necessary beyond just rewriting code. Before turning to C/C++ solutions, there are several strategies that can be employed to optimize the performance of a Python program:

- What steps should be taken to proceed with performance optimization in a Python program?
- Which profiling schemes can be utilized, and how should the profiling output be interpreted?
- What are the bottlenecks in a program? How can each bottleneck be identified? What strategies are required to optimize these bottlenecks?
- If I/O operations are a bottleneck, what techniques can be employed to optimize I/O performance?
- What is the cost for executing a Python statement and what are the corresponding costs in C/C++? How much performance improvement can be expected from rewriting a specific piece of Python code in C/C++?
- How to accurately estimate the cost of Python code, in terms of the bytecode, and the source code of the Python interpreter?
- Which tools are available for compiling Python code? Which one should be chosen for a particular project?

- How would Python JIT compilation improve the performance? In which scenarios are compiled languages exclusively suitable, where Python JIT compilation or other compilation techniques are not applicable?
- How can programming techniques such as memoization and lazy evaluation be used to optimize performance?

9.1 Principles for performance optimization

Optimization strategies

To enhance the execution speed of a program, there are multiple strategies one can employ, including improving algorithms, optimizing implementation, and utilizing parallel computing techniques.

Improving algorithms are often the most effective approach. By applying improved algorithms, one may achieve substantial speed improvements or handle larger problem sizes without the need for additional computational resources.

Parallel computing can also enhance the speed of program execution and enable the handling of larger systems. However, parallelization can introduce additional complexity and challenges, such as non-deterministic behavior, increased fault rates, and the scalability limit.

Optimizing code implementation, without altering the underlying algorithms, falls in between. It encompasses techniques that are relatively simpler to manage than designing new algorithms or developing parallelism. This option focuses on refactoring code to be more hardware-friendly, improving data structures for efficient data access, and utilizing external tools or libraries to accelerate the program.

Optimization procedure

Typically, the optimization of a Python code implementation can be carried out through the following steps:

1. Establishing a correct version as a reference point, which forms the foundation for unit tests to ensure the correctness of future refactoring.
2. Establishing benchmark tests to verify the effectiveness of the optimization.
3. Profiling the program to identify the slow parts.
4. Optimizing one function at a time, and regularly running tests to ensure correctness.
5. Versioning each effective optimization, which can be used for future reference and potential rollback.

Cost estimation

It is highly beneficial to develop an understanding of the computational costs associated with various computation operations.

Knowing the costs of Python and C/C++ statements can provide valuable insights for program optimization. This knowledge can help us to identify which parts of

the code can be executed more efficiently and to estimate the potential performance improvements if rewriting code in C/C++. With a better understanding of these costs, we may achieve an appropriate design of the data structure and algorithm from the beginning.

Operating system-related costs are another significant factor affecting performance, which are often overlooked. These overheads may be encountered in both Python and C/C++ programs. Rewriting Python programs in C/C++ is unlikely to minimize this overhead. They are difficult to optimize using conventional methods.

9.1.1 Cost comparison between Python and C/C++ operations

CPU can execute calculations at a very high speed. The working frequency of a CPU is typically in the range of several GHz. This implies that each CPU cycle takes a very short amount of time, typically around 0.5 nanoseconds or even less. Most computer devices cannot match the processing speed of the CPU. Executing single operations in a program is generally slower than the CPU's operating speed. To measure the speed in relation to CPU performance, we can use CPU cycles as the unit of measurement.

How fast can Python execute its instructions? Table 9.1 exhibits a cost estimation (in CPU cycles) for common Python operations. For comparison, a single operation in C/C++ typically has an overhead of less than 10 CPU cycles. The estimations do not consider the impact of cache misses, which is a critical factor in the performance of C/C++ code. A cold (previously unaccessed) memory access may result in a penalty exceeding 100 CPU cycles. These performance penalties can be observed in Python program as well.

When comparing individual operations in C/C++ to those in Python, how much performance difference can one expect? Below, we provide a summary and comparison for the costs associated with Python operations and their corresponding C/C++ operations.

Atomic instructions

Regular atomic Python instructions typically consume 50 - 100 CPU cycles. The following cases are examples of atomic Python operations that fall in this category:

- Arithmetic expression, such as `a * b`;
- Indexing an element in a list or a tuple;
- Referencing an attribute of an object;
- A single iteration step in the clause `for i in range(n)`.

The corresponding operations in C/C++ typically consume only a few CPU cycles. For these operations, the performance difference between native C/C++ program and Python code lies between 10 and 100 times.

Please note that a single Python statement can be a complex operation that involves multiple atomic instructions to execute. For instance, the `import module` statement in Python is a complex operation. The `dis` library can tell us how a Python statement can be decomposed into individual instructions. By summing up the costs

Table 9.1 Cost estimation for Python operations.

Operation	Code example	Estimated costs / CPU cycles
Assignment	a = ...	10 - 20
Referencing a local variable		10 - 20
Referencing a constant	200	10 - 20
Referencing a global variable		100
Referencing an attribute	object.a	100
Creating a tuple	(3, 5, 7)	50 - 100
Creating a list	[3, 5, 7]	100
Creating a dict	{3: 5}	100
Creating a set	{3, 5, 7}	100
Create an instance	object = cls()	> 200
Create a NumPy array	a = np.asarray()	> 1000
Indexing list / tuple	a[i]	50
Indexing array	a[i]	200
Indexing tensor	h[i,j,k,l]	400
Dict lookup	dct[key]	100
Dict insert	dct[key] = val	100
Single arithmetic expr.	+, *, /, &, ==, abs(), ...	50 - 100
List append	lst.append(3)	100
List comprehension	[... for x in ...]	20 - 50
Dict comprehension	{x: ... for x in ...}	100
Unpacking sequence	*tupl	100
Unpacking dict	**dct	100 - 200
Range iterator	for _ in range(n)	10 - 20
Enumerate	for i, _ in enumerate(...)	100
zip	for x, y in zip(xs, ys)	100
Branching	if ... else ...	10 - 20
Data type conversion	float(3.2), np.array(lst)	100 - 10,000
Call function	fn(...)	> 100
Call method	object.method(...)	> 100
Display	print()	> 10,000
eval		> 10,000
exec		> 10,000

of individual atomic instructions, we can estimate the overall cost of a complex Python operation. We will explore more details of the `dis` library and the optimization of Python statements in Section 9.3.

Dictionary lookup

Dictionary lookup in Python is slightly slower than regular atomic instructions, estimated to be around 100 CPU cycles or more, depending on the data types of

the keys. The process of dictionary lookup involves two steps: hashing and key lookup. The hashing functions for Python built-in data types are implementation in C. Their implementations can be found in various Python source code files named `{datatype}object.c`, under the name `{datatype}_hash`.

For scalar objects like integers, floats, booleans, and single elements of `bytes`, the hashing process is extremely fast, with a cost estimated to be around 10 - 30 CPU cycles. After taking into account the subsequent lookup function and other Python object-related operations, the overall cost of dictionary lookup remains around 100 CPU cycles. In the case of containers, such as tuples, sets, or strings, the hashing cost is *proportional to the number of elements within the object*.

`unordered_map` is the C++ equivalent of the Python dictionary. There are various implementations of `unordered_map` and hashing functions in C++ [2]. Their performance can vary on different tasks. The efficiency of `unordered_map` is largely influenced by the performance of the hash function. For standard C++ data types, accessing `unordered_map` using `std::hash` is estimated to be near 100 CPU cycles, which is comparable to the performance of a Python dictionary. Therefore, converting a Python dictionary to a C++ `unordered_map` is unlikely to yield significant performance improvements.

Referencing a variable

Referencing a global variable in Python requires a dictionary lookup in the global namespace. Its cost is comparable to retrieving a value from a dictionary using a string-type key, which is approximately 100 CPU cycles.

Unlike global variables, local variables within a function are not stored in a dictionary. Instead, they are accessed directly through indexing on a C array, a process that typically requires about 1 CPU cycle. The subsequent operations performed by the Python interpreter increase the cost by approximately 10 CPU cycles. As a result, accessing a local variable can be 5 - 10 times faster than accessing a global variable.

In C/C++, referencing or assigning local variables requires approximately 1 CPU cycle. The cost is slightly higher to reference an attribute of a class, which would take 2 - 3 CPU cycles. C/C++ is roughly 10 times faster than Python for referencing local variables.

Indexing an element of a vector in C++ (or array in C) consumes only 1 CPU cycle in the ideal case (in the vectorized code). This can result in performance improvements of up to 100 times compared to the explicit iteration in Python.

Vectorized operations in NumPy arrays

Vectorized operations in NumPy can be as fast as the well-optimized C/C++ implementation. A rough estimation for the cost is 3 to 5 CPU cycles per element. However, operating individual elements of a NumPy array are inefficient, being slower than the operations on regular Python lists or tuples.

A single arithmetic expression in C/C++ takes 3 to 5 CPU cycles, which is nearly 10 times faster than arithmetic expressions in Python. In a vectorized scenario, the cost can be reduced to just 1 CPU cycle per individual arithmetic expression. If the

vectorized code satisfies the SIMD (Single Instruction Multiple Data) pattern, the CPU can utilize SIMD instructions to execute multiple operations in a single cycle. For instance, the AVX2 instruction set can process 16 floating-point operations in one cycle. This suggests a great potential for performance optimization in arithmetic-intensive Python code. With proper optimization, a performance enhancement of more than 1000-fold could be achieved.

Branching

The cost of the `if ... else ...` branching in Python is relatively low, around 10 CPU cycles. This estimation does not account for the cost of evaluating the condition. Evaluating the condition requires multiple instructions and consumes more than 100 CPU cycles.

Branching in C/C++ is often regarded as computationally expensive due to its high latency. Certain hardware supports speculative execution based on branch predictions. If the prediction is accurate, the cost of branching is reduced to only 1 CPU cycle, a situation often encountered in loop control statements. We will discuss the latency of branching in more detail later.

Function calls

The process of a Python function call typically takes at least 100 CPU cycles. Additionally, the Python interpreter needs to reference each input argument before issuing the call, which takes approximately 10 - 20 CPU cycles per argument. Function calls involve the process of transferring execution frames and the environment to the callee. During this process, the callee performs operations such as dispatching the input parameters, allocating them to the local variable array. These processes are the main source of overhead in function calls, although they are all implemented in C and do not involve any operations of the Python interpreter.

If the function parameters include variable-length arguments (`*args`) and keyword arguments (`**kwargs`), the function calling process requires the construction of a tuple or a dictionary as intermediate variables, adding an extra 50 to 100 CPU cycles.

The cost of a direct function call in C/C++ is approximately 10 CPU cycles, which is about 10 times faster than in Python. However, calling a *virtual method* in C/C++ is significantly slower than a direct function call, requiring nearly 100 CPU cycles. If the methods of a Python class are translated to virtual methods of a C/C++ class, the performance benefits might not be substantial.

Data type conversion

The costs of data type conversions in Python vary significantly. The implicit data type conversions within the arithmetic expressions, such as converting an integer to a float ($8 \rightarrow 8.0$) in the expression `8*2.5`, have negligible overhead. Explicit data type conversions, like `int(1.25)` or `float(15)` for scalar variables, have a cost comparable to regular function calls, ranging from 100 to 200 CPU cycles. When type conversions involve strings, such as `float('3.75')` or `str(20)`, the cost is much higher, which may require 200 - 2000 CPU cycles. Conversion between `numpy.ndarray` and the Python

list or tuple is particularly costly. For example, converting a list to an array using `np.array(lst)` not only incurs the overhead of creating a new `numpy.ndarray`, but also requires explicit data type conversions for each element in the list. This conversion process can take thousands or more CPU cycles to complete.

Data type conversion between characters/string and numerical values is expensive in C/C++ as well. Particularly, it can take 100 - 1000 CPU cycles to perform these conversions using `std::cin` or `scanf` in C/C++. That is saying, rewriting a text parser in C/C++ to process string data and handle type conversions might not lead to significant performance improvements.

I/O operations

Reading a chunk of data using the `.read()` method of a Python file object is limited by the disk performance. Writing chunked data is generally faster because the write operation is buffered. The read and write operations in Python only add a negligible overhead to the highly optimized `read` and `write` functions provided by the operating system. Using C/C++ code does not offer a performance advantage in this context.

Running shell commands

The overhead of executing shell commands in Python is substantial. Functions like `subprocess.call` or `os.system` can incur an overhead of nearly 1 millisecond. When function calls in the same memory space are applicable, we should opt for direct function calls rather than executing shell commands. This technique has been explored in Chapter 8.

In summary, rewriting Python code in C/C++ can offer different levels of improvement depending on the specific tasks. Normally, rewriting Python code in C/C++ can yield a speedup of 10 to 50 times.

9.1.2 Hardware and operating system overhead

The overhead associated with hardware and operating systems (OS) can affect the performance of both Python programs and the rewritten C/C++ programs.

Memory allocation

In general, memory allocation is an expensive operation. The cost of memory allocation is not a fixed value. The time taken to allocate memory depends on various factors, including the memory allocator library, the operating system, the hardware architecture, the size to be allocated, and the runtime system load. As a rough estimate, we might experience an overhead within the range of 100 - 10,000 CPU cycles. When using the `malloc` function to allocate memory, the cost is not negligible. The penalty becomes even more severe in parallel program since `malloc` is not scalable. To address this issue, Python implements its own memory allocator, `_PyObject_Malloc`, which is used to allocate memory for objects that are smaller than 512 bytes. When optimizing computationally intensive Python code using C/C++, one can improve performance by avoiding frequent memory allocations. One such

example in quantum chemistry programs is the integral computation, which we will discuss in Chapter 12.

Page fault

Programs interact with main memory through small *pages*, and a *page fault* arises when the required page is not available in main memory [3,4]. When a page fault occurs, the OS and hardware have to perform a costly work to initialize the page, which can consume thousands of CPU cycles (roughly a few microseconds). Page faults are commonly encountered when accessing data in newly allocated memory, such as the data buffer created by the `numpy.empty` function. Filling up the newly allocated data buffer takes more time compared to reusing a previously allocated one. In the benchmarking tests below, you can observe the overhead associated with filling data into a newly allocated array.

```
In [1]: buf = np.empty((1000,1000,1000))

In [2]: %time buf[:] = 0.0
CPU times: user 193 ms, sys: 1.73 s, total: 1.93 s
Wall time: 1.95 s

In [3]: %time buf[:] = 0.0
CPU times: user 582 ms, sys: 0 ns, total: 582 ms
Wall time: 581 ms
```

The first assignment triggers numerous page faults and frequently pause for the page initialization before executing the data filling operations. The second test primarily involves data transfer between the CPU and memory, reflecting the memory bandwidth. The cost difference in these tests can be attributed to the impact of page faults.

To reduce the occurrence of page faults, it is recommended to reuse memory buffers as much as possible and maintain a minimal memory footprint within the program. For instance, many NumPy functions provide the `out` keyword, which serves as a buffer for storing the results that are returned. By utilizing the `out` buffer, we can decrease the memory footprint and enhance the performance of NumPy functions.

Context switch

Context switches are a fundamental mechanism employed by OS to share CPU resources among multiple processes or threads. They occur implicitly within the OS kernel space, and are not directly manageable by user programs. A context switch involves the saving of the current state of a running process (or thread) and the restoration of another state. Context switch is a resource-intensive task. Each switch can add an overhead of 1 - 10 microseconds to the computation time.

In Python, for threads created by the `threading` module, the overhead of context switches is generally not a concern. This is largely due to the Global Interpreter Lock (GIL), which limits the parallel execution of Python bytecode. However, in scenarios where the `multiprocessing` module or OpenMP is employed, requesting too many

Table 9.2 Latency and throughput of various hardware.

Device	Latency	Throughput
CPU register	1 cycle	10,000 MIPS/core
L1 cache	4-5 cycles	
L2 cache	10-20 cycles	
L3 cache	50 cycles	
Memory	200-300 cycles	10 - 20 GB/s
LAN	1 μ s	10 GB/s
SSD	0.1 ms	1 - 10 GB/s
HDD	10 ms	100 MB/s - 1 GB/s
Internet (WAN)	100 ms	1 MB/s - 10 MB/s

processes or threads can result in more frequent context switches and increased overheads in system kernel space.

9.1.3 Latency and throughput

Latency and throughput are two critical metrics that influence code performance:

- Latency refers to the time that computer takes to complete an operation.
- Throughput (or bandwidth for certain devices) measures the number of operations that can be completed within a specific period of time.

In the simplest scenario, the computer would execute requests one after another, synchronously. The faster the computer processes a request, the higher throughput achieved. In other words, the time taken to complete one task (latency) is inversely related to the throughput.

However, reducing latency is generally very challenging for various technical reasons. This can limit the overall throughput in the synchronous execution mode. Some computer hardware can work in *concurrent mode*, which allows the computer to start new tasks without waiting for the completion of the previous task. The concurrent mode decouples the latency and throughput, allowing for high throughput even if the latency for executing individual tasks remains high.

Table 9.2 exhibits the latency and throughput of hardware and devices that are related to the optimization of quantum chemistry programs. We will further analyze their latency and throughput below.

CPU

CPU employs pipelines [5] to achieve low latency. Modern X86 CPUs have long pipelines with 20 - 30 stages to complete one operation. CPUs run at a high clock frequency to shorten the time period of each stage. If an operation, such as branching, stalls the pipeline, the penalty is significant. The CPU would have to flush and restore the entire pipeline, resulting in a latency of 20 - 30 CPU cycles.

On the other hand, each CPU core contains multiple execution units. This feature allows the CPU to execute multiple instructions in each cycle, achieving parallel execution. The throughput, measured in millions of instructions per second (MIPS), can exceed the operational frequency of the CPU.

CPU cache

The latency of retrieving data from memory is a critical factor affecting CPU performance. To mitigate this issue, CPU chips are equipped with various types of caches, such as L1I (L1 instruction) cache, L1D cache (L1 data), L2 cache, TLB (Translation Lookaside Buffer) cache.

The L1 cache is specifically optimized for speed, with data indexing incurring only 4 to 5 cycles of latency. However, the trade-off is the limited size of the L1 cache. For instance, modern X86 CPUs typically have only 32 KB of L1 cache per core. Due to the complexities in designing a fast L1 cache, it is unlikely to have larger L1 cache in the future. The L2 cache is slower than the L1 cache, with a latency of 10 - 20 cycles. The L3 cache facilitates data sharing among different CPU cores. It typically has a latency that falls between the L2 cache and the main memory, around 50 CPU cycles. For more detailed information about the latency of CPU cache, one can refer to the Intel documents *Intel 64 and IA-32 Architectures Optimization Reference Manual* [6].

It is not necessary to closely monitor the throughput of the cache. The L1 and L2 caches can operate at the same speed as the CPU working frequency, achieving a throughput of 1 or 2 cache accesses per cycle.

Memory

Accessing data from cold memory, which occurs in the case of a cache miss, has a latency of 200 to 300 CPU cycles. Memory bandwidth is around 20 GB/s per channel. After accounting for multiple channels, the overall throughput of memory accessing can reach nearly 100 GB/s. It should be noted that the memory bandwidth is shared by all CPU cores. The average bandwidth per CPU core is actually quite limited.

Storage

The latency of accessing data on a hard disk drive (HDD) or a solid-state drive (SSD) is a critical factor in performance. The latency of an HDD is around 10 milliseconds. SSDs perform better, being approximately 100 times faster than HDDs, with a latency of about 0.1 milliseconds. Manipulating individual data on HDDs or SSDs is highly discouraged, as it could result in the waste of millions of CPU cycles. For storage-related operations, it is more efficient and advisable to process data in larger blocks or batches.

Network

The performance of different network devices can vary significantly. Some network connections can achieve data transfer rates faster than those of disk access. High-

speed network devices, like InfiniBand, can offer sub-microsecond latency and high bandwidth, which is comparable to the bandwidth of main memory.

9.1.4 Strategies for optimizing latency and throughput

Achieving both low latency and high throughput, although highly desirable, is extremely difficult. The trend in hardware development is shifting from latency-oriented to throughput-oriented. This shift occurs because maintaining low latency is not friendly to energy efficiency. A large number of integrated circuits (ICs) and a significant amount of energy consumption may only result in marginal improvements in latency. For example, modern CPUs are integrating more and more lightweight cores to provide better parallel performance, rather than focusing on boosting the working frequency. This trend should be kept in mind when designing algorithms or implementing new programs.

Although latency and throughput are often tightly coupled, a crucial consideration in cost analysis is to determine whether a specific operation in whether a specific operation in the program is primarily limited by latency or throughput. When optimizing such operations, one may need to prioritize and optimize one factor over the other. Generally speaking, if the goal is to complete as many operations as possible in certain period of the time, such as loading a large integral tensor from disk, the optimization should focus on throughput. On the other hand, if the priority is the time required to complete a particular operation, minimizing latency should be the focus. For example, the request to access data in remote storage is in this category.

Different optimization strategies can be considered to improve latency and throughput. For latency, the common strategies include:

- *Utilizing low-latency cache.* The CPU cache is subject to this category, which helps to hide the latency of memory access. Message queues and various in-memory databases are frequently used to hide the latency of network access.
- *Enhancing cache utilization.* This strategy focuses on the effectiveness of cache. To reduce the cache miss rates, algorithms and data structures may need to be redesigned to enhance the temporal and spatial locality of data.
- *Asynchronous execution.* To hide the latency of I/O, I/O operations can be executed in background to overlap with computational tasks. For more details, refer to Chapter 10.
- *Prefetching data.* Some hardware is capable of making simple predictions about which data will be used in the near future and read them in advance. When the access pattern of the data is predictable, software prefetching in the program can be performed to asynchronously load data into cache in advance. More discussions can be found in Section 9.6.3.
- *Function inline and code unrolling.* These techniques can reduce the overhead of function call and branching. By inline, we mean removing unnecessary function calls and replacing them with the actual code blocks. Unrolling means expanding loops to reduce the branching count. More details can be found in Section 9.4.1.1

- *Memory mapping.* Normally, data must be transferred from external devices, such as disk, to a memory buffer before they can be accessed by the processor. The technique of memory mapping allows the processor to directly access the data on external devices as if it were accessing data in memory. This eliminates the overhead of copying data and the associated system calls. When this technique is used for accessing data from disk, further details can be found in Chapter 6 and Section 9.6.
- *Direct memory access (DMA).* Transferring data between a memory buffer and external devices would consume valuable CPU resources if the CPU was used to manage the memory buffer. Certain hardware devices, such as GPU and network adaptors, support the DMA mode to directly write data to a specific memory space without involving CPU. One application of this strategy is the utilization of pinned memory in GPU programming (see more details in Chapter 11). The DMA mode can achieve faster data transfers and reduced communication delays. By using DMA to manage the data transfer process, CPU resources can be freed to perform other tasks, further improving the overall efficiency of the system.

When optimizing program for throughput, the following strategies are often utilized:

- *Utilizing high throughput buffer.* For instance, shared memory can serve as a buffer to facilitate communication between different processes. Further details are discussed in Section 10.9.4 of Chapter 10.
- *Parallel computation.*
- *Accessing data in batches or chunks.* Hardware devices are optimized to process data in batches or blocks, rather than on an individual element basis. For instance, modern CPUs are engineered to read data from memory in the unit of cache-line, which is typically 64 bytes in length. Even if only a few bytes from a cache line are needed, the entire line will be loaded into the cache. Likewise, data on disk are organized into blocks, typically with a size of 512 bytes or 1 KB. Operations involving disk reading or writing are executed on a block-by-block basis.
- *Accessing data sequentially and continuously.* Random access is slow, due to the higher possibilities of cache misses, and the additional traffic caused by accessing the entire cache line or storage block.
- *Compact data representation.* This refers to compressing data in terms of sparsity or other characteristics to minimize the volume of data.

9.1.5 Computation bound and I/O bound

Computation bound and *I/O bound* are terms to describe whether the performance of a process is limited by the speed of the CPU or by the speed of input/output (I/O) operations.

- **Computation Bound:** A process is considered computation bound if it spends the majority of execution time performing arithmetic calculations, or other CPU-

intensive tasks. Its performance is directly linked to the efficiency of the core subsystem (CPU and memory).

- I/O Bound: A process is considered I/O bound if it spends a significant amount of time waiting for data to be read from or written to storage devices or the network. Its performance is mainly limited by the speed of the I/O subsystems (storage and network) rather than the processing capacity of the CPU.

In quantum chemistry programs, the evaluation of integrals is a typical example of a computation-bound task, which we will discuss in Chapter 12. Tasks such as coupled-cluster programs, covered in Chapter 15, can be considered I/O bound when the intermediate tensors are stored on disk or distributed across different computer nodes.

Determining whether a program is computation-bound or I/O-bound can help us choose the appropriate optimization strategies. If a Python program involves computation-bound code, rewriting them in C/C++ could enhance performance. Employing parallel execution is another effective strategy to accelerate computation-bound programs. For I/O-bound programs, we can consider the asynchronous execution to improve performance.

In computation-bound code, there is a special case known as memory-bound, where the performance of the program is heavily influenced by cache size and memory bandwidth. In this scenario, computational resources on CPU cannot operate at their maximum capacity because a considerable amount of time is spent waiting to retrieve data from the main memory. For example, the FCI (Full Configuration Interaction) program, as discussed in Chapter 14, is of this category.

9.1.5.1 Is a program computation-bound or memory-bound?

To determine whether a Program is computation bound or memory bound, a simple method is to measure the IPC (instructions per cycle) and cache-miss rates using the profiler `perf`. A low IPC or a high cache-miss rate typically suggests that the program is memory-bound. Modern X86 CPUs are able to execute 4 or more IPC at maximum load. A low IPC implies that the CPU is frequently stalled due to data dependencies or delays in memory access. A high cache-miss rate indicates that the CPU frequently waits to retrieve data from the main memory.

For example, here is the output of `perf` for the matrix transpose operation

```
np.transpose(a).copy()
```

695.31 msec	task-clock	#	0.999 CPUs utilized
2	context-switches	#	0.003 K/sec
0	cpu-migrations	#	0.000 K/sec
6,039	page-faults	#	0.009 M/sec
2,126,239,844	cycles	#	3.058 GHz
1,097,502,672	instructions	#	0.52 insn per cycle
152,534,052	branches	#	219.376 M/sec
2,655,018	branch-misses	#	1.74% of all
branches			

228,338,188	L1-dcache-loads	# 328.398 M/sec
182,549,765	L1-dcache-load-misses	# 79.95% of all L1-
dcache accesses		
80,043,333	LLC-loads	# 115.119 M/sec
15,678,968	LLC-load-misses	# 19.59% of all LL-
cache accesses		

In this profiling, we observe a low IPC (labeled as `insn per cycle`) and extremely high L1 cache miss rate. Apparently, such a matrix transposition operation is a memory-bound task. We will explore `perf` in more detail in Section 9.2.4.

Due to the continuous evolution of hardware architecture and the increase in computational power, some problems that were traditionally considered computation-bound are, in fact, shifting towards memory-bound. For instance, the quantum chemistry FCI problem is traditionally viewed as a floating-point computation-intensive problem. However, due to the utilization of the SIMD instructions on modern CPUs, the cost of data transfer has become the dominant factor in the FCI program. Another example is the evaluation of analytical integrals. In the past, algorithms for analytical integral evaluation were developed and optimized based on the floating-point operation counts. When these algorithms are ported to GPUs, the performance is actually found to be limited by the size of the GPU cache and the GPU memory bandwidth [7].

9.1.5.2 How to accelerate memory-bound applications?

Increasing the CPU working frequency can offer some bonus to memory-bound problems, but the improvement is typically limited. This is because the bottleneck lies primarily in the data transfer between the CPU and memory, rather than the processing speed of the CPU itself. Increasing the number of CPU cores can sometimes improve the speed of cache-bound programs, as more CPU cores mean access to more cache. However, the performance improvement scales up slowly, and in some cases, it may even decrease with an increasing number of cores. This decline is due to the issues introduced by multi-core parallelism, such as race conditions, memory bandwidth contention, and cache coherence issues. These factors can introduce overhead that offsets the bonus of the additional cores.

Optimizing memory-bound programs can be quite complex. To develop efficient programs for modern hardware, the hardware architecture is no longer completely transparent to the data structure and algorithm. C/C++ compilers can only moderately utilize the instruction and cache characteristics of the CPU. Therefore, cache-aware algorithms and data structures are generally required to minimize the penalty of slow memory access.

Compared to the processing speed of CPU, moving data around is quite expensive in modern computer architecture. In some cases, it may be necessary to prioritize memory and cache efficiency over the number of operations. To write hardware-friendly programs, it is crucial to have a comprehensive understanding of the characters of CPU and memory. Readers can refer to the book *What Every Programmer Should Know About Memory* [4] for more details of CPU and memory architecture.

The common consideration in cache-aware algorithms is the memory access pattern, which includes:

- The cache size, or the amount of data that can be cached.
- Data locality, such as the strides in high-dimensional arrays.
- Cache coherence.

9.1.5.3 Optimizing for cache size

As previously demonstrated, transposing a matrix using the naive implementation,

```
a.T.copy(order='C')
```

is not cache-friendly for large matrices. In the tests on a 10000×10000 matrix, the cache-miss rate for the L1 cache is approximately 80%. This high cache-miss rate indicates that the CPU must frequently access data from memory.

Loop tiling is an effective technique to solve high cache-miss rates. Loop tiling divides the giant matrix into smaller blocks and performing matrix transposition within each of the small blocks. If the individual block can fit into cache, the data within each block is likely to be found in cache. This improves cache utilization and reduces latency in memory access.

The following tiled algorithm performs 50% faster than the naive transpose code.

```
def transpose(a):
    block_size = 200
    arow, acol = a.shape
    out = np.empty((acol, arow), a.dtype)
    for c0 in range(0, acol, block_size):
        c1 = min(acol, c0 + block_size)
        for r0 in range(0, arow, block_size):
            r1 = min(arow, r0 + block_size)
            out[c0:c1,r0:r1] = a[r0:r1,c0:c1].T
    return out

a = np.ones((10000, 10000))
b = a.T.copy('C') # 479 ms
b = transpose(a) # 311 ms
```

The CPU used in this test has a 1024 KB L2 data cache. To store both input and output matrices in the cache, each tile can occupy a maximum of 512 KB of cache space. This is equivalent to 64k float64 elements, or a block with dimensions of up to 256×256 elements. Therefore, a block size smaller than 256×256 is suitable for this problem.

9.1.5.4 Optimizing data locality

When the CPU fetches data from memory, it first loads that data into the cache. The cache line is the smallest unit of data that can be transferred between the main

memory and the cache. The size of cache line is 64 bytes on X86 CPU architectures. When a cache line is loaded, it retrieves not only the data at the requested memory location but also the surrounding data within the same cache line. Accessing nearby data within the same cache line does not require new access requests to memory. Consequently, accessing consecutive memory locations is efficient as the data for subsequent memory accesses are likely already present in the cache.

As we discussed in Chapter 2, the shape of high-dimensional arrays (tensors) and the loop order to traverse the tensor have a significant impact on program efficiency. When traversing the tensor, it is preferred to access consecutive memory in the inner loop to maximize the cache efficiency. When an array is stored in row-major order, it is advisable to iterate over the last index in the inner-most loop. For column-major arrays, iterating over the first index in the innermost loop is recommended.

Let us use the matrix multiplication program to illustrate the impacts of strides and loop order to the program performance. The formula for matrix multiplication is

$$C_{ij} = \sum_k A_{ik} B_{jk}. \quad (9.1)$$

A naive implementation of the matrix multiplication would be like the `matmul_nn_v1` function below.

```
import numpy as np
from transpose import transpose
import numba
@numba.njit
def matmul_nn_v1(a, b):
    arow, acol = a.shape
    brow, bcol = b.shape
    assert acol == brow
    c = np.zeros((arow, bcol))
    for i in range(arow):
        for j in range(bcol):
            c_ij = 0
            for k in range(acol):
                c_ij += a[i,k] * b[k,j]      # (1)
            c[i,j] = c_ij
    return c

def matmul_nn_v2(a, b):
    arow, acol = a.shape
    brow, bcol = b.shape
    assert acol == brow
    b_t = transpose(b)                  #(2)
    return matmul_nt(a, b_t)
```

```
@numba.njit(cache=True)
def matmul_nt(a, b):
    arow, acol = a.shape
    brow, bcol = b.shape
    assert acol == bcol
    c = np.zeros((arow, brow))
    for i in range(arow):
        for j in range(brow):
            c_ij = 0
            for k in range(acol):
                c_ij += a[i,k] * b[j,k]
            c[i,j] = c_ij
    return c

@numba.njit
def matmul_nn_v3(a, b):
    arow, acol = a.shape
    brow, bcol = b.shape
    assert acol == brow
    c = np.zeros((arow, bcol))
    for i in range(arow):
        for k in range(acol):
            a_ik = a[i,k] # (3)
            for j in range(bcol):
                c[i,j] += a_ik * b[k,j]
    return c

m = n = k = 5000
a = np.random.rand(m, k)
b = np.random.rand(k, n)

c = matmul_nn_v1(a, b) # 215 s
c = matmul_nn_v2(a, b) # 142 s
c = matmul_nn_v3(a, b) # 77.4 s
```

In `matmul_nn_v1`, we utilize a temporal variable `c_ij` in line (1) to represent the matrix element `c[i,j]`. This local variable reduces the overhead of accessing the matrix element `c[i,j]` in the inner-most loop. In this function, matrix `b` is accessed in a non-contiguous manner, leading to an inefficient memory access pattern. Each time `b[k,j]` is loaded, a whole cache line containing 8 elements is transferred, even though only one element is actually needed. When accessing the subsequent element `b[k+1,j]`, it might not be present in the cache line previously loaded for `b[k,j]`. Unless the cache line containing `b[k+1,j-1]` is still available, accessing `b[k+1,j]` will lead to a

cache miss and the transfer of a new cache line from the main memory, increasing the latency of the computation.

In `matmul_nn_v2`, we reorder the elements in the matrix to ensure sequential memory access. By transposing the `b` matrix at line (2), we impose the contiguous access to the `b` matrix in the inner-most loop. The inner-most loop now iterates over the rows of the transposed `b` matrix, which are contiguous in memory, rather than the columns of the original `b` matrix. This is essentially a new type of matrix multiplication `matmul_nt`, where `t` means the transposition for the second matrix,

$$C_{ij} = \sum_k A_{ik} B_{jk}. \quad (9.2)$$

Matrix multiplication has the complexity $O(n^3)$ and the transpose is of $O(n^2)$. For sufficiently large problems, taking the transpose introduces a negligible overhead on the computation time.

By swapping the loop order of `k` and `j`, as shown in the function `matmul_nn_v3`, we can also achieve the sequential access to the elements of `b`. In this implementation, a temporal variable `a_ik` in line (3) is used to reduce the cost of repeatedly accessing `a[i,k]` in the inner-most loop. The `matmul_nn_v3` function issues two memory access requests for `c[i,j]` and `b[k,j]` in the innermost loop, which is equal to that in `matmul_nn_v1`. Both memory accesses walk over the contiguous indices of the `b` and `c` matrices. The `matmul_nn_v3` implementation achieves contiguous memory access, without introducing additional memory and computation overhead. We can expect `matmul_nn_v3` to be the most efficient among the three versions of matrix multiplication. Despite the overhead of transposing the matrix, `matmul_nn_v2` exhibits better memory access efficiency compared to `matmul_nn_v1`, making it more efficient.

In the performance benchmark, calling the three Python functions directly will not show performance differences. This is because manipulating individual elements of a NumPy array is slow, which adds too much noise to the performance benchmark. We therefore compile the three implementations with Numba JIT (Just-In-Time) compiler. For sufficient large matrices, the performance benchmark shows that `matmul_nn_v2` is about 50% faster than `matmul_nn_v1`. `matmul_nn_v3` performs surprisingly 3 times faster than `matmul_nn_v1`. This significant improvement can primarily be attributed to the auto-vectorization performed by the JIT compiler. We will explore more about the Numba JIT compilation in Section 9.4.1. After disabling auto-vectorization, the performance difference between `matmul_nn_v3` and `matmul_nn_v2` is less than 3%. This difference can be attributed to the overhead of the matrix transposition in `matmul_nn_v2`.

To further improve the cache efficiency of the matrix multiplication program, cache size should also be considered. By applying the loop tiling technique, we achieve a 30% speedup on top of `matmul_nn_v3` (without disabling auto-vectorization).

```
@numba.njit(cache=True)
def matmul_nn_tiling(a, b):
```

```
'''Matrix multiplication a * b with loop tiling'''
arow, acol = a.shape
brow, bcol = b.shape
assert acol == brow
c = np.zeros((arow, bcol))
block_size = 200

for i0 in range(0, arow, block_size):
    i1 = min(i0 + block_size, arow)
    for j0 in range(0, bcol, block_size):
        j1 = min(j0 + block_size, bcol)
        for k0 in range(0, acol, block_size):
            k1 = min(k0 + block_size, acol)
            for i in range(i0, i1):
                for k in range(k0, k1):
                    a_ik = a[i,k]
                    for j in range(j0, j1):
                        c[i,j] += a_ik * b[k,j]
return c
```

9.1.5.5 Optimizing cache coherence

Cache coherence maintains data consistency across multiple CPU caches in a multi-core system. When data within a cache line is accessed by multiple CPU cores, cache synchronization is triggered to ensure consistency whenever any part of the cache line is modified. Specifically, if a CPU core alters the status of the cache line, the other cores must stall, waiting for the cache line to be synchronized across the system. This ensures that all cores have access to the most recent version of the data. The process of maintaining cache coherence can introduce latency and overhead, which may offset the performance gains of multi-core parallelism.

In Chapter 14 on the FCI theory, we will encounter a program that assigns data to a memory buffer based on a precomputed lookup index table. The relevant code block, which is simplified for demonstration purposes, is presented below:

```
from cython.parallel import prange

def build_d(fciwfn, norb, lookup):
    ma, mb = fciwfn.shape
    d = np.zeros((norb,norb,ma,mb))
    for I in prange(mb, nogil=True, schedule='dynamic', chunksize=1):
        for a, i, J, sign in lookup[I]:
            for K in range(ma):
                d[i,a,K,I] += sign * fciwfn[K,J]
    return d
```

When multi-core parallelism is implemented for the variable `I`, the cache line for the last index of the `d` tensor is shared among multiple CPU cores. If one thread updates the `d` tensor, cache synchronization might need to be performed to ensure that all cores have a consistent view of the data.

If the `d` tensor is accessed by multiple CPU cores with multiple memory controllers from different CPU chips, a situation known as non-uniform memory access (NUMA), the overhead for cache synchronization can become more significant. This is a common scenario for programs running on workstations (or cluster nodes) with multiple CPU sockets. To improve cache coherence, we can rearrange the loop order and rewrite the code as follows:

```
from cython.parallel import prange

def build_d(fciwfn, norb, lookup):
    ma, mb = fciwfn.shape
    d = np.zeros((norb,norb,ma,mb))
    for K in prange(ma, nogil=True, schedule='dynamic', chunksize=1):
        for I in prange(mb):
            for a, i, J, sign in lookup[I]:
                d[i,a,K,I] += sign * fciwfn[K,J]
    return d
```

9.1.6 Instruction level parallelism

Modern X86 CPUs can execute various instructions in parallel [8], such as pipeline, super-scalar execution, out-of-order execution, speculative execution, etc. Among these instruction level parallelism (ILP) techniques, the super-scalar instruction SIMD (Single Instruction, Multiple Data) provides the capability to execute multiple arithmetic operations simultaneously. This parallelism feature is a critical factor in the performance optimization of quantum chemistry programs.

SIMD can significantly increase the arithmetic computation throughput of the CPU. The throughput is determined by the width of SIMD registers and the number of execution units, such as the multiplication (MUL) units, the addition (ADD) units, and the fused-multiply-add (FMA) units. The MUL and ADD units each provide one floating-point throughput, while the FMA unit can deliver two (one for multiplication and one for addition). The total throughput of SIMD instructions for one CPU core can be calculated as:

$$\text{Throughput} = \frac{\text{reg. width}}{\text{sizeof(date type)}} \times \text{execution units} \times \begin{cases} 2 & \text{FMA} \\ 1 & \text{MUL or ADD} \end{cases}. \quad (9.3)$$

Table 9.3 shows some typical floating-point throughputs of SIMD instructions. For example, each AVX2 register can handle 256 bits of data, corresponding to 4 `double` values. With two FMA units, the CPU can execute FMA operations in two

Table 9.3 Typical parameters and throughput of SIMD instructions.

Instructions	AVX-512	AVX2	AVX
Register width	512	256	256
FMA units	2	2	0
MUL units	2	2	2
ADD units	2	2	2
FLOP throughput:			
float32	64	32	16
float64	32	16	8

AVX2 registers simultaneously. Hence, the overall FLOP (floating point operation) throughput equals to $4 \times 2 \times 2 = 16$.

Different CPUs have different specifications for instruction sets and execution units, leading to different SIMD throughput. Based on the CPU microarchitectures, we can identify the supported SIMD instruction sets, the number of execution units. Combining with the information of CPU frequency and the number of CPU cores, we can estimate the theoretical peak FLOP throughput

$$\text{SIMD throughput} \times \text{CPU frequency} \times \text{CPU cores.} \quad (9.4)$$

The base frequency of the CPU we used in the tests is 3.3 GHz. The peak performance of the CPU, measured at the level of AVX2 instructions, is $3.3 \times 16 = 52.8$ GFLOPs per core. The FLOP count of matrix multiplication is $2N^3$. Therefore, our previous matrix multiplication implementation achieved only 3.3% of the CPU capacity. Even with the auto-vectorized version which utilizes SIMD instructions, the FLOP efficiency reaches only 6.1%.

In fact, it is very challenging for an application to achieve more than 50% of the peak FLOP performance due to various overheads. Only a few highly optimized functions and libraries, such as the `gemm` (general matrix multiplication) function provided by the OpenBLAS library, are capable of delivering up to 90% of the peak performance.

Please note that blindly using SIMD instructions may not necessarily result in a speedup of the program. There is an (in)famous CPU frequency trap associated with SIMD instructions [9]. When the CPU executes AVX-512 instructions, the overall CPU frequency may decrease, leading to reduced performance for other instructions. If the arithmetic calculations in a program are not intensive enough, the acceleration obtained through SIMD may not outweigh the performance loss caused by the decreased frequency. This can lead to a decrease in program performance. The decision to employ SIMD instructions in the program should be based on the density of arithmetic calculations.

9.2 Profiling

Profiling is a commonly performed task in program optimization. By analyzing profiling results, we can identify the hotspots in a program. Optimization efforts should focus on these hotspots first.

We will use the Schmidt orthogonalization algorithm to illustrate the techniques involved in program profiling and optimization. Given an overlap matrix \mathbf{S} corresponding to a set of basis vectors \mathbf{u} ,

$$S_{mn} = \mathbf{u}_m \cdot \mathbf{u}_n. \quad (9.5)$$

we can determine an upper-triangular coefficient matrix \mathbf{C} to represent the orthonormal basis \mathbf{v}

$$\mathbf{v}_i = \sum_m C_{mi} \mathbf{u}_m, \quad (9.6)$$

$$\sum_{mn} C_{mi} S_{mn} C_{nj} = \delta_{ij}. \quad (9.7)$$

The \mathbf{C} matrix can be obtained through the following Schmidt orthogonalization process

$$\tilde{\mathbf{v}}_j = \mathbf{u}_j - \sum_k \mathbf{v}_k \mathbf{v}_k \cdot \mathbf{u}_j, \quad (9.8)$$

$$\mathbf{v}_j = \frac{\tilde{\mathbf{v}}'_j}{\sqrt{\tilde{\mathbf{v}}_j \cdot \tilde{\mathbf{v}}_j}}. \quad (9.9)$$

where

$$\tilde{\mathbf{v}}_j \cdot \tilde{\mathbf{v}}_j = S_{jj} - \sum_k d_{kj}^2, \quad (9.10)$$

$$d_{kj} = \mathbf{v}_k \cdot \mathbf{u}_j = \sum_m C_{mk} S_{mj}. \quad (9.11)$$

These equations are translated into the function `schmidt_orth` as follows:

```
@profile
def schmidt_orth(s):
    n = s.shape[0]
    cs = np.zeros((n, n))
    for j in range(n):
        fac = s[j,j]
        for k in range(j):
            dot_kj = np.dot(cs[:,k], s[:,j])
            cs[:,j] -= dot_kj * cs[:,k]
            fac -= dot_kj * dot_kj
```

```

if fac <= 0:
    raise RuntimeError(f'schmidt_orth fail. (j={j} (fac={fac}))')
fac = fac**-.5
cs[j,j] = fac
for i in range(j):
    cs[i,j] *= fac
return cs

```

9.2.1 Benchmark tests

Benchmark tests assess the overall runtime performance of a program. These tests require specialized designs, which distinguish them from unit tests and feature tests:

- Benchmark tests are designed to highlight the hotspot and minimize the noise of irrelevant parts.
- The tests can include various problem sizes to identify performance bottlenecks and limitations as the system scales. Many cache efficiency issues might not be accurately identified unless the problem size is sufficiently large.
- To avoid bias towards specific workloads, benchmark tests can include multiple scenarios, such as worst-case tests, best-case tests, and some tests from real-world applications.
- Allow a warm-up period before collecting performance data to ensure system stabilization.
- Collect results together with system configurations, such as hardware specifications, Python version, versions of dependent libraries and compilers.

9.2.2 Python built-in profiling tools

In Jupyter notebook, we can use `%time` and `%timeit` magics to measure the performance of individual Python statements. The `%time` magic command is based on the Python `time` module, and the `%timeit` magic command utilizes the `timeit` module. The two magic commands can be employed in both line and cell modes within Jupyter notebooks.

```

In [1]: from schmidt import schmidt_orth

In [2]: s = np.eye(500)

In [3]: %time c = schmidt_orth(s)
CPU times: user 586 ms, sys: 20.1 ms, total: 606 ms
Wall time: 588 ms

In [4]: %timeit c = schmidt_orth(s)
562 ms ± 750 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

```

```
In [5]: %%time
    for _ in range(10):
        schmidt_orth(s)
CPU times: user 5.6 s, sys: 8.04 ms, total: 5.61 s
Wall time: 5.6 s
```

The output from the `%time` magic command displays timing information labeled as `CPU time` and `Wall time`. `CPU time` indicates the actual amount of time the processor spends on executing the program. It does not include the time spent waiting for I/O operations and periods when the CPU is idle. The term `CPU time` refers to the actual amount of time the processor spends on executing a program. It does not include the time spent waiting for I/O operations or periods when the CPU is idle. On the other hand, `Wall time` denotes the total time that has elapsed in the real world, including the time spent on I/O operations and any other delays. Furthermore, the information on `CPU time` encompasses:

- `sys`: Time spent in the OS kernel space for managing system resources and interacting with hardware.
- `user`: Time spent executing program code, excluding functions in the kernel space.
- `total`: The total time consumed by the program, comprising both user space and kernel space time.

The default output of the `%timeit` magic command reports the average wall time of a code snippet after being executed multiple times. We can utilize the `%timeit -c` command to only measure the CPU time instead. Typically, the time reported by `%timeit` will be marginally less than the wall time reported by the `%time` command. This discrepancy arises because executing the same code multiple times minimizes the effects of initialization, cache misses, and other factors.

If the CPU total time is significantly less than the wall time, it indicates that the CPU is not fully utilized. There are several potential reasons for this:

- The program may be generating a large number of I/O requests, causing the CPU to wait for I/O operations to complete.
- The system might be engaged with other tasks, which can delay the execution of the current program.
- In a multiprocessing parallel program, the main process might be blocked while waiting for synchronization with other processes.
- If the code involves mutexes (see Chapter 10), the execution may be delayed due to the waiting for the lock to be released.

The CPU time can also exceed the wall time, which suggests that the program employs multithreading for parallel computation. Ideally, the ratio of the total CPU time to the wall time should correspond to the number of parallel threads used in the computation.

The `sys` time is another informative metric in the CPU time output. An efficient program should spend as little time in the kernel space as possible. A high `sys` time indicates that the operations in the kernel space have consumed a substantial amount of resources. Unfortunately, most operations in the kernel space cannot be parallelized, which is a limitation to the scalability of the program. In quantum chemistry programs, common reasons for high `sys` time include:

- Heavy I/O operations.
- Frequent memory allocation, deallocation, and page faults.
- Excessive threads or processes running in the system, leading to a high number of context switches.

For a complex Python program, we also want to know the details of its execution efficiency. The Python built-in module `cProfile` is a useful tool for identifying functions that consume the most execution time. In Jupyter notebook, we can use the `%prun` magic command to profile Python code with `cProfile`.

```
In [6]: %prun
    for i in range(5):
        schmidt_orth(s)
1871263 function calls in 3.181 seconds

Ordered by: internal time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
          5    2.179    0.436    3.180    0.636 schmidt.py:3(schmidt_orth)
  623750    0.729    0.000    0.729    0.000 {built-in method numpy.core.
 _multiarray_umath.implement_array_function}
  623750    0.218    0.000    1.001    0.000 <__array_function__ internals
 >:177(dot)
  623750    0.053    0.000    0.053    0.000 multiarray.py:740(dot)
          5    0.000    0.000    0.000    0.000 {built-in method numpy.zeros}
          1    0.000    0.000    3.181    3.181 <string>:1(<module>)
          1    0.000    0.000    3.181    3.181 {built-in method builtins.
 exec}
          1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
 Profiler' objects}
```

The output of `cProfile` is sorted by the execution time spent within each function, as indicated by the line `Ordered by: internal time`. The performance statistics can be sorted by other metrics, such as cumulative time or the number of calls. According to the `internal time`, a significant amount of the computation time is spent within the `ufunc` and `dot` functions from the NumPy library. Identifying the most time-consuming functions is a crucial step in the optimization process. Next, we can shift the focus to the functions that demand the most execution time.

9.2.3 Line profiler

The `cProfile` tool does not provide detailed information about the execution efficiency within a function. To obtain more precise timing within a function, we can use the third-party tool `line-profiler`. This tool offers a line-by-line analysis of a function.

To use `line_profiler` from the command line, we can add the `@profile` decorator to the function we want to profile.

```
@profile
def schmidt_orth(s):
    ...
```

Then execute the following command:

```
$ kernprof -lv schmidt_v1.py
```

The command line option `-v` enables the display of performance statistics after the program execution. The line profiler saves the profiling data in a file with a `.lprof` suffix. To review existing performance statistics, we can use the `line_profiler` module to view the `.lprof` file:

```
$ python -m line_profiler schmidt_v1.py.lprof
```

In a Jupyter notebook, we can load the `line_profiler` module and execute the `%lprun` magic command to profile functions. The `-f` option of `%lprun` specifies the function to be profiled. Multiple `-f` options can be specified to profile several functions simultaneously. Taking the `schmidt_orth` function as an example, the following illustrates how to use the line profiler:

```
In [7]: %load_ext line_profiler

In [8]: %lprun -f schmidt_orth schmidt_orth(s)
Timer unit: 1e-09 s

Total time: 0.812276 s
File: /home/ubuntu/code-examples/schmidt/schmidt_v1.py
Function: schmidt_orth at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
=====
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
3					def schmidt_orth(s):
4	1	5043.0	5043.0	0.0	n = s.shape[0]
5	1	263741.0	263741.0	0.0	cs = np.zeros((n, n))
6	500	81657.0	163.3	0.0	for j in range(n):
7	500	154634.0	309.3	0.0	fac = s[j,j]
8	124750	20412051.0	163.6	2.5	for k in range(j):

```

 9    124750  327342723.0   2624.0      40.3      dot_kj = np.
dot(cs[:,k], s[:,j])
10   124750  362190394.0   2903.3      44.6      cs[:,j] -=
dot_kj * cs[:,k]
11   124750  39026926.0    312.8       4.8      fac -= dot_kj
* dot_kj
12
13     500    203630.0     407.3      0.0      if fac <= 0:
14                                raise
RuntimeError(f'schmidt_orth fail. {j=} {fac=}')
15     500    209276.0     418.6      0.0      fac = fac**-.5
16     500    163808.0     327.6      0.0      cs[j,j] = fac
17   124750  17723304.0    142.1      2.2      for i in range(j):
18   124750  44498893.0    356.7      5.5      cs[i,j] *= fac
19     1      131.0      131.0      0.0      return cs

```

The output from the line profiler is quite intuitive. It displays the total costs, average costs, and relative costs for each line of code. Among these metrics, the average cost (labeled as `Per Hit`) and the relative cost (labeled as `% Time`) are especially informative.

A few lines in a function often consume the majority of the execution time. The `% Time` column helps us identify these critical lines. Optimization should be prioritized for these code blocks. If the hotspots involve external functions, we can include those functions in the tracking list and then rerun the line profiler.

Does the cost of a slow Python statement agree with the cost estimation? The `Per Hit` time here can help us perform this analysis. The `Per Hit` time is measured in nanoseconds, as indicated by the line `Timer unit: 1e-09 s`. In this example, the statements

```
dot_kj = np.dot(cs[:,k], s[:,j])
```

and

```
cs[:,j] -= dot_kj * cs[:,k]
```

together consume 85% of the total execution time. The `Per Hit` time indicates that the average execution time of the two NumPy statements is 2600 and 2900 nanoseconds, respectively. Given the problem size of 500 in this instance, the per-element costs amount to approximately 5.5 nanoseconds, nearly 16 CPU cycles. This is much slower than the cost estimation of NumPy ufunc statements (3 - 5 CPU cycles). Here are some possible reasons for the discrepancy:

- The testing system might be too small. The overhead from function calls and profiling overshadows the `Per Hit` time.
- The matrices are accessed column-wise, whereas the array storage is in the row-major order. This access pattern mismatch leads to cache misses and consequently, higher costs.

Based on this analysis, we slightly optimize the code, leading to the cache-friendly implementation

```
def schmidt_orth(s):
    n = s.shape[0]
    cs = np.zeros((n, n))
    for j in range(n):
        fac = s[j,j]
        for k in range(j):
            dot_kj = np.dot(cs[k], s[j])           # (1)
            cs[j] -= dot_kj * cs[k]                # (2)
            fac -= dot_kj * dot_kj

        if fac <= 0:
            raise RuntimeError(f'schmidt_orth fail. {j=} {fac=}')
        fac = fac**-.5
        cs[j,j] = fac
        cs[j,:j] *= fac
    return cs.
```

We then profile this function using an overlap matrix consisting of 2000×2000 elements. The `Per Hit` time for the statements at line (1) and line (2) is 3000 - 3200 nanoseconds. The per-element costs are decreased to 5 CPU cycles, which agrees with the cost estimation of NumPy functions. This cost suggests that unless techniques beyond NumPy ufuncs are utilized, there is no room for further optimization.

When analyzing the `Per Hit` time for other lines, we observe that this value is significantly higher than the cost estimation presented in Table 9.1 in the previous section. For example, the cost for each step in the for-loop statement is 160 nanoseconds (approximately 500 CPU cycles). However, we already know that the cost of iteration is around 50-100 CPU cycles. This indicates that the line profiler introduces an overhead of more than 400 CPU cycles.

The line profiler utilizes the Python `PyEval_SetTrace()` hook to trace events within a function. It adds overhead to every opcode (operation code) event of the function it tracks. Opcodes are the disassembled operations that Python interpreter executes, which we will discuss in Section 9.3. A single Python statement can involve multiple opcode events. This overhead can significantly impact the profiling accuracy. Consequently, the `% Time` value for simple Python statements may be overestimated, whereas this value for lines with a high `Per Hit` might be underestimated. This error should be taken into account when using the line profiler.

9.2.4 Sampling profiler

Both the line profiler and `cProfile` utilize Python’s trace hook mechanism to monitor events within a Python program. These profiling methods have substantial overhead, which can result in inaccurate profiling results. In contrast, sample-based profilers

gather runtime information of a Python program periodically, rather than monitoring each function call or instruction. This method does not interfere with the execution of the Python interpreter, thereby not affecting the performance of the Python program. It provides accurate runtime performance statistics.

Sampling profilers can be used to evaluate the performance of C/C++ extensions in Python.

9.2.4.1 *py-spy*

The sampling profiler *py-spy* [10] supports both the Python code and the C/C++ extensions. For example, to profile the `schmidt_orth` program, we use the following command.

```
$ py-spy record -o profile.svg python schmidt_v2.py
```

The option `-o profile.svg` requires to save the profiling results in a flame graph (Fig. 9.1) with the filename `profile.svg`.

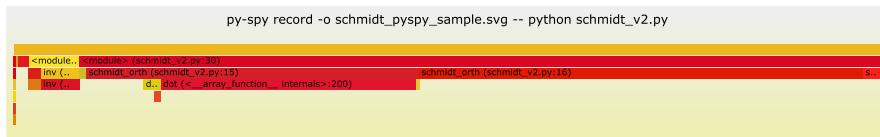


FIGURE 9.1

Flame graph of the *py-spy* profiling results.

The profiling plots are convenient for identifying the functions that consume significant amounts of time. For instance, in the case of the `schmidt_orth` function, two specific lines within the `schmidt_v2.py` file are responsible for 38% and 51% of the total computation time, respectively. Compared to the 92% cost for the `schmidt_orth` function, the two lines together account for 95% of the function's execution time. This proportion of time is higher than what is reported by the `line-profiler`. The difference can largely be attributed to the overhead of Python's trace hook in the `line-profiler` tool.

To profile C/C++ code in Python, the *py-spy* profiler must be executed with the `--native` option. Similar to the output for Python, the profiling plots can highlight specific lines in the C or C++ source code.

A powerful feature of *py-spy* is its capability to analyze a running Python program. We can use the following command to attach the *py-spy* profiler to a specific process:

```
$ py-spy record -o profile.svg --pid <PID>
```

After gathering sufficient data, you can press `Ctrl-C` to interrupt the profiling process without terminating the running process.

9.2.5 Perf

The `perf` profiler is a powerful tool for profiling C/C++ code [11]. In most Linux distributions, it can be easily installed through the package management systems. We can use `perf` to profile the C/C++ extensions in Python. Generally, `perf` is used to identify slow functions and code blocks, as well as to monitor CPU utilization statistics.

The feature of identifying slow code is similar to the functionality provided by `py-spy record --native`. Taking the `schmidt_orth` function as an example, the following command profiles the stack and saves the samples to the `perf.data` file.

```
$ perf record python schmidt_v2.py
```

To view the profiling results, we can use the `perf report` or `perf annotate` commands. The `perf report` command displays the sample count for each function in a text-based format. It provides an overview of the most frequently executed functions. On the other hand, the `perf annotate` command can attribute the sample counts to specific lines of the source code.

The following is a sample of `perf report`:

```
# Samples: 42K of event 'cycles'
# Event count (approx.): 37014407361
#
# Overhead  Command  Shared Object
#           Symbol
# .....  .....
#
#          15.91%  python    libopenblas64_p-r0-15028c96.3.21.so      []
#                      dot_compute
#          7.59%  python    _multiarray_umath.cpython-310-x86_64-linux-gnu.so []
#                      DOUBLE_subtract_AVX512F
#          6.59%  python    _multiarray_umath.cpython-310-x86_64-linux-gnu.so []
#                      DOUBLE_multiply_AVX512F
#          6.19%  python    python3.10                                []
#                      _PyEval_EvalFrameDefault
#          3.37%  python    _multiarray_umath.cpython-310-x86_64-linux-gnu.so []
#                      ufunc_generic_fastcall
#          2.08%  python    _multiarray_umath.cpython-310-x86_64-linux-gnu.so []
#                      PyArray_NewFromDescr_int
#          1.87%  python    _multiarray_umath.cpython-310-x86_64-linux-gnu.so []
#                      PyArray_DiscoverDTypeAndShape_Recursive
[...]
```

The output shows that the most time-consuming function is `dot_compute`. This function is provided by OpenBLAS and has been thoroughly optimized. It is nearly impossible to improve the performance through direct optimization of `dot_compute`. The

`dot_compute` function is called by the `np.dot` function, which accounts for nearly 40% of the total computational cost, as indicated by `line-profiler` and `py-spy`. This implies that a considerable amount of resources may be consumed by various overhead factors. Reducing the overhead can be the focus of future optimization. It can be achieved via techniques such as Just-In-Time (JIT) compilation, Cython compilation, or writing C/C++ extensions, etc. We will discuss these methods in detail in subsequent sections.

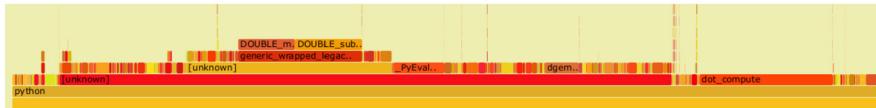


FIGURE 9.2

Flame graph of the `perf` profiling results.

The `perf` profiling results can also be visualized using flame graphs (see Fig. 9.2). This requires to record samples with the `-g` option. By default, `perf record` only records the samples for the functions on the call stack. When the `-g` option is configured, `perf record` will also collect the samples of all the child functions in the call chain. Then we can utilize the open-source tool FlameGraph [12] to generate a flame graph from the `perf.data` file

```
$ perf record -g python schmidt_v2.py
$ git clone https://github.com/brendangregg/FlameGraph
$ perf script | FlameGraph/stackcollapse-perf.pl > out.perf
$ cat out.perf | FlameGraph/flamegraph.pl > perf.svg
```

Now let's examine the CPU utilization statistics provided by `perf`. These statistics show how efficiently the program utilizes the available CPU resources. By running the `perf stat` command

```
$ perf stat python schmidt_v2.py'
```

we can obtain the following statistics:

Performance counter stats for 'python schmidt_v2.py':		
11,295.77 msec	task-clock	# 0.998 CPUs utilized
52	context-switches	# 0.005 K/sec
3	cpu-migrations	# 0.000 K/sec
22,369	page-faults	# 0.002 M/sec
34,860,872,792	cycles	# 3.086 GHz
56,693,690,277	instructions	# 1.63 insn per cycle
11,688,583,612	branches	# 1034.776 M/sec
26,498,322	branch-misses	# 0.23% of all
branches		

```
11.314251488 seconds time elapsed
```

```
11.260627000 seconds user
```

```
0.036014000 seconds sys
```

By combining the statistics of each event in the statistics with the cost estimation we introduced previously, we can roughly estimate the overall overhead of various operations. For example, given the overhead of 1000 to 10,000 cycles for a single page fault, the 22,369 page-faults result in an overall penalty of 22 million to 220 million cycles. This contributes approximately 0.7% to the total number of cycles, as reported in the line `cycles`.

The `perf stat` command provides the statistics for many types of events, which can be listed using the `perf list` command. Apart from the above statistics for default configurations, we can use the option `-d` and `-e` to include additional metrics in the profiling output. For example, by running the command

```
$ perf stat -d -d -e instructions,L1-dcache-stores,L1-dcache-store-misses,  
LLC-stores,LLC-store-misses python schmidt_v2.py
```

we can obtain the output:

```
Performance counter stats for 'python schmidt_v2.py':  
  
      56,793,613,373      instructions  
      9,362,101,055      L1-dcache-stores  
<not supported>    L1-dcache-store-misses  
          600,183        LLC-stores  
         201,100        LLC-store-misses  
     16,468,021,779      L1-dcache-loads  
     4,810,295,651      L1-dcache-load-misses  # 29.21% of all L1-  
dcache accesses  
     148,302,646       LLC-loads  
      60,267,214       LLC-load-misses      # 40.64% of all LL-  
cache accesses  
<not supported>    L1-icache-loads  
     2,777,017,686      L1-icache-load-misses  
    16,576,195,387      dTLB-loads  
      7,306,971       dTLB-load-misses      # 0.04% of all dTLB  
cache accesses  
     144,804,462       iTLB-loads  
      137,492        iTLB-load-misses      # 0.09% of all iTLB  
cache accesses  
  
11.486423819 seconds time elapsed
```

```
11.433589000 seconds user
0.052007000 seconds sys
```

In these events, we are particularly interested in four groups of metrics.

- `instructions` count (the first line in the output). A lower number of instructions usually indicates a lower cost. In addition, in the same line as the `instructions` metric, we can find the IPC metric, denoted as `insn per cycle`. This metric reflects the CPU's efficiency in executing instructions. A low IPC could be a result of excessive branching that interrupts the pipeline, a high cache miss rate, or CPU idling while waiting for data.
- `branches` and `branch misses`. Branch misses occur when the CPU incorrectly predicts the outcome of a branch instruction. The mis-prediction can stall the CPU pipeline, leading to a significant performance penalty. If the number of branch misses or the branch miss rate is high, it is likely that the program is running inefficiently.
- The effectiveness of L1 cache, including events
 - `L1-dcache-loads`,
 - `L1-dcache-load-misses`,
 - `L1-dcache-stores`,
 - `L1-dcache-store-misses`,
 - `L1-icache-load-misses`.

The metrics `L1-dcache-loads` and `L1-dcache-stores` reflect the overall cost of accessing data. L1 cache misses encompass both data cache misses and instruction cache misses. When an L1 cache miss occurs, data must be retrieved from the L2 cache, which has a latency exceeding 10 cycles. A high rate of cache misses is a signal of poor CPU utilization.

- The effectiveness of L3 cache, including events
 - `LLC-loads`,
 - `LLC-load-misses`,
 - `LLC-stores`,
 - `LLC-store-misses`.

The term `LLC` stands for the last level cache. Each `LLC-loads` and `LLC-stores` indicates L2 cache misses and a latency of 50 cycles for accessing data from L3 cache. On the other hand, `LLC-load-misses` and `LLC-store-misses` indicate that data needs to be fetched from the main memory, leading to a penalty of more than 200 cycles. These penalties cannot be hidden by the CPU pipeline or out-of-order execution. By comparing these metrics with latency estimation, we can determine the amount of CPU time wasted on waiting for data.

We previously mentioned that a low IPC value typically indicates poor efficiency. However, does this imply that a high IPC value guarantees good efficiency?

In a numerical computation program, the most commonly executed instructions include arithmetic calculations, data access operations, and branch conditions. These

types of instructions constitute the majority of the total instruction count. To estimate the number of instructions dedicated to data access, we can approximate this by summing the `L1-dcache-loads` and `L1-dcache-stores` metrics. By subtracting the counts of data accessing and branching from the total instruction count, we can approximate the number of instructions used for arithmetic calculations. In the current example, the instructions consumed by branching and data access amount to approximately 37.6 billion. This represents 66% of the total 56.8 billion instructions. As a result, we can deduce that the efficiency of the arithmetic operations is roughly $1.64 \times \frac{1}{3} \approx 0.5$ instructions per cycle.

In quantum chemistry programs, arithmetic calculations are the main source of computational workload. When optimizing the program, the focus should be on enhancing the *arithmetic operations per cycle* rather than the IPC, and on reducing the number of branching and data access instructions.

What can we do to reduce the overhead of non-arithmetic instructions, more specifically, the branching and data access operations?

To reduce the overhead of branching, we can employ the techniques such as function inline and loop unrolling. Although these methods will not directly increase the performance of Python script code, they are particularly useful when applying Python compilation techniques such as JIT (Just-In-Time compilation) or Cython. We will explore these techniques in more details in Section 9.4.

To reduce the instructions required for data accessing, aligning memory and then utilizing SIMD store/load instructions is an effective method, as it allows for accessing multiple data within a single instruction. However, this strategy cannot be directly implemented in Python code, even when using JIT or Cython compilation. It is only applicable in extensions written in native C or C++ code.

9.3 Python level optimization

9.3.1 The `dis` module

Before execution, the Python interpreter compiles Python code into bytecode, which is an internal representation of the program. The interpreter then reads this bytecode and executes the instructions within a stack-based virtual machine. The `dis` module can be used to display the bytecode of the disassembled Python source code. This allows us to see the step-by-step instructions that the Python interpreter follows when executing a program. For example, the arithmetic expression `x * 2` is compiled into the following bytecode instructions:

```
In [1]: import dis
In [2]: dis.dis('x * 2')
 2           0 LOAD_FAST              0 (x)
              2 LOAD_CONST              1 (2)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
```

The output has six columns.

- The first column shows the line numbers in the original code.
- In the second column, which is empty in this example, we may find `>>` symbols. They label the jump points for loops and if-else branches elsewhere.
- The third column is the address of the instruction.
- The fourth column is the operation code (opcode) name.
- The fifth column is the parameter for the operation.
- The last column contains annotations to aid interpretation of the bytecode.

The implementation of opcode can be found in the source code file `ceval.c` (it is moved to `generated_cases.c.h` as of Python 3.12). We can analyze the source to obtain the rough idea about the computational efforts of each instruction. For example, the implementation of the `LOAD_FAST` instruction is:

```
TARGET(LOAD_FAST) {
    PyObject *value = GETLOCAL(oparg);
    if (value == NULL) {
        format_exc_check_arg(PyExc_UnboundLocalError,
                             UNBOUNDLOCAL_ERROR_MSG,
                             PyTuple_GetItem(co->co_varnames, oparg));
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);
    FAST_DISPATCH();
}
```

The core of `LOAD_FAST` is the `GETLOCAL` macro, which is expanded into a simple array indexing operation (`fastlocals[i]`). This particular operation is very efficient, requires only 1 CPU cycle to execute. The macros `Py_INCREF` and `PUSH` are also simple statements, each taking approximately 1 CPU cycle. The `FAST_DISPATCH` macro executes the basic `read-eval` loop of Python interpreter, introducing an overhead around 10 CPU cycles. Therefore, the overall cost of `LOAD_FAST` instruction is estimated to be 10 - 20 CPU cycles.

We can examine the costs of individual instructions by running a simple test in Jupyter notebook. For example, to estimate the cost of the arithmetic instruction `BINARY_MULTIPLY`, we execute the baseline test:

```
In [1]: %timeit
        for _ in range(1000000):
            pass
21.1 ms ± 293 µs per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

and then the test for the actual operation:

```
In [2]: %%timeit
    x = 1.
    for _ in range(1000000):
        pass
        x * 2
48.5 ms ± 235 µs per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

The difference of the two tests represents the cost of executing `x * 2` one million times. The total time spent is 27.4 milliseconds, corresponding to 85 million CPU cycles on a computer with a CPU clock speed of 3.1 GHz. After excluding the costs of `LOAD_FAST` and `LOAD_CONST` instructions, which consume 30 - 40 CPU cycles, we obtain a rough estimation for the arithmetic instruction `BINARY_MULTIPLY`, which is 40 - 50 CPU cycles.

Using the `dis` module, we can obtain the disassembled bytecode, which can then be used to estimate the computational costs. By employing this approach to analyze the source code in `ceval.c`, we derive the cost estimation for Python instructions in Section 9.1.1. Applying bytecode analysis to an entire Python program is generally impractical. Typically, bytecode information is only used to ensure that specific code statements are effectively optimized.

9.3.2 Performance-friendly Python code

Knowing how the Python virtual machine executes bytecodes and the cost of each Python instruction, it's easier to reason why certain coding styles perform better and are recommended. Below are some common scenarios and the recommended Python code practices.

Using `enumerate` for index and object iteration

Using `enumerate` to simultaneously generate the index and the iteration object is more efficient and more elegant than iterating over a `range` and then indexing elements.

```
# Recommended:
for i, x in enumerate(a_list):
    ...

# Not recommended:
for i in range(len(a_list)):
    x = a_list[i]
    ...
```

Based on the disassembled bytecode of the two methods, it is clear that the `enumerate` function requires only one instruction `UNPACK_SEQUENCE` to obtain both `i` and `x`, while the range-indexing version requires two load operations and one indexing operation.

```
# for i, x in enumerate(a_list):
#     ...
    >>   8 FOR_ITER              4 (to 18)
        10 UNPACK_SEQUENCE      2
        12 STORE_NAME            2 (i)
        14 STORE_NAME            3 (x)
        16 JUMP_ABSOLUTE         4 (to 8)

# for i in range(len(a_list)):
#     x = a_list[i]
#     ...
    >>   12 FOR_ITER             6 (to 26)
        14 STORE_NAME             3 (i)
3       16 LOAD_NAME              2 (a_list)
        18 LOAD_NAME              3 (i)
        20 BINARY_SUBSCR
        22 STORE_NAME              4 (x)
4       24 JUMP_ABSOLUTE          6 (to 12)
```

Similarly, when iterating over multiple lists, the recommended method is to use the `zip` function because it involves fewer operations than the range-then-index method.

```
# Recommended:
for x, y in zip(list1, list2):
    ...

# Not recommended:
for i in range(len(list1)):
    x = list1[i]
    y = list2[i]
    ...
```

Using generator instead of explicitly constructing list or tuple

If applicable, it is preferred to use generator expressions instead of creating lists or tuples and then iterating over them. Generator expressions produce elements on demand, while creating lists or tuples requires additional memory consumption.

```
# Recommended:
sum(func(i) for i in a_list)

# Not recommended:
sum([func(i) for i in a_list])
```

List comprehension for creating new lists

The list comprehension uses the specialized instruction LIST_APPEND to add elements to the target list. It involves only one operation during the iteration. In contrast, the loop-then-append method requires loading the target list for each iteration, accessing the .append method and calling it.

```
# Recommended:
result = [x * 2 for x in a_list]

# Not recommended:
result = []
for x in some_list:
    result.append(x * 2)
```

Dictionary comprehension for creating dictionaries

Similar to list comprehension, dictionary comprehension also benefits from a specialized instruction MAP_ADD and the simplified iteration structure. It is more efficient than other methods for dictionary initialization.

```
# Recommended:
result = {k: func(k) for k in a_list}

# Not recommended:
result = dict((k, func(k)) for k in a_list)

# Not recommended:
result = {}
for k in a_list
    result[k] = func(k)
```

Merging dictionaries

The dictionary unpacking operation (**) utilizes the DICT_UPDATE instruction to sequentially update the dictionaries in the order they are specified. This process does not involve the for-loop iteration over the elements within the dictionaries. It is an efficient method for combining multiple dictionaries into a single one.

```
# Recommended:
result = {**dict1, **dict2}

# Not recommended:
result = {}
for key in dict1:
    result[key] = dict1[key]
for key in dict2:
```

```
result[key] = dict2[key]
```

Swapping two variables

Python interpreter has a specialized `ROT_TWO` instruction for swapping variables, which is suitable for this case.

```
# Recommended:  
x, y = y, x  
  
# Not recommended:  
temp = x  
x = y  
y = temp
```

Concatenating strings

The most efficient method for concatenating two strings is by using `str1 + str2`. However, when it comes to concatenating multiple strings, the `{string}.join` method is more efficient. This is because the `sum` function for concatenation essentially turns the process into a series of `str1 + str2` operations. Each of these operations involves creating and eliminating temporary objects.

```
# Recommended:  
combined = ''.join(['s', 't', 'r', 'i', 'n', 'g', 's'])  
  
# Not recommended:  
combined = sum(['s', 't', 'r', 'i', 'n', 'g', 's'])
```

Using local variables

The overhead of creating and accessing local variables is generally low. Accessing local variables is significantly faster than referencing global variables. A typical situation where global variables are frequently referenced is during function calls. For instance, when a function like `math.cos` is called multiple times within a loop, the global variable `math` and its `cos` attribute are accessed in every iteration. This referencing process can take approximately 200 CPU cycles for each iteration. It is advisable to assign such a function to a local variable. By doing so, the costs for repeated access to global variables or object attributes can be substantially reduced.

```
# Efficient  
def f():  
    cos = math.cos  
    return [cos(i) for i in range(10000)]  
  
# Inefficient
```

```
def f():
    return [math.cos(i) for i in range(10000)]
```

9.3.3 Utilizing tensor operations

Python is slow for element-wise arithmetic operations on arrays and lists with for-loop iterations. One optimization approach is to substitute the element-wise code with NumPy vector operations. Vector operations in NumPy can achieve speeds that are 10 times faster or more than the naive for-loop implementation.

NumPy offers various efficient vectorization functions and tensor operations. By eliminating the for-loops and utilizing tensor operations, one can achieve performance that matches or even exceeds that of Python compilation techniques or C/C++ implementations. However, to reach this level of performance, the program might need a complete redesign to meet the requirements of vectorization and tensor operations.

For instance, the `schmidt_orth` function in Section 9.2 can be rewritten using matrix blocks and matrix multiplications. The overlap matrix in the block representation is:

$$\mathbf{S} = \begin{pmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} \\ \mathbf{S}_{21} & \mathbf{S}_{22} \end{pmatrix}. \quad (9.12)$$

Assuming the \mathbf{S}_{11} block can be orthogonalized with the coefficient matrix \mathbf{C}_{11} , we can then transform the entire overlap matrix:

$$\begin{pmatrix} \mathbf{C}_{11}^T & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} \\ \mathbf{S}_{21} & \mathbf{S}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{C}_{11} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{C}_{11}^T \mathbf{S}_{12} \\ \mathbf{S}_{12} \mathbf{C}_{11} & \mathbf{S}_{22} \end{pmatrix}. \quad (9.13)$$

We then project \mathbf{C}_{11} out of the remaining basis.

$$\begin{pmatrix} \mathbf{1} & \mathbf{0} \\ -(\mathbf{C}_{11}^T \mathbf{S}_{12})^T & 1 \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{C}_{11}^T \mathbf{S}_{12} \\ \mathbf{S}_{12} \mathbf{C}_{11} & \mathbf{S}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{1} & -\mathbf{C}_{11}^T \mathbf{S}_{12} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{S}}_{22} \end{pmatrix}. \quad (9.14)$$

This transformation modifies the overlap matrix block \mathbf{S}_{22}

$$\tilde{\mathbf{S}}_{22} = \mathbf{S}_{22} - \mathbf{S}_{12} \mathbf{C}_{11} \mathbf{C}_{11}^T \mathbf{S}_{12}. \quad (9.15)$$

By applying the `schmidt_orth` function to $\tilde{\mathbf{S}}_{22}$, we obtain the sub-block of the coefficient matrix, \mathbf{C}_{22} . Combining the two steps of orthogonalization transformation, we obtain the orthogonalization coefficients.

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{1} & -\mathbf{C}_{11}^T \mathbf{S}_{12} \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{C}_{11} & -\mathbf{C}_{11} \mathbf{C}_{11}^T \mathbf{S}_{12} \mathbf{C}_{22} \\ \mathbf{0} & \mathbf{C}_{22} \end{pmatrix}. \quad (9.16)$$

These equations can be translated into

```
def schmidt_orth_vectorized(s):
    n = s.shape[0]
    m = n // 2
    c_block11 = schmidt_orth(s[:m,:m])
    s_block12 = np.dot(c_block11.T, s[:m,m:])
    c_block12 = -np.dot(c_block11, s_block12)
    s_block22 = s[m:,m:] - np.dot(s_block12.T, s_block12)
    c_block22 = schmidt_orth(s_block22)
    cs = np.zeros((n, n))
    cs[:m,:m] = c_block11
    cs[m:,m:] = c_block22
    cs[:m,m:] = np.dot(c_block12, c_block22)
    return cs
```

This vectorized version eliminates the for-loop by leveraging the matrix multiplication function `np.dot`. Its performance is nearly twice as fast as the previous version, `schmidt_v2.py`. The `line-profiler` shows that the bottleneck of the function is the two calls to `schmidt_orth` for calculating \mathbf{C}_{11} and \mathbf{C}_{22} . To further enhance performance, we can replace the two calls to `schmidt_orth` with the calls to the function itself, leading to the recursive version as illustrated below:

```
def schmidt_orth_recursive(s):
    n = s.shape[0]
    if n < 8:
        return schmidt_orth(s)

    n = s.shape[0]
    m = n // 2
    c_block11 = schmidt_orth_recursive(s[:m,:m])
    s_block12 = np.dot(c_block11.T, s[:m,m:])
    c_block12 = -np.dot(c_block11, s_block12)
    s_block22 = s[m:,m:] - np.dot(s_block12.T, s_block12)
    c_block22 = schmidt_orth_recursive(s_block22)
    cs = np.zeros((n, n))
    cs[:m,:m] = c_block11
    cs[m:,m:] = c_block22
    cs[:m,m:] = np.dot(c_block12, c_block22)
    return cs
```

The recursive function is nearly 60 times faster than the previous version (`schmidt_v2.py`). According to the `perf stat` results, the most significant improvement comes from the reduction in L1D cache accesses and branch instructions. This enhancement is primarily attributed to the efficient matrix multiplication function in the NumPy library, which effectively utilizes the cache and SIMD instructions.

9.3.4 Optimizing tensor indexing efficiency

Indexing a single element of a NumPy array is inefficient, especially when working with high-dimensional arrays. The process of indexing involves translating high-dimensional indices into a one-dimensional address. This conversion requires multiple steps, including boundary checks for each index, iterating over indices and strides, and calculating the sum of the products of indices and strides.

To improve the efficiency of indexing, one strategy is to simplify the process of address computation by decreasing the dimensionality of the array. For instance, in the matrix multiplication program we discussed in Section 9.1.5.4, matrix elements are accessed using two subscripts. By introducing an intermediate vector, matrix indexing is converted into one-dimensional array indexing, which can enhance the program performance.

```
def matmul_nn_v3(a, b):
    arow, acol = a.shape
    brow, bcol = b.shape
    assert acol == brow
    c = np.zeros((arow, bcol))
    for i in range(arow):
        c_i = c[i]
        for k in range(acol):
            a_ik = a[i,k]
            b_k = b[k]
            for j in range(bcol):
                c[j] += a_ik * b_k
    return c
```

Another method to enhance indexing efficiency is through the use of indirect indexing. Although indirect indexing is generally unfavorable due to poor cache utilization and ineffective data prefetching, there are circumstances where it can outperform direct indexing methods for high-dimensional arrays. For example, in the integral computation program which we will discuss in Chapter 12, we will encounter a recursion program to evaluate elements in a tensor:

```
lkl = lk + ll
for lx in range(1, ll+1):
    for ly in range(ll+1-lx):
        for lz in range(ll+1-ly):
            lsum = lx + ly + lz
            for kx in range(lkl+1-lsum):
                for ky in range(lkl+1-lsum-kx):
                    for kz in range(lkl+1-lsum-kx-ky):
                        a[kx,ky,kz,lx,ly,lz] = a[kx+1,ky,kz,lx-1,ly,lz] \
                            + Xcd * a[kx ,ky,kz,lx-1,ly,lz]
```

The computational cost of this recursion program is primarily determined by the cost of tensor indexing. In this scenario, indirect indexing can be made more efficient than direct indexing. Specifically, we pre-compute the addresses for each tensor element involved in the nested iterations. We then transform the tensor into a one-dimensional array and iterate over the one-dimensional array using the cached indirect indices.

```
def cache_indexing(lk, ll, a):
    lk1 = lk + ll
    for lx in range(1, ll+1):
        for ly in range(ll+1-lx):
            for lz in range(ll+1-lx-ly):
                lsum = lx + ly + lz
                for kx in range(lk1+1-lsum):
                    for ky in range(lk1+1-lsum-kx):
                        for kz in range(lk1+1-lsum-kx-ky):
                            idx0.append((kx , ky, kz, lx , ly, lz))
                            idx1.append((kx+1, ky, kz, lx-1, ly, lz))
                            idx2.append((kx , ky, kz, lx-1, ly, lz))

    idx0 = np.ravel_multi_index(np.array(idx0).T, a.shape)
    idx1 = np.ravel_multi_index(np.array(idx1).T, a.shape)
    idx2 = np.ravel_multi_index(np.array(idx2).T, a.shape)
    return idx0, idx1, idx2

idx0, idx1, idx2 = cache_indexing(lk, ll, a)
assert sub.flags.c_contiguous
a_view = a.ravel()
a_view[idx0] = a_view[idx1] + Xcd * a_view[idx2]
```

9.4 Compiling Python code

Pure Python code is slow, primarily due to the overhead of dynamic typing and line-by-line interpretation at runtime. Compilation techniques can resolve these issues and accelerate the performance of Python code. Common compilation methods include JIT (Just-in-Time) compilation and AOT (Ahead-of-Time) compilation. JIT compilation is a process that translates Python bytecode into machine code (binary code) during runtime, just before executing a specific piece of code. Numba [13], JAX [14], and PyPy are representative examples that employ JIT compilation. For AOT compilation, popular tools include Cython [15], Pythran [16], Nuitka [17], ShedSkin [18], and Codon [19]. However, for scientific applications, PyPy, Nuitka, Shedskin, and Codon are not suitable choices due to their limited support for NumPy functionality. In this section, we will explore the optimization techniques utilized in Numba, JAX, Cython, and Pythran.

9.4.1 Numba

Numba is a JIT compilation tool specialized for numerical computation tasks. It is particularly effective at accelerating arithmetic operations and loops. However, it is not well-suited for code segments that extensively utilize object-oriented programming or the dynamic features of Python. Numba supports JIT compilation for both CPU and GPU architectures. We will focus on the CPU aspect in this section and defer the discussion on GPU to Chapter 11.

The basic usage of Numba is straightforward. It simply requires adding the `@numba.jit` decorator to the function we want to optimize. Numba will compile the target function during the first run, incurring some warm-up overhead.

As described in the Numba documentation [20], Numba utilizes the `llvmlite` library [21] to convert Python code into LLVM IR (Intermediate Representation). It then uses LLVM (Low-Level Virtual Machine) compiler to execute the generated IR. In this procedure, it does not generate any C or C++ code files as the intermediate layer. Consequently, traditional code inspection tools, such as GDB (GNU Debugger) or perf annotations, are not applicable. The following Numba commands can be utilized to dump LLVM IR:

```
$ numba --annotate --dump-llvm coulomb_2e_OS.py
$ numba --dump-optimized coulomb_2e_OS.py
$ numba coulomb_2e_OS.py --annotate-html coulomb_2e_OS.html
```

In principle, it is possible to analyze the LLVM IR to determine whether the JIT compilation yields reasonable machine code. However, we will skip this analysis because it requires a deep understanding of LLVM, which is beyond the scope of this book.

There are several keyword options for the Numba JIT decorator that can impact runtime performance [22]:

- *nopython mode*. When the `nopython` mode decorator `@numba.njit` is utilized, Numba completely discards the dynamic nature of Python objects and generates code with the highest performance.
- *Function inline*. `Inline` can reduce the overhead of function calls. Numba JIT features the keyword `inline='always'` to facilitate function inline. The `inline` keyword can be set to other values to enable conditional inline [23].
- *Parallel Execution*. By using the decorator `@numba.njit(parallel=True)` along with the `numba.prange` iterator, Numba can execute the iterator in parallel using multiple threads. As shown by the following example, each thread accumulates its own partial sum, which is then combined across threads by a cross-thread reduction.

```
@numba.njit(parallel=True)
def do_sum_parallel(A):
    n = len(A)
    acc = 0.
```

```
for i in numba.prange(n):
    acc += np.sqrt(A[i])
return acc
```

- *Function signatures.* One challenge in JIT optimization is the warm-up overhead, particularly when compiling complex functions. Employing the option `@numba.njit(cache=True)` to cache the LLVM IR can help reduce the warm-up overhead. Furthermore, specifying an explicit function signature can simplify the type inference process, thereby improving warm-up speed. However, please note that neither function signatures nor LLVM IR caching can boost the runtime performance.
- *Fast math operations.* Configuring `fastmath=True` enables less precise but faster mathematical operations. It is important to enable this option to make LLVM compiler utilize FMA (Fused Multiply-Add) instructions.

In some scenarios, one needs to apply Numba JIT compilation to optimize the code that involves NumPy array objects and functions. Below are several strategies for optimizing code associated with NumPy.

- The NumPy array class and many NumPy functions are reimplemented in the Numba package to better suit JIT compilation. Numba can identify these NumPy functions and select the most appropriate implementation from its internal code library [24]. Manually implementing NumPy functions and then using JIT to compile the code is generally unnecessary and unlikely to lead to performance improvements.
- Numba can automatically expand the NumPy ufuncs and broadcasting code into iteration code. When dealing with arrays that have a unit stride, i.e. continuous memory access, Numba not only expands the ufuncs but also turns on vectorization option for the LLVM compiler. In such cases, one can achieve significant performance improvements through Numba’s JIT compilation. However, in the case of non-contiguous memory access, the performance difference between the JIT compilation and NumPy ufuncs is not substantial.
- When processing the loop expansion of NumPy broadcasting code, LLVM is unable to accurately detect the potential data dependencies or memory overlaps. Consequently, it opts for a conservative approach and does not SIMD-vectorize the broadcasting operation. If we manually expand the broadcasting code within the program, the LLVM compiler would see more context about the variables within the iteration. This extra context enhances the likelihood for the LLVM compiler to successfully utilize SIMD vectorization. Manual loop expansion for broadcasting code is generally recommended.
- Array strides, loop order, cache effectiveness are important factors to consider, even when using Numba JIT compilation, as we discussed in Section 9.1.5.
- Loop unrolling is useful in optimizing the performance of Numba JIT code.

9.4.1.1 Unrolling

The unrolling technique does not improve the performance of pure Python code. It is an effective method for optimizing the Numba JIT-compiled code. The unrolling technique can reduce the number of iterations and minimize the overhead associated with loop control logic. Additionally, it reduces the likelihood of stalls in the CPU pipeline.

There is no keyword for loop unrolling in Numba. Instead, one must explicitly write out the unrolled statements for specific loop indices. One essential consideration in loop unrolling is to determine which index to unroll. If a short loop is unrolled, the iteration may frequently fall into the edge cases where the unrolling does not apply. On the other hand, unrolling the innermost loop might confuse the compiler. The compiler might mistakenly detect memory overlap and thereby refuse to apply SIMD vectorization. Generally, it is preferable to *unroll a long loop in an outer loop*. For example, in the `matmul_nn_tiling` function below, we choose to unroll a loop in the middle with a length of 200. The unrolled code runs approximately twice as fast as the previous implementation in Section 9.1.5.4.

```
@numba.njit(cache=True)
def matmul_nn_tiling_unrolled(a, b):
    '''Matrix multiplication a * b with loop tiling'''
    arow, acol = a.shape
    brow, bcol = b.shape
    assert acol == brow
    c = np.zeros((arow, bcol))
    block_size = 200

    for i0 in range(0, arow, block_size):
        i1 = min(i0 + block_size, arow)
        for j0 in range(0, bcol, block_size):
            j1 = min(j0 + block_size, bcol)
            for k0 in range(0, acol, block_size):
                k1 = min(k0 + block_size, acol)
                for i in range(i0, i1-3, 4):
                    for k in range(k0, k1):
                        a_ik0 = a[i+0,k]
                        a_ik1 = a[i+1,k]
                        a_ik2 = a[i+2,k]
                        a_ik3 = a[i+3,k]
                        for j in range(j0, j1):
                            b_kj = b[k,j]
                            c[i+0,j] += a_ik0 * b_kj
                            c[i+1,j] += a_ik1 * b_kj
                            c[i+2,j] += a_ik2 * b_kj
                            c[i+3,j] += a_ik3 * b_kj
```

```
    for i in range(i+4, i1):
        for k in range(k0, k1):
            a_ik = a[i,k]
            for j in range(j0, j1):
                c[i,j] += a_ik * b[k,j]
    return c
```

In certain scenarios, simply unrolling a single loop might not produce satisfactory performance. Especially when the loop size is short, the overhead of handling edge cases can dominate the computation. In such cases, we can aggressively unroll the entire function, derive specific functions for every possible problem size. This exhaustive unrolling method can be programmatically implemented using the metaprogramming technique discussed in Chapter 5.

Taking the `schmidt_orth` function as an example, we can implement the following function generator `unroll_schmidt_orth` to automatically generate the unrolled code based on the original Python implementation of `schmidt_orth`. Compared to the original version, computation statements, except for the iteration index, are all converted into string representations. Given a problem size denoted by `n`, by running the string-hybrid code, we can obtain an unrolled Python function represented as a string. We then use the `exec` function to compile and register the generated Python code.

```
import jinja2
import numpy as np
from numba.pycc import CC

unroll_tpl = jinja2.Template('''
def schmidt_orth_n{{ n }}(s):
{%- for line in code %}
{{ line }}
{%- endfor %}
''')

def unroll_schmidt_orth(n):
    code = []
    code.append(f'cs = np.zeros(({n}, {n}))')
    for j in range(n):
        code.append(f'fac = s[{j},{(j)}]')
        for k in range(j):
            code.append('dot_kj = 0.')
            for i in range(n):
                code.append(f'dot_kj += cs[{k},{(i)}] * s[{j},{(i)}]')
        for i in range(n):
            code.append(f'cs[{j},{(i)}] -= dot_kj * cs[{k},{(i)}]')
    code.append('fac -= dot_kj * dot_kj')
```

```

        code.append('fac = fac**.5')
        code.append(f'cs[{j}][{j}] = fac')
        for i in range(j):
            code.append(f'cs[{j}][{i}] *= fac')
        code.append('return cs.T')

    exec(unroll_tpl.render(n=n, code=code))
    return vars()[f'schmidt_orth_n{n}']

cc = CC('schmidt_unrolled')
for i in range(8):
    fn = unroll_schmidt_orth(i)
    print(fn.__name__)
    cc.export(fn.__name__, 'f8[:, :](f8[:, :])')(fn)
cc.compile()

```

The function unrolling can result in a significant amount of code, which may lead to a longer warm-up time in JIT compilation. To reduce the warm-up cost, we use the AOT compilation here [25]. The AOT compilation outputs an importable Python module, named

`schmidt_unrolled.cpython-310-x86_64-linux-gnu.so`

for our Python 3.10 environment. We then develop a function named

`schmidt_orth_small_size`

tailored for small-sized `schmidt_orth`. When working with small overlap matrices, the unrolled function achieves nearly 150-fold speedups compared to the original Python function `schmidt_orth`.

```

import numpy as np
import schmidt_unrolled

schmidt_orth_small_size = [
    getattr(schmidt_unrolled, f'schmidt_orth_n{n}') for n in range(8)

def schmidt_orth_recursive_unrolled(s):
    n = s.shape[0]
    if n < 8:
        return schmidt_orth_small_size[n](s)

    n = s.shape[0]
    m = n // 2
    c_block11 = schmidt_orth_recursive(s[:m, :m])
    s_block12 = np.dot(c_block11.T, s[:m, m:])

```

```
c_block12 = -np.dot(c_block11, s_block12)
s_block22 = s[m:,m:] - np.dot(s_block12.T, s_block12)
c_block22 = schmidt_orth_recursive(s_block22)
cs = np.zeros((n, n))
cs[:m,:m] = c_block11
cs[m:,m:] = c_block22
cs[:m,m:] = np.dot(c_block12, c_block22)
return cs
```

Although unrolling can often improve performance, it is important to consider the trade-off between the benefits of unrolling (as well as function inline) and the potential impacts on the instruction cache performance. The use of inline and unrolling techniques significantly increases the number of instructions that the CPU has to load and decode. This can increase pressure on the cache, as the decoded instructions are stored in the L1I cache. The penalty of an L1I cache miss is comparable to that of an L1D cache miss, which implies at least a latency associated with accessing the L2 cache. Consequently, the total penalty of L1I cache misses might exceed that of L1D cache misses.

9.4.1.2 Numba vectorization issues

Numba does not directly support assembly code or SIMD intrinsics for vectorization. It relies on the underlying LLVM to analyze the memory layout and vectorization pattern to determine if the program is suitable for SIMD vectorization. Occasionally, LLVM may incorrectly identify data overlapping. This causes the Numba JIT to overlook cases that are actually SIMD vectorizable, such as the tiled program `matmul_nn_tiling` shown in Section 9.1.5.4.

To check whether SIMD vectorization is utilized by the Numba JIT, we can add the option `--debug-only=loop-vectorize` to LLVM compiler [26]

```
import llvmlite.binding as llvm
llvm.set_option('', '--debug-only=loop-vectorize')
```

This option will provide details on how LLVM examines vectorization. For example, the output below indicates that the compiler identifies a candidate for vectorization but refuses to vectorize the code.

```
LV: Can't vectorize due to memory conflicts
```

Another method to check for SIMD vectorization is to inspect the assembly code produced by Numba's JIT compilation. We can utilize the `find_instr` function below to search for specific assembly instructions or SIMD registers. For instance, to identify AVX-level vectorization, we can search for the `ymm` registers.

```
import numpy as np
from matmul import matmul_nn_tiling
```

```
def find_instr(func, keyword, sig=0, limit=5):
    count = 0
    for l in func.inspect_asm(func.signatures[sig]).split('\n'):
        if keyword in l:
            count += 1
            print(l)
            if count >= limit:
                break
    if count == 0:
        print('No instructions found')

matmul_nn_tiling(np.eye(3), np.eye(3))
find_instr(matmul_nn_tiling, 'ymm')
```

Vectorization often fails due to the uncertainty in array bounds when the bounds are dynamically generated during the loop. It is not clear why LLVM fails to identify the array bounds. To circumvent the limitations of the LLVM boundary checker, one approach is to simplify array indexing, such as using the temporary one-dimensional array `cp` in the following example.

Please note the configuration `fastmath=True` in this example. How does the `fastmath` mode come to impact the SIMD vectorization? Many CPUs support the FMA (fused-multiply-add) instruction. FMA (as well as other arithmetic SIMD instructions) is considered to affect the precision of numerical computations. LLVM only generate FMA instructions when the `fastmath` mode is enabled. This is a subtle difference between the LLVM compiler and the GCC compiler in handling FMA instructions.

9.4.2 Cython

Cython is an excellent AOT tool that translates Python code into intermediate C code, which is then compiled using GCC (or Clang). Such tools are often referred to as transpilers, as they only perform the source-to-source translation rather than direct compilation to machine code. The main advantage of having C code as an intermediate layer is that it allows us to inspect the intermediate C code with GDB and standard C program profilers.

Compiling Python code with Cython is straightforward, as we have discussed in Section 8.2.2 of Chapter 8. However, Cython-compiled pure Python code only provides limited performance improvement.

The strategy for optimizing Cython code aims to eliminate any footprints of Python objects. Following this idea, we can employ various techniques to make Cython generate more efficient code.

- *Type declaration for every variable*, including all function arguments and return values. Cython cannot accurately infer data types. Without precise data type information, Cython treats these variables as Python objects. Cython’s type declarations adopt a C-style syntax, indicated by the `cdef` prefix, for example, `cdef int`, `cdef double`, or `cdef class`.
- Using *typed memoryview* to specify the structure of a NumPy array, including data type, dimension, and strides. For example, in the type declaration

```
cdef double[:, ::1] a_view = a
```

the notation `double[:, ::1]` indicates that the array is of `double` type and is two-dimensional. The `::1` in the last index means a unit stride for that last dimension, indicating that the memory view is C-contiguous. Cython memoryview supports various stride modes [27]. The two commonly used layouts in programming are C-contiguous, `[:, ::1]`, and Fortran-contiguous, `[:, 1:, :]`. Cython is able to optimize the indexing efficiency for the two particular stride modes.

- *Explicit loops for NumPy ufuncs and broadcasting operations*. Unlike Numba, Cython cannot automatically optimize NumPy ufuncs or broadcasting operations. Cython treats them as regular Python statements. To achieve better performance, we must manually expand the loops for NumPy ufuncs and broadcasting operations.
- *Avoiding any Python syntax sugar*. Cython transpiler is unable to handle most Python syntax sugar. It typically treats them as regular Python statements and

generate the slow and Python-involved code. For example, the Python syntax for iterating over an iterable is not supported by Cython.

```
for x in array:  
    ...
```

Even though the types of `x` and `array` are both explicitly declared, Cython cannot fully eliminate the Python footprints and generate efficient C code for this iteration. The generated code first extracts an element from the iterable (`array`) as an intermediate Python object, and then utilizes the Python-C API to bind the intermediate variable to `x`. Similarly, other loop iterators such as `enumerate` and `zip` are unsupported syntax sugar in Cythonization. The most efficient method for iteration is the *range-then-index* approach, which is completely opposite to the practice of optimizing pure Python code.

```
for i in range(len(array)):  
    x = array[i]  
    ...
```

- Using *Cython compiler directives* to disable Python-specific checks [28]. The compiler directives can be annotated within the source code or specified via the command line. Below, we outline several settings within the source code that are beneficial for numerical calculations.

```
#cython: boundscheck=False  
#cython: wraparound=False  
#cython: overflowcheck.fold=False  
#cython: cdivision=True
```

- *Parallelism with OpenMP threads* [29]. To activate OpenMP parallelization in Cython, we need to replace the `range` statement with the `cython.parallel.prange` function, and include the following compiler directives in the `.pyx` file

```
#distutils: extra_compile_args=-fopenmp  
#distutils: extra_link_args=-fopenmp
```

- Other optimization techniques, as discussed in Numba optimization (Section 9.4.1), which include but are not limited to optimizing array strides, adjusting loop order, improving cache utilization, and implementing loop unrolling.

When using OpenMP parallelization in Cython, the reduction convention can be somewhat unusual and prone to errors. Cython is not as intelligent as Numba when inferring reduction operations. Numba can automatically determine whether a reduction is needed based on the scope of a variable and avoid applying reduction to private variables inside the `prange` context. In contrast, Cython infers reduction merely based on the presence of inplace operators (`+=`, `-=`, `*=`, etc.). For example, in the code snippet below, even though `x` is just a temporary variable within the `prange` context, Cython mistakenly perform reduction for both `x` and `v` at the end of the `prange`.

```
cdef double v = 0
cdef double x
for i in cython.parallel.prange(n, nogil=True):
    x = 0.
    x += a[i] * a[i]
    v += x
```

To help Cython accurately identify reduction operations, we should avoid using inplace operators on local variables inside the `prange` context. The above code snippets should be corrected accordingly.

```
cdef double v = 0
cdef double x
for i in cython.parallel.prange(n, nogil=True):
    x = 0.
    x = x + a[i] * a[i]
    v += x
```

By applying all the optimization tricks mentioned above, we can implement the Cython code for the `schmidt_orth` problem.

```
#cython: boundscheck=False
#cython: wraparound=False
#cython: overflowcheck.fold=False
#cython: cdivision=True
#cython: language_level=3
#distutils: extra_compile_args=-fopenmp
#distutils: extra_link_args=-fopenmp

import numpy as np
from cython.parallel import prange

def schmidt_orth(double[:, ::1] s):
    cdef int n = s.shape[0]
    _cs = np.zeros((n, n))
    cdef double[:, ::1] cs = _cs
    cdef int i, j, k
    cdef double dot_kj, fac
    cdef double[:1] dot_kj_buf = np.empty(n)

    for j in range(n):
        fac = s[j,j]
        for k in prange(j, schedule='static', nogil=True):
            dot_kj = 0.
            for i in range(n):
```

```

dot_kj = dot_kj + cs[k,i] * s[j,i]
fac -= dot_kj * dot_kj
dot_kj_buf[k] = dot_kj

for i in prange(n, schedule='static', nogil=True):
    for k in range(j):
        cs[j,i] -= dot_kj_buf[k] * cs[k,i]

if fac <= 0:
    raise RuntimeError(f'schmidt_orth fail. {j=} {fac=}')

fac = fac**-.5
cs[j,j] = fac
for i in range(j):
    cs[j,i] *= fac
return _cs.T

```

Raw output: [schmidt_v2_cython.c](#)

```

+01: #cython: boundscheck=False
+02: #cython: wraparound=False
+03: #cython: overflowcheck.fold=False
+04: #cython: cdivision=True
+05: #cython: language_level=3
+06: #distutils: extra_compile_args=-fopenmp
+07: #distutils: extra_link_args=-fopenmp
+08:
+09: import numpy as np
10: from cython.parallel import prange
11:
+12: def schmidt_orth(double[:, ::1] s):
+13:     cdef int n = s.shape[0]
+14:     _cs = np.zeros((n, n))
+15:     cdef double[:, ::1] cs = _cs
16:     cdef int i, j, k
17:     cdef double dot_kj, fac
+18:     cdef double[:,::1] dot_kj_buf = np.empty(n)
19:
+20:     for j in range(n):
+21:         fac = s[j,j]
+22:         for k in prange(j, schedule='static', nogil=True):
+23:             dot_kj = 0.
+24:             for i in range(n):
+25:                 dot_kj = dot_kj + cs[k,i] * s[j,i]
+26:             fac -= dot_kj * dot_kj
+27:             dot_kj_buf[k] = dot_kj
28:
+29:         for i in prange(n, schedule='static', nogil=True):
+30:             for k in range(j):
+31:                 cs[j,i] -= dot_kj_buf[k] * cs[k,i]
32:
+33:         if fac <= 0:
+34:             raise RuntimeError(f'schmidt_orth fail. {j=j} {fac=fac}')
+35:         fac = fac**-.5
+36:         cs[j,j] = fac
+37:         for i in range(j):
+38:             cs[j,i] *= fac
+39:     return _cs.T

```

FIGURE 9.3

Cython annotations visualized in HTML.

Is our Cython code reasonably implemented in the sense that all unnecessary Python statements have been eliminated? We can use Cython annotations to analyze the code:

```
$ cython -a schmidt_v2_cython.pyx
```

This command generates an annotated result in an HTML file, which can be viewed in a web browser (Fig. 9.3). The Cython annotations use color coding to highlight the presence of Python code within the Cythonized code. Deeper yellow (gray in print version) means more Python footprint, thus less efficient. The goal is to eliminate or weaken the yellow-highlighted sections as much as possible. In the annotations of `schmidt_orth` function, we can observe yellow lines corresponding to the allocation of the NumPy array, error handling, the function signature line, and the return statements. If the problem size is large enough, the impact of these yellow lines on performance is not significant, as the majority of the execution time is spent inside the loops. However, for small problem sizes, the overhead introduced by these yellow lines may not be negligible. Therefore, while Cython is a useful tool for enhancing the performance of slow Python functions, it may not be the ideal choice for small-sized problems.

9.4.3 Pythran

Pythran is an AOT tool that translates Python code into intermediate C++ code. The intermediate C++ code can be compiled into an importable Python extension.

Pythran does not require extensive modifications to the Python code. It only requires the inclusion of function signature annotations within the Python code, known as *Pythran directives*. These directives assist Pythran in inferring the data types and optimizing the function. For example, the Pythran directive for the `schmidt_orth` function is

```
#pythran export schmidt_orth(float64[:, :])
def schmidt_orth(s):
    ...
```

Clearly, the Pythran directives in the comment line do not affect the execution of the original Python code. When using OpenMP parallelization in Pythran, OpenMP directives like

```
#pragma omp for schedule(static)
```

can be injected into Python code as comment lines. This is an advantage of Pythran over Cython, as its input is simply standard Python code. One does not need to memorize specific details as if learning a new language.

Pythran does not require specifying the type details of local variables, nor does it require adherence to any particular Python syntax sugar. It utilizes the C++ *expression template* technique to infer types, generate loops, optimize array indexing,

Table 9.4 Comparison of various Python compilation techniques.

	Cython	Pythran	Numba
Compilation mode	AOT	AOT	JIT, AOT
Intermediate language	C	C++	LLVM IR
Compilation speed	Fast	Medium	Slow
Execution speed	Medium	Fast	Fast
NumPy specialization	Moderate	Yes	Yes
OpenMP parallelism	Yes	Yes	Yes
GPU acceleration	No	No	Yes
Optimization hints	Annotations in C	No	Annotations in LLVM IR
Python profiling	Limited	No	No
Breaking original code	Yes	No	No
Python feature support	All	Limited	All
SIMD vectorization	Yes	Yes	Limited
Ease of use	No	Yes	Yes

and employ SIMD vectorization. Pythran has an internal implementation of NumPy `ndarray` using C++ templates. In most cases, the expression template technique is able to generate efficient iteration code for NumPy ufuncs and broadcasting operations. Unlike Numba and Cython, manually expanding the broadcasting operations does not improve performance. Exceptions are found in only a few cases. One example is the code for handling the fancy-index or a slice of an array. We observe that the manual expansion code is more efficient than the code generated by C++ expression templates.

There are not many optimization tricks one can adopt in Pythran. One approach is to change C++ compiler and compilation options. The C++ compiler and compilation options for Pythran can be configured in `~/.pythranrc`. For example,

```
[compiler]
cflags=-std=c++11 -O3 -march=native -ffast-math
```

More discussions on the compilation options are available in Section 9.5.1

Pythran only supports a subset of the Python syntax and libraries. The supported modules and functions are detailed in its documentation [30]. When compiling a Python file, Pythran needs to compile all dependent modules into a single extension. Importing Python modules within a Pythran extension is not supported. If there are any unsupported functions in the dependent modules, compilation will fail. Consequently, Pythran is not an suitable choice for compiling large-scale projects that involve multiple Python modules.

9.4.4 Comparison of Cython, Pythran, and Numba

Table 9.4 summarizes the technical aspects of Numba, Cython, and Pythran. Below, we provide a detailed comparison for the three technologies.

Ease of use

The usage of Numba and Pythran simply requires a line of decorator or directive. Moreover, the two tools do not alter the original Python code, making it easier to debug and test. In contrast, Cython introduces a new language, albeit with syntax close to Python. There is a learning curve to master the new language, especially when aiming for optimal performance. Additionally, Cython code cannot be directly executed by a Python interpreter. Debugging Cython code is not as straightforward as debugging with the other two tools.

Support for Python features

Numba and Cython support most Python language features, although using these features may result in decreased performance. Python modules can be imported into programs that utilize Numba and Cython compilation. However, Pythran does not support importing other Python modules or functions into the Pythran-compiled code. This limitation restricts the scope of applications where Pythran can be effectively utilized.

Optimization hints

Both Cython and Numba offer the feature of code annotation in HTML. These annotations are useful for analyzing the complexity and efficiency of each Python statement in the compiled code. Annotations in Cython go a step further by highlighting the code that may not be optimally optimized. In the C++ code generated by Pythran, comment lines are included to map the C++ code back to the original Python statements. However, the extensive use of C++ templates in Pythran makes code analysis more challenging.

Profiling

None of the three compilation methods work with the conventional Python profilers, such as cProfile or Line profiler. However, the py-spy profiler can interpret the line directives embedded in cythonized code. In the flame graph generated by py-spy, the costs of C function calls can be aggregated and attributed to the corresponding lines in the Cython .pyx file.

NumPy specialization

Numba and Pythran specialize the treatment of NumPy functionalities. All NumPy statements in the code, such as array objects and function calls, are replaced with their custom implementations. By doing so, the performance of NumPy operations is automatically optimized and the overhead of calling NumPy is minimized. In Cython, NumPy operations are either treated as regular Python objects, or linked to the shared library provided by NumPy. Unless explicitly expanding NumPy ufuncs and broadcasting code, the performance of NumPy operations in Cython is nearly the same as in pure Python. Cython only specializes the array indexing code through memoryviews, which provide efficient access to individual elements comparable to that at the C level.

SIMD vectorization

When SIMD vectorization is possible, Cython and Pythran are winners because they can leverage the capabilities of advanced C and C++ compilers. Programs compiled by these two are more likely to utilize SIMD vectorization. In some cases, the NumPy implementations in Pythran can be more eager to utilize SIMD, which makes it more efficient than the Cython compilation. Because of this feature, Cython offers an option to use Pythran as a backend to optimize NumPy operations [31]. SIMD vectorization is less straightforward in Numba. To ensure correctness, Numba JIT tends to not utilize SIMD vectorization.

Compilation cost

If compilation is required only once or a few times, its cost is generally not a primary concern. However, during the development and optimization phases of a program, the speed of compilation can have certain impacts. Among the three, Numba JIT typically has the longest compilation time, whereas Cython is the fastest. The slowness of Numba JIT can be attributed to its compiler front-end, which is implemented in Python. The code translation in Cython and Pythran is rapid, with negligible cost. Compiling the C code generated by Cython is faster than compiling the template-intensive C++ code produced by Pythran.

Execution speed

In theory, JIT compilation has the potential to deliver code with best performance. JIT compilation has the capability to access and collect extensive runtime information. This enables the compiler to perform cross-function optimization and tailor optimizations to the specific underlying architecture. However, the advantages of JIT compilation are restricted by the complexity of optimization process, resource constraints, the cost of compilation, and various other factors. Consequently, in practical applications, JIT compilation may only utilize a limited scope of information, leading to compromised performance.

9.5 Optimization with compiled languages

The development of Python JIT and AOT compilers allows us to achieve the performance of compiled languages without the need to write code in those languages. Despite the significant capabilities of these tools, there remain tasks that are straightforward in compiled languages but inconvenient or unfeasible in Python compilation. Below are some typical scenarios:

Indexing array elements using pointers

There are multiple ways to use pointers in C/C++ to simplify the indexing of high-dimensional arrays. However, direct manipulation of pointers is not feasible in Python. To improve the efficiency of array indexing in Python, we have to slice a lower-dimensional array from a high-dimensional array, obtaining an array view

of its sub-block. This method, however, introduces overhead in managing the lower-dimensional arrays.

Memory management

In compiled languages, we can employ various memory management techniques to improve memory efficiency, such as using more efficient memory allocators and aligning memory addresses for SIMD operations. These methods are not directly accessible in Python code.

Interactions with compilers

In C and C++, we can use *function attributes* [32], and *compiler directives* to interact with the compiler, such as the function attribute for memory alignment:

```
double roots[8] __attribute__((aligned(32)));
```

and the compiler directive

```
#pragma omp simd
```

for the execution of SIMD instructions. These features are not available in Python compilation tools.

Hardware-specific instructions and operations

In C and C++, we can explicitly invoke SIMD instructions using *compiler intrinsics* [33]. These compiler intrinsics can also be invoked in Cython, where they are treated as regular external C functions. However, a direct control over SIMD instructions in Python code using JIT compilation is inconvenient.

When optimizing a program under these circumstances, using compiled languages should be a natural choice than the Python compilation techniques. To integrate compiled languages into Python code, we can utilize the `ctypes` or `cffi` package, as discussed in Chapter 8.

9.5.1 GCC compiler

In addition to the optimization of the program code itself, compilers and compilation options can also impact program performance. The GCC (GNU Compiler Collection) and LLVM compilers offer a long list of compilation flags, some of which can significantly affect performance optimization. Here, we will use GCC as an example to demonstrate the compiler options that are beneficial for optimizing Python extensions. Although the names of these options might vary slightly in LLVM, the principles are largely comparable.

- -O2. The -O2 optimization level is the default setting in Cython and Pythran. This level strikes a balance among performance, portability, code size, and compilation speed. In fact, if you want to release a program for general platforms, -O2 is the recommended optimization level.

- `-O3`. The `-O3` flag introduces several additional optimizations beyond those applied at the `-O2` level. The details of the additional optimizations can be found in the documentation (`man gcc`). In quantum chemistry applications, the following additional options are particularly important:
 - `-ftree-loop-vectorize`,
 - `-ftree-slp-vectorize`,
 - `-fversion-loops-for-strides`.
- `-ftree-loop-vectorize` and `-ftree-slp-vectorize`. The two options facilitate the detection of vectorization capabilities on the AST (Abstract Syntax Tree). If no pointer dependencies or memory overlaps are detected, the compiler will generate SIMD vectorization code. Please note that these options merely enable the detection of SIMD vectorization. They do not guarantee to produce the SIMD operations. Proper design and implementation are still required to ensure that the code is SIMD vectorizable.
- `-fversion-loops-for-strides`. By enabling this option, the compiler will generate multiple versions of the code for different array strides. This is particularly beneficial for optimizing memory access in the case of unit stride. In AOT compilation, such as with Cython, the access pattern of an array might not be determined in advance. Therefore, it is beneficial to allow the compiler to generate multiple code versions to optimize for different stride patterns. However, this compilation option does not offer significant assistance to JIT compilation. In JIT compilation, specialized treatments for strides can often be applied during the generation of the IR.
- `-march=native`. The `-march=native` flag instructs the GCC compiler to optimize the code specifically for the hardware of the current platform. It enables the compiler to utilize all the supported instructions and features available on the machine. For Cython and Pythran compilation, the simplest method to generate more optimized machine code is to combine the `-O3` flag with `-march=native`. However, the machine code generated with this option is not portable and may not function on different machines. This option should not be used for public release packages.
- `-ffast-math`. This option enhances performance for specific mathematical functions, such as `sqrt`, `log`, `sin`, `cos`, etc. In the `-ffast-math` mode, these functions are approximated by faster but less accurate algorithms. In some compilers, SIMD instructions are generated only when `-ffast-math` is enabled as SIMD parallelization can break the associative property [34]. However, GCC allows SIMD vectorization without checking the `-ffast-math` option.

How can we enable these compilation options in Python compilation? Numba JIT provides several environment variables [35] for adjusting compilation settings. The default configurations in Numba are already quite aggressive, making it generally unnecessary to customize compiler options. The compiler options are primarily used in the AOT compilation processes with Cython and Pythran. Below are some available methods to configure the compiler options:

- Setting the environment variable `CFLAGS` and `CXXFLAGS`, applicable to both Cython and Pythran.
- Adding compilation flags in `setup.py`, applicable to both Cython and Pythran.

```
from setuptools import setup, Extension
from Cython.Build import cythonize
setup(
    ext_modules=cythonize([
        Extension('schmidt', ['schmidt_v2_cython.pyx'],
                  extra_compile_args=['-fopenmp', '-O3', '-march=native'])
    ]),
)
```

```
from setuptools import setup, Extension
from pythran.dist import PythranExtension
setup(
    ext_modules=[
        PythranExtension('schmidt', ['schmidt_v2_pythran.py'],
                        extra_compile_args=['-fopenmp', '-O3', '-march=native'])
    ],
)
```

- Adding Cython compiler directives with the prefix `distutils:`. These directives are automatically applied when building extensions using `setup.py` or the `cythonize` command.

```
#distutils: extra_compile_args=-fopenmp -O3 -march=native
```

- The configuration file `~/.pythranrc` for Pythran. It only affects the `pythran` command when executed from the command line. The `PythranExtension` in `setup.py` does not read settings from this configuration file.

9.6 Optimization for I/O

In most quantum chemistry applications, I/O operations are primarily related to file I/O. Certain applications running on HPC clusters may also engage in network I/O via MPI parallelism. Optimizing I/O efficiency involves enhancing both bandwidth and latency. Several strategies can be employed to optimize I/O, including:

- Improving data layout and storage format. This approach makes data easier to access and transfer, effectively reducing latency.
- Compressing data. This reduces the storage size and enhances bandwidth efficiency for data transfer.
- Overlapping computation with I/O operations to hide latency.
- Utilizing data caching to improve both bandwidth and latency.

9.6.1 Storage layout

HDF5 format

Many quantum chemistry methods require the storage of large-sized arrays as intermediate data on disk. HDF5 offers a data storage layout that is compatible with the NumPy array, supporting the access to small segments within an array.

Arrays in an HDF5 file are stored in the row-major format. This format is advantageous for sequentially accessing large volumes of data on disk. However, when accessing a sub-array of a high-dimensional array, non-contiguous data access may become inevitable, which is not optimal for disk access. Disk operations are typically conducted at the level of a file page, which is usually 4 KiB in size. Even if only a single data element is accessed, the entire file page (and possibly multiple adjacent pages) must be read or written. Therefore, when reading or writing a small block of the array, the actual I/O operations involved may significantly exceed the amount of data needed. To mitigate this issue, chunked storage can be employed. Chunk storage (Fig. 9.4) divides the physically adjacent data into small blocks, enhancing the efficiency of accessing subsets of the data.

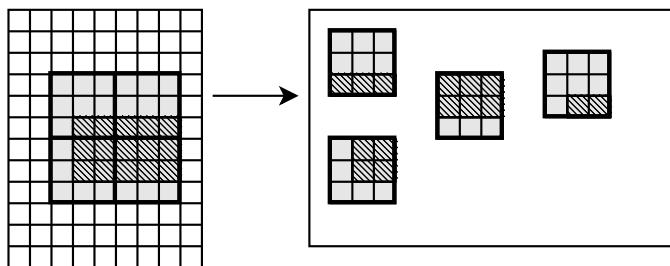


FIGURE 9.4

HDF5 chunked storage.

In the following tests, by using the chunked datasets, the second test achieves a throughput of 252 MB/s. In comparison, the first one, which does not utilize chunked storage, only delivers a throughput of 14.4 MB/s.

```
In [1]: import h5py

In [2]: %time
        with h5py.File('test_plain.h5', 'w') as f:
            d = f.create_dataset('test', (20000,20000), dtype='f8')
            d[:, :600] = 1.
CPU times: user 369 ms, sys: 1.11 s, total: 1.48 s
Wall time: 6.66 s

In [3]: %time
        with h5py.File('test_chunk.h5', 'w') as f:
```

```

d = f.create_dataset(
    'test', (20000,20000), dtype='f8', chunks=(200,200))
d[:, :600] = 1.
CPU times: user 321 ms, sys: 56.9 ms, total: 378 ms
Wall time: 380 ms

```

The tests were performed on an AWS EC2 c5d.xlarge type instance. This VM is equipped with 16 GB of physical memory and SSD local storage. To limit the available memory (2 GB in these tests), we allocated a giant array in a separate process before running the tests. To eliminate the influence of the file-system cache, we manually cleared the cache before each test using the command:

```
$ sync; echo 1 > /proc/sys/vm/drop_caches
```

Although chunked storage improves the efficiency of accessing data subsets, it introduces additional complexity and new overhead. In HDF5, a chunk is treated as the atomic object. When any element in a chunk is accessed, the entire chunk needs to be read from disk. Accessing data that spans across chunk boundaries can lead to some overhead. Aligning the access pattern with the shape of the chunk can help in reducing this overhead. For example, in the following, the first test needs to read 200 chunks while the second test needs to read 400 chunks. This difference results in a 50% difference in performance.

```

In [4]: %time
    with h5py.File('test_chunk.h5', 'r') as f:
        f['test'][200:400]
CPU times: user 13.7 ms, sys: 13.8 ms, total: 27.5 ms
Wall time: 52.2 ms

In [5]: %time
    with h5py.File('test_chunk.h5', 'r') as f:
        f['test'][610:810]
CPU times: user 14.4 ms, sys: 28.9 ms, total: 43.2 ms
Wall time: 86.6 ms

```

The second property of chunked storage is that the chunks of a dataset are stored randomly on the disk. Consequently, sequential access to the dataset actually involves some randomness in both reading and writing operations. In the following example, when reading 640 MB contiguous data, chunked storage incurs a 50% overhead compared to the plain row-major storage.

```

In [6]: %time
    with h5py.File('test_plain.h5', 'r') as f:
        a = f['test'][5000:9000]
CPU times: user 38 ms, sys: 495 ms, total: 533 ms
Wall time: 1.65 s

```

```
In [7]: %time
    with h5py.File('test_chunk.h5', 'r') as f:
        a = f['test'][5000:9000]
CPU times: user 142 ms, sys: 658 ms, total: 800 ms
Wall time: 2.49 s
```

Memory mapping

When utilizing memory mapping (memmap) for array storage, the storage layout also has a critical impact on the I/O efficiency. Memmap allows for data storage in either row-major or column-major format. It can outperform HDF5 when accessing data sequentially. This is due to the fact that memmap incurs lower overhead for data movement in memory. However, memmap lacks the capability to store data in chunks, leading to a significant performance decline when accessing non-contiguous data. To evaluate the I/O performance of memmap relative to the HDF5 format, we employ similar tests to those previously conducted for HDF5.

```
In [10]: %time
    d = np.memmap('test_plain.mmap',
                  mode='w+', shape=(20000,20000), dtype=float)
    d[:,600] = 1.
    d.flush()
CPU times: user 84 ms, sys: 992 ms, total: 1.08 s
Wall time: 1.28 s

In [11]: %time
    d = np.memmap('test_plain.mmap',
                  mode='r+', shape=(20000,20000), dtype=float)
    d[5000:9000] = 0.
    d.flush()
CPU times: user 122 ms, sys: 365 ms, total: 487 ms
Wall time: 6.23 s

In [12]: %time
    d = np.memmap('test_plain.mmap',
                  mode='r', shape=(20000,20000), dtype=float)
    a = d[5000:9000]
CPU times: user 14.6 ms, sys: 19.4 ms, total: 33.9 ms
Wall time: 33.5 ms

In [13]: %time
    d = np.memmap('test_plain.mmap',
                  mode='r', shape=(20000,20000), dtype=float)
    a = d[5000:9000].copy()
```

```
CPU times: user 103 ms, sys: 378 ms, total: 481 ms
Wall time: 1.81 s
```

Please note the difference between `numpy.memmap` and `h5py` when handling array slicing. Slicing a memmap object only generates a memoryview. The actual I/O operations are postponed until the data is explicitly accessed. This feature simplifies programming, as it eliminates the need to explicitly call file read/write commands. The operating system can automatically synchronize the necessary data between the memory and the memory-mapped file. However, this also means that we lose the flexibility to control when the I/O operations occur. Unless we make a copy of the segment of the memmap object, we are unable to accurately prefetch the required data for upcoming computation tasks. In some applications, this may decrease the overall performance.

For example, let's consider the following test of matrix multiplication on two memmap objects, which takes 42 seconds to complete. The computation time would only be 29 seconds if both arrays are entirely in memory. The additional 13 seconds is the overhead of loading data from the `memmap` files.

```
In [14]: %%time
a = np.memmap('test_a.mmap',
              mode='r', shape=(40000,10000), dtype=float)
b = np.memmap('test_b.mmap',
              mode='r', shape=(40000,10000), dtype=float)
c = a.T.dot(b)
CPU times: user 3min 47s, sys: 9.34 s, total: 3min 56s
Wall time: 42.2 s
```

By restructuring the matrix multiplication problem into a series of 10000×10000 sub-matrix multiplications, and employing the `iterate_with_prefetch` function (as described in Section 9.6.3) to overlap computation and I/O, the computation time can be reduced to 37 seconds.

```
a = np.memmap('test_a.mmap', mode='r', shape=(40000,10000), dtype=float)
b = np.memmap('test_b.mmap', mode='r', shape=(40000,10000), dtype=float)
def loader(task):
    sub_a = a[task*10000:(task+1)*10000].copy()
    sub_b = b[task*10000:(task+1)*10000].copy()
    return sub_a, sub_b

c = np.zeros((10000,10000))
for task, (sub_a, sub_b) in iterate_with_prefetch(range(4), loader):
    c += sub_a.T.dot(sub_b)
# Wall time: 37.1 s
```

Table 9.5 Performance comparison between HDF5 and NumPy memmap.

	HDF5	NumPy memmap
Sequential access	Decent performance	Slightly more efficient than HDF5
Random access	Moderate loss with chunk configuration	Significant loss if data size exceeds available memory
Frequent access	High overhead due to system calls	Lower overhead
Cache utilization	Relies on file-system cache	Based on memory pages, more manageable and efficient
Synchronization	Additional cost to move data from memory to disk	Automatic sync between memory and disk, no overhead

HDF5 vs memmap

When accessing large array objects, which technique is preferable, HDF5 or memmap? Table 9.5 outlines several factors for consideration. Generally, the HDF5 format can deliver satisfactory performance across a wide range of scenarios. Nonetheless, memmap proves superior in specific situations, including:

- Sequential data access. Memmap benefits from fewer data movement operations between system buffers.
- Frequent read and write operations on hot (frequently accessed) pages. Once the hot pages are mapped into memory, accessing them with memmap requires only memory operations. Furthermore, any modifications made to the memmap array are automatically synchronized to the file. This eliminates the overhead of data transmission in HDF5, which involves moving data from memory to disk.
- When the system has sufficient memory, memmap can efficiently utilize the system memory to cache data. If a memmap file can be entirely cached in memory, operating it is nearly as fast as direct memory access.

9.6.2 Compressing data

Compressing data is a widely used optimization technique to reduce I/O pressure.

Quantum chemistry data often exhibit specific symmetries, such as the Hermitian property of the Hamiltonian matrix, and the 8-fold permutation symmetry in electron repulsion integrals. Moreover, raw data in quantum chemistry calculations often contain a significant number of values that are nearly zero. To compress quantum chemistry data, we can leverage the symmetry properties to reduce the volume of data, and then exploit the sparsity to achieve further compression.

One option for storing compressed data is to use the compression filters provided by the HDF5 library [36]. This method requires a few extra parameters to configure the compression filters when creating the HDF5 dataset, as shown in the following example.

```

import h5py
import hdf5plugin

eri = np.ones((5050, 5050))
with h5py.File('test_lzf.h5', 'w') as f:
    d = f.create_dataset('test', eri.shape, dtype='f8', chunks=(100,100),
                         compression='lzf')
    d[:] = eri

with h5py.File('test_blosc.h5', 'w') as f:
    d = f.create_dataset('test', eri.shape, dtype='f8', chunks=(100,100),
                         **hdf5plugin.Blosc())
    d[:] = eri

```

There are several lossless compression filters available: Blosc, LZF, and LZ4. These compressors can achieve a throughput of approximately 1 GB/s using a single CPU thread (actually only one thread is supported in Python). The `h5py` library natively supports the LZF filter. To use Blosc or LZ4 compressors, the `hdf5plugin` extension is required [37]. The PyTables library provides a comprehensive guide on the relationship between compressor performance and chunk size [38]. Generally, the influence of chunk size on the performance of each compressor is relatively small.

Another method for compressing data involves utilizing the `scipy.sparse` module to transform the data into a sparse format, such as CSR (Compressed Sparse Row), BSR (Block Sparse Row), or COO (Coordinate) format. Sparse formats are not compatible with high-dimensional arrays. It is necessary to reshape the high-dimensional array into a matrix before applying a sparse format. Sparse arrays cannot be directly stored in an HDF5 file. To store a sparse array, we need to save the non-zero elements along with the metadata which indicates the locations of the non-zero elements. Below are helper functions for saving and loading sparse arrays.

```

from scipy.sparse import csr_array, bsr_array, coo_array

def save_sparse(h5obj, a):
    if isinstance(a, (csr_array, bsr_array)):
        h5obj['data'] = a.data
        h5obj['indices'] = a.indices
        h5obj['indptr'] = a.indptr
        h5obj['shape'] = a.shape
    elif isinstance(a, coo_array):
        h5obj['data'] = a.data
        h5obj['row'] = a.row
        h5obj['col'] = a.col
        h5obj['shape'] = a.shape
    else:

```

```

        raise NotImplementedError

def load_sparse(h5obj):
    assert 'data' in h5obj
    data = h5obj['data'][()]
    shape = h5obj['shape'][()]
    if 'indices' in h5obj:
        indices = h5obj['indices'][()]
        indptr = h5obj['indptr'][()]
        if data.ndim == 1:
            return csr_array((data, indices, indptr), shape=shape)
        else:
            return bsr_array((data, indices, indptr), shape=shape)
    elif 'row' in h5obj:
        row = h5obj['row'][()]
        col = h5obj['col'][()]
        return coo_array((data, (row, col)), shape=shape)
    else:
        raise NotImplementedError

```

9.6.3 Overlapping computation and I/O

To overlap computation and I/O, we can utilize non-blocking I/O operations and data prefetching.

In a blocking program, tasks are executed sequentially, one after another. The program would pause and wait for the I/O operation to finish before proceeding to the next task. Conversely, non-blocking I/O allows the program to initiate an I/O operation and then continue executing other tasks without waiting for the I/O operation to complete. The program can periodically check the status of the I/O operation and perform other tasks in the meantime. Using the `ThreadPoolExecutor` class from the `concurrent.futures` module is a convenient way to achieve non-blocking I/O. `ThreadPoolExecutor` can execute an I/O task in a separate thread and return a `future` object. By invoking the `future.result()` method, the program can be blocked until the background task is completed. The code below is an example of how to use the `ThreadPoolExecutor` class to overlap computation and I/O operations.

```

from concurrent.futures import ThreadPoolExecutor

with h5py.File('test_block.h5', 'r+') as f:
    def save_data(task_id, data):
        f[f'test/{task_id}'] = data

    with ThreadPoolExecutor() as worker:
        for task_id in range(100):

```

```
# code to generate data
data = np.ones((10000,100))
# save data asynchronously
future = worker.submit(save_data, task_id)
```

Using non-blocking I/O, we can implement the data prefetching technique to hide the latency of data access. Before the data is actually needed, we can start a thread to read data from storage in the background. When the data is required, we can use the `.result()` method to retrieve the preloaded data. For example

```
with h5py.File('test_plain.h5', 'r') as f:
    block_size = 200
    rows = f['test'].shape[0]

    def loader(row0):
        row1 = min(row0 + block_size, rows)
        return f['test'][row0:row1]

    with ThreadPoolExecutor(max_workers=1) as worker:
        future = worker.submit(loader, task)
        data1 = prefetch(task)
        do_some_computation()
        data2 = prefetch(task)
        do_some_computation()
        computing_with_data(data1.result())
        computing_with_data(data2.result())
        ...
```

The data prefetch technique can be combined with the loop iteration code. In each iteration, when the instruction of loading data is issued, we can initiate a prefetch operation for the next iteration and use the `.result()` method to access the prefetched data for the current step. To simplify this process, we implement a wrapper named `iterate_with_prefetch` as follows.

```
from concurrent.futures import ThreadPoolExecutor

def iterate_with_prefetch(tasks, loader):
    with ThreadPoolExecutor(max_workers=1) as worker:
        task_to_run = None
        for task in tasks:
            if task_to_run is None:
                future = worker.submit(loader, task)
                task_to_run = task
            continue
```

```

        data = future.result()
        future = worker.submit(loader, task)
        yield task_to_run, data
        task_to_run = task

    data = future.result()
    yield task_to_run, data

```

Using `iterate_with_prefetch`, we can write the data-loading and computing tasks in the following way.

```

with h5py.File('test_plain.h5', 'r') as f:
    block_size = 200
    rows = f['test'].shape[0]

    def loader(row0):
        row1 = min(row0 + block_size, rows)
        return f['test'][row0:row1]

    tasks = range(0, rows, block_size)
    for task, data in iterate_with_prefetch(tasks, loader):
        # code to process data

```

There also exist other techniques to achieve asynchronous I/O, such as the `asyncio` library, which allows for concurrent execution of computation and I/O operations. `asyncio` can greatly simplify the implementation of asynchronous code, making the code more comprehensive. However, HDF5 does not support `async` functions. We will not explore the use of `asyncio` in this context.

9.7 Precomputation and memoization

Avoiding redundant computations can minimize computational costs. If the same calculations are performed multiple times in a program, the results can be pre-computed and stored for later use.

There are several strategies to implement precomputation. For instance, if certain code snippets are repeated, one may adjust the data structure and the order of loops to compute these snippets in advance only once. If a function yields a limited number of possible outcomes, these outcomes can be precomputed and tabulated. When the function is called, the results can be retrieved via a table lookup.

Function-based precomputation can be performed in a different way. The first time a function is called with a specific input, its result can be stored in a cache. For subsequent calls with the same inputs, the result is fetched from the cache rather than being recomputed. This approach is known as *memoization*.

Memoization requires that the result is determined solely by the inputs, and the function does not affect other functions. This ties the concept of memoization closely to functional programming. In this section, we will explore memoization, functional programming, and associated techniques.

9.7.1 LRU cache

Implementing memoization in Python is straightforward. It just requires a Python dictionary to provide caching and lookup capabilities to the function. Here is a simple example of the memoization decorator.

```
def cache(f):
    cache = {}
    def f_cached(*args):
        if args in cache:
            return cache[args]
        result = f(*args)
        cache[args] = result
        return result
    return f_cached

@cache
def factorial(n):
    if n <= 1:
        return 1
    return factorial(n-1) * n
```

If we cache all values without any restrictions, the cache might end up consuming a large amount of memory. To address this issue, we need a strategy to decide which data should be kept in cache and which should be evicted. There are multiple strategies available for managing cache, such as:

- FIFO (First-In/First-Out) to expire the oldest entry;
- LIFO (Last-In/First-Out) to expire the newest entry;
- LRU (Least Recently Used) to keep the most recently accessed entry;
- LFU (Least Frequently Used) to keep the most frequently accessed entry.

Due to the widely existing temporal locality in data, the LRU cache strategy is well-suited and extensively utilized in a variety of applications, both in software and hardware contexts. Python offers an implementation of the `lru_cache` in the standard library `functools`. Beyond the `lru_cache`, the `cachetools` library [39] provides implementations for other cache management strategies.

Can `lru_cache` be applied to any Python functions?

When using `lru_cache` in Python, there are several technical considerations we should be aware of:

- All *input variables must be hashable and comparable*. This is the basic requirement for dictionary lookup.
- *Input variables should be immutable*.
- *Cached outputs cannot be freed*. Please note that if a cached object references other objects, those objects will not be freed until the cached object is evicted. This affects not only the memory usage of memoization but also certain resources in the operating system [40]. Typically, we should avoid caching or referencing resources such as file descriptors or sockets, as they are limited in the system.
- *Cached outputs must behave consistently* if they are accessed multiple times. Certain objects returned by concurrent functions [41] are not suitable for caching. This includes objects like the `Future` object, the `Thread` object (provided by the `threading` library), or the `coroutine` object (returned by `async` functions). When these objects are consumed, for example, by calling the `.join()` method on a `Thread`, their states are altered. Consuming these objects more than once can cause errors or unexpected outcomes. Similarly, other objects in the outputs, such as generators and iterators, should also not be cached. Although the `lru_cache` function does not explicitly prevent caching these results, we should avoid optimizing these functions with LRU cache.
- *Cached outputs should be immutable*. The function we want to memoize might return a mutable object, such as a NumPy array. Python does not enforce the immutability of cached objects. We should avoid modifying objects returned by memoized functions.

To safely apply memoization, the functional programming style is preferred in both the caller and the cached function. Before delving into functional programming, let's first examine how to manage the function input parameters for memoization.

The function input arguments must be hashable. In Python, most immutable data types, such as numbers, strings, and tuples, are hashable. We can use the built-in function `hash()` to obtain the hash value of these objects. However, mutable objects, including lists, sets, dictionaries, and NumPy arrays, are not hashable. To utilize these mutable objects in memoization, we should convert them into their corresponding immutable data types. Below are some typical conversions:

- Convert a list to a tuple;
- Change a set to a `frozenset`;
- Transform a NumPy array into bytes (utilizing the `ndarray.tobytes()` method).

When caching NumPy arrays, we may need to address issues related to the rounding of floating-point numbers. If minor errors in floating-point numbers are acceptable, we can eliminate insignificant decimal places using the `np.round` function. This approach may result in small negative values being rounded to `-0.0`. To remove the negative zeros, we can simply add `0.0` to the rounded array. For example, to limit the array to only 6 decimal places, we can hash the array with the statement

```
hash((a.round(6) + 0.0).tobytes())
```

Python does not have a built-in immutable (frozen) dictionary. To hash inputs that are presented in a dictionary format, a common workaround is to convert the dictionary into a tuple of key-value pairs. Alternatively, the `namedtuple` class from the `collections` module can be used.

```
class CustomHash:  
    ...  
    def __hash__(self):  
        return hash((tuple(self.list), frozenset(self.set),  
                    self.array.tobytes(), tuple(self.dict.items())))
```

`hash()` function can generate a hash value for certain mutable objects, such as the instance of a user-defined Python class.

```
In [1]: class Empty:  
        pass  
  
In [2]: print(hash(Empty()))  
Out[2]:  
8733822284401
```

Python automatically implements the `__hash__` and `__eq__` methods for user-defined classes. By default, the `__hash__` method generates unique hash values for different instances, even if they are created from the same class and contain the same data. Consequently, user-defined classes with the default `__hash__` implementation are not suitable for memoization techniques. An instance of such a class will not be recognized by the cache lookup, even if an identical instance is already present in the cache. As a result, calling the memoized function with such instances will always trigger a call to the underlying function. To accurately identify identical instances, it is essential to override the `__hash__` and `__eq__` methods to encode the essential attributes.

On the other hand, how can we disable memoization for certain objects? This can be achieved by setting the `__hash__` method:

```
class NotHashable:  
    __hash__ = None
```

This will lead to an `unhashable type` error when passing the instance to the memoized function.

9.7.2 Functional programming

In our previous examples, the program contains several statements that modify variables and states of the program, such as assigning a value to an element of an array. These statements must be executed in a strict order as they are implemented. This style of programming is known as *imperative programming*.

In contrast to imperative programming, programs can also be written in a style that relies only on pure functions. A pure function is a mathematical procedure that

computes a result *without any side effects* (no mutation of non-local variables or input arguments). By properly defining pure functions f_1, f_2, \dots , the entire computation can be viewed as a series of nested function calls:

```
f_n(..., f_2(f_3(f_1(x), f_2(x)), f_4(x)), ...)
```

This style of programming is known as *functional programming* (FP).

FP has several advantages in terms of program optimization:

- Memoization. Pure functions always produce the same output for the same input. This deterministic nature facilitates the use of memoization technique.
- Precomputation with closures. Closure can be used to store precomputed states. By capturing variables from the outer function, the closure retains the computed data. When the closure is invoked, these states do not require recomputation.
- Parallelism. FP makes it easier to break down the computation tasks into smaller, independent units. These units can be executed in parallel, without concern for the order of execution.
- Vectorization for immutable data. FP encourages the use of immutable data, which simplifies certain optimization operations, such as vectorization. One common obstacle to vectorization is the uncertainty regarding whether the output of an arithmetic operation overlaps with the memory address of another operand. With guaranteed data immutability, applying vectorization becomes straightforward.
- Lazy evaluation. FP supports lazy evaluation, which means expressions are evaluated only when their values are actually needed. This strategy can reduce computation cost by avoiding unnecessary calculations.

If you are interested in exploring FP techniques in Python, the book *SICP in Python* [42] can be a good starting point.

Completely abandoning assignment statements, as required by the pure FP style, might be inconvenient in Python. Actually, we only need to ensure that assignments do not introduce side effects. If an assignment statement is utilized to create local variables only, without modifying any existing objects, it is considered side-effect-free. In fact, such assignments can be translated into function calls, making the code equivalent to the assignment-free FP style. For instance, in the code below, instead of employing the assignment `dx = point1 - point2`, we can achieve the same outcome with an auxiliary function that binds the local variable `dx`.

```
def euclidean_distance(point1: ndarray, point2: ndarray) -> float:
    dx = point1 - point2
    return sum(dx * dx) ** .5

def euclidean_distance_assignment_free(point1, point2):
    def norm(dx):
        return sum(dx * dx) ** .5
    return norm(point1 - point2)
```

Let's now revisit the `schmidt_orth` function we discussed earlier.

```
def schmidt_orth(s):
    n = s.shape[0]                      # (1)
    cs = np.eye(n)                      # (2)
    for j in range(n):
        fac = s[j,j]                    # (3)
        for k in range(j):
            dot_kj = np.dot(cs[k], s[j])      # (4)
            cs[j] -= dot_kj * cs[k]          # (5)
            fac -= dot_kj * dot_kj          # (6)
        cs[j] /= fac**.5                  # (7)
    return cs.T
```

The statements in lines (1) to (4) create local variables that are not subject to any side effects. Lines (5) and (7) involve inplace operations, which introduce side effects as they modify the `cs` array. Depending on the mechanism how the local variable `fac` is linked to the input element `s[j,j]`, line (6) might change the value of `s[j,j]` and introduce side effects. To eliminate the side effects in lines (5) to (7), we can use a regular assignment statement instead of inplace modification, such as

```
fac = fac - dot_kj * dot_kj
```

Although the same variable name `fac` is used on both sides of the assignment, the instance on the left now points to a new local object, not the one in the expression `fac - dot_kj * dot_kj`. We thus derive the side-effect-free implementation.

```
def schmidt_orth_side_effects_free(s):
    n = s.shape[0]
    cs = []
    guess = np.eye(n)
    for j in range(n):
        fac = s[j,j]
        vec = guess[j]
        for k in range(j):
            dot_kj = np.dot(cs[k], s[j])
            vec = vec - dot_kj * cs[k]
            fac = fac - dot_kj * dot_kj
        cs = cs + [vec / fac**.5]
    return np.array(cs).T
```

This implementation can be further simplified to:

```
def schmidt_orth_side_effects_free(s):
    n = s.shape[0]
    cs = []
    guess = np.eye(n)
    for j in range(n):
```

```

dot_kj = [np.dot(cs[k], s[j]) for k in range(j)]           # (1)
fac = s[j,j] - sum(x * x for x in dot_kj)                 # (2)
vec = guess[j] - sum(dot_kj[k] * cs[k] for k in range(j)) # (3)
cs = cs + [vec / fac**.5]                                   # (4)
return np.array(cs).T

```

In this version, we can identify the sub-tasks within the function and the dependencies between these tasks, as shown in Fig. 9.5. Notably, tasks in lines (2) and (3) are independent, which allows for parallel execution. By substituting the NumPy library with either the Dask or JAX library, we can facilitate the parallel execution of these independent tasks across multiple devices and machines.

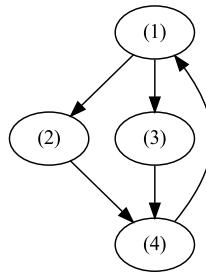


FIGURE 9.5

Task dependencies in the function `schmidt_orth_side_effects_free`.

9.7.3 Dynamic programming

Recursion is a widely utilized technique in functional programming. It allows us to recursively decompose complex problems into more manageable sub-problems. By solving these smaller problems, we can eventually resolve the initial problem. To optimize the efficiency of this process, we can use memoization to avoid unnecessary recalculations. The combination of recursion and memoization establishes the basis for the dynamic programming (DP) methodology.

To develop a DP solver, we can start by addressing the following questions:

- How to formulate the problem, particularly the definition of input and output?
- What are the boundaries of the problem?
- Given the solution of a sub-problem, how to derive the solution by enlarging the problem size by a unit?

Once we have clear answers to these questions, we can then translate these answers into program code and let the computer search for the solution.

For example, let us consider the problem of deriving the Slater determinant basis in the full configuration interaction (FCI) method. The problem states that: given n orbitals and m electrons, the determinant basis is formed by all possible combinations

of m occupied orbitals chosen from the n orbitals. Clearly, the number of possible determinants is equal to the binomial coefficient $\binom{n}{m}$.

- Inputs and outputs: The inputs of the problem (n, m) are the number of orbitals n and the number of electrons m . The output is a list of determinants. Each determinant is represented by a list of occupied orbitals.
- Boundaries: The boundaries are found at $(t, 0)$ and (t, t) . The value at $(t, 0)$ is an empty list, while the value at (t, t) is a single list, which contains all t orbitals.
- Recursion: The solution for (n, m) can be derived from the sub-problems $(n-1, m-1)$ and $(n-1, m)$. This is achieved by adding the orbital n in each determinant from the results of $(n-1, m-1)$ and keeping all determinants from $(n-1, m)$.

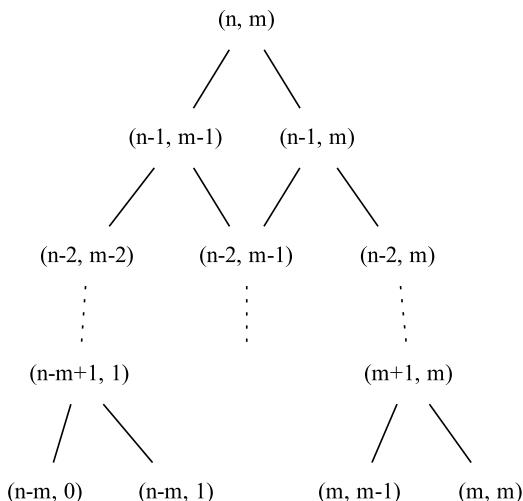


FIGURE 9.6

The dynamic programming recursion for generating FCI determinant bases.

Fig. 9.6 illustrates the process of generating determinants through a recursive procedure. The above rules for creating FCI determinant bases are translated into the DP code as follows.

```

from typing import List

class Determinant:
    def __init__(self, occupied_orbitals: List):
        self.occupied_orbitals = occupied_orbitals

    def __repr__(self):
  
```

```

        return f'Det({self.occupied_orbitals})'

    @classmethod
    def vacuum(cls):
        return cls([])

    @classmethod
    def fully_occupied(cls, n):
        return cls(list(range(n)))

    def add_occupancy(self, orbital_id):
        assert orbital_id not in self.occupied_orbitals
        return Determinant(sorted(self.occupied_orbitals + [orbital_id]))

@lru_cache(1000)
def dets(n, m) -> List[Determinant]:
    assert n >= m
    if m == 0:
        return [Determinant.vacuum()]
    elif n == m:
        return [Determinant.fully_occupied(n)]
    return (dets(n-1, m) +
            [det.add_occupancy(n-1) for det in dets(n-1, m-1)])

```

So far, we have demonstrated the top-down DP approach using recursion and memoization. A DP problem can also be solved using a bottom-up (iterating) approach. This method begins with smaller sub-problems and iteratively builds up the solution for the original problem. In the bottom-up approach, intermediate DP states are usually stored in an array. Nested loops are then employed to walk through array, systematically generating the DP states. The comparison between the top-down and bottom-up approaches is displayed in Table 9.6.

The top-down approach computes only the necessary intermediate DP states as needed. In contrast, the bottom-up approach often requires the computation of nearly all sub-problems before arriving at the final solution. This sometimes lead to additional, unnecessary workloads. However, the recursion in the top-down approach has a high overhead due to the extensive number of function calls, the increased size of the calling stack, the costs of hashing, and the costs of lookup operations in the cache table. In terms of efficiency, modern computers are more efficient to handle loop and array indexing operations than function calls and dictionary lookups. Therefore, the bottom-up approach tends to be more efficient than the top-down approach.

Converting recursion to iteration essentially involves rewriting the top-down DP code to bottom-up DP code. This conversion is generally a difficult and error-prone step in the optimization process. Before proceeding to write iteration code, a necessary step is to analyze the recursive code in order to:

Table 9.6 Comparison of the top-down and bottom-up dynamic programming approaches.

	Top-down approach (recursion)	Bottom-up approach (iteration)
Execution	A function calls itself	Instructions are repeatedly executed in nested loops
Context management	Each recursive call creates a new execution context	A single execution context
Memory usage	High due to the call stack	Low as it uses a fixed amount of memory
Efficiency	Less efficient if not optimized	More efficient in terms of CPU time and memory space

- Identify the dependencies between different DP states.
- Determine the order of the nested for-loops and the direction for each iteration.
- Establish the problem size, which determines the dimensions of the array used for storing DP states.

In the case of the FCI determinants program, the DP state array has two dimensions. The array size is $(n_{\text{orbitals}}+1, n_{\text{electrons}}+1)$, representing the possible states associated with 0, ..., n_{orbitals} orbitals and 0, ..., $n_{\text{electrons}}$ electrons. The dependencies between states are illustrated in Fig. 9.7.

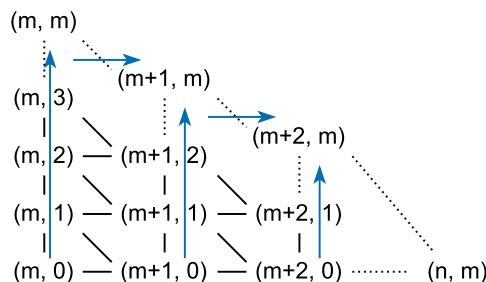


FIGURE 9.7

The bottom-up iteration for generating FCI determinant bases.

Based on these analyses, we can develop the iteration code in the following steps:

1. Create an N-dimensional array, where N represents the number of arguments in the recursion function. Each element in the array represents one DP state. The size of the DP state array is determined from the previous analysis.
2. Create N-level nested for-loops to iterate over all elements in the DP state array. The direction of the iteration is determined based on the previous analysis.

3. Copy the code block of the recursion function into the nested loops. Replace the function calls with array indexing of the DP state array. Replace the return statement with an assignment statement and proceed to the next loop iteration.
4. Run tests to validate the iteration code. If tests fail, check whether DP states are generated in the correct order. If necessary, adjust and the iteration direction or the order of the for-loops. To avoid accessing an uninitialized DP state, the DP state array can be initialized with certain invalid values. These values can help identify whether the dependent DP states are correctly initialized.

Now, we can proceed to convert the FCI recursion program into the iteration program:

```
dets = np.empty((n_orbitals+1, n_electrons+1), dtype=object)

for m in range(n_electrons+1):
    for n in range(m, n_orbitals+1):
        if m == 0:
            dets[n,m] = [Determinant.vacuum()]
        elif n == m:
            dets[n,m] = [Determinant.fully_occupied(n)]
        else:
            dets[n,m] = dets[n-1,m] + [
                det.add_occupancy(n-1) for det in dets[n-1,m-1]]
```

9.8 Optimization with lazy evaluation

Normally, programs are written in the eager evaluation style, where the program must compute the results of an expression before moving on to the next instruction. In some situations, the results of certain computations might not be used later in the code, resulting in unnecessary computation. This is where lazy evaluation becomes beneficial. Lazy evaluation is a programming strategy that postpones the evaluation of an expression until its value is actually needed. By postponing the evaluation until necessary, lazy evaluation can enhance the efficiency of a program by avoiding unnecessary computations.

In the Python language, there are several tools that offer the basic functionality of lazy evaluation, such as iterators (`range`, `zip`, `map`, `enumerate`) and generators. These tools do not generate the entire sequence of elements up front. They only produce the required elements during iterations.

In addition to these Python features, lazy evaluation is frequently employed using additional function wrappers to defer computations. For example, instead of directly constructing logging messages, it is common to define a series of logging methods

(e.g., `debug`, `info`, `warning`) to defer the construction of the logging message. This approach is adopted because formatting or converting certain objects into strings can be a resource-intensive operation. By deferring the message construction within the custom logging method, we can bypass some of the unnecessary formatting operations.

```
def debug(message, args):
    if LOGLEVEL >= DEBUG:
        print(message.format(args))
```

Lazy evaluation is particularly useful in more complex scenarios such as concurrent computation, automatic differentiation, and optimization of computational graphs. To effectively implement lazy evaluation, we can follow the guidelines below:

1. Identify the statements within the system that require lazy evaluation.
2. Determine the function calls required by these statements and convert them into lazy functions. A lazy function returns a lazy object, also referred to as a *Promise*.
3. Promises can be passed to other lazy functions, forming a chain of promises. It's important to recognize that Promises cannot be directly utilized in standard (eager) functions. Before passing a promise to a standard function, its value must be explicitly computed.
4. Evaluating a promise would trigger the evaluation of all subsequent promises.

Let's consider the `schmidt_orth` function as an example to see how lazy evaluation can be implemented in a program. In the implementation below, we employ the `defer` function to generate a promise object, which is essentially a Python function.

```
import numpy as np

def defer(func, *args):
    promise = lambda: func(*[force_eval(x) for x in args])
    return promise

def force_eval(x):
    if callable(x):
        x = x()
    return x

def dot(a, b):
    return defer(np.dot, a, b)

def subtract(a, b):
    return defer(np.subtract, a, b)

def multiply(a, b):
```

```

        return defer(np.multiply, a, b)

def divide(a, b):
    return defer(np.divide, a, b)

def power(a, b):
    return defer(np.power, a, b)

def array(a):
    return defer(lambda: np.array([x() for x in a]))

def schmidt_orth_lazy(s):
    n = s.shape[0]
    cs = []
    guess = np.eye(n)
    for j in range(n):
        fac = s[j,j]
        vec = guess[j]
        for k in range(j):
            dot_kj = dot(cs[k], s[j])
            vec = subtract(vec, multiply(dot_kj, cs[k]))
            fac = subtract(fac, multiply(dot_kj, dot_kj))
        cs = cs + [divide(vec, power(fac, .5))]
    return defer(np.transpose, array(cs))

if __name__ == '__main__':
    s = np.eye(3)
    cs = schmidt_orth_lazy(s)
    print(cs())

```

Now, let's update the implementation to demonstrate how lazy evaluation helps to avoid unnecessary calculations. Here, we replace the `defer` function with the `Promise` class. This modification enhances the object-oriented feature of the lazy object, making it easier to extend in future developments. Within the `Promise` class, we incorporate a counter to monitor the executed operations.

```

...
class Promise:
    count = 0

    def __init__(self, func, *args):
        self._func = func
        self._args = args

```

```
def __repr__(self):
    return f'Deferred {self._func}'

def __call__(self):
    return self.compute()

def compute(self):
    Promise.count += 1
    args = [force_eval(x) for x in self._args]
    return self._func(*args)

def force_eval(x):
    if isinstance(x, Promise):
        x = x.compute()
    return x

def array(a):
    def build_array(a):
        if isinstance(a, Promise):
            return np.array(a.compute())
        if not isinstance(a, (tuple, list)):
            return np.array(a)
        # Nested list should be considered for a complete implementation.
        # For simplicity, we only consider a plain list.
        return np.array([force_eval(x) x for x in a])
    return Promise(build_array, a)

def schmidt_orth_lazy(s, idx=slice(None)):
    n = s.shape[0]
    cs = []
    guess = np.eye(n)
    for j in range(n):
        fac = s[j,j]
        vec = guess[j]
        for k in range(j):
            dot_kj = dot(cs[k], s[j])
            vec = subtract(vec, multiply(dot_kj, cs[k]))
            fac = subtract(fac, multiply(dot_kj, dot_kj))
        cs = cs + [divide(vec, power(fac, .5))]
    return Promise(np.transpose, array(cs[idx]))

if __name__ == '__main__':
    s = np.eye(3)
    cs = schmidt_orth_lazy(s)
```

```

    cs()
    print(Promise.count) # 113
    Promise.count = 0
    cs = schmidt_orth_lazy(s, idx=-1)
    cs()
    print(Promise.count) # 94

```

In the test, we observe that calculating only the last orthogonal basis requires fewer operations, which is 94, compared to selecting the last one after calculating all orthogonal bases, which totals 113. The difference in operation counts indicates that lazy evaluation “intelligently” avoids 15% of the unnecessary calculations, even though we do not know specifically which calculations are skipped. For this particular problem, it should be noted that the operation counts of the two cases are actually the same if employing memoization for the `Promise.compute` method. In order to demonstrate the effects of lazy evaluation, we did not use the memoization technique to optimize operation counts.

Within the `Promise` class, one can implement more features, such as concurrent computation, automatic differentiation, dry-run methods, sanity checks, visualization of the calling graph, and dynamical schedulers for distributing computations, and so forth. These features are commonly found in tools like PyTorch, JAX, Dask, and Joblib. The foundation of these tools is the technique of lazy evaluation with the `Promise` object.

As an example, we introduce the `parallel_compute` method to demonstrate the capability for concurrent computation.

```

import time
from concurrent.futures import ThreadPoolExecutor, Future
import numpy as np
import networkx as nx

concurrent_executor = ThreadPoolExecutor(max_workers=4)

class Promise:
    _edges = []

    def __init__(self, func, *args):
        self._func = func
        self._args = args
        self._future = None
        Promise._edges.extend(
            # A directed edge, means self depends on x
            [(x, self) for x in args if isinstance(x, Promise)])]

    def __repr__(self):
        return f'Deferred {self._func}'


```

```
def compute(self):
    return self._func(*[force_eval(x) for x in self._args])

def __call__(self):
    return self.compute()

# Many features can be done in lazy evaluation mode, such as automatic
# differentiation, visualization of the call graph, etc

def submit(self):
    def task():
        time.sleep(0.1)
        # dependent tasks will be blocked by the .result() method in
        # force_eval() function.
        return self._func(*[force_eval(x) for x in self._args])
    self._future = concurrent_executor.submit(task)
    return self._future

def result(self):
    if self._future is None:
        return self.compute()
    else:
        return self._future.result()

def parallel_compute(self):
    # Build a dependency tree based on directed edges.
    tree = nx.DiGraph(Promise._edges)
    Promise._edges.clear()
    # topological_sort generates an iterator to traverse the dependency
    # tree. Leaves are evaluated first.
    futures = [task.submit() for task in nx.topological_sort(tree)]
    print(f'{len(futures)} tasks to execute')
    # self is the last task (the root) in the dependency tree.
    return futures[-1].result()

def force_eval(x):
    if isinstance(x, Promise):
        return x.result()
    return x

if __name__ == '__main__':
    s = np.eye(3)
    cs = schmidt_orth_lazy(s)
```

```
t0 = time.perf_counter()
print(cs.parallel_compute())
t1 = time.perf_counter()
print(t1 - t0)
```

To illustrate the effects of concurrent execution, we insert `time.sleep(0.1)` into each computation task. In this scenario, we have a total of 23 tasks. Executing these tasks sequentially would approximately take 2.3 seconds. However, by employing lazy evaluation and parallel computing with 4 workers, the execution time is reduced to just 0.6 seconds. Clearly, operations are executed concurrently.

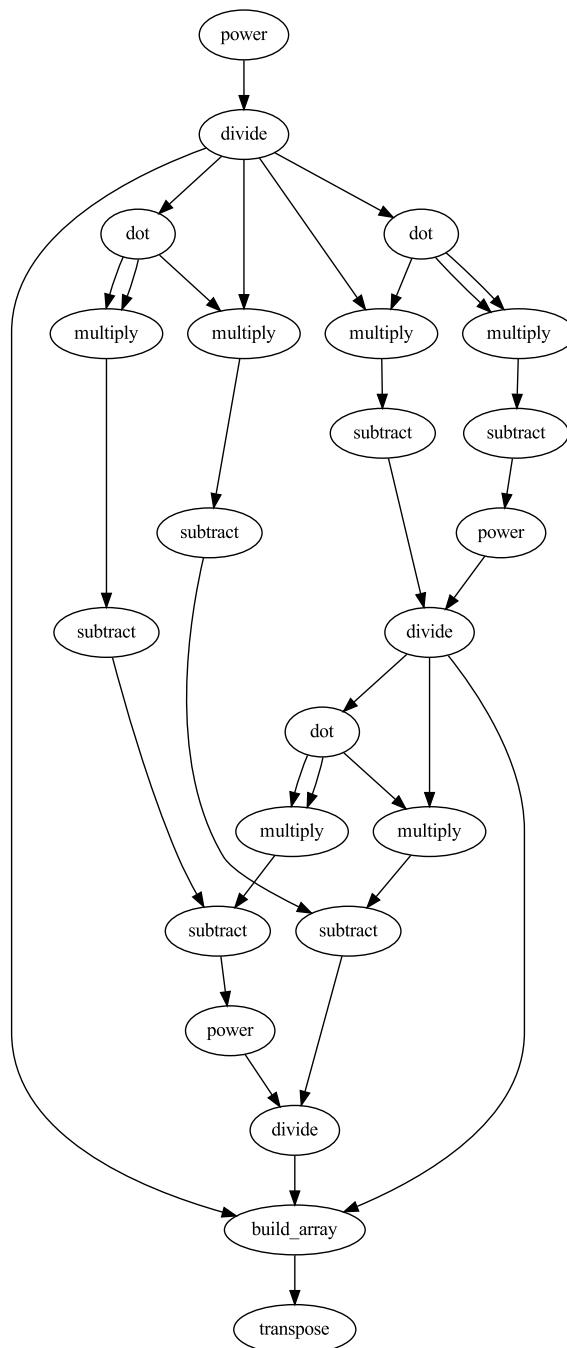
Lazy evaluation generates a computational graph, which acts as a dependency tree for tasks (Fig. 9.8). The `parallel_compute` method is essentially a job scheduler: it traverses the computational graph and then launches tasks. The dependency tree is represented as a directed graph, where directed edges maintained in the attribute `Promise._edges`. We use the `topological_sort` function provided by `networkx` to walk through all the nodes of the directed graph. This process begins with the most dependent node and moves towards the root node.

The `parallel_compute` method utilizes the `ThreadPoolExecutor` (for more details, see Chapter 10) to launch tasks. The outcome of the computation is represented by a `Future` object. When the `Future` object is evaluated via its `.result()` method, the computation is paused until all the dependent tasks are completed. In addition to the `ThreadPoolExecutor` as demonstrated in this example, other executors can be used to launch the actual computations, such as the Celery workers (Chapter 7), the ECS workers (Chapter 7), or SLURM job runners, etc. If GPUs or other specialized hardware are available, the job scheduler can be modified to allocate jobs onto these devices. This enables heterogeneous computation in a simple manner.

Summary

This chapter covers various techniques and methodologies for Python program optimization, including profiling techniques, performance analysis, Python code optimization strategies, Python compilation techniques, I/O optimization, precomputation and lazy evaluation.

In this chapter, we compared the cost estimation of common Python operations and their corresponding operations in C/C++. The cost estimation reveals the potential improvement one might expect from rewriting a Python program in C/C++. Python executes bytecode, which is typically about 10 times slower than the corresponding operations in C++. For arithmetic computations, C++ programs can potentially achieve speeds up to 100 times faster than Python arithmetic computation code. However, when the overhead in Python programs arises from the kernel space of the operating system, such as the memory management, C/C++ programs do not have significant performance advantages.

**FIGURE 9.8**

Task tree generated by lazy evaluation for the function `schmidt_orth_lazy`.

Certain Python coding practices, such as efficient iteration, list and dictionary comprehensions, and the use of tensor operations, can enhance efficiency. These optimizations should be implemented before considering the compilation of Python code or rewriting it in C/C++. The next step in optimization is the Python compilation technique, which reduces the overhead associated with the dynamic interpretation of Python bytecode. Python programs can be compiled using tools such as Numba, Cython, and Pythran. To achieve optimal efficiency, these compilation techniques also require certain coding adaptations. In this step, the core principle is to eliminate Pythonic code and reduce uncertainty in data structures and data types. Additionally, to achieve efficient numerical computations, it is necessary to fine-tune the loop structure, array indexing, and array views to enable SIMD vectorization in Python compilation. Numba's JIT (Just in Time) compilation generally offers efficiency improvements for regular Python code, but it is more conservative in utilizing SIMD vectorization. Cython and Pythran are more adept at compiling numerical computation code that leverages SIMD vectorization. If the optimization involves precise control over memory management, interaction with compilers, and hardware-specific optimizations, it is still necessary to write C/C++ code directly for these operations.

Identifying the hotspots and analyzing the bottlenecks are major challenges in program optimization. This chapter introduced several profilers to identify the hotspots in a program. Commonly used Python profilers include cProfile and line-profiler. They may slightly reduce the execution speed of Python programs and affect the accuracy of profiling. The py-spy profiler minimizes these impacts as it profiles Python programs based on event samples. To profile the C/C++ extensions of a Python program, both py-spy and perf profilers are commonly used.

Program bottlenecks, based on their characteristics, can generally be categorized as computation-bound, I/O-bound, and memory-bound. We demonstrated how these bottlenecks can be identified through profiling. Different types of bottlenecks require different optimization strategies. To improve the I/O performance of a program, the optimization strategies include redesigning the storage layout, reducing data transfer, and using asynchronous computation to overlap computation and I/O operations. For computation-bound and memory-bound applications, the focus of optimization should be on data locality, cache utilization, and cache coherence.

Precomputing and lazy evaluation are also useful methods to optimize programs. Although they cannot speed up the execution of individual Python statements, these techniques can help the program eliminate unnecessary computations, reducing the computational load.

No techniques can magically produce optimal performance. When optimizing a program, it is essential to identify the sources of bottlenecks and apply the appropriate techniques to address specific performance issues. Optimizing Python programs often requires knowledge of C/C++ programming and understanding of operating systems. It is highly recommended to acquire a basic understanding of C/C++ programming and operating system principles.

References

- [1] M. Shannon, Implementation plan for speeding up CPython, <https://github.com/markshannon/faster-cpython/blob/master/plan.md>, 2024.
- [2] M. Ankerl, Comprehensive C++ hashmap benchmarks 2022, <https://martin.ankerl.com/2022/08/27/hashmap-bench-01/>, Aug. 2022.
- [3] R.E. Bryant, D.R. O'Hallaron, Computer Systems: A Programmer's Perspective, 2nd edition, Addison-Wesley Publishing Company, USA, 2010.
- [4] U. Drepper, What every programmer should know about memory, <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>, Nov. 2007.
- [5] Wikipedia Contributors, Instruction pipelining, https://en.wikipedia.org/wiki/Instruction_pipelining, 2024.
- [6] Intel Corporation, Intel 64 and IA-32 architectures software developer manuals, <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>, 2024.
- [7] A. Asadchev, E.F. Valeev, High-performance evaluation of high angular momentum 4-center Gaussian integrals on modern accelerated processors, *The Journal of Physical Chemistry A* 127 (51) (2023) 10889–10895, <https://doi.org/10.1021/acs.jpca.3c04574>, pMID: 38090753.
- [8] Wikipedia Contributors, Instruction-level parallelism, https://en.wikipedia.org/wiki/Instruction-level_parallelism, 2024.
- [9] T. Downs, Gathering Intel on Intel AVX-512 transitions, <https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html>, 2020.
- [10] B. Fred, py-spy: sampling profiler for Python programs, <https://github.com/benfred/py-spy>, 2024.
- [11] B. Gregg, perf examples, <https://www.brendangregg.com/perf.html>, 2024.
- [12] B. Gregg, Flame graphs, <https://www.brendangregg.com/flamegraphs.html>, 2024.
- [13] S.K. Lam, A. Pitrou, S. Seibert, Numba: a LLVM-based Python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1–6, <https://doi.org/10.1145/2833157.2833162>.
- [14] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python+NumPy programs, <http://github.com/google/jax>, 2018.
- [15] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, K. Smith, Cython: the best of both worlds, *Computing in Science & Engineering* 13 (2) (2011) 31–39, <https://doi.org/10.1109/MCSE.2010.118>.
- [16] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, A. Raynaud, Pythran: enabling static optimization of scientific Python programs, *Computational Science and Discovery* 8 (1) (2015) 014001, <https://doi.org/10.1088/1749-4680/8/1/014001>.
- [17] K. Hayen, Nuitka Contributors, Nuitka the Python compiler, <https://nuitka.net/>, 2024.
- [18] M. Dufour, the Shed Skin contributors, Shed skin documentation, <https://shedskin.readthedocs.io/en/latest/documentation.html>, 2024.
- [19] A. Shajii, G. Ramirez, H. Smajlović, J. Ray, B. Berger, S. Amarasinghe, I. Numanagić, Codon: a compiler for high-performance Pythonic applications and DSLs, in: Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Association for Computing Machinery, New York, NY, USA, 2023, pp. 191–202, <https://doi.org/10.1145/3578360.35580275>.
- [20] Numba Developers, Numba architecture, <https://numba.readthedocs.io/en/stable/developer/architecture.html>, 2024.

- [21] Continuum Analytics, llvmlite: a lightweight LLVM-Python binding for writing JIT compilers, <https://llvmlite.readthedocs.io/en/latest/index.html>, 2024.
- [22] Numba Developers, Numba performance tips, <https://numba.pydata.org/numba-doc/dev/user/performance-tips.html>, 2024.
- [23] Numba Developers, Numba developer manual - notes on inlining:w, <https://numba.readthedocs.io/en/stable/developer/inlining.html>, 2024.
- [24] Numba Developers, Extending numba - implementing @overload for numpy functions, <https://numba.readthedocs.io/en/stable/extending/overloading-guide.html#implementing-overload-for-numpy-functions>, 2024.
- [25] Numba Developers, Numba user manual - compiling code ahead of time, <https://numba.readthedocs.io/en/stable/user/pycc.html>, 2024.
- [26] Numba Developers, Numba faq: why is my loop not vectorized?, <https://numba.readthedocs.io/en/latest/user/faq.html#why-has-my-loop-not-vectorized>, 2024.
- [27] S. Behnel, R. Bradshaw, D.S. Seljebotn, G. Ewing, W. Stein, G. Gellner, Cython documentation - typed memoryviews, <https://cython.readthedocs.io/en/latest/src/userguide/memoryviews.html?specifying-more-general-memory-layouts=>, 2024.
- [28] S. Behnel, R. Bradshaw, D.S. Seljebotn, G. Ewing, W. Stein, G. Gellner, et al., Cython documentation - compiler directives, https://cython.readthedocs.io/en/latest/src/userguide/source_files_and_compilation.html#compiler-directives, 2024.
- [29] S. Behnel, R. Bradshaw, D.S. Seljebotn, G. Ewing, W. Stein, G. Gellner, Cython documentation - using multiple threads, https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html#using-multiple-threads, 2024.
- [30] P. Brunet, S. Guelton, Pythran documentation - supported modules and functions, <https://pythran.readthedocs.io/en/latest/SUPPORT.html>, 2024.
- [31] S. Behnel, R. Bradshaw, D.S. Seljebotn, G. Ewing, W. Stein, G. Gellner, et al., Cython documentation - pythran as a numpy backend, [https://cython.readthedocs.io/en/latest/src/userguide\(numpy_pythran.html](https://cython.readthedocs.io/en/latest/src/userguide(numpy_pythran.html), 2024.
- [32] GNU Compiler Collection Team, Gcc documentation - declaring attributes of functions, <https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html>, 2024.
- [33] Intel Corporation, Intel intrinsics guide, <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>, 2024.
- [34] S. Byrne, Beware of fast-math, <https://simonbyrne.github.io/notes/fastmath/>, 2015.
- [35] Numba Developers, Numba reference manual - compilation options, <https://numba.pydata.org/numba-doc/dev/reference/envvars.html#compilation-options>, 2024.
- [36] The HDF Group, HDF5 Filter Plugins, https://support.hdfgroup.org/documentation/hdf5/latest/_h5_p_l_u_g.html, 2024.
- [37] European Synchrotron Radiation Facility, hdf5plugin, <http://www.silx.org/doc/hdf5plugin/latest/index.html>, 2024.
- [38] PyTables maintainers, Pytables user's guide - optimization tips, <https://www.pytables.org/usersguide/optimization.html>, 2024.
- [39] T. Kemper, cachetools, <https://github.com/tkem/cachetools>, 2024.
- [40] M. Kerrisk, limits.conf(5) — Linux manual page, <https://www.man7.org/linux/man-pages/man5/limits.conf.5.html>, 2024.
- [41] Python Software Foundation, The Python standard library - concurrent execution, <https://docs.python.org/3/library/concurrency.html>, 2024.
- [42] J. Denero, SICP in Python, <https://wizardforcel.gitbooks.io/sicp-in-python/content/>, 2024.

Parallel computation

10

Why do we need parallel computation in Python programs? The answer is straightforward: to improve the program performance. Although performance can be enhanced by using faster hardware, optimized libraries, or more efficient Python interpreters, parallel execution remains an attractive and effective method.

However, parallel programming is fundamentally different from serial programming. It presents several unique challenges:

- *Correctness*. Unlike serial programs, parallel programs can produce more errors or entirely different outcomes due to the non-deterministic nature of parallel computing. The issue of reproducibility can cause significant difficulties when debugging a parallel program. Additionally, race conditions in parallel programs occasionally lead to incorrect outcomes, which may not be easy to detect.
- *Complexity*. Parallel programming introduces new computation models and abstractions, which in turn bring new technical challenges such as race conditions, deadlocks, and communication problems. The code for parallel programs is significantly more complex to understand and maintain compared to serial programs.
- *Efficiency*. The efficiency of a parallel program is influenced by various factors, including hardware architecture, computation models, software, and operating systems. In some instances, blindly adding parallel resources can actually reduce the efficiency of parallel programs. The hardware, operating systems, and other underlying components are not transparent for developing efficient parallel programs,

There are various techniques and tools available to enable parallelism in Python programs, including:

- Instruction level parallelism (SIMD vectorization) for fine-grained parallel processing.
- OpenMP, which provides automated threading parallelization.
- The `threading` module for managing concurrent threads.
- The `multiprocessing` module for parallelism across processes.
- Message Passing Interface (MPI) for parallelism across processors and computer nodes.
- Coroutines and asynchronous modules, such as `asyncio`, for concurrent execution.
- Distributed computation frameworks, such as `dask`, `Celery`, and `Ray`.
- GPU-acceleration tools, such as `pyCUDA`, `cupy`, `pytorch`, and others.

The SIMD vectorization, GPU acceleration, and distributed computation are covered in Chapters 9, 11, and 7, respectively. We will discuss the remaining technical topics in this chapter. This chapter will address the following critical questions, which are essential for mastering any parallel computation technique:

- How to launch multiple concurrent workers or tasks?
- How to achieve high occupancy of the workers?
- How to exchange information between workers or tasks?
- How to manage the states and results of a computation task?

10.1 Multithreading

Threading level parallelism enables multiple threads to execute concurrently and access the same memory space. This allows different threads to exchange information conveniently through the shared memory space. In Python, the `threading` module and the `concurrent.futures.ThreadPoolExecutor` class are available for implementing threading level parallelism.

10.1.1 The `threading` module

The core functionality of the `threading` module is the `threading.Thread` class, which is responsible for creating a new thread. By calling the `.start()` method of a `Thread` instance, we can initiate the thread. To ensure synchronization, we can use the `.join()` method, which will halt the main thread until the created thread completes its execution.

Let us consider the parallelization of computing Coulomb matrix as an example. This is a common computational task in quantum chemistry, for which we will present more theoretical background in Chapter 13. Essentially, this problem is a tensor contraction between a four-index tensor `eri` (short for electron-repulsion integrals) and a square matrix `dm` (representing the density matrix). By using the `einsum` function, we can simplify the computation of Coulomb matrix program into the following `get_j` function.

```
def get_j(eri, dm):
    return np.einsum('ijkl,ji->kl', eri, dm)
```

In real application, the `eri` tensor is usually too large to be computed and stored in memory all at once. Instead, its elements are often computed on-the-fly. To demonstrate parallel computation techniques, we will omit the details of the `eri` tensor computation and use a placeholder `compute_eri` function to simulate the real-time calculation of a sub-block of the `eri` tensor. Here is a potential parallel implementation of the `get_j` function, which employs multithreading to distribute the workload for the third index of the `eri` tensor:

```

def compute_eri(k, n):
    # Mimic the computation of ERIs
    np.random.seed(k)
    return np.random.rand(n,n,n)

output = np.zeros((n,n))
def get_j_task(k, dm):
    output[k] += np.einsum('jil,ij->l', compute_eri(k, n), dm)

threads = [threading.Thread(target=get_j_task, args=(k, dm))
           for k in range(n)]
for t in threads:
    t.start()
for t in threads:
    t.join()

```

The `threading` module offers fundamental components for multithreading. Users need to handle thread management, data exchange, and error management. Let's consider some technical questions regarding the multithreading parallelism.

How to manage data sharing between threads?

While sharing data between threads is not difficult, there are two primary concerns regarding data access to shared content:

- *Race conditions*, where multiple threads write to the same memory address.
- *Read-before-write issue*, which occurs when a thread reads an incomplete value of a shared variable before another thread has finished writing to that variable.

To address these issues, the `threading` module provides tools such as locks, semaphores, events, and condition variables. These tools help control the program flow and safeguard shared contents in multithreaded parallel programs. We will discuss these tools in Section 10.2.

How to monitor tasks running on child threads?

The `threading.Thread` class by itself is not sufficient to monitor the status of tasks executed on child threads. It does not offer mechanisms to manage the output or the exceptions of a task running on child threads. If an error occurs on a child thread, the main thread is not able to handle the error.

How to ensure threading efficiency?

Several factors can impact the efficiency of a multithreaded function:

- *Global Interpreter Lock (GIL)*. The GIL is a major constraint for the multithreading parallel performance. The GIL is acquired by the Python interpreter when executing bytecode. It is only released in certain scenarios, such as when performing

ing I/O operations or invoking external C libraries. Consequently, multithreading cannot facilitate the parallel processing of bytecode and might even decrease performance under certain circumstances. However, the situation is different for programs engaged in numerical computations. Despite the potential impact of the GIL on bytecode execution, multithreading can often lead to a considerable speedup. This is because many numerical functions, particularly those from libraries like NumPy and other tensor libraries, are implemented in compiled languages like C and C++. These functions can operate without the participation of Python interpreter, and the GIL is released during their execution.

- *Number of threads.* A limitation of multithreading is the overhead of context switching, which can reduce the effective CPU resources available for the computation. When the number of threads far exceeds the number of CPU cores, the cost of context switching can slow down the program.
- *Cache coherence.* Cache coherence can become a bottleneck in multithreaded programs. Let's consider the following version of the `get_j_task` function:

```
output = np.zeros((n, n))
def get_j_task(l, dm, output):
    output[:, l] += np.einsum('jik,ij->k', eri[:, :, :, l], dm)
```

When performing multithreading parallelization for this task, each thread needs to update a column of the global variable `output` matrix. The `output` matrix is stored in a row-major order. Accessing columns disrupts the sequential memory operations. The adjacent memory addresses may be accessed by different CPU cores, leading to *false sharing* in CPU caches [1]. This situation forces the hardware to frequently lock and synchronize cache lines, which significantly reduces memory access efficiency. For more details of these technical issues, readers may refer to the book *What Every Programmer Should Know About Memory* [2].

Clearly, there are several shortcomings of the `threading.Thread` class. It lacks a concise way to monitor tasks for their running status, errors and execution results. Additionally, it is unable to manage and reuse the threading resources. In practice, the `threading.Thread` class is rarely used by itself for multithreading parallelization. It primarily serves as the fundamental component of multithreading parallelization. Other multithreading tools, built upon the `threading` module, can create more robust frameworks for parallel execution. One such example is the `ThreadPoolExecutor` offered by the `concurrent.futures` module. It provides a significant improvement over the functionality of `threading.Thread` and is frequently used for parallel computation.

10.1.2 ThreadPoolExecutor

The `ThreadPoolExecutor` class offers a `.submit()` method to dispatch tasks and returns a `Future` object representing the unfinished result. To obtain the result, we can use the `.result()` method of the `Future` object. It is important to explicitly call the

.result() method for each Future object to ensure the task execution, even if we do not need the output of the task. Unlike in a serial program where errors are always displayed, the ThreadPoolExecutor does not automatically raise errors from child threads. The absence of error messages does not necessarily imply that errors have not occurred on child threads.

To illustrate how to use ThreadPoolExecutor, let's once more use the `get_j` function as an example. The sample code below parallelizes over the first index of the `eri` tensor. The partially contracted results produced by the parallel tasks are then aggregated by the `np.sum` function.

```
def get_j_task(i, dm):
    n = dm.shape[0]
    return np.einsum('jk1,j->k1', compute_eri(i, n), dm[:,i])

with concurrent.futures.ThreadPoolExecutor(max_workers=8) as scheduler:
    fs = [scheduler.submit(get_j_task, i, dm) for i in range(n)]
    output = np.sum([f.result() for f in fs])
```

Upon exiting the `with` context of `ThreadPoolExecutor`, the main thread will be blocked until all workers have finished their tasks and the scheduler has shut down. If we intend to asynchronously execute certain code in the main thread while leaving the `ThreadPoolExecutor` performing some operations in the background, the asynchronous part must be placed inside the `with` context block. This ensures that the asynchronous code is not blocked by the `ThreadPoolExecutor.__exit__` method.

While the `ThreadPoolExecutor` offers more functionalities than the `Thread` class, does it resolve all the issues associated with `threading.Thread`? What are the advantages of using `ThreadPoolExecutor` over `threading.Thread`?

- The GIL limitation for threads is not resolved by the `ThreadPoolExecutor`.
- Better management of thread resources. As the name suggests, `ThreadPoolExecutor` manages threads through a pool. Each thread acts as a worker that can be reused, which reduces the overhead of thread creation and context switching. By default, the maximum number of workers is limited to 32.
- Enhanced task management through the `Future` object. With the `Future.result()` method, we can retrieve the outcome of a task and capture any errors or exceptions that occurred during its execution. Additionally, the `Future` object allows us to track the state of a task, which can be `PENDING`, `RUNNING`, `CANCELLED`, or `FINISHED`. We can call the `Future.cancel()` method to cancel a task if needed.
- Task Scheduling. The `ThreadPoolExecutor` offers basic functionality as a task scheduler. It manages and executes tasks using a first-in-first-out (FIFO) queue, which operates in a single producer multiple consumers execution mode. The number of consumers, or workers, is determined by the `max_workers` argument when initializing the executor. Once a task is completed, the consumers immediately move on to the next task in the queue until all tasks have been completed. Dynamic load balancing is achieved through the producer-consumer pattern. This

pattern is a highly useful model for parallel computing. We will explore this pattern in more detail in Section 10.3.

- Higher-level interfaces. The `ThreadPoolExecutor` provides higher-level interfaces that simplify the design of parallel execution algorithms. One only needs to memorize the `.submit()` method for executing a single task and the `.map()` method for processing a batch of tasks. These concepts are applied in other parallel task execution models as well. For example, when switching to the `concurrent.futures.ProcessPoolExecutor`, the same interface convention can be followed. The higher-level interface helps us focus on the computation algorithm itself, rather than the tedious syntax and technical details like thread creation and synchronization operations.

While the `ThreadPoolExecutor` exhibits many great features, it can potentially introduce the problem of additional memory overhead. This is because the arguments and results of the parallel tasks are stored in `Future` objects, leading to extra memory consumption. This becomes problematic if the program generates a large number of `Future` objects. If the `Future` objects cannot be consumed and freed, the corresponding temporary objects can continue to occupy memory. For instance, in the previous `get_j` program, we created a list of `Future` objects for all parallel tasks. The `ThreadPoolExecutor` workers continuously write the output of each task, which is an n^2 -sized matrix, into the `Future` objects. Even though the matrix of the final result contains only n^2 elements, the list of `Future` objects for intermediates occupies the space of n^3 elements. This leads to unnecessary memory usage and can potentially impact the performance of the program.

Using the `ThreadPoolExecutor.map` method to process batch jobs can mitigate the memory usage issue to some extent. This method returns a generator of `Future` objects. Unlike the list container for `Future` objects, iterating over the generator allows for continuously consuming the `Future` objects it yields. The memory of the consumed `Future` objects can be immediately freed. For example, in the `get_j` program, we could iterate over the generator object returned by the `map` function, which eliminates the need to retain the n temporary matrices in the memory. The output is accumulated at the same time that the `Future` objects are processed:

```
output = np.zeros((n,n))
with concurrent.futures.ThreadPoolExecutor(max_workers=8) as scheduler:
    mapped = scheduler.map(get_j_task, range(n), itertools.repeat(dm, n))
    for v in mapped:
        output += v
```

Even though the generator returned by the `ThreadPoolExecutor.map` method can help reduce memory usage, it does not finalize the issue of excessive memory usage caused by `Future` objects. The `Future` objects for all tasks are initialized and submitted in the `map` method. As the workers process the tasks, their results are continuously written into the corresponding `Future` objects. If these `Future` objects are not promptly consumed by the main thread, they can still occupy a significant amount of memory.

Apart from parallel data processing with similar tasks, the `ThreadPoolExecutor` is also suitable for running different tasks concurrently. A common scenario for concurrent tasks is the coordination of CPU-intensive computations and I/O operations. In this scenario, the I/O operations are offloaded to the worker of the `ThreadPoolExecutor`, while the main thread is used to perform CPU computations. This functionality was demonstrated in the `iterate_with_prefetch` function in Section 9.6.3 of Chapter 9.

10.2 Lock and thread synchronization

In our previous `get_j` example using the `ThreadPoolExecutor`, we create numerous `Future` objects to store the intermediate matrices. If these intermediates are not consumed immediately, they could lead to substantial memory consumption. To avoid storing the intermediate matrices, they should be consumed as soon as they are generated. However, there is a risk of race conditions when multiple threads access the same memory location simultaneously. To resolve race conditions, a mutex (mutual exclusion lock) is the most common option to protect the shared variable, which is the `output` matrix in this example.

```
mutex = threading.Lock()
output = np.zeros((n,n))
def get_j_task(i, dm):
    j_priv = np.einsum('jk1,j->k1', eri[i], dm[:,i])
    with mutex:
        output += j_priv

with concurrent.futures.ThreadPoolExecutor(max_workers=8) as scheduler:
    mapped = scheduler.map(get_j_task, range(n), itertools.repeat(dm, n))
```

We place the inplace updates for the `output` matrix within the `with mutex` context. The `with` statement for the mutex automatically triggers the `lock.acquire()` operation before entering the critical section and the `lock.release()` operation upon completion. The mutex guarantees that the shared `output` matrix is modified by only one thread at any particular time.

10.2.1 Mutex

Mutex is an effective solution for protecting shared variables from race conditions. A mutex lock can be in one of the two states:

- *held*, indicating that a thread is accessing the critical section. When a thread calls `lock.acquire()`, the lock is set to *held*. This action blocks any other threads that attempt to acquire the lock.

- *not held*, indicating that the critical section is unoccupied at the moment. When a thread calls `lock.release()`, the lock is set to *not held*. This action allows one of the waiting threads to acquire the lock and enter the critical section.

In a scenario where multiple threads are attempting to enter the critical section, each thread calls `lock.acquire()` to request control of the mutex. Only one thread can hold the mutex and execute the code within the critical section at any given time.

Do all operations on shared variables require protection from race conditions with a mutex? The answer is no. Not all operations on shared variables are vulnerable to race conditions. For example, atomic operations, which can be completed in a single step without the possibility of interruption, do not require a mutex. In the Python documentation [3], we can find several atomic operations in Python. They are mostly the operations for built-in data types (such as integers, lists, and dictionaries). However, some operations that appear to be atomic, like `i = i + 1`, are not. When in doubt, it is always safer to use a mutex lock.

In C/C++ programs, manipulating mutexes is often considered a heavy operation due to the involvement of system calls and the potential thread context switching required during the acquisition or release of a lock. As we discussed in Chapter 9, context switching is a fundamental feature of the operating system with high overhead. Such overhead can sometimes be higher than the cost of executing hundreds of bytecode instructions by the Python interpreter. This raises the question: Is it necessary to minimize the use of mutexes in Python to reduce the overhead of system calls?

The short answer is no. Actually, the GIL in Python prevents frequent thread context switching. The GIL is periodically released to enable thread switching, typically every 5 milliseconds. Even if a mutex lock is frequently acquired or released, it is typically executed by the same thread. This operation only involves a few CPU instructions and, in most cases, does not trigger a system call. The cost is often similar to executing one or a few regular Python bytecode instructions. As a result, the use of mutexes in Python has a minimal impact on overall performance. This is a notable difference in the use of mutexes between Python and C/C++ programs.

10.2.2 Event and condition variable

In addition to mutex locks, the Python `threading` module provides various mechanisms for coordinating and synchronizing threads. These mechanisms extend the basic functionality of mutex locks and enable more complex synchronization patterns between threads.

- *Semaphore*: A semaphore restricts the number of concurrent accesses to a shared resource. It allows a certain number of threads to hold the semaphore at the same time, and blocks any additional threads until a semaphore is released.
- *Barrier*: A barrier blocks a group of threads until they have all reached the barrier point.

- *Event*: An event is a synchronization primitive that allows a thread to notify other threads that a specific condition has been met.
- *Condition variable*: A condition variable allows threads to wait for a condition to be met before continuing execution.

Events and condition variables are particularly useful techniques. Let's explore some typical scenarios where events and condition variables are applied.

Read-after-write synchronization is a common scenario in multithreading programs, where one thread needs access to certain shared resources produced by other threads. In such cases, we need to block the reader until it receives a signal from the thread of writer, indicating that the required condition has been fulfilled. Events and condition variables can make threads wait for a condition to be fulfilled or to signal the condition to other waiting threads.

Another scenario of using event and condition variables involves the coordination of data processing threads and a data prefetching thread. In this scenario, the computation threads need to wait for the I/O thread to fully read the data into a buffer before they can operate on it. The code snippet below is a prototype of this parallel computation scenario.

```
ready_for_read = threading.Event()
ready_for_read.clear() # mark the initial state: unsafe for reading
ready_for_write = threading.Event()
ready_for_write.set() # mark the initial state: safe to write

def computation_task(buffer):
    while True:
        ready_for_read.wait()
        # code to manipulate data in buffer
        ...
        ready_for_read.clear()
        # notify the I/O thread that data in buffer are consumed,
        # the buffer can be reused
        ready_for_write.set()
        # Other code which do not need to access buffer
        ...

def prefetch(buffer):
    while True:
        ready_for_write.wait()
        # code to read data
        ...
        ready_for_write.clear()
        ready_for_read.set()
```

To manage this synchronization, we can employ an event `ready_for_read` to indicate whether the data has been completely loaded into the buffer. The status of the event can be checked using the `ready_for_read.is_set()` function and cleared using the `ready_for_read.clear()` function. If the event is cleared, the `buf_ready.wait()` call will block the calling thread, preventing it from proceeding until the I/O thread calls `ready_for_read.set()` to reset the event. Moreover, the I/O thread needs to wait for the buffer to be freed for reuse. To facilitate this, we utilize another event `ready_for_write` to indicate that the data has been completely processed by the computation thread and the buffer is available for writing.

This scenario can also be handled using condition variables. A condition variable provides three operations to manage and update its status:

- The `.wait()` method indicates that a thread should “wait until the condition is met”.
- The `.notify()` method wakes up a single blocked thread when the condition is met.
- The `.notifyall()` method wakes up all blocked threads associated with the condition variable when the condition is met.

Similar to the `Event` objects, we need condition variables `ready_for_read` and `ready_for_write` to represent the status of the buffer. In addition to the two condition variables, it is necessary to introduce a flag `buf_empty` to explicitly indicate whether the buffer is empty. The prototype of the program is illustrated below.

```
mutex = threading.Lock()
ready_for_read = threading.Condition(mutex)
ready_for_write = threading.Condition(mutex)
buf_empty = True

def computation_task(buffer):
    global buf_empty
    while True:
        with mutex:
            if buf_empty:
                ready_for_read.wait()
                # code to manipulate data in buffer
                ...
                buf_empty = False
                ready_for_write.notify()
            # Other code which do not need to access buffer
            ...

def prefetch(buffer):
    global buf_empty
    while True:
```

```
with mutex:  
    if not buf_empty:  
        ready_for_write.wait()  
    # code to read data into the buffer  
    ...  
    buf_empty = False  
    ready_for_read.notify()
```

Why do we need the `buf_empty` flag to explicitly control whether to call the `ready_for_write.wait()`? What issues would the program encounter without this flag? We will leave these problems for the reader to explore.

In the Python implementation, a condition variable is tied to a mutex lock. A thread must acquire the associated mutex lock before manipulating the condition variable. This can be done explicitly or by using the context manager for the condition variables. When a thread executes the `ready_for_read.wait()` statement, the associated mutex lock is released, even though the thread remains blocked (by a different lock within the condition variable). This setup allows other threads to enter the critical section and modify the shared resources. Once the condition is met, the waiting thread will reacquire the mutex lock before proceeding with the subsequent code.

In the above example, it appears that the event and the condition variable exhibit similar functionalities and could be used interchangeably. This might lead to the question: when should we use a condition variable and when should we use an event?

If the concurrent program involves a single producer and single consumer, such as the I/O thread and the computation thread in the above example, we just need to ensure that the producer or the consumer threads wait for signals from the other. In such a scenario, both the event and the condition variable are sufficient to provide the necessary functionalities. You can use whichever technique you are most comfortable with.

However, when the dependency between the threads becomes more complex, such as in programs with multiple consumers or multiple producers, the condition variable is often the better option as it offers more precise control over the threads. For example, the `Queue` class in the Python standard library `queue` accommodates multiple producers and multiple consumers. The core functionality of the `Queue` class utilizes condition variables, as illustrated below.

```
from collections import deque  
class Queue:  
    def __init__(self, max_size):  
        self._queue = deque([])  
        self.max_size = max_size  
        self.mutex = threading.Lock()  
        self.not_full = threading.Condition(self.mutex)  
        self.not_empty = threading.Condition(self.mutex)
```

```

def put(self, item):
    with self.mutex:
        while len(self._queue) >= self.max_size:
            self.not_full.wait()

        self._queue.append(item)
        self.not_empty.notify()

def get(self):
    with self.mutex:
        while len(self._queue) == 0:
            self.not_empty.wait()

        item = self._queue.popleft()
        self.not_full.notify()
        return item

```

A `Queue` instance stores its tasks within the `_queue` attribute. Producers can add tasks via the `put` method, while consumers use the `get` method to retrieve tasks for processing. The condition variables `not_empty` and `not_full` can block the threads of producers and consumers to maintain the size of the `_queue`.

10.3 Producer-consumer model

While introducing the `mutex` in a parallel program can solve race conditions, it inevitably makes the program more complex and increases the risk of *deadlocks*. A deadlock is a situation where two or more tasks are each waiting for the other to release a resource. How can one circumvent the issue of race conditions without introducing `mutex` locks? The producer-consumer model, when used with a queue, presents an elegant solution to this problem.

Consider the `get_j` function in Section 10.2 as an example. We can adopt the producer-consumer model in this case. Some functions can act as producers, which only generate and push intermediate results to a queue. A dedicated function operates as the consumer, which only interacts with the queue to process these intermediate results. The queue is an important intermediate buffer that decouples the producers and consumers, and isolates the producers from each other. This setup allows the producers to operate at their full speed, without being blocked by `mutex` locks. The shared resource, which is the `matrix output` in this example, is only accessed by a single consumer. This approach effectively eliminates race conditions and the need for mutexes. By utilizing the `Queue` class we developed in the previous section, we can create the `get_j` program based on the producer-consumer model.

```
from concurrent.futures import ThreadPoolExecutor
import numpy as np

def compute_eri(i, n):
    # Mimic the function to compute ERIs
    np.random.seed(i)
    return np.random.rand(n,n,n)

def get_j_task(i, dm, q):
    '''The producer'''
    n = dm.shape[0]
    jmat = np.einsum('jkl,j->k1', compute_eri(i,n), dm[:,i])
    q.put(jmat)

def reduce(n, q):
    '''The consumer'''
    output = np.zeros((n,n))
    for i in range(n):
        output += q.get()
    return output

n = 200
dm = np.identity(n)
q = Queue(max_size=8)
with ThreadPoolExecutor(max_workers=16) as scheduler:
    fut = scheduler.submit(consumer, n, q)
    for _ in scheduler.map(get_j_task, range(n), [dm]*n, [q]*n):
        pass
print(fut.result().sum())
```

It should be noted that the queue in this example imposes a limit on the level of parallelization. The maximum degree of parallelization is 8, determined by the capacity of the queue object `q` and constrained by the `q.get()` and `q.put()` methods. The `ThreadPoolExecutor` is only responsible for initiating the consumers and producers. Even if 16 workers are allocated in the `ThreadPoolExecutor`, no additional tasks for `get_j_task` can be executed until there are available slots in the `q`. The consumer `reduce` can process data as soon as it becomes available, rather than waiting for all data to be generated before starting processing. Each time an item in `q` is consumed, the `q.put()` method inserts a new one to the queue. The `get_j_task` is then completed, releasing the worker for the next task.

Based on the producer-consumer model, it becomes feasible to develop more advanced parallelization frameworks. The pipeline executor is one such example, which enables the concurrent execution of a series of dependent tasks.

10.4 Pipeline executor

Let's consider a program that is processed by a series of independent operations `op0`, `op1`, ... `op_last`. By appropriately connecting these functions, a streaming processing system can be formed, similar to the Unix pipeline. When a batch of tasks move through the stream (Fig. 10.1), the system executes a series of intermediate operations and produces a list of results. This process is essentially the same as the expression

```
[op_last(... op2(op1(op0(task))))... for task in tasks]
```

To parallelize these tasks, one can utilize the `ThreadPoolExecutor`. However, if there are numerous intermediate operations but only a limited number of tasks, the simple `ThreadPoolExecutor` parallelization over tasks may not be able to fully harness the parallel capability.

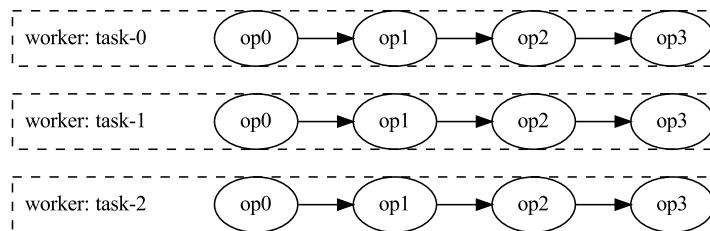


FIGURE 10.1

Concurrent tasks executed with multiple intermediate functions.

We can design a pipeline executor to stream the operations and tasks in parallel. A pipeline is a chain of producers and consumers. Each operation in the pipeline can act as a consumer, fetching tasks from the task queue for execution. These operations can also serve as producers, passing their output into the task queue for subsequent consumers to process. Basically, the `pipeline` function provides the following functionality:

```
from concurrent.futures import ThreadPoolExecutor

def pipeline(ops, tasks):
    with ThreadPoolExecutor() as q:
        def proc0():
            return [queue.submit(ops[0], task) for task in tasks]

        def proc1():
            return [queue.submit(ops[1], task.result()) for task in proc0()]

        def proc2():
            return [queue.submit(ops[2], task.result()) for task in proc1()]

        def proc3():
            return [queue.submit(ops[3], task.result()) for task in proc2()]

    return [proc3().result() for _ in range(len(tasks))]
```

```

        return [queue.submit(ops[2], task.result()) for task in proc1()]
    ]

    ...

    return [future.result() for future in proc_last()]

```

However, this basic version might not be able to efficiently parallelize the task streaming execution. Resource utilization could be low, and tasks in a sequence (e.g., `op0 -> op1 -> op2 ...`) may not be executed continuously. Instead, the `proc?` functions often stay in a pending status, waiting for the completion of the preceding `proc` functions, which leads to delays in execution.

Let's inspect the process within this `pipeline` function to find out why the parallel execution is blocked. The `proc0` function iterates over the list and submits all tasks. No blocking occurs in this step. It creates a list of `Future` objects, which is then passed to the `proc1` function. In `proc1`, the method `task.result()` is invoked to acquire the actual outcome of the operation `ops[0]`. If any of the `Future` objects are not finished, it blocks the list comprehension, thus stalling the return statement of the `proc1` function. The `proc2` function relies on the output of `proc1` and it cannot start scheduling any tasks for the operation `ops[2]` until `proc1` has completed the list comprehension. In other words, the `proc2` function will start submitting tasks only after all tasks for `ops[0]` (created by `proc0`) have been completed. Even if the upstream dependencies for certain tasks of `ops[2]` are met, these tasks cannot be submitted due to the blocking in `proc1`. Similarly, the subsequent operations in the `ops` list cannot be executed promptly due to the same blocking issue.

In the `pipeline` executor, directly calling `task.result()` should be avoided as it can block the pipeline. Instead, we can introduce a closure to wrap the `task.result()` method, deferring the blocking operation until its value is referenced during runtime. This approach is somewhat similar to the technique of lazy evaluation.

```

from concurrent.futures import ThreadPoolExecutor, Future
def _ensure(task):
    if isinstance(task, Future):
        return task.result()
    else:
        return task

def scheduler(queue, fn, tasks):
    def fn_with_future(task):
        return fn(_ensure(task))
    return [queue.submit(fn_with_future, task) for task in tasks]

def pipeline(ops, tasks):
    with ThreadPoolExecutor() as queue:
        def proc0():

```

```

        return scheduler(queue, ops[0], tasks)

    def proc1():
        return scheduler(queue, ops[1], proc0())

    def proc2():
        return scheduler(queue, ops[2], proc1())

    ...

    return [future.result() for future in proc_last()]

```

With this enhancement, the streaming processing can rapidly submit all tasks and fill the task queue as:

```

ops[0](tasks[0]), ops[0](tasks[1]), ops[0](tasks[2]), ...
ops[1](ops[0](tasks[0])), ops[1](ops[0](tasks[1])), ...

```

If the parallel capacity is limited, for example, allowing only 4 tasks to be executed simultaneously, tasks that are queued later will remain in a pending state due to resource constraints, even if all upstream dependencies have been satisfied. As a result, downstream tasks cannot be executed promptly, leading to delays in consuming the results produced by upstream tasks. The upstream results have to be stored in memory for an extended period, leading to inefficient memory usage and decreased memory management efficiency.

To address this problem, we can implement multiple queues for streaming execution. By assigning different consumers to different queues, we can leverage the independent task scheduling of each queue. Once the upstream dependencies of a task are resolved, the task can be immediately scheduled for execution by the queue associated with the consumer. The following version of the `pipeline` function illustrates this idea.

```

def pipeline(ops, tasks):
    with ThreadPoolExecutor() as queue1:
        def proc0():
            return scheduler(queue1, ops[0], tasks)

    with ThreadPoolExecutor() as queue2:
        def proc1():
            return scheduler(queue2, ops[1], proc0())

    with ThreadPoolExecutor() as queue3:
        def proc2():
            return scheduler(queue3, ops[2], proc1())

    with ThreadPoolExecutor() as ...:

```

```
...  
  
return [future.result() for future in proc_last()]
```

Please note that the `with` statements are nested in this example. This is a direct outcome of using the `ThreadPoolExecutor`. When the scope of a `with` context is exited, the `ThreadPoolExecutor` enforces a synchronization operation (equivalent to calling the `.result()` method) on all submitted tasks. Without nested `with` contexts, the synchronization operation applied to the `proc0` function would block the execution of `proc1` and any subsequent functions.

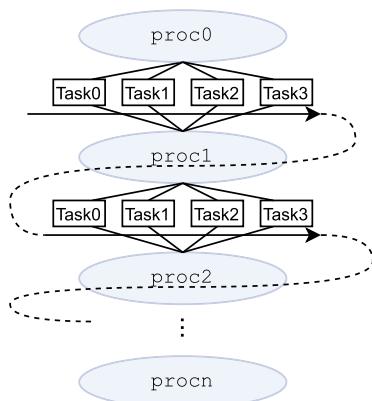


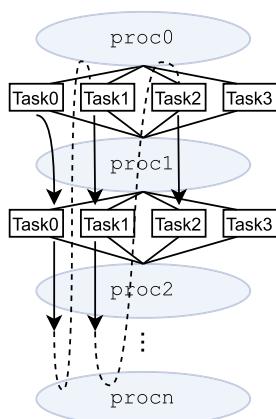
FIGURE 10.2

Traversal of tasks using breadth-first search.

The scheduler previously demonstrated prioritizes the submission of any available tasks before proceeding to the next operation in the `ops` list. If we view each operation in `ops` as a node in a tree, the previous scheduler operates in a manner similar to a *breadth-first search* traversal of the tree (Fig. 10.2). Hence, we refer to it as the `bf_scheduler`. Alternatively, the traversal method of the task scheduler can be modified to resemble a *depth-first search* (DFS) pattern (Fig. 10.3). Here, we can create a `df_scheduler` which leverages the lazy evaluation characteristic of generators to change the task submission order.

```
def df_scheduler(queue, fn, tasks, args):  
    def fn_with_future(task):  
        return fn(_ensure(task), *args)  
    for task in tasks:  
        yield queue.submit(fn_with_future, task)
```

When the `df_scheduler` is applied to the `pipeline` function, the downstream tasks will recursively reference the elements of the generator created by each `proc` function

**FIGURE 10.3**

Traversal of tasks using depth-first search.

until they reach the root tasks defined in the `proc0` function. Each element of the generator is created only when referenced, and the remaining elements are not created in advance. As a result, for each element in the input variable `tasks` of the `pipeline` function, the operations specified in the `ops` list will be sequentially submitted to the task queue, following the dependency chain.

For instance, let's consider a scenario where the `pipeline` function consists of only three functions: `proc0`, `proc1`, and `proc2`, each utilizing the `df_scheduler` to submit tasks.

```
def pipeline(ops, tasks):
    with ThreadPoolExecutor() as queue1:
        def proc0():
            return df_scheduler(queue1, ops[0], tasks)

        with ThreadPoolExecutor() as queue2:
            def proc1():
                return df_scheduler(queue2, ops[1], proc0())

            with ThreadPoolExecutor() as queue3:
                def proc2():
                    return df_scheduler(queue3, ops[2], proc1())

    return [future.result() for future in proc2()]
```

In `proc2`, accessing an element from the `tasks` generator triggers the `yield` statement in `proc1`. This action, in turn, draws an element from the `tasks` generator created by `proc0`. When the generator of `proc0` is accessed, the task submission command

`queue.submit(fn_with_future, task)` is executed. This command appends a job to the task queue and passes a `Future` object to `proc1` through the generator. Subsequently, `proc1` receives the `Future` object and utilizes it to submit a task. The resultant `Future` object is then passed on to `proc2`. After processing the received `Future` object, `proc2` proceeds to the next element of the tasks generator, triggering the above cycle once again. `proc0`, `proc1`, and `proc2` will generate new tasks and sequentially add them to the task queue. Eventually, the task queue may be populated as follows:

```
ops[0](tasks[0]), ops[1](Future(ops[0])), ops[2](Future(ops[1])),
ops[0](tasks[1]), ops[1](Future(ops[0])), ops[2](Future(ops[1])), ...
```

The `df_scheduler` can actively digest the results of upstream tasks, solving the issue of excessive memory usage that might occur with the `bf_scheduler`. However, `df_scheduler` may cause certain running tasks being stuck in an inactive state, as they await the completion of upstream tasks. These tasks occupy thread workers but are unable fully utilize the computational resources, resulting in a waste of computational power.

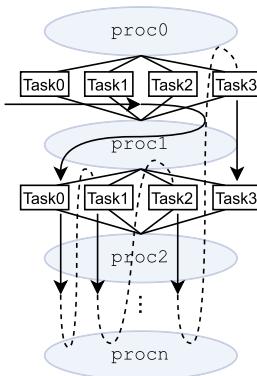


FIGURE 10.4

Traversal of tasks using a hybrid of breadth-first and depth-first search.

To strike a balance between parallel efficiency and memory utilization, we can design new schedulers that implement different strategies. For instance, the `df_warmup_scheduler` presented below employs a hybrid approach (Fig. 10.4).

```
def df_warmup_scheduler(queue, fn, tasks, args, max_workers=8):
    '''Launches several tasks at beginning then fills task queue like
    df_scheduler'''
    def fn_with_future(task):
        return fn(_ensure(task), *args)

    future_buf = deque()
```

```

for task in tasks:
    future_buf.append(queue.submit(fn_with_future, task))
    if len(future_buf) > max_workers:
        yield future_buf.popleft()
while future_buf:
    yield future_buf.popleft()

```

It initially submits a subset of tasks like a `bf_scheduler`, then shifts to the `df_scheduler` mode for the remaining tasks. This scheduler is designed to quickly occupy the available workers in the thread executor at beginning with the non-blocking tasks. As initial tasks are completed, it then gradually refills the queue in the style of the depth-first scheduler. The underlying assumption is that by immediately processing the initial tasks, the upstream dependencies required by subsequent tasks will likely be satisfied by the time they are scheduled. This method aims to minimize the likelihood of tasks becoming stuck in inactive states. The resulting task queue may have the following population

```

ops[0](tasks[0]), ops[0](tasks[1]), ...,
ops[0](tasks[7]), ops[1](Future(ops[0])), ops[2](Future(ops[1])),
ops[0](tasks[8]), ops[1](Future(ops[0])), ...

```

Based on the `df_warmup_scheduler`, we can further develop a `dynamic_scheduler` that launches only those tasks ready for immediate execution. Subsequent tasks are deferred and submitted only after the completion of upstream tasks. The task queue is not completely filled by initial tasks. To facilitate launching of tasks upon the completion of upstream tasks, the `LaunchInFuture` class is introduced. This class offers a `submit` method, which allows a task (represented as a `Future` object) to be executed as a callback upon the completion of another task.

```

class LaunchInFuture(Future):
    def __init__(self, queue, fn):
        self.queue = queue
        self.fn = fn
        super().__init__()

    def submit(self, task): # (1)
        future = self.queue.submit(self.fn, task) # (2)
        future.add_done_callback(self.pass_result) # (3)

    def pass_result(self, value: Future): # (4)
        self.set_result(value.result())

def dynamic_scheduler(queue, fn, tasks, args, max_workers=8):
    '''Dynamic scheduler submits tasks only if they are ready to proceed'''
    def fn_with_future(task):
        return fn(_ensure(task), *args)

```

```
future_buf = deque()
for task in tasks:
    # Block the submission to prevent the task queue being filled with
    # too many operations of the same type
    if len(future_buf) >= max_workers:
        future_buf.popleft().result()
    if not isinstance(task, Future):
        future = queue.submit(fn_with_future, task)
    else:
        future = LaunchInFuture(queue, fn_with_future)           # (5)
        task.add_done_callback(future.submit)                      # (6)
    future_buf.append(future)
yield future
```

In line (6), `add_done_callback` appends a subsequent operation to the current task. The subsequent operation, `future.submit` which is defined in line (1), will be submitted after the current task is completed. The argument `task` in line (1) refers to the `task` object mentioned in line (6). To retrieve the results of the task launched in line (2), another callback is registered in line (3), which transfers the results back to the instance `future` created in line (5). In the callback function `pass_result` in line (4), the argument `value` is actually the instance `future` created in line (2).

Finally, to enhance the readability and the usability of the `pipeline` function, we refine the `pipeline` function and transform it into a recursive style. The refactored version of the `pipeline` function now supports general streaming processing computations that incorporate various task schedulers.

```
def pipeline(ops, tasks, scheduler=dynamic_scheduler):
    ...
    Streams operations and tasks
    ...
    queue, fn, args = ops[0]
    if queue is None:
        with ThreadPoolExecutor() as queue:
            return pipeline([(queue, fn, args)], *ops[1:], tasks, scheduler)

    if len(ops) == 1:
        # Call list() to traverse through the entire scheduler generator
        # which
        # will launch all tasks and their upstream tasks.
        return [_ensure(f) for f in list(scheduler(queue, fn, tasks, args))]
    ]

    return pipeline(ops[1:], scheduler(queue, fn, tasks, args), scheduler)
```

In Chapter 14, we will use the Full Configuration Interaction (FCI) program as an example to demonstrate how to use the pipeline executor to parallelize complex computational tasks.

10.5 Asynchronous program

In a synchronous program, tasks are executed one after another. The program flow has to wait for the completion of one operation before proceeding to the next. By using the `Future` object to reference the result of a task, a program can carry out operations asynchronously, as shown in the examples of the `ThreadPoolExecutor` in the previous sections. Strictly speaking, this is a technique of non-blocking execution, which is different from asynchronous programming. However, since the two concepts are often closely related, we will not differentiate between them in this book. The book *Parallel Programming with Python* by Jan Palach covers more detailed discussions on these concepts.

10.5.1 Similarity between asynchronous programming and lazy evaluation

The asynchronous computation approach is closely related to the lazy evaluation approach we discussed in Chapter 9. In lazy evaluation, we explicitly invoke the `defer` function to create a `Promise` object, which represents a result that is not yet evaluated. The `Promise` object plays a similar role to the `Future` object. By transforming the `defer` function into a decorator, as demonstrated below, we can unify the structure of the lazy evaluation and the asynchronous computation code.

```
from pychem_book.chap09.lazy_eval import Promise
executor = concurrent.futures.ThreadPoolExecutor()
lazy_mode = True

def defer(f):
    @functools.wraps(f)
    def deferred_f(args):
        if lazy_mode:
            return Promise(f, args)
        else:
            return executor.submit(f, args)
    return deferred_f
```

It is worth mentioning that we employ the `ThreadPoolExecutor` in the above example to illustrate the functionality. Other executors, such as the `dask.distributed` function, can also be utilized to achieve asynchronous computation.

We can use the `defer` decorator for the `get_j` function. The program can seamlessly switch between lazy evaluation mode and asynchronous mode, depending on the configuration of the global flag `lazy_mode`.

```
@defer
def compute_eri(i, n):
    ...
    return eri

@defer
def get_j_task(i, dm):
    n = dm.shape[0]
    eri = compute_eri(i, n).result()
    return np.einsum('jk1,j->k1', eri, dm[:,i])

@defer
def get_j(dm):
    n = dm.shape[0]
    output = np.zeros((n,n))
    for i in range(n):
        output += get_j_task(i, dm).result()
    return output

get_j(dm).result()
```

10.5.2 Asynchronous program with coroutines and `asyncio`

Asynchronous programming in Python is often associated with concurrent coding that utilizes coroutines and the `asyncio` module [4]. This method involves the use of the `async` and `await` keywords, along with an event loop to manage task execution.

Coroutines are functions that are defined using the `async` keyword in the function definition. A coroutine uses the `await` keyword to pause its execution and yield the computation resources to other coroutines. Later, the event loop can resume the paused coroutine and continue its execution. Asynchronous programming with coroutines is an elegant and effective solution for executing multiple concurrent tasks. It offers several advantages:

- Low overhead in task switching. To achieve concurrency, coroutines yield to others, rather than being suspended and resumed by expensive thread context switches.
- Simplicity. The code structure of asynchronous code remains very similar to that of conventional synchronous code.
- No race conditions. Coroutines are scheduled by an event loop running in a single thread. Resources are accessed by only one coroutine at any time.

A primary limitation of the coroutine technique is the lack of support for parallel computation with multiple CPU cores. This limitation makes coroutine unsuitable for accelerating computation-intensive quantum chemistry applications. As such, we do not intend to discuss this technique in depth. We will briefly demonstrate the coroutine version of the `get_j` function as a complement to the discussions on the threading-level asynchronous programming. If you are interested in this technique, more explanations and examples can be found in the books *Python Parallel Programming Cookbook* and *Fluent python*.

Starting from the `defer`-decorated `get_j` program discussed in the previous section, we just need to substitute the `defer` decorator with the `async` keyword and replace the `.result()` call to the statement with `await`. This modification yields the coroutine version of `get_j` as follows:

```
async def compute_eri(i, n):
    ...
    return eri

async def get_j_task(i, dm):
    n = dm.shape[0]
    eri = await compute_eri(i, n)
    await asyncio.sleep(0.5)
    return np.einsum('jkl,j->kl', eri, dm[:,i])

async def get_j(dm):
    n = dm.shape[0]
    output = np.zeros((n,n))
    fs = [asyncio.ensure_future(get_j_task(i, dm)) for i in range(n)]
    for fut in fs:
        output += await fut
    return output

n = 200
dm = np.identity(n)
co = get_j(dm)
output = asyncio.get_event_loop().run_until_complete(co)
print(output.sum())
```

In the `get_j_task` function, the statement `asyncio.sleep(0.5)` temporarily pauses the execution of the individual tasks. Despite this intentional delay, there is no noticeable slowdown, which proofs that the code is executed concurrently. When running this asynchronous program, you might notice that only one CPU core is engaged. The executing time is comparable to that of a synchronous program running in a single thread without any pauses.

10.6 Multiprocessing

Multiprocessing parallelization is a popular strategy for parallel computation in Python, as it overcomes the limitations of the GIL in threading parallelization. The multiprocessing parallelization can be achieved with the Python standard library `multiprocessing`.

The `multiprocessing` module plays a similar role to the `threading` module. It provides the fundamental tools for process-based parallelization. For instance, the `multiprocessing` module offers a `Process` class for spawning child processes. The usage of `multiprocessing.Process` is analogous to the `threading.Thread` class. Child processes can be initiated with the `start` method and synchronized using the `join` method.

The `multiprocessing` module also encounters the task management issues similar to those in the `threading` module. For example, if an error occurs in the child processes, the error cannot be handled by the parent process. To enhance the capabilities of multiprocessing parallelization, the `concurrent.futures` module also introduces the `ProcessPoolExecutor` for managing the process resources and tasks. The `ProcessPoolExecutor` and `ThreadPoolExecutor` provide similar APIs, facilitating the adaptation of code from `ThreadPoolExecutor` to `ProcessPoolExecutor`.

Given the similarities between the `multiprocessing` and `threading` modules, as well as their respective pool executors, we will omit the basic usage of these multiprocessing parallelization tools. The Python official documentation offers a comprehensive guideline on multiprocessing programming [5]. Our focus in this section will be on data communication within the multiprocessing context.

The efficiency of multiprocessing parallel computation is significantly influenced by the Inter-Process Communication (IPC) method employed by each library. In the `multiprocessing` module, the default methods are anonymous `pipe` for one-way communication and `socket` for duplex communication. In both methods, data are serialized and deserialized using `pickle` during the data transfer process. It is important to note that not all Python data types can be transferred between processes. We must ensure that the data intended for transfer are compatible with `pickle` serialization, as discussed in Chapter 6. Besides `pipe` and `socket`, shared memory and memory-mapped files are also commonly used for data transfer. However, the two techniques support fewer data types.

10.6.1 Data communication in `multiprocessing.Process`

Let's first examine the transfer of function arguments between the parent process and the child processes. When the child processes are created using the `multiprocessing.Process` class, there is no need for IPC to transfer function arguments from the parent process to the child process.

The operating system employs the copy-on-write (COW) technique to enable the data transfer from the parent process to child processes. This technique allows the parent process and the child processes to share the memory space in a read-only mode. If a process needs to modify the shared space, a private copy of the relevant

memory pages is created in its private memory space. Therefore, upon the creation of the child process, it can access all data owned by the parent process, including the function arguments, without any overhead.

COW effectively solves the problem of transferring function arguments to child processes. The `multiprocessing.Process` method is as efficient as multithreading parallelism regarding the data transfer of function arguments. Additionally, COW allows the passing of any arguments to child processes, including those objects that are not pickleable. For example, we can pass an iterator to the child processes. However, each child process consumes its own iterator without interfering with each other. This is different to the case in multithreading, where the state of the iterator is shared among different threads.

```
In [1]: import time
        from multiprocessing import Process
        from threading import Thread
        def f(thread_id, tasks):
            for task in tasks:
                print(thread_id, task)
                time.sleep(.1)

In [2]: iters = enumerate(range(5))
        ps = [Thread(target=f, args=(i, iters)) for i in range(3)]
        [p.start() for p in ps]
        [p.join() for p in ps]
Out[2]:
0 (0, 0)
1 (1, 1)
2 (2, 2)
0 (3, 3)
1 (4, 4)

In [3]: iters = enumerate(range(5))
        ps = [Process(target=f, args=(i, iters)) for i in range(3)]
        [p.start() for p in ps]
        [p.join() for p in ps]
Out[3]:
0 (0, 0)
1 (0, 0)
2 (0, 0)
0 (1, 1)
1 (1, 1)
2 (1, 1)
...
```

How would the parent process get the output of the child processes? Similar to the child threads created by `threading.Thread`, the output of functions in the child processes cannot be directly returned to the parent process. In the case of multithreading, functions in child threads can modify input arguments, thus transferring information back to the main thread. Although not recommended in general, this method enables data exchange between the main thread and the child threads. In multiprocessing, when the function within a child process modifies the arguments, a local copy is produced due to the COW mechanism. Consequently, the parent process is unable to access the modifications made to the arguments by the child process.

To enable data exchange between the parent process and child processes, the `multiprocessing` module offers the `Manager` class. This class mimics several Python data structures, such as lists and dictionaries. When a child process makes changes, `Manager` can send the modification back to the parent process, thereby enabling data sharing among processes. The `Manager` handles data transfer through sockets or pipes, which automate the data transfer between a server-client pair. To utilize the `Manager` data transfer service, one needs to properly initialize and terminate the `Manager`, for example, by employing a `with` context [6].

```
from multiprocessing import Process, Manager

def append(lst, elem):
    lst.append(elem)

with Manager() as manager:
    l = manager.list()
    p = Process(target=append, args=(l, 'a'))
    p.start()
    p.join()
    l = list(l) # convert to regular list
assert len(l) == 1
```

The `Manager.list` provides an API comparable to the Python standard list. However, it should not be simply considered as a direct replacement for the standard list. When used with `multiprocessing.Process`, we need to realize the additional requirements:

- `Manager.list` is only accessible within the `with` context of the `Manager`. To access the list beyond this scope, it must be converted to a standard list object, as shown in the above example.
- Elements of `Manager.list` must be picklable. Communication within `Manager.list` occurs through an anonymous pipe or a socket pair. Both methods require the elements to be picklable.

In quantum chemistry programs, there is frequently a need to distribute large volumes of data, particularly NumPy arrays, across processes. In this circumstance, `multithreading.SharedMemory`, `multithreading.Array` and `memmap` files are viable options for data sharing. We can use these tools to create a data buffer for a NumPy

array, which can then be passed to the child processes along with other function arguments. When the child processes alter these NumPy arrays, the modifications are directly written to the underlying shared memory or the corresponding memory mapped files. The following example demonstrates how to implement shared NumPy arrays using these techniques.

```
from multiprocessing import Array, Process
from multiprocessing.shared_memory import SharedMemory

def get_j_task(k, dm, output):
    n = dm.shape[0]
    output[k] = np.einsum('ijl,ji->l', compute_eri(k, n), dm)

def get_j(dm, output):
    output[:] = 0.
    ps = [Process(target=get_j_task, args=(k, dm, output))
          for k in range(n)]
    [p.start() for p in ps]
    [p.join() for p in ps]
    return output

if __name__ == '__main__':
    n = 50
    dm = np.identity(n)

    shm = Array(dm.dtype.char, dm.size)
    output = np.ndarray(dtype=dm.dtype, shape=dm.shape,
                        buffer=shm.get_obj())
    print(get_j(dm, output).sum())

    shm = SharedMemory(create=True, size=dm.nbytes)
    shm.unlink()
    output = np.ndarray(dtype=dm.dtype, shape=dm.shape, buffer=shm.buf)
    print(get_j(dm, output).sum())

    output = np.memmap('/dev/shm/output', dtype=dm.dtype, shape=dm.shape,
                      mode='w+')
    print(get_j(dm, output).sum())
```

10.6.2 Data communication in ProcessPoolExecutor

`ProcessPoolExecutor` abstracts away the technical details of inter-process data transfer. Its APIs are similar to those in `ThreadPoolExecutor`, one simply needs to call the `submit` method to initiate parallel tasks. The `submit` method yields a `Future` ob-

ject, representing the task in progress. Both the outcomes and any exceptions can be retrieved via the `Future.result()` method.

Despite the API similarities between `ProcessPoolExecutor` and `ThreadPoolExecutor`, their relationships to the underlying threading and multiprocessing modules are distinct. Scenarios utilizing `threading.Thread` can be seamlessly transitioned to `ThreadPoolExecutor`, without any modifications. However, `ProcessPoolExecutor` cannot fully replace `multiprocessing.Process` due to the COW mechanism, which grants several unique features to `multiprocessing.Process`.

The `ProcessPoolExecutor` utilizes a task queue to dispatch data to child processes. The task queue in `ProcessPoolExecutor` relies on the anonymous pipe or socket pairs to send and receive data. All Python objects are converted with the `pickle` serialization and deserialization when transferred by the task queue. As a result, `ProcessPoolExecutor` is not suitable for parallelizing an arbitrary function. We need to ensure the function, its arguments, and the result are picklable.

Due to the constraints of `pickle` serialization, some `multiprocessing.Process` parallel tasks cannot be directly executed by the `ProcessPoolExecutor`. The incompatible tasks for the `ProcessPoolExecutor` can be classified into three categories:

- Tasks that are completely incompatible. For example, `ProcessPoolExecutor` does not work when the task involves a closure, or the arguments contain generator objects. In such cases, the algorithm has to be redesigned.
- Tasks for which alternatives are available. For example, if the function arguments contain lists, dictionaries, Locks, or Queues, they can be replaced with equivalents provided by `multiprocessing.Manager`.
- Tasks that can be executed by the `ProcessPoolExecutor` may introduce potential bugs. For example, shared NumPy arrays that depend on shared memory or `memmap` files are picklable. When a shared-memory NumPy array is passed to the `ProcessPoolExecutor` workers, the contents of the NumPy array is correctly distributed, but the capability for data sharing is lost. Data written to these arrays in child processes will not be reflected in the arrays in the parent process.

Enabling shared memory for NumPy arrays in the `ProcessPoolExecutor` requires extra coding. Since the contents of the shared-memory NumPy arrays can be accessed by all processes, the only information that needs to be transferred is the filename, the datatype, and the shape of the array. This information can be broadcast using the default IPC methods provided by the `ProcessPoolExecutor`. For example, the data sharing via the `SharedMemory` object can be implemented as

```
from multiprocessing.shared_memory import SharedMemory

def get_j_task(k, dm, output_attr):
    filename, dtype, shape = output_attr
    shm = SharedMemory(filename, create=False)
    output = np.ndarray(dtype=dtype, shape=shape, buffer=shm.buf)
    n = dm.shape[0]
```

```

output[k] = np.einsum('ijl,ji->l', compute_eri(k, n), dm)

def get_j(dm):
    n = dm.shape[0]
    shm = SharedMemory(create=True, size=dm.nbytes)
    output = np.ndarray(dtype=dm.dtype, shape=dm.shape, buffer=shm.buf)
    output[:] = 0.
    output_attr = (shm.name, output.dtype, output.shape)
    with ProcessPoolExecutor(max_workers=4) as executor:
        futures = [executor.submit(get_j_task, i, dm, output_attr)
                   for i in range(n)]
        [f.result() for f in futures]
    shm.unlink()
    return output.copy()

```

Thanks to the helper function `np.memmap`, the data sharing code using `memmap` files is simple

```

def get_j_task(k, dm, output_attr):
    filename, dtype, shape = output_attr
    output = np.memmap(filename, shape=shape, dtype=dtype, mode='r+')
    n = dm.shape[0]
    output[k] = np.einsum('ijl,ji->l', compute_eri(k, n), dm)

def get_j(dm):
    n = dm.shape[0]
    output = np.memmap('/dev/shm/output', dtype=dm.dtype,
                      shape=dm.shape, mode='w+')
    output[:] = 0.
    output_attr = ('/dev/shm/output', output.dtype, output.shape)
    with ProcessPoolExecutor(max_workers=4) as executor:
        futures = [executor.submit(get_j_task, i, dm, output_attr)
                   for i in range(n)]
        [f.result() for f in futures]
    return output

```

Please note that this approach cannot be applied to NumPy arrays that utilizes the `multiprocessing.Array` buffer because there is no filename associated with the shared `multiprocessing.Array` object.

When comparing `ThreadPoolExecutor` and `multiprocessing.Process` for parallelism, the `ProcessPoolExecutor` often incurs higher initialization overhead. If functions take large NumPy arrays as arguments, employing the shared memory technique can reduce the overhead. However, even with the use of shared memory for transferring large arrays, data sharing with `ProcessPoolExecutor` remains less efficient

than using `multiprocessing.Process` parallelism. We will analyze this further in Section 10.7.

10.6.3 Locks in the multiprocessing program

In Section 10.2, we explored the use of locks and related synchronization primitives for thread synchronization. Since child threads share the same memory space, threading locks are naturally applicable to both `threading.Thread` and `ThreadPoolExecutor` parallelism. However, the situation changes with multiprocessing parallelism, where each process operates its own memory space. This raises the question: How can we use locks to protect shared resources?

When child processes are spawned using `multiprocessing.Process`, we can still utilize locks and synchronization primitives, similar to their applications in threading parallelization. The key difference is that one must use the synchronization primitives provided by the `multiprocessing` module. For instance, the `multiprocessing.Process` variant of the `get_j` function in Section 10.2 can be implemented as follows.

```
from multiprocessing import Lock, Process
import numpy as np

def compute_eri(i, n):
    # Mimic the function to compute ERIs
    np.random.seed(i)
    return np.random.rand(n,n,n)

def get_j_task(i, dm, output, lock):
    n = dm.shape[0]
    jmat = np.einsum('jkl,j->kl', compute_eri(i, n), dm[:,i])
    with lock:
        output += jmat

def get_j(dm):
    output = np.memmap('/dev/shm/output', dtype=dm.dtype,
                      shape=dm.shape, mode='w+')
    output[:] = 0.
    lock = Lock()
    ps = [Process(target=get_j_task, args=(i, dm, output, lock))
          for i in range(n)]
    [p.start() for p in ps]
    [p.join() for p in ps]
    return output

if __name__ == '__main__':
    n = 50
    dm = np.identity(n)
```

```
print(get_j(dm).sum())
```

The `multiprocessing.Lock` object is created within the shared memory space and distributed to each child process upon its creation. When a process acquires or releases the lock, it modifies the state of the shared memory. These changes are visible to all child processes.

Please note that `multiprocessing.Lock` objects cannot be serialized using `pickle`. They cannot be passed as arguments when using the `ProcessPoolExecutor`. To overcome this limitation, we can use the alternative provided by the `Manager` class. The `Lock` instance of `Manager` class can transfer the lock status between processes in the background. We can modify the above example to utilize the `ProcessPoolExecutor`:

```
def get_j_task(i, dm, output_attr):
    filename, dtype, shape = output_attr
    output = np.memmap(filename, shape=shape, dtype=dtype, mode='r+')
    n = dm.shape[0]
    jmat = np.einsum('jkl,j->kl', compute_eri(i, n), dm[:, i])
    with lock:
        output += jmat

def get_j(dm):
    output = np.memmap('/dev/shm/output', dtype=dm.dtype,
                       shape=dm.shape, mode='w+')
    with Manager() as mgr, ProcessPoolExecutor(max_workers=4) as scheduler:
        lock = mgr.Lock()
        output_attr = ('/dev/shm/output', output.dtype, output.shape, lock)
        for _ in scheduler.map(
            get_j_task, range(n), [dm]*n, [output_attr]*n):
            pass
    return output
```

10.7 Inter-process communication

Inter-Process communication (IPC) encompasses a range of mechanisms that enable processes to exchange information [7]. In the context of multiprocessing computation in Python, several IPC mechanisms are commonly employed:

- Pipes. Pipes provide a one-way communication channel between two processes. Python supports two types of pipes:
 - Anonymous pipes, which are created using the `os.pipe` function;
 - Named pipes, which are established with the `os.mkfifo` function.
- Sockets. Sockets support bidirectional communication. Pipes and sockets form the foundation of the communication for functions and methods implemented in

`multiprocessing` module and the `ProcessPoolExecutor` class. The Python standard library `socket` offers a low-level interface for socket communication. However, direct invocation of the `socket` module in Python parallel programming is uncommon. Various communication tools are built on top of the `socket` module.

- Signals. Signals are primarily utilized to interrupt a running program. Common signals include `SIGTERM` (issued by the `kill` command) and `SIGKILL` (triggered by `kill -9`). Signals cannot be used for data transfer.
- Memory Mapped Files. This technique maps a file directly to the memory, allowing for the manipulation of the file as if it were memory itself.
- Shared Memory. Shared memory is a segment in memory that can be accessed by multiple processes. On Linux systems, this memory is mounted to the `/dev/shm` directory.
- Message Passing. This technique is commonly found for communication among distributed workers. This category includes various techniques, such as MPI (Message Passing Interface) and RPC (Remote Procedure Call). RPC was previously discussed in Chapter 6. We will explore MPI in more detail in Section 10.9.
- Message Queue: This technique enables the indirect communication between processes. Python does not have a native interface to access the message queue provided by the operating system [8]. This functionality is usually implemented using third-party message queues, such as RabbitMQ and ZMQ.

Different IPC mechanisms exhibit different features and performance. Shared memory and memory-mapped files (`memmap`) are more efficient than pipes and sockets. In the following, we will evaluate the performance of these IPC techniques in transferring NumPy arrays. This will provide us a concrete idea of the communication overhead in parallel programs that utilize the `multiprocessing` module and the `ProcessPoolExecutor` class.

To perform these benchmarks, we first develop some helper functions in the following:

```
import multiprocessing
from contextlib import contextmanager
def timing(f):
    def timing_f(*args):
        t0 = time.perf_counter()
        f(*args)
        t1 = time.perf_counter()
        return t1 - t0
    return timing_f

def communicate(server, client, channel, args):
    if isinstance(channel, tuple):
        r, w = channel
    else:
        r = w = channel
```

```

proc = multiprocessing.Process(target=client, args=(r, *args))
proc.start()
server(w, *args)
proc.join()

class Guard:
    def __init__(self):
        self.ready_for_write = multiprocessing.Event()
        self.ready_for_write.set()
        self.ready_for_read = multiprocessing.Event()
        self.ready_for_read.clear()

    @contextmanager
    def for_send(self):
        self.ready_for_write.wait()
        yield
        self.ready_for_write.clear()
        self.ready_for_read.set()

    @contextmanager
    def for_recv(self):
        self.ready_for_read.wait()
        yield
        self.ready_for_read.clear()
        self.ready_for_write.set()

```

The `timing` function is a decorator designed to measure the execution time of a function. The `communicate` function sets up a pair of processes, acting as the server and client, which utilize the specified `channel` for data exchange. To eliminate the risk of race conditions when accessing shared memory and `memmap`, we employ the synchronization algorithm described in Section 10.2 within the context managers of the `Guard` class. This class ensures that the server and client access the shared resources in an alternating manner.

There are two methods for transferring NumPy arrays through anonymous pipes. One method is to send the data buffer along with the shape and data types of the NumPy array, and then rebuild the array on the client. The second method employs pickle serialization to encode the NumPy array object, which is then sent and deserialized on the client side. No matter which method is chosen, data transfer cannot be accomplished in a single pass due to the limitations of the pipe's buffer size. Therefore, the client needs to read the data in multiple passes and then concatenate them in a buffer. There are several ways to allocate the buffer. We can utilize the in-memory I/O technique (provided by the `io.BytesIO` class, as discussed in Chapter 6) to manage the buffer. This approach is also utilized by the `multiprocessing` module. The program below illustrates the first method for transferring NumPy arrays.

```
import io
def pipe_recv(fd, size, count):
    for _ in range(count):
        buf = io.BytesIO()
        remaining = size
        while remaining > 0:
            dat = os.read(fd, remaining)
            buf.write(dat)
            remaining -= len(dat)
        buf = np.ndarray(shape=(size,), dtype='c', buffer=buf.getvalue())

def pipe_send(fd, size, count):
    a = np.ones(size, dtype='c')
    buf = a.data
    for _ in range(count):
        start = 0
        while start < size:
            m = os.write(fd, buf[start:])
            start += m

@timing
def pipe_transfer(size, count):
    r, w = os.pipe()
    communicate(pipe_send, pipe_recv, (r, w), (size, count))
    os.close(r)
    os.close(w)
```

The second method adds the code of data serialization on top of the above implementation. This program simulates the actions performed by the `ProcessPoolExecutor`. As we will find in the throughput benchmark, the overhead introduced by serialization is significant, nearly equal to the duration required for the data transfer.

```
def pipe_pickle_recv(fd, size, count):
    for _ in range(count):
        buf = io.BytesIO()
        remaining = int.from_bytes(os.read(fd, 8), sys.byteorder)
        while remaining > 0:
            dat = os.read(fd, remaining)
            buf.write(dat)
            remaining -= len(dat)
        pickle.loads(buf.getvalue())

def pipe_pickle_send(fd, size, count):
    a = np.ones(size, dtype='c')
```

```

for _ in range(count):
    buf = pickle.dumps(a)
    os.write(fd, len(buf).to_bytes(8, sys.byteorder))
    start = 0
    while start < size:
        m = os.write(fd, buf[start:])
        start += m

@timing
def pipe_pickle_transfer(size, count):
    r, w = os.pipe()
    communicate(pipe_pickle_send, pipe_pickle_recv, (r, w), (size, count))
    os.close(r)
    os.close(w)

```

Similar to the two options available for pipes, there are two methods for transferring NumPy arrays through sockets. The method with data serialization incurs an overhead comparable to that observed with pipes. Therefore, we only implement the method that utilizes sockets to transfer the data buffer of NumPy arrays directly.

```

def socket_recv(sock, size, count):
    for _ in range(count):
        buf = io.BytesIO()
        remaining = size
        while remaining > 0:
            dat = sock.recv(remaining)
            buf.write(dat)
            remaining -= len(dat)
        buf = np.ndarray(shape=(size,), dtype='c', buffer=buf.getvalue())

def socket_send(sock, size, count):
    a = np.ones(size, dtype='c')
    buf = a.data
    for _ in range(count):
        start = 0
        while start < size:
            m = sock.send(buf[start:])
            start += m

@timing
def socket_transfer(size, count):
    r, w = socket.socketpair()
    communicate(socket_send, socket_recv, (r, w), (size, count))
    r.close()

```

```
w.close()
```

When using shared memory (shm) to transfer a NumPy array, it is preferable to transfer only the data buffer of the array and rebuild the array on the client side. This can be achieved by copying the data to the shared memory on the server side. On the client, the shared memory is directly mapped to the NumPy array, thereby eliminating the need to copy the data again when rebuilding the array.

In the following program, we directly distribute the shared memory object to the child process, thanks to the COW technique. Please note that such a mechanism is incompatible with the `ProcessPoolExecutor` because the shared memory object cannot be serialized and distributed.

```
def shm_recv(shm, size, count, guard):
    for _ in range(count):
        with guard.for_recv():
            buf = np.ndarray(shape=(size,), dtype='c', buffer=shm.buf)

def shm_send(shm, size, count, guard):
    data = np.ones(shape=(size,), dtype='c')
    for _ in range(count):
        with guard.for_send():
            buf = np.ndarray(shape=(size,), dtype='c', buffer=shm.buf)
            buf[:] = data

@timing
def shm_transfer(size, count):
    shm = SharedMemory(create=True, size=size)
    communicate(shm_send, shm_recv, shm, (size, count, Guard()))
    shm.unlink()
```

To prevent the server and client from modifying the shared memory at the same time, the program employs the `Guard` class for synchronization. In practice, this synchronization step might be unnecessary if the data has been copied to the shared memory before the child processes are spawned.

For programs that use the `ProcessPoolExecutor`, shared memory can be transferred via the filesystem. We can distribute the filename associated with the shared memory and rebuild the shared memory object in each process. Subsequently, the shared memory object can be linked to a NumPy array to serve as the data buffer. The following code snippet implements this procedure.

```
def shm_file_recv(shm_file, size, count, guard):
    for _ in range(count):
        with guard.for_recv():
            shm = SharedMemory(name=shm_file)
            buf = np.ndarray(shape=(size,), dtype='c', buffer=shm.buf)
```

```

def shm_file_send(shm_file, size, count, guard):
    data = np.ones(shape=(size,), dtype='c')
    for _ in range(count):
        with guard.for_send():
            shm = SharedMemory(name=shm_file)
            buf = np.ndarray(shape=(size,), dtype='c', buffer=shm.buf)
            buf[:] = data

@timing
def shm_file_transfer(size, count):
    shm = SharedMemory(create=True, size=size)
    communicate(shm_file_send, shm_file_recv, shm.name,
                (size, count, Guard()))
    shm.unlink()

```

The `memmap` technique exhibits many similarities to the shared memory approach in the context of transferring NumPy arrays. By using the `np.memmap` function, we create NumPy arrays that can be directly passed to child processes. The `memmap` arrays can be manipulated on both the server and client sides without requiring any additional operations.

```

def np_memmap_recv(buf, size, count, guard):
    for _ in range(count):
        with guard.for_recv():
            pass

def np_memmap_send(buf, size, count, guard):
    data = np.ones(shape=(size,), dtype='c')
    for _ in range(count):
        with guard.for_send():
            buf[:] = data

@timing
def np_memmap_transfer(size, count):
    mmap_file = tempfile.mktemp()
    buf = np.memmap(mmap_file, shape=(size,), dtype='c', mode='w+')
    communicate(np_memmap_send, np_memmap_recv, buf,
                (size, count, Guard()))
    os.remove(mmap_file)

```

Directly distributing the `memmap` object is not applicable with `ProcessPoolExecutor`. For `ProcessPoolExecutor`, it is necessary to use the filesystem for data transfer, similar to the method implemented in the `shm_file_transfer` test. We can send the

Table 10.1 Throughput (GB/s) of each IPC technique for transferring NumPy arrays.

Array size	0.1 MB	1 MB	10 MB	100 MB
pipe	2.92	0.96	1.03	1.02
pipe-pickle	2.52	0.71	0.38	0.47
socket	4.64	1.53	1.83	1.21
socket-pickle	3.04	0.95	0.78	0.53
shm	2.96	8.3	9.76	5.03
shm-via-file	0.48	1.73	2.53	2.56
np.memmap	3.08	7.83	9.36	5.14
memmap-via-file	0.37	0.96	1.16	1.08
shm w/o lock	18.2	14.2	10.9	5.03
shm-file w/o lock	2.03	2.98	3.17	2.64

filename of the `memmap` file to the child process, which will then use this filename to reconstruct the `memmap` arrays.

```
def memmap_recv(mmap_file, size, count, guard):
    for _ in range(count):
        with guard.for_recv():
            buf = np.memmap(mmap_file, shape=(size,), dtype='c', mode='r+')

def memmap_send(mmap_file, size, count, guard):
    data = np.ones(shape=(size,), dtype='c')
    for _ in range(count):
        with guard.for_send():
            buf = np.memmap(mmap_file, shape=(size,), dtype='c', mode='w+')
            buf[:] = data

@timing
def memmap_transfer(size, count):
    mmap_file = tempfile.mktemp()
    communicate(memmap_send, memmap_recv, mmap_file,
                (size, count, Guard()))
    os.remove(mmap_file)
```

These testing programs were executed to transfer a total of 1 GB of NumPy arrays, with individual array sizes ranging from 100 KB to 100 MB. The performance of each IPC technique is presented in Table 10.1. From these results, we can identify several important insights:

- Array size significantly influences performance. Pipes and sockets are more efficient for smaller arrays, while shared memory and `memmap` exhibit better performance for larger arrays. The turning point is around 1 MB.

- The poor performance of shared memory and memmap for small arrays can largely be attributed to the overhead of event locks. These locks are frequently acquired and released in tests for small arrays, introducing non-negligible costs. However, locks are unnecessary if data has been copied to the shared memory before the child process begins to access the data. This can lead to significantly improved performance, particularly for the transfer of small arrays, as shown by the row labeled “shm w/o lock”.
- For pipes and sockets, serialization introduces noticeable overhead, particularly for large arrays. Pickle serialization can lead to increased memory usage. The overhead for small arrays is minimal, since the data for deserialization is likely already present in the CPU L2 cache. However, for large arrays, the serialization process requires additional memory, which leads to increased memory management costs and a higher possibility of cache misses. In such cases, the cost of serialization can exceed the data transfer itself.
- The performance of pipes and sockets can be considered comparable, although pipes are slightly slower than sockets in these tests. Their performance is highly dependent on the system configuration.
- Transferring data through the filesystem is significantly slower than the methods that directly share objects with the child process. The tests labeled “shm” and “np.memmap” correspond to the direct-sharing methods, while those labeled “shm-via-file” and “memmap-via-file” refer to the filesystem-based methods. The direct-sharing methods exhibit similar performance, being 2–8 times faster than the filesystem-based methods. The overhead of filesystem-based data transfer can be attributed to two primary factors. Firstly, when the child process receives the filename of the shared memory (or the memmap object) it must perform the expensive system call `mmap` to map the file into virtual memory. Secondly, the newly created data object might encounter severe memory page faults and cache misses, which further slow down subsequent operations. The “memmap-via-file” tests incur an even greater penalty, as their memory is linked to files on the disk.

`multiprocessing.Process` exhibits superior performance over the `ProcessPoolExecutor` in terms of the data transfer efficiency, especially when transferring large arrays to child processes. The inefficiencies of `ProcessPoolExecutor` can be attributed to the following reasons:

- The overhead associated with pickle serialization.
- The relatively slow performance of pipes and sockets when transferring large volumes of data.
- The overhead from costly `mmap` system calls, especially when utilizing shared memory or memmap files for small arrays.
- The penalties related to cold cache and memory management when large arrays are shared via the filesystem.

If the `multiprocessing.Process` parallelization is used in your program, the choice of data transfer method should be considered carefully. For large-sized function arguments, specifically arrays exceeding 10 MB, utilizing shared memory or

memory-mapped are more efficient. Conversely, for smaller function arguments, the default mechanism based on pipes and pickle serialization is preferable over shared memory or memmap files, owing to its lower overhead.

10.8 OpenMP

OpenMP is a widely used parallel programming framework for C/C++ and Fortran programs. It facilitates data parallelization through simple *OpenMP directives* within the program. However, pure Python code cannot directly utilize the OpenMP directives. The OpenMP parallelism in Python relies on the underlying C extensions, which can be implemented in various approaches as discussed in Chapter 8 and Chapter 9.

Since OpenMP parallelism is only implemented in compiled languages, Python has limited capabilities to manage OpenMP functionalities. Typically, in a Python program, we only configure the number of threads and set the OpenMP parallel schedule.

A common method for configuring OpenMP is through the environment variables. For example, the environment variable `OMP_NUM_THREADS` can be used to set the number of OpenMP threads. `OMP_SCHEDULE` can adjust the scheduling strategy for OpenMP parallel execution [9]. Similarly, the environment variable `NUMBA_NUM_THREADS` can be used to configure the level of parallelism for the Numba JIT-compiled parts of the program. The environment variables `MKL_NUM_THREADS` and `OPENBLAS_NUM_THREADS` are commonly used to control the number of threads used by the BLAS library (for NumPy and SciPy functions).

Setting OpenMP environment variables is equivalent to specifying default parameters for OpenMP parallelism. If there is a need to dynamically adjust OpenMP parameters during runtime, we can directly interact with OpenMP shared libraries using `ctypes`. OpenMP parallelism is usually provided by libraries like GNU's `libgomp.so` or Intel's `libiomp.so`. By calling OpenMP APIs from these libraries, such as `omp_set_num_threads`, we can dynamically set the number of threads. For example, the following `ctypes` code can set the number of threads to 4 and the scheduling policy to `guided` for GNU OpenMP libraries:

```
import ctypes
omp_sched = { # refer to the omp_sched_t enum in OpenMP documentation
    'static': 1,
    'dynamic': 2,
    'guided': 3,
    'auto': 4,
}
chunk_size = 8
# The path should be replaced with the actual path on the user's system.
gomp = ctypes.CDLL('path/to/libgomp')
```

```
gomp.omp_set_num_threads(4)
gomp.omp_set_schedule(omp_sched['guided'], chunk_size)
```

Based on this approach, the `threadpoolctl` library [10] offers more accessible Python APIs for managing OpenMP threads. However, `threadpoolctl` does not support configuring OpenMP schedules. We can use the code snippet provided above, along with `threadpoolctl`, to manage the OpenMP parallelism.

Different Python modules might use OpenMP libraries from different vendors, versions, and file locations. When loading OpenMP libraries with `ctypes.CDLL`, it is necessary to specify the correct file path. The `threadpoolctl` library offers an automated method to identify the OpenMP library paths that have been loaded in Python. These paths can be retrieved by calling the `threadpoolctl.threadpool_info()` function.

10.9 MPI

The Message Passing Interface (MPI) is a fundamental tool in high-performance computing (HPC) for designing parallel programs. It is optimized for efficient network communication in parallel applications. MPI can integrate hardware and software to maximize communication performance, especially for inter-process communication across different nodes. Due to its requirements for a stable and high-performance network, MPI is commonly used in HPC supercomputers. When computing resources are provided by cloud workers, although technically feasible, MPI parallelism is often not the preferred option. In such environments, distributed executors like Dask and Ray are more suitable alternatives (see Chapter 7 for further discussions).

While popular MPI libraries are implemented in C, direct access to the C/C++ APIs of the MPI library is not necessary. The Python `mpi4py` library [11] provides a comprehensive interface to MPI, enabling the implementation of MPI parallelism entirely in Python.

Installing `mpi4py` can be somewhat complex as it does not come with a pre-built wheel. Executing

```
pip install mpi4py
```

would compile the package from source code, which requires a pre-installed MPI library and the corresponding `mpicc` compiler. If you are using Python via conda, a convenient method to install `mpi4py` is through the conda-forge channel:

```
conda install -c conda-forge mpi4py openmpi
```

This command installs the `mpi4py` library with the Open MPI backend.

On HPC clusters, it is common to encounter multiple MPI libraries, various compilers, and different versions of Python interpreters. If a specific combination of compiler, MPI library, and Python version is used to compile `mpi4py`, you must use

the same combination to run the `mpi4py` program, just like running the C/C++ or Fortran MPI programs.

10.9.1 Naming conventions in MPI4Py

`mpi4py` supports a variety of MPI backends, such as Open MPI and MPICH. Different MPI libraries may offer slightly different functionalities. `mpi4py` has integrated the most common MPI APIs, up to the MPI-2 standard, into Python.

The `mpi4py` library initializes a default `Communicator` instance when it is imported. This instance can be accessed using the import statement:

```
from mpi4py.MPI import COMM_WORLD as comm
```

In the following text, we will use the `comm` object to represent the default `COMM_WORLD` communicator, unless otherwise specified.

The `Communicator` class in `mpi4py` closely mirrors the MPI standard APIs. The prefixes `MPI_` and `MPI_Comm_` in the MPI APIs are typically omitted when translating them into the methods of the `Communicator` class from the `mpi4py.MPI` module. Methods of the `Communicator` class can be grouped into the following categories:

- *Point-to-point communication* methods, such as `comm.send`, `comm.recv`, which enable the direct data exchange between processes.
- *Collective communication* methods for communications between multiple processes, such as `comm.gather`, `comm.scatter`, `comm.alltoall`, `comm.allreduce`.
- *Process management* methods, such as `comm.Barrier` for synchronization, `comm.Abort` for termination, `comm.split` for creating new communicators.
- *MPI information retrieval methods*, such as `comm.Get_rank`, and `comm.Get_size`. The `comm` object provides two attributes `comm.rank` and `comm.size` as shortcuts for the two methods. The `comm.rank` is the ID of each MPI process, ranging from 0 to $n - 1$ where n represents the total number of processes, `comm.size`, launched by the MPI program.

We will not document the details of each MPI communication method here, as comprehensive documentation and explanations are readily available in established MPI parallel programming references, such as the book *Parallel Programming: for Multicore and Cluster Systems* by Rauber and Rünger [12], and Open MPI documentation.

The MPI API follows certain naming conventions for its functions. For instance, a function name prefixed with the letter `I` denotes a non-blocking operation. This type of function returns control to the caller immediately and performs data transfers in the background. `mpi4py` inherits these naming conventions when defining its method names. Additionally, it has introduced conventions to distinguish between the handling of general Python objects and buffer objects (such as NumPy arrays and `bytes` objects).

- The *general mode*: For methods that handle data as general Python objects, the first letter of the method name is in lowercase. Examples are `comm.isend`,

`comm.gather`. In this mode, Python objects are automatically pickled by senders and unpickled by receivers. This incurs additional computational costs and memory footprint due to the pickle and unpickle operations.

- The *buffer mode*: For methods that support objects with a contiguous memory buffer, the first letter of the method name is in uppercase, such as `comm.Isend`, `comm.Allreduce`. This mode is significantly more efficient than the general mode as it eliminates the overhead of pickling the objects.

A NumPy array can be transferred using two methods in `mpi4py`. One approach is to serialize the NumPy array as a Python object and transfer it using the general mode. The other method is to transfer the elements of a NumPy array via the buffer mode and the metadata (`dtype` and `shape`) via the general mode. In buffer mode, several options are available to specify the buffer [13]:

- For transferring a single NumPy array object, the array can be directly passed to the MPI functions. `mpi4py` can automatically determine the data type and buffer object.
- To explicitly specify the buffer object and the data type, a list can be specified. Both the two-element list structure `[data_buffer, data_type]` and the three-element list structure `[data_buffer, size, data_type]` are valid formats.
- For collective communication operations that include the suffix `v`, such as `comm.Allgatherv`, a four-element list `[data, count, displ, datatype]` can be supplied.

Here is an example of a simple wrapper function designed to broadcast a NumPy array across all nodes using the buffer mode:

```
rank = comm.Get_rank()

def broadcast(arr):
    if rank == 0:
        # The master node sends the array shape and data type
        comm.bcast((arr.shape, arr.dtype.char))
    else:
        # The other nodes receive the shape and data type
        shape, dtype = comm.bcast(None)
        # An empty array is created to receive the data
        arr = np.empty(shape, dtype=dtype)
        # The actual data of the array is broadcasted
        # MPI automatically discovers the data type
        # The array data is directly written into the buffer
        comm.Bcast(arr)
    return arr
```

10.9.2 SPMD MPI programs in Python

The MPI program in Python can be implemented using the Single Program Multiple Data (SPMD) design pattern. This is the most commonly adopted approach in MPI programs written in C/C++ or Fortran. In this design pattern, all MPI processes execute the same program. The program utilizes the `comm.rank` to differentiate between processes, and executes different code blocks for different processes.

On the other hand, the Multiple Program Multiple Data (MPMD) design pattern offers the flexibility of running different programs on the different MPI processes. The interpreted nature of Python makes it well-suited for this design pattern. For example, the `mpi4py.futures` module provides an `MPIPoolExecutor` class, which allows the execution of distinct tasks on individual MPI processes.

In this section, we will explore the SPMD MPI program. Given that all processes execute the same function, we must differentiate the roles of each MPI process, such as the sender or receiver, and implement the appropriate logic for each process. For instance, the `get_j` function can be implemented in this manner:

```
# filename: get_j_mpi.py
from mpi4py.MPI import COMM_WORLD as comm
rank = comm.Get_rank()
size = comm.Get_size()

def compute_eri(i, n):
    # Mimic the function to compute a sub-block of the ERI tensor
    return np.random.rand(n,n,n)

def get_j(dm):
    if rank == 0:
        n = dm.shape[0]
        comm.bcast(n)
    else:
        n = comm.bcast(None)
        dm = np.empty((n, n))
    comm.Bcast(dm)

    try:
        output = np.zeros((n, n))
        for i in range(0, n, size):
            output += np.einsum('jk1,j->k1', compute_eri(i, n), dm[:,i])
    except Exception:
        comm.Abort(1)                                # (1)

    jmat = np.zeros((n, n))
    req = comm.Iallreduce(output, jmat) # Communication in background
    req.wait()
```

```

    return output

if rank == 0:
    dm = np.identity(150)
    output = get_j(dm)
else:
    get_j(None)

```

The error-exception system in Python allows for capturing errors at one point and postponing their handling at another. However, in Python MPI programs, it is particularly important to handle exceptions promptly. If an error occurs before essential MPI communication operations and error handling is postponed, the program might skip these MPI operations. This can easily lead to deadlocks and the hanging of the MPI program. The MPI daemon is unable to terminate the MPI execution unless some worker processes exit or crash. To gracefully terminate a Python MPI program, it might be necessary to include additional error checks for the communication operations, and issue a manual call to `comm.Abort` within the error handling code, as shown in line (1).

This MPI Python program can be launched using the following command:

```
$ mpirun -n 8 --host node1,node2 python get_j_mpi.py
```

This command specifies that 8 MPI processes are equally distributed on two nodes, which are identified by the hostnames `node1` and `node2`. All 8 processes will execute the same Python program `get_j_mpi.py`.

10.9.3 Integrating MPI into serial program

In many cases, we only need to parallelize a few computationally intensive functions, which represent just a small portion of a serial program. The SPMD design requires careful planning of the parallel algorithm from the beginning of the program. This approach can make the program unnecessarily complex, particularly in the areas that involve shared resources, such as I/O operations.

To simplify MPI parallel computation within a serial program, `mpi4py` offers the `mpi4py.futures.MPIPoolExecutor` class. This class provides functionalities similar to those of the `ProcessPoolExecutor`. However, as of `mpi4py` version 3.1, the `MPIPoolExecutor` is limited to intra-node MPI parallelism.

By applying the Multiple Program Multiple Data (MPMD) model, we can enable the inter-node MPI parallelization in a serial program. In this approach (Fig. 10.5), the MPI master process runs the serial program as a single process, while the remaining MPI processes act as daemons, standing by to execute tasks. These tasks include both the data and the instance of the function to be executed. When a parallel session is reached, the master process distributes both the data and the functions to the worker processes, engaging in parallel computation. When the daemon worker receives a task, it decodes the function and then performs the computation. Consequently, the

MPI program is capable of executing any parallel tasks sent by the master, not just the pre-defined SPMD code.

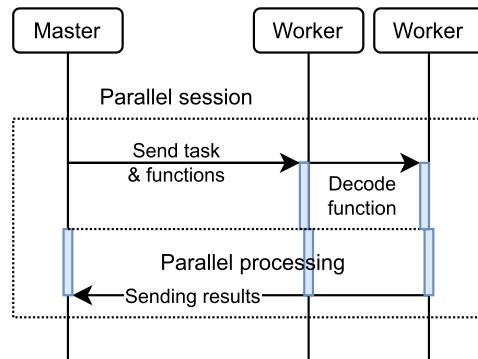


FIGURE 10.5

Parallel execution of a serial program using MPI Multiple Program Multiple Data (MPMD) executors.

To implement the daemon workers, the following technical aspects should be considered:

1. Daemon workers must run an infinite loop to continuously receive tasks sent by the master process.
2. The daemon workers need a mechanism to exit the infinite loop. This can be achieved by receiving a specific termination signal from the master process.
3. Tasks (including both data and function) should be properly serialized for network communication. `mpi4py` utilizes the Python built-in `pickle` module for object serialization. However, certain functions, such as closures, cannot be pickled. When encountering such issues, `cloudpickle` can be utilized for serialization of the complex objects.
4. To avoid the unintended mixing of messages, a unique communication tag should be employed in the `send` and `recv` functions.

The code below illustrates how a simple daemon worker is implemented in the file `mpi_worker.py`.

```

$ cat mpi_worker.py
from mpi4py.MPI import COMM_WORLD as comm
TAG = 92815
def wait():
    assert comm.rank != 0
    func = None
    while True:
        label, args = comm.recv(tag=TAG)

```

```

        match label:
            case 'Terminate': return
            case 'Func': func = args
            case 'Args':
                result = func(*args)
                comm.send(result, dest=0, tag=TAG)

if __name__ == '__main__':
    wait()

```

In the master process, we can implement several helper functions to enhance the functionality. For instance, we can create an MPI version of the `map` function.

```

import itertools
from mpi4py.MPI import COMM_WORLD as comm

size = comm.Get_size()
workers = range(1, size)
TAG = 92815

if comm.rank == 0:
    import atexit
    from time import sleep

def wait_all(requests, timeout=None):
    if timeout is None:
        return [req.wait() for req in requests]

    if not all(req.test() for req in requests):
        elapsed, interval = 0, 0.1
        while elapsed < timeout: # (1)
            sleep(interval)
            if all(req.test() for req in requests):
                break
            elapsed += interval
        else:
            print('Response from receivers timeout')
            comm.Abort()
            raise TimeoutError
    return [req.wait() for req in requests]

def shutdown():
    reqs = [comm.isend(('Terminate', None), i, tag=TAG)
            for i in workers]
    wait_all(reqs, timeout=0.5)

```

```

atexit.register(shutdown)

def _build_chunks(iterable):                                     # (2)
    iterable = iter(iterable)
    while True:
        chunk = list(itertools.islice(iterable, size))          # (3)
        if not chunk:
            return
        yield chunk

def map(func, *iterables):                                       # (4)
    assert comm.rank == 0
    for i in workers:
        comm.send(('Func', func), dest=i, tag=TAG)
    results = []
    for args_list in _build_chunks(zip(*iterables)):
        reqs = [comm.isend('Args', args), i, tag=TAG]           # (5)
                    for args, i in zip(args_list[1:], workers)]
        reqs = wait_all(reqs, timeout=0.5)
        results.append(func(*args_list[0]))
        for i in workers[:len(reqs)]:
            results.append(comm.recv(source=i, tag=TAG))
    return results

```

In the map function, we might need to partition the tasks, represented by the `iterables` in line (4), into smaller batches for processing. The generator `_build_chunks` in line (2) is designed to regroup the tasks based on the number of MPI processes available. This is achieved by the `itertools.islice` function, in line (3), which selects the first `comm.size` tasks from the task list each time the generator is invoked. Consequently, the tasks are distributed in a manner that aligns with the number of MPI processes, ensuring a balanced workload.

The non-blocking function `MPI.isend` is invoked to distribute tasks, as shown in line (5). Since the Python GIL is released during MPI data transfers, this method allows for the overlapping of multiple communication operations. The non-blocking function returns an `MPI.Request` object. We can use its `.test` method to check if a send or receive operation has been completed. It is important to ensure that all non-blocking send operations are completed before proceeding to subsequent operations. This synchronization is achieved using the `wait_all` function. As a rule of thumb for network communication, a timeout mechanism should always be applied. To avoid deadlocks in the MPI operations, as shown in line (1), the `wait_all` periodically checks the status of data transfers until the timeout is reached.

Tasks are distributed using MPI point-to-point operations. This option is inefficient for transferring tasks with large arguments. In such scenarios, we can consider to move the data-intensive transfers inside the custom function `func` itself. By doing

this, collective communication operations can be utilized to efficiently transfer large datasets when the workers are executing the custom function.

To terminate the workers gracefully, we implement a `shutdown` function and register it to the `atexit` module. This function is responsible for sending a termination request to all workers when the master process exits. Although the MPI driver can terminate workers using the `SIGTERM` and `SIGKILL` signals, this approach might result in the improper release of resources. For example, files that have not been flushed or closed may lose their contents. To prevent deadlocks during the termination phase, we use a non-blocking approach to send termination requests, followed by a `wait_all` operation with a timeout.

The MPI version of the map function can be integrated into serial programs. Taking the function `get_j` as an example, we can implement the `get_j_mpi_map.py` module as follows.

```
import itertools
import numpy as np
import mpi_map

def compute_eri(i, n):
    # Mimic the function to compute ERIs
    np.random.seed(i)
    return np.random.rand(n,n,n)

def get_j_task(i, dm):
    n = dm.shape[0]
    return np.einsum('jkl,j->kl', compute_eri(i, n), dm[:, :, i])

def get_j(dm):
    n = dm.shape[0]
    output = mpi_map.map(get_j_task, range(n), itertools.repeat(dm, n))
    jmat = sum(output)
    return jmat
```

This module can be used as a regular Python module within a sequential program.

```
$ cat run.py
import numpy as np
import get_j_mpi_map

n = 25
dm = np.identity(n)
out = get_j_mpi_map.get_j(dm)
print(out.shape)
```

When executing this Python program, we use the MPMD style command to launch the processes. Different programs are separated using the delimiter “:”, as show below:

```
$ mpirun -n 1 python run.py : -n 7 python mpi_worker.py
```

In MPMD, the order in which the programs are specified in the `mpirun` command is critical. To ensure that rank 0 within the MPI `COMM_WORLD` is assigned to the serial program, the serial program must be specified first.

Based on the MPMD approach, additional parallel functionalities can be implemented, such as a logging system and mechanisms for error handling. If you are interested in exploring these concepts further, a practical example of these techniques is available in the Python package `schwimmbad` [14].

10.9.4 Shared memory

When studying the `multiprocessing` module in Section 10.6, we introduced shared memory for data transfer due to its high efficiency. Naturally, one might consider how to harness shared memory in MPI programs to improve data transfer efficiency.

It should be noted that, the term *shared memory* within the MPI standard differs from the one used in the `multiprocessing` context. In the MPI standard, shared memory is referred to as global memory, which is consistent with the one-sided communication model, also known as the Remote Memory Access (RMA) model [15]. This model aims to simplify the management of memory across nodes and reduce the latency of data access. The `mpi4py` library offers the `Win` class to manage RMA, which acts as the communication handler, similar to the `COMM_WORLD` Communicator in the two-sided communication APIs. Methods such as `Get`, `Put`, and `Accumulate` within the `Win` class are employed in a manner similar to the C API of MPI RMA. The use of MPI RMA is documented in various MPI literature, such as the book *Parallel Programming: for Multicore and Cluster Systems* by Rauber and Rünger [12]. However, a detailed discussion of this topic is beyond the scope of our current text.

If the goal is to enhance the efficiency of data exchange within the same node using shared memory, we can consider the following three scenarios and their corresponding solutions:

- Optimizations by MPI back-ends. Certain MPI implementations, such as Open MPI, automatically optimize the communication for processes residing on the same node [16]. These libraries are able to detect whether shared memory can be utilized and then adapt the data transfer operations accordingly. If the intention is to utilize shared memory in this manner, there is no need to modify the MPI program.
- Manual management of intra-node communication. This approach is analogous to the strategy how MPI processes handle file operations. On each node, we can operate on a single shared memory file for each shared memory object. By invoking the `comm.Split_type(MPI.COMM_TYPE_SHARED)` function, processes are reorganized

into groups, creating intra-node communicators. For each intro-node communicator, specific code can then be implemented to manually handle the shared memory using the Python `mmap` module or the `multiprocessing.SharedMemory` class.

- MPI and X hybrid method. Here, X represents any intra-node parallelization technique, such as OpenMP, `ThreadPoolExecutor`, `ProcessPoolExecutor`. In this hybrid approach, MPI is utilized exclusively for inter-node communication, while the intra-node parallelism is managed by the other technique. This arrangement can lead to a more efficient use of memory resources within the node, as the parallelization is handled by the method most suitable for the intra-node scenario.

Summary

In this chapter, we briefly discussed parallel computation techniques. We demonstrated how to utilize multithreading, asynchronous programming, multiprocessing, OpenMP, and MPI parallelism in parallel computation programs. We also discussed issues related to data communication and synchronization in parallel computation. Our discussion covered inter-thread and inter-process synchronization, the producer-consumer model, the pipeline executor, and methods for data exchange in inter-process communication.

Python provides the `threading` module and the `ThreadPoolExecutor` for multithreading parallelism. The `ThreadPoolExecutor` offers very useful APIs, and it can almost completely replace the use of the `threading` module in multithreading parallelization. To overcome the impact of Python's Global Interpreter Lock (GIL) on the efficiency of multithreading, the `multiprocessing` module or the `ProcessPoolExecutor` can be utilized to implement multiprocessing parallelism. The `ProcessPoolExecutor` has a similar API to the `ThreadPoolExecutor` and is convenient to use. However, it cannot fully replace the standard `multiprocessing` library in the same way the `ThreadPoolExecutor` does to the `threading` module. Because of the copy-on-write technique, `multiprocessing` is more efficient and robust in data transmission. When we need to further use more nodes to parallelize Python programs, besides the distributed job executor introduced in Chapter 7, the `mpi4py` package can help us utilize the MPI library to achieve cross-node parallelism. By leveraging the dynamic language features of Python along with the MPI MPMD (Multiple Program Multiple Data) paradigm, we can integrate MPI into a serial program. This integration creates a new approach to utilize MPI parallelism beyond the traditional MPI SPMD (Single Program Multiple Data) framework.

Data communication presents a complex technical challenge. Common solutions for inter-process data transfer include pipes, sockets, shared memory, MPI, RPC (Remote Procedure Call), and message queues. When transmitting small NumPy arrays, pipes and sockets exhibit lower latency. However, for transferring larger arrays within the same node, pipes and sockets have high overhead, while shared memory offers the best performance.

Synchronization across workers is another challenge in parallel computing programs. When different workers need to simultaneously access the same resources, it is necessary to consider the potential race conditions or read-before-write errors. Python offers several synchronization primitives, such as mutexes, events, and condition variables, to handle these issues. However, employing these synchronization primitives can make the program complicated and difficult to maintain. In many cases, the producer-consumer model can simplify the method of data communication in the parallel program and avoid the use of synchronization primitives. Furthermore, the producer-consumer model can act as a task scheduler, helping to balance the computation load in parallel computing. Based on the producer-consumer approach, we introduced a pipeline executor to execute even more complex tasks in parallel.

References

- [1] Wikipedia Contributors, False sharing, https://en.wikipedia.org/wiki/False_sharing, 2024.
- [2] U. Drepper, What every programmer should know about memory, <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>, Nov. 2007.
- [3] Python Software Foundation, Python frequently asked questions - what kinds of global value mutation are thread-safe?, <https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe>, 2024.
- [4] Python Software Foundation, The Python standard library: asyncio - asynchronous I/O, <https://docs.python.org/3/library/asyncio.html>, 2024.
- [5] Python Software Foundation, multiprocessing - programming guidelines, <https://docs.python.org/3/library/multiprocessing.html#programming-guidelines>, 2024.
- [6] Python Software Foundation, multiprocessing - sharing state between processes, <https://docs.python.org/3/library/multiprocessing.html#sharing-state-between-processes>, 2024.
- [7] Wikipedia Contributors, Inter-process communication, https://en.wikipedia.org/wiki/Inter-process_communication, 2024.
- [8] M. Kerrisk, queue(7) — Linux manual page, <https://man7.org/linux/man-pages/man7/queue.7.html>, 2024.
- [9] OpenMP Architecture Review Board, OpenMP API specification: environment variables, <https://www.openmp.org/spec-html/5.0/openmpch6.html#openmpse50.html>, 2020.
- [10] Joblib, threadpoolctl: thread-pool controls, <https://github.com/joblib/threadpoolctl>, 2024.
- [11] L. Dalcin, Y.-L.L. Fang, mpi4py: status update after 12 years of development, Computing in Science & Engineering 23 (4) (2021) 47–54, <https://doi.org/10.1109/MCSE.2021.3083216>.
- [12] T. Rauber, G. Rnger, Parallel Programming: for Multicore and Cluster Systems, 1st edition, Springer Publishing Company, Incorporated, 2010.
- [13] L. Dalcin, MPI for Python tutorial, <https://mpi4py.readthedocs.io/en/stable/tutorial.html>, 2024.
- [14] A.M. Price-Whelan, D. Foreman-Mackey, schwimmbad: a uniform interface to parallel processing pools in Python, The Journal of Open Source Software 2 (17) (2017) 357, <https://doi.org/10.21105/joss.00357>.
- [15] L. Dalcin, MPI for Python - one-sided communications, <https://mpi4py.readthedocs.io/en/stable/overview.html#one-sided-communications>, 2024.

- [16] Open MPI Development Team Faq, Tuning the run-time characteristics of MPI shared memory communications, <https://www.open-mpi.org/faq/?category=sm>, 2024.

GPU programming

11

Graphics Processing Units (GPUs) are equipped with thousands of cores, allowing them to execute tasks in parallel. This architecture enables GPUs to achieve low-cost and high-throughput computational capabilities. When applied to floating-point intensive applications, such as linear algebra programs, GPUs can significantly outperform CPUs. Using GPUs with data parallelization is an attractive option to enhance the performance of quantum chemistry applications.

How can we enable GPU acceleration in a Python program? If the GPU runtime environment is properly configured, the GPU-enabled Python packages, such as PyTorch, CuPy, and JAX, are generally straightforward to use. However, to fully leverage the computation capabilities of GPU hardware, a deeper understanding of GPU architecture, runtime, hardware specifications, and the CUDA programming model is essential.

This chapter provides guidelines for integrating GPU computation into Python applications, focusing on:

- How can a NumPy-based program be incrementally transformed into a GPU program?
- How can special GPU features, such as pinned memory and asynchronous computation, be utilized at the Python level to accelerate GPU computation?
- If encountering low utilization of the GPU hardware, or a lack of required functionality, how can one customize GPU kernels and launch kernels in Python?

Several packages and technologies are presented in the discussion of these technical processes.

11.1 Configuring GPU runtime environment

The rapid development of GPU hardware has led to a wide range of toolkits for GPU programming. An inappropriate combination of these toolkits can potentially lead to compatibility issues. Consequently, configuring the GPU runtime environment for Python applications becomes a complex task.

Unlike regular Python applications, neither Conda packages nor Python wheels alone are sufficient to manage GPU applications. The GPU hardware specs and runtime configurations may not be completely transparent to users when running a GPU

application. To correctly install and run Python GPU applications, we need a basic understanding of GPU runtime environments and architecture, including:

- The choices of frameworks and vendors for GPU applications, such as CUDA for Nvidia GPUs, and OpenCL and ROCm for AMD GPUs.
- The versions of CUDA toolkits and runtime, such as CUDA 11 and CUDA 12. Applications developed using a newer version of the CUDA toolkit may not be compatible with older versions of the CUDA runtime. To ensure compatibility and optimal performance, it is recommended to employ the same version of CUDA for both compilation and the CUDA runtime.
- Multiple versions of CUDA *Compute Architectures* and *Compute Capabilities* [1,2]. We need to ensure that the Compute Capability of a GPU application, such as `sm_70`, `sm_80`, and `sm_90`, is compatible with the compute architecture of the GPU hardware, such as Volta, Ampere, and Hopper. Binary compatibility is not guaranteed across different versions of Compute Capabilities.
- The distinction of *real* and *virtual* architecture [3]. GPU applications may contain a real binary specific to hardware, such as the `cubin` file in CUDA, for a specific hardware, and/or virtual intermediate instruction sets such as PTX (Parallel Thread Execution) code in CUDA. The CUDA runtime can perform JIT compilation to convert virtual code into real instructions for the specific GPU hardware in use. However, some applications may be compiled without targeting a virtual architecture, resulting in code that lacks portability across different GPU architectures.
- Specific GPU models. Even within the same micro-architecture, such as Hopper, individual GPUs may vary in hardware specifications, especially in terms of memory size and the double precision computation capacity. Some applications are specifically optimized for particular GPU models. Even if the GPU Compute Capability, the CUDA version, and other software prerequisites are met, these applications might not function as expected.

Below, we outline some common methods to identify and configure the CUDA runtime.

Selecting a CUDA version

Typically, the latest version of CUDA is recommended as it provides the best compatibility and supports a wider range of Nvidia GPU products. However, there are circumstances where using an older version of CUDA might be necessary. For instance, some libraries may not yet have a binary release compatible with the most recent version of CUDA.

The process of installing CUDA toolkits is documented in the official Nvidia documentation [4] and will not be repeated here. When running a GPU instance in the cloud, the cloud provider may supply a Linux OS with several pre-installed CUDA versions. For example, AWS cloud offers deep learning images that come with both CUDA 11 and CUDA 12, with CUDA 12 set as the default environment. To switch

to the other version, we can modify the `PATH` and `LD_LIBRARY_PATH` environment variables.

```
$ export PATH=/usr/local/cuda-12.0/bin:${PATH}  
$ export LD_LIBRARY_PATH=/usr/local/cuda-12.0/lib64:${LD_LIBRARY_PATH}
```

Alternatively, we can modify the symbolic link for the default CUDA installation location `/usr/local/cuda` [5]:

```
$ sudo ln -fs /usr/local/cuda-12.0 /usr/local/cuda
```

Configuring runtime environments for Docker containers

Python applications are often executed within Docker containers. However, merely using the default Docker command to start a CUDA-enabled image does not properly load the CUDA runtime. To run CUDA applications within a container, the Nvidia Container Toolkit is necessary [6]. After installing this toolkit, CUDA runtime environments will be integrated into the Docker configuration file, `/etc/docker/daemon.json`, as shown in the following example:

```
{  
    "runtimes": {  
        "nvidia": {  
            "path": "/usr/bin/nvidia-container-runtime",  
            "runtimeArgs": []  
        }  
    }  
}
```

To utilize the CUDA runtime, we can then specify the `--runtime` option:

```
docker run --runtime=nvidia
```

If you are utilizing GPU instances in the cloud, the required CUDA runtime for Docker might already be pre-installed in the image of the virtual machine. To confirm whether the CUDA runtime toolkit for Docker is enabled, you can check the existence of the following executable file:

```
$ which nvidia-container-toolkit
```

Checking compute capability

A GPU application can be designed to target multiple *Compute Capability* classes, and thereby incorporate executable instructions for each class. However, this strategy can significantly increase the program size. To reduce the program size, some GPU applications are compiled only for specific GPU hardware or the necessary Compute Capabilities. When executing a pre-compiled GPU application, it is necessary to en-

sure that the hardware supports the required Compute Capability [2]. The following command can be used to list the supported PTX files within a CUDA library [7]:

```
$ cuobjdump --all-fatbin /path/to/library.so
```

Selecting appropriate software versions

Due to the variety of CUDA versions and Compute Capabilities, some Python GPU applications are packaged in multiple versions to accommodate different runtime environments. Users should choose the version that best matches their system's specifications. For instance, the libraries discussed in this chapter, such as PyTorch [8], CuPy [9], and JAX [10], all provide multiple CUDA backends for installation.

11.2 Architecture-independent optimization

In Python scientific applications, when a large number of linear algebra operations are performed, the performance can significantly benefit from the computational power of GPUs. On CPU platforms, these operations are often developed using the NumPy and SciPy libraries. To optimize the performance, the first step is to rewrite the NumPy functions or refactor the required code blocks to incorporate functions that can take advantage of GPU hardware. At this stage, it is not necessary to consider the specific GPU architecture. A practical strategy is to utilize libraries such as CuPy, PyTorch, or JAX, which can serve as drop-in replacements for the existing NumPy/SciPy code.

To illustrate this process, let us revisit the `get_j` function presented in Chapter 10. This function can be simply computed using the `einsum` function. However, to demonstrate the functionality of each Python GPU package, as well as the optimization process for a GPU application, we have intentionally refrained from using `einsum` in this example.

```
def get_j(eri, dm):
    '''Perform einsum("ijkl,ji->kl", eri, dm)'''
    n = dm.shape[0]
    output = np.zeros((n, n))
    for k in range(n):
        for l in range(n):
            for i in range(n):
                for j in range(n):
                    output[k,l] += eri[i,j,k,l] * dm[j,i]
    return output
```

Please be aware that utilizing GPU computation for this function might not lead to a significant performance improvement. In this scenario, the bottleneck is the data transfer between the host and the GPU device, which limits the computational power of the GPU.

11.2.1 CuPy

Transitioning from a NumPy-based program to the program that employs CuPy is straightforward. CuPy offers most of the functionalities available in NumPy. For a detailed comparison between CuPy APIs and their corresponding NumPy APIs, one can refer to the CuPy documentation [11,12]. In most scenarios, we can simply change the module name from `numpy` to `cupy`, such as

```
np.dot -> cupy.dot
np.zeros -> cupy.zeros
np.stack -> cupy.stack
np.transpose -> cupy.transpose
```

One notable API difference [12] between CuPy and NumPy is the return type of reduction functions such as `cupy.count_nonzero`, `cupy.sum`, `cupy.max`, and `cupy.argmax`. When a NumPy reduction function is called, it returns a scalar, whereas the CuPy equivalent returns a zero-dimensional array. When mixing CuPy code with NumPy code, this discrepancy might lead to compatibility issues, primarily due to the differences between scalars and zero-dimensional arrays, as discussed in Section 2.2.2 of Chapter 2. This is not a flaw in CuPy. Rather, it is an intentional design choice to reduce the overhead of synchronization between the CPU and the GPU. CPUs and GPUs can execute computations asynchronously. When reduction results are used as scalars for computations on the CPU, it is necessary to synchronize them from the GPU to the CPU. This synchronization blocks the execution of either the CPU or the GPU. Often, these reduction results serve as intermediate data for subsequent tensor operations on the GPU. The CuPy zero-dimensional array does not affect these computations on the GPU.

Let's use the `get_j` function as an example to illustrate the process of converting from NumPy to CuPy. In the CuPy-based program, we need to manage the location of the array data. To ensure that the data resides in the device (GPU) memory, we can use the function `cupy.asarray()`. Similarly, the `cupy.asnumpy()` function allows for the transfer of data back to the host (CPU) memory. Aside from these changes, the algorithm and the code structure remain essentially the same as the CPU version.

```
import cupy as cp
def get_j(eri, dm):
    n = dm.shape[0]
    eri = cp.asarray(eri) # Transfer to memory on GPU
    dm = cp.asarray(dm)
    output = cp.zeros((n, n))
    for k in range(n):
        for l in range(n):
            for i in range(n):
                for j in range(n):
                    output[k,l] += eri[i,j,k,l] * dm[j,i]
    return cp.asnumpy(output)
```

When working with CuPy, you may encounter certain issues that are not present in NumPy-based programs:

- *Memory Management.* To minimize the overhead of memory allocation and initialization on the GPU, CuPy utilizes a *memory pool* to manage memory on the GPU device. When a CuPy array object is deleted, CuPy does not immediately free the memory of the array. Instead, the memory is reserved in the memory pool for reuse, which helps CuPy reduce the overhead of memory allocation. The memory pool is only expanded when new arrays cannot fit into the existing memory pool. In some cases, you may encounter an `OutOfMemoryError` when creating a new CuPy array object, even if the `nvidia-smi` command shows sufficient free GPU memory. To solve this problem, you may need to manually adjust the size of the memory pool [13] using the method

```
cupy.get_default_memory_pool().set_limit()
```

Once the memory pool is expanded, it does not automatically shrink. Consequently, the program may occupy more GPU memory than it actually requires. To address this issue, you can call

```
cupy.get_default_memory_pool().free_all_blocks()
```

to free unused memory blocks and reduce the memory pool size.

- *Asynchronous Execution.* Except for a few data transfer functions, such as `cupy.asarray()` and `cupy.asnumpy()`, most functions in the CuPy library execute asynchronously. When a CuPy function is called, it is appended to a queue for execution on the GPU. The control flow then immediately returns to the Python interpreter to process the next statement. This asynchronous execution can mislead Python profilers. Python profilers typically record the time duration of each Python statement executed on the CPU. They only capture the time it takes for the CuPy function to submit the job to the GPU device, not the execution time on the GPU itself. Therefore, the timing data reported by the CPU profilers are unreliable for CuPy functions. To address this problem, CuPy has developed a profiler module. This module introduces the `cupyx.profiler.benchmark()` and `cupyx.profiler.profile()` methods to measure the timing of Python functions or code segments that involve GPU operations [14]. Generally, the profiler tools in CuPy have limited functionality and are not as comprehensive as the conventional Python profiling tools.

11.2.2 PyTorch

PyTorch tensor objects and APIs are highly similar to those of NumPy, allowing most NumPy functions to be seamlessly replaced by their PyTorch counterparts. However, compared to the similarity between CuPy and NumPy, PyTorch APIs exhibit more differences. Below are some commonly encountered incompatibility issues when using PyTorch to replace NumPy:

- PyTorch lacks certain functions in NumPy, such as the `np.array_equal` and `np.append` functions.
- Unless the `torch.set_default_tensor_type` function is explicitly configured, the default data type for tensor objects in PyTorch is `float32`, in contrast to the default data type `float64` in NumPy. For example, functions like `torch.zeros(5)`, `torch.rand(5)`, and `torch.as_tensor([1., 2., 3.])` all yield `float32` tensors by default.
- When multiple indices are applicable, NumPy functions use the keyword `axis` to specify which index of a tensor to operate on. The corresponding keyword argument in PyTorch is `dim`.
- Some PyTorch functions have different function signatures compared to their NumPy counterparts. For instance, `np.unique` and `torch.unique` follow different conventions for their return values [15]. Direct replacement of `np.unique` with `torch.unique` in a program might lead to unexpected results.

There is no comprehensive documentation like CuPy that tracks all the differences between the PyTorch and NumPy APIs. It is recommended to consult the PyTorch documentation to ensure compatibility before substituting any NumPy functions with PyTorch equivalents. Apart from this step, converting a CPU program to its PyTorch version can be accomplished in a manner similar to the approach previously demonstrated with CuPy.

Consider the `get_j` function as an example. To utilize PyTorch, we first need to set the default data type. Next, we add statements for data transfer between the host and the device. Transferring data to GPU memory can be accomplished using the `torch.as_tensor` function. It is important to include the keyword `device='cuda'` in this function, because PyTorch creates Tensor objects in the host memory by default. Alternatively, we can use the `torch.from_numpy` function to convert NumPy arrays to Tensor objects, and then call the general-purpose conversion method `to` to transfer the data to the GPU.

```
import torch
torch.set_default_tensor_type(torch.float64)
def get_j(eri, dm):
    n = dm.shape[0]
    eri = torch.as_tensor(eri, device='cuda') # Transfer to GPU
    dm = torch.as_tensor(dm, device='cuda')
    output = torch.zeros((n, n), device='cuda')
    for k in range(n):
        for l in range(n):
            for i in range(n):
                for j in range(n):
                    output[k,l] += eri[i,j,k,l] * dm[j,i]
    return output.to(device='cpu').numpy()
```

Similar to the CuPy library, to reduce the overhead of memory allocation, PyTorch also caches and reuses memory for tensor objects created on the GPU device.

However, the library does not automatically release the cached memory. You will need to manually free the unused memory blocks as well [16].

11.2.3 JAX

The JAX library includes a module, `jax.numpy`, which closely follows the NumPy API. The design and data manipulation approach employed by JAX differ significantly from those of NumPy. Directly translating a program from NumPy to JAX could lead to a significant performance drop. An efficient implementation with JAX often requires a substantial modification to the original program.

JAX is slow for element-wise operations. The preferred approach is to use a vector-oriented coding style. The official document *How to Think in JAX* [17] offers a guideline to help us better understand the JAX coding style. The implementation of `get_j` presented in the previous sections is not suitable for JAX. We can use NumPy ufuncs and broadcasting rules to perform vectorized operations. After eliminating explicit iterations, we can apply JAX JIT compilation to offload the calculations to GPU devices.

```
import jax.numpy as jnp
jax.config.update('jax_enable_x64', True)
@jax.jit
def get_j(eri, dm):
    return (eri.transpose(2, 3, 1, 0) * dm).sum(axis=(2, 3))
```

Please note that JAX functions use the `float32` data type by default, similar to PyTorch. To utilize the `float64` precision, we must configure the library with the statement

```
jax.config.update('jax_enable_x64', True)
```

to change the default data types.

11.3 Architecture-aware optimization

In the previous section, we simply replaced NumPy functions with the GPU-accelerated equivalents. However, the specific features of GPU architecture were not taking into account. Certain optimization techniques related to GPU architecture can be utilized without the need to access the CUDA API in C++ programs. These features are accessible through Python APIs provided by libraries such as CuPy and PyTorch. This section will focus on the use of pinned memory and GPU streams in CuPy. Although the API of PyTorch is slightly different, the underlying principles are similar.

11.3.1 Pinned memory

Pinned memory, also known as page-locked memory, is a type of memory that cannot be paged out to disk by the operating system. The use of pinned memory in data transfer can enhance the data transfer performance.

Normally, the memory used in a program is pageable. However, GPU cannot directly access data in pageable host memory. When data is transferred from pageable host memory to device memory, the data movement occurs in two stages. Initially, the data is copied from the pageable memory to a page-locked buffer, and subsequently, it is transferred from the buffer to the device. CPU is involved in this two-step data transfer process, specifically in the step that moves data between the pageable memory and the page-locked memory. This step can be skipped if the data on the host is created in the pinned memory from the beginning. This eliminates the latency and overhead associated with the memory management on CPU [18].

CuPy offers several functions in the `cupy.cuda.pinned_memory` module to manipulate pinned memory. To integrate more seamlessly with NumPy, CuPy also provides high-level APIs in the `cupyx` package to create NumPy arrays backed by pinned memory. Some commonly used APIs in `cupyx` include

- `cupyx.empty_pinned`
- `cupyx.empty_like_pinned`
- `cupyx.zeros_pinned`
- `cupyx.zeros_like_pinned`

Please note that the outputs of these functions are NumPy arrays rather than CuPy arrays on the GPU. We can use these functions to create NumPy array buffers in pinned memory and pass them to NumPy functions using the `out` keyword argument. NumPy functions then write data directly into the pinned memory.

For example, the `eri` tensor we passed to the `get_j` function can be initialized on the pinned memory.

```
import cupy as cp
import cupyx
pinned_buffer = cupyx.empty_pinned((n,n,n,n))
rng = np.random.default_rng()
eri_cpu = rng.random((n, n, n, n), out=pinned_buffer)

# host to device transfer
eri_gpu = cp.empty_like(eri_cpu)
eri_gpu.set(eri_cpu)
output = get_j(eri_gpu, dm)
```

In this example, we use the `random()` method from the random number generator `rng` to fill the `pinned_buffer`. Next, we create a CuPy array of the required size on the GPU and use the `.set()` method to transfer the data from the pinned memory to the device.

The two lines of code that create an empty array and call the `.set()` method might appear inelegant, but they are crucial for utilizing pinned memory. We cannot simplify the process by using functions like `cupy.asarray(eri_cpu)` to combine the creation of a CuPy array and data transfers into a single step. This is because the `cupy.asarray` function is an asynchronous operation. Data might be written to the input array before the data transfer is complete. To avoid potential race conditions within these asynchronous operations, *CuPy always creates a copy of the input NumPy array* within the `cupy.asarray` function, even when pinned memory is used. This introduces an overhead in data transfer.

To transfer data from the device to the pinned memory, we can create a pinned buffer and utilize the `.get()` method of the CuPy array object, using the `out` keyword option to specify the destination of the data.

```
output_cpu = output.get(out=cupyx.empty_pinned((n, n)))
```

Pinned memory is utilized in various scenarios beyond simply optimizing data transfers. For instance, it is a prerequisite for asynchronous data transfers and the zero-copy technique, which allows the GPU to directly access the host memory's address. However, the use of pinned memory comes with certain drawbacks. Pinned memory is always resident in physical memory, which can reduce the available paged memory for other applications. Moreover, pinning system memory is a resource-intensive operation, more costly than regular system memory allocations. When transferring small arrays, the use of pinned memory is unnecessary.

11.3.2 CUDA stream

A GPU stream is a queue of operations to be executed by the GPU. The GPU can execute multiple streams in parallel, thus enabling the concurrent execution of multiple operations. Using GPU streams can enable more efficient utilization of the GPU. A common scenario for using GPU streams is the overlapping of computation and data transfer, which reduces the overall execution time.

CUDA operates with a default stream, known as the NULL stream. All processors on the GPU execute tasks queued within this stream. CuPy functions, in their standard behavior, utilize the NULL stream, causing the entire GPU to perform the same set of instructions. To manage multiple GPU streams, the `cupy.cuda.stream` module can be used. For instance, a dedicated stream can be created to handle asynchronous data transfers. There are several key aspects regarding asynchronous data transfers:

- The host memory must be allocated as pinned memory, which is a requirement for asynchronous transfer [19].
- Modifications to the contents of the pinned memory should be avoided until the `synchronize` method for the corresponding stream has been called.
- Data transferred via a non-default stream may not be immediately accessible by the default (NULL) stream. To ensure that the data is ready for the default stream to process, call the `synchronize` method on the non-default stream before accessing it on the default stream.

Let's consider the transfer of the giant `eri` tensor from the host to the device as an example. The following code snippet illustrates the process of asynchronous data transfer:

```
pinned_buffer = cupyx.empty_pinned((n,n,n,n))
rng = np.random.default_rng()
eri_cpu = rng.random((n, n, n, n), out=pinned_buffer)

sm = cp.cuda.stream.Stream(non_blocking=True)
eri_gpu = cp.empty_like(eri_cpu)
eri_gpu.set(eri_cpu, stream=sm)
...
sm.synchronize()                                     # (1)
output = get_j(eri_gpu, dm)
```

Here, a new stream is created with the setting `non_blocking=True`. This allows the stream to operate asynchronously alongside the default stream. CuPy functions can be configured to use this new stream by assigning the new stream to the `stream` keyword argument. In this example, the data transfer is placed on the new stream while the computation of `get_j` is placed on the default stream. If the two streams are executed concurrently, the default stream might attempt to access memory addresses that have not yet been filled with the required data, leading to a read-before-write error. To ensure that the data is ready in the buffer, as shown in line (1), the new stream is synchronized before proceeding to the `get_j` function.

Alternatively, we can use the context manager provided by the `Stream` class to manage the execution of streams. Within the `Stream` context, we don't need to specify the `stream` keyword for the CuPy functions. CuPy functions will be executed on the new stream unless they are explicitly associated with a different stream object.

```
with cp.cuda.stream.Stream(non_blocking=True) as sm:
    eri_gpu = cp.empty_like(eri_cpu)
    eri_gpu.set(eri_cpu)
    output = get_j(eri_gpu, dm)
sm.synchronize()
```

In this code example, the `get_j` function is called within the context of the newly created stream. Functions within `get_j` are enqueued in the new stream, similar to the other two CuPy operations. Functions within the same stream are executed sequentially.

It should be noted that the role of the context manager is simply to set the default stream. It does not block the execution of other streams, nor does it automatically synchronize upon exiting the context manager. To avoid potential race conditions or conflicts between the new stream and the default stream, it is advisable to explicitly issue a `synchronize()` command after exiting the context manager.

As mentioned earlier, most CuPy functions operate asynchronously, which can lead to inaccuracies in timing and profiling of GPU programs. To obtain precise timing measurements for a specific function, one can enforce synchronization by calling the method `.synchronize()` on the NULL stream, which is accessible via `cupy.cuda.Stream.null.synchronize`. This ensures that all preceding asynchronous operations are completed.

11.4 Custom GPU kernels in Python

If we need to implement features that are not available in CuPy or any existing libraries, customizing GPU kernels becomes necessary. Essentially, there are two main considerations related to custom kernels:

- How to design the kernel to achieve the best performance?
- How to embed the custom kernel into a Python program?

To develop and embed custom kernels in Python, several options are available, including: Numba [20], CuPy RawKernel [21], PyCUDA [22,23], CUDA-Python [24], and `ctypes`. Each approach provides different levels of abstraction for integrating CUDA programming capabilities.

Numba offers a highly Pythonic method for translating Python code into GPU kernels. It allows users to write GPU programs without extensive knowledge of the CUDA programming model or GPU architecture.

CuPy custom kernel, as well as PyCUDA, and CUDA-Python allow users to integrate raw CUDA source code into Python while abstracting away the complexity of runtime configurations. These tools require a certain level of understanding of the CUDA C++ programming model.

The `ctypes` module can be combined with a native CUDA C++ project. This combination offers the greatest flexibility and potential for GPU functionalities and performance. However, using the `ctypes` module requires a deeper understanding of the GPU programming toolkit and more effort to maintain both the C++ backend and the Python interface.

Before exploring custom CUDA kernels, we would like to emphasize that in many scenarios, the bottleneck of an application lies in the data exchange between the host and the device, or the bandwidth of the GPU on-board memory. Developing a custom kernel may not necessarily improve the performance. As long as the performance penalty is acceptable, using a pure Python implementation with CuPy or PyTorch functions is still a favorable option. They are much easier to maintain than custom kernels.

11.4.1 Kernel in CUDA code

The GPU parallelization model, as shown in Fig. 11.1, is organized into a hierarchical structure consisting of grids, blocks, and threads. A grid is a collection of blocks that

execute the same kernel code. Within each block, a group of threads runs the same instructions, each operating on different data elements.

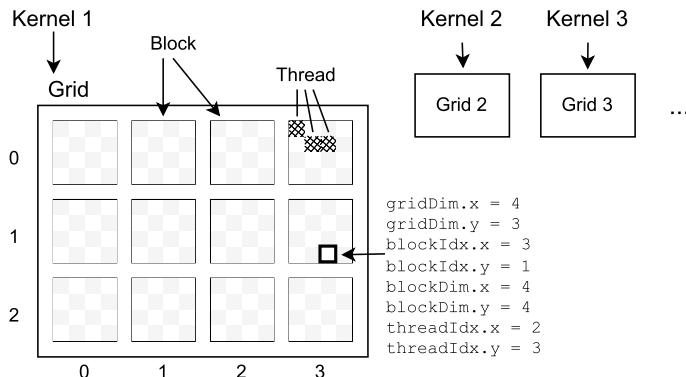


FIGURE 11.1

GPU parallelization model: a hierarchical structure consisting of grids, blocks, and threads.

Essentially, implementing a GPU kernel is a task that maps the data parallelization problem to the GPU parallelization model. Here, we will briefly discuss the design and optimization of CUDA code for the `get_j` function. This CUDA kernel will later be integrated into Python code. If you are interested in the details of GPU architecture and optimization methods, the official manual, *CUDA C Programming Guide* [25], and textbooks, such as *CUDA by Example: An Introduction to General-Purpose GPU Programming* [26] are excellent resources that provide comprehensive coverage of GPU programming concepts.

Our initial version is derived from the CPU code presented in Section 11.2. In this version, the loop indices `i` and `j` are mapped to CUDA threads. High-dimensional arrays are represented by one-dimensional arrays, and their addressing is converted from the multi-dimensional format. This is a common technique for handling multi-dimensional arrays in CUDA kernels. It is crucial to ensure the contiguous property of arrays on the Python side.

```

#include <cuda_runtime.h>
__global__ void _kernel_v1(double *output, double *eri, double *dm, int n)
{
    size_t ij = blockIdx.x * blockDim.x + threadIdx.x;
    size_t nn = n * n;
    if (ij > nn) return;
    size_t i = ij / n;
    size_t j = ij % n;
    size_t k1;
    double dm_ji = dm[j*n+i];
    for (k1 = 0; k1 < nn; ++k1) {
        ...
    }
}
  
```

```

        atomicAdd(&output[kl], eri[(i*n+j)*nn+kl] * dm_ji);
    }
}

```

This GPU kernel performs a reduction add operation on the elements of the output array. Since the output array is shared among CUDA threads, directly writing data to the array could lead to race conditions. We employ the CUDA atomic operation `atomicAdd` to ensure a race-condition-free reduction. Although `atomicAdd` is optimized at the hardware level, performing a reduction across CUDA blocks remains a highly costly operation and should generally be avoided.

To eliminate the cross-thread reduction, the second version of the CUDA kernel swaps the loop order of `ij` and `kl`. The indices `k` and `l` are then mapped to the CUDA threads. By implementing these changes, the reduction sum is accumulated in a register within each thread.

```

__global__ void _kernel_v2(double *output, double *eri, double *dm, int n)
{
    size_t kl = blockIdx.x * blockDim.x + threadIdx.x;
    size_t nn = n * n;
    if (kl > nn) return;
    size_t i, j;
    double s = 0.;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            s += eri[(i*n+j)*nn+kl] * dm[j*n+i];
        }
    }
    output[kl] = s;
}

```

In this implementation, each thread needs to repeatedly read elements from the `dm` array. Disregarding the effects of the cache in the GPU processors, this leads to a total of n^4 read operations from global memory. Accessing global memory incurs high latency. Therefore, it is desirable to minimize or avoid accessing data through global memory in a GPU kernel.

To optimize global memory access, the technique of tiling and shared memory can be utilized. Shared memory is a small, low-latency memory space located on CUDA stream processors. Threads within the same block can then access data from the shared memory, which effectively reduces the need for global memory accesses. The tiling technique involves dividing the dataset into small chunks that can fit into the shared memory. These techniques can be viewed as a way to manually control the behavior of the GPU cache. By implementing these techniques, we can develop a new version of the CUDA kernel.

```

#define BLOCK 16
__global__ void _kernel_v3(double *output, double *eri, double *dm, int n)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    size_t k1 = blockDim.x * blockDim.x * blockDim.y + ty * blockDim.x + tx;
    size_t nn = n * n;
    size_t i, j, i0, j0;
    __shared__ double dm_t[BLOCK][BLOCK];

    double s = 0.;
    for (i0 = 0; i0 < n; i0 += BLOCK) {
        for (j0 = 0; j0 < n; j0 += BLOCK) {
            __syncthreads();
            dm_t[tx][ty] = dm[(j0+ty)*n+i0+tx];
            __syncthreads();
            for (i = 0; i < BLOCK; ++i) {
                for (j = 0; j < BLOCK; ++j) {
                    s += eri[((i0+i)*n+j0+j)*nn+k1] * dm_t[i][j];
                }
            }
        }
    }
    output[k1] = s;
}

```

This implementation is a simplified version that assumes the dimensions of the relevant tensors and arrays are integer multiples of the `BLOCK` size, which is 16. For a comprehensive implementation, one must include the code to check for unaligned elements near the boundary. Nonetheless, the core of the tiling algorithm with shared memory remains unchanged. Here, each tile is a 16×16 square matrix. The shared memory buffer `dm_t` is allocated to accommodate this size. To establish the mapping between the 2D shared memory and the CUDA threads, when launching this kernel, CUDA blocks should be configured to contain 16×16 threads

```
dim3 threads(16, 16);
```

Each thread within a CUDA block is responsible for reading one element from the global memory to fill the shared memory. It is crucial to maintain exclusive access to the shared memory while it is being modified. To achieve this, the `__syncthreads()` function is used to synchronize all threads in the block, ensuring that they reach the same point in execution. This synchronization prevents the shared memory from being accessed in write and read modes simultaneously.

By employing the tiling algorithm, the required global memory access for reading the `dm` array is reduced to $\frac{1}{256}n^4$. Consequently, the GPU kernel needs to access

only n^4 elements from the global memory for the `eri` tensor. When compared to `kernel_v2`, this approach approximately halves the total number of global memory accesses.

Unlike algorithm design for CPUs, GPU algorithms are better suited for a *throughput-oriented* approach. Although FLOP counting remains an important metric, it is often not the primary constraint on GPU kernel performance. Optimization efforts should focus on factors such as data access latency, the frequency of global memory accesses, and cache hit rates.

11.4.2 Numba CUDA JIT

Numba provides the option of JIT compilation to generate GPU kernels from Python functions. This method is the simplest way to develop a new kernel, even without the need to understand GPU parallelization model. For example, the GPU kernel of the `get_j` function can be created with the `numba.cuda.jit` decorator.

```
from numba import cuda
@cuda.jit(fastmath=True)
def naive_kernel(output, eri, dm):
    n = dm.shape[0]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                for l in range(n):
                    output[k,l] += eri[i,j,k,l] * dm[j,i]

def get_j(eri, dm):
    n = dm.shape[0]
    output = np.empty_like(dm)
    grids = ((n*n+255) // 256,)
    threads = (256,)
    naive_kernel[grids, threads](output, eri, dm)
    return output
```

The GPU kernel generated by JIT employs the following function signature [27]:

```
func[griddim, blockdim, stream, sharedmem](*arguments)
```

The kernel function is identical to the kernel launch syntax utilized in CUDA C++:

```
func<<<griddim, blockdim, sharedmem, stream>>>(*arguments)
```

The Numba JIT for regular Python code hides many details of GPU parallelization model, which generally lead to suboptimal performance. Achieving improved performance with Numba JIT is possible but requires to incorporate more GPU-specific languages and features into the Python JIT code. To utilize advanced CUDA features,

Numba offers comprehensive CUDA APIs through The `numba.cuda` module [20]. For instance, the atomic addition operation can be performed using the `cuda.atomic.add` function, and the `cuda.shared.array` function can be employed to allocate arrays in shared memory.

```
from numba import cuda
@cuda.jit(fastmath=True)
def _kernel_v1(output, eri, dm):
    n = dm.shape[0]
    ij = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if ij > n * n: return
    i, j = divmod(ij, n)
    for k in range(n):
        for l in range(n):
            cuda.atomic.add(output, (k,l), eri[i,j,k,l] * dm[j,i])

def get_j(eri, dm):
    n = dm.shape[0]
    output = np.zeros_like(dm)
    grids = ((n*n+255) // 256,)
    threads = (256,)
    _kernel_v1[grids, threads](output, eri, dm)
    return output

BLOCK = 16

@cuda.jit(fastmath=True)
def kernel_v3(output, eri, dm):
    n = dm.shape[0]
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    k, l = cuda.grid(2)
    dm_t = cuda.shared.array((BLOCK, BLOCK), dtype='f8')
    s = 0.
    for i0 in range(0, n, BLOCK):
        for j0 in range(0, n, BLOCK):
            cuda.syncthreads()
            dm_t[tx,ty] = dm[j0+ty,i0+tx]
            cuda.syncthreads()
            for i in range(BLOCK):
                for j in range(BLOCK):
                    s += eri[i0+i,j0+j,k,l] * dm_t[i,j]
    output[k,l] = s
```

```
def get_j_v3(eri, dm):
    n = dm.shape[0]
    assert n % BLOCK == 0
    output = np.empty_like(dm)
    grids = ((n**2+BLOCK**2-1) // BLOCK**2,)
    threads = (BLOCK, BLOCK)
    kernel_v3[grids, threads](output, eri, dm)
    return output
```

To be fair, compared to writing raw CUDA kernel code, using the Numba CUDA APIs in the Python JIT code does not reduce the complexity of GPU kernel development. It only simplifies the compilation configuration and the implementation of Python interface. From this perspective, implementing raw CUDA kernels using tools such as CuPy or PyCUDA presents a similar level of complexity.

11.4.3 CuPy custom kernel

CuPy, PyCUDA, and CUDA-Python are the tools that allow us to embed raw CUDA kernel in Python programs.

Among the three, CUDA-Python, provided by Nvidia, is less commonly used. CUDA-Python offers a comprehensive set of CUDA APIs, with naming conventions and API structures that are almost identical to those of the CUDA C++ APIs. Theoretically, it can accomplish nearly everything that CUDA C++ can do. However, CUDA-Python only exports CUDA API functions to Python, without adapting to Python language conventions. When using CUDA-Python, one needs to pay attention to data types, error code, and other details that are typically associated with C++ programming. The compilation configuration also requires a deep understanding of the CUDA binary and runtime framework. Using CUDA-Python alone does not simplify the development process compared to implementing everything in C++ and then exporting to Python via `ctypes` or `FFI`.

Both CuPy and PyCUDA offer a better balance between CUDA programming and Python integration. With CuPy and PyCUDA, one can just focus on the CUDA kernel code. These tools automatically handle the remaining tasks, including the compilation configuration and the Python interface. They provide APIs to access most of the essential CUDA features, though not as extensively as CUDA-Python. These APIs are located in the `cupy.cuda.runtime` module in CuPy and the `pycuda.driver` module in PyCUDA. For instance, the zero-copy memory access technique, which will be used in Section 11.4.4, requires a direct call to the APIs of the `cupy.cuda.runtime` module.

Let's consider the CuPy `RawKernel` as an example to explore how to integrate a raw CUDA kernel into a Python program. Although the API naming and package structure of PyCUDA differ from CuPy, their usage is quite similar. It is not difficult to implement the same functionality with PyCUDA based on the CuPy example.

Using the source code of a raw CUDA kernel developed in Section 11.4.1, we can initialize a `RawKernel` instance as follows:

```
kernel_v2 = cupy.RawKernel('''
extern "C" __global__
void _kernel_v2(double *output, double *eri, double *dm, long long n)
{
    size_t k1 = blockIdx.x * blockDim.x + threadIdx.x;
    size_t nn = n * n;
    if (k1 > nn) return;
    size_t i, j;
    double s = 0.;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            s += eri[(i*n+j)*nn+k1] * dm[j*n+i];
        }
    }
    output[k1] = s;
}
''', '_kernel_v2')

def get_j(eri, dm):
    n = dm.shape[0]
    eri = cupy.asarray(eri)
    dm = cupy.asarray(dm)
    output = cupy.empty_like(dm)
    grids = ((n*n+255) // 256,)
    threads = (256,)
    kernel_v2(grids, threads, (output, eri, dm, n))
    return output
```

The kernel function generated by CuPy has the following function signature:

```
func(griddim, blockdim, args, shared_mem=0, stream=None)
```

If you intend to use multiple streams to launch the kernel, the approach is similar to that used with other CuPy functions. You can either use a context manager to set the default stream for the custom kernel or explicitly pass a `Stream` instance to the kernel.

Unlike the Numba JIT, which automatically handles the data types of the arguments, we must explicitly define the data types of the arguments with respect to their counterparts in Python. This requirement is explained in the CuPy documentation [28]. For instance, the integer `n` is specified as the `long long` type to align with the Python integer.

You might be curious about the compiling and runtime performance of the embedded raw CUDA kernel. Let's address some common questions here:

When is the raw CUDA code compiled? The compilation of raw CUDA code is done lazily. It occurs the first time the `RawKernel` object is invoked.

Is the compilation performed every time we call the kernel? No, the kernel is compiled only once and cached during the same run. CuPy also caches the compiled PTX assembly code to speed up the compilation for future runs [14].

Which Python tool exhibits the best runtime performance among CuPy RawKernel, PyCUDA, and Numba? If the same algorithm is used and they are implemented in a reasonable manner, the performance of CuPy RawKernel, PyCUDA, and Numba with CUDA APIs is similar. While these tools offer different interfaces and language specifications for implementing CUDA kernels, the final GPU kernels are all generated by the nvrtc compiler. Their efficiency is almost identical. The differences mainly lie in the Python interfaces. The performance differences in the interfaces are minimal.

In some scenarios, utilizing CUDA features specialized for C/C++ can improve the interaction between the host code and the GPU kernel. However, CuPy and PyCUDA are limited to customizing and launching CUDA kernels. They do not support the integration of C/C++ functions executed on the CPU. To access the full spectrum of CUDA features, the `ctypes` library can be utilized.

11.4.4 Integrating CUDA kernels using `ctypes`

If you are comfortable with CUDA C++ programming, the `ctypes` module is always a favorable option to integrate GPU kernels into the Python project. The GPU applications can be developed independently from the Python project and compiled as a standard C/C++ shared library. As discussed in Chapter 8, C/C++ shared libraries can be dynamically loaded into Python using the `ctypes` module or other interfacing tools.

In this approach, a C function wrapper serves as a bridge between the GPU kernel and Python. For instance, we can add C function wrappers to the kernels developed in Section 11.4.1 as follows:

```
$ cat get_j.cu
#include <stdio.h>
#include <cuda_runtime.h>
__global__ void _kernel_v1(double *output, double *eri, double *dm, int n)
...
extern "C"
void kernel_v1(double *output, double *eri, double *dm, int n)
{
    int grids = (n*n+255)/256;
    int threads = 256;
    _kernel_v1<<<grids, threads>>>(output, eri, dm, n);
    cudaError_t err = cudaGetLastError();
    if(err != cudaSuccess)
        fprintf(stderr, "CUDA error %s\n", cudaGetErrorString(err));
}
```

```
}

extern "C"
void kernel_v2(double *output, double *eri, double *dm, int n)
{
    ...

extern "C"
void kernel_v3(double *output, double *eri, double *dm, int n)
{
    int grids = (n*n+255)/256;
    dim3 threads(16, 16, 1);
    _kernel_v3<<<grids, threads>>>(output, eri, dm, n);
    cudaError_t err = cudaGetLastError();
    if(err != cudaSuccess){
        fprintf(stderr, "CUDA error %s\n", cudaGetErrorString(err));
    }
}
```

Compiling the CUDA code with the following command produces the shared library `libget_j.so`.

```
$ nvcc -g -shared -Xcompiler=-fPIC -arch=sm_60 -o libget_j.so get_j.cu
```

The compilation flags `-shared` and `-Xcompiler=-fPIC` are essential for creating a library that can be dynamically loaded by `ctypes`. Atomic operations such as `atomicAdd` are not supported in old versions of CUDA Compute Capability. The `-arch=sm_60` flag (or higher versions of `sm`) is necessary to enable the `atomicAdd` function in `_kernel_v1`. We can then load the compiled shared library and invoke the C function `kernel_v1` using `ctypes`, which in turn launches the GPU kernel.

The remaining question in the `ctypes` interface is how to pass Python objects, particularly the pointers to array objects in GPU memory, to the C function. The GPU-based tensor libraries CuPy, PyTorch, and JAX employ different treatments for GPU memory pointers:

- The device address of a CuPy array object can be obtained through the `.data.ptr` attribute.
- The device address of a PyTorch tensor object can be accessed using the `.data_ptr()` method.
- JAX does not support such functionality and does not allow direct modification of its contents. GPU pointers are not available in JAX.

It should be noted that the device addresses provided by CuPy and PyTorch are integers. They must be cast to the appropriate pointer type before being passed to the underlying C functions. For example, in the interface function `get_j`, the addresses of CuPy arrays are transformed as follows:

```

gpu_ext = ctypes.CDLL('./libget_j.so')
def get_j(eri, dm):
    n = dm.shape[0]
    eri = cp.asarray(eri)
    dm = cp.asarray(dm)
    output = cp.zeros_like(dm)
    gpu_ext.kernel_v1(ctypes.c_void_p(output.data.ptr),
                      ctypes.c_void_p(eri.data.ptr),
                      ctypes.c_void_p(dm.data.ptr), ctypes.c_int(n))
    return output

```

In the `get_j` function, the NumPy array `eri` is converted into a CuPy array using the `cupy.asarray` function. This conversion transfers the array data to GPU memory. Subsequently, the GPU function `kernel_v1` reads the data from GPU memory. This two-step approach is inefficient as it introduces unnecessary write and read operations.

To improve the data transfer efficiency, one approach is to send the `eri` array using asynchronous data transfer before calling the `get_j` function. In fact, a more suitable approach for this scenario is the use of the CUDA zero-copy technique [29], also known as the direct memory access (DMA) technique. Zero-copy allows GPU threads to directly access the host memory. This eliminates the need to store data in GPU memory and thereby reduces the overhead of one write and one read operation for each element. However, the speed of zero-copy memory access is constrained by the bandwidth of NVLink or the PCIe bus [30,31]. If data needs to be accessed multiple times, caching data in GPU memory is a better option as GPU memory can provide much higher bandwidth.

To utilize the zero-copy technique, the host memory must be pinned and mapped. The special host memory can be allocated using the low-level CUDA APIs in the `cupy.cuda.runtime` module. Subsequently, this host memory is converted into a Python buffer object using the `cp.cuda.PinnedMemoryPointer` class, which can be assigned to an empty NumPy array. The following Python code snippet illustrates these steps:

```

def empty_mapped(shape, dtype=float, order='C'):
    nbytes = np.prod(shape) * np.dtype(dtype).itemsize
    mem = cp.cuda.PinnedMemoryPointer(
        cp.cuda.PinnedMemory(nbytes, cp.cuda.runtime.hostAllocMapped), 0)
    out = np.ndarray(shape, dtype=dtype, buffer=mem, order=order)
    return out

```

In the corresponding C function, the CUDA API `cudaHostGetDevicePointer` is required to bind the memory address on the GPU to the memory page on the host. The bound address can then be passed to the GPU kernel and utilized as a regular on-device address.

```
extern "C"
void kernel_v2_mapped(double *output, double *eri_cpu, double *dm, int n)
{
    double *eri_gpu;
    cudaError_t err = cudaHostGetDevicePointer(&eri_gpu, eri_cpu, 0);
    if(err != cudaSuccess){
        fprintf(stderr, "CUDA error %s\n", cudaGetStringError(err));
    }
    kernel_v2(output, eri_gpu, dm, n);
    err = cudaGetLastError();
    if(err != cudaSuccess){
        fprintf(stderr, "CUDA error %s\n", cudaGetStringError(err));
    }
}
```

Finally, in the Python function, we remove the CuPy array conversion code. The address of the memory-pinned NumPy array is now directly passed to the CUDA kernel.

```
def get_j_zero_copy(eri, dm):
    n = dm.shape[0]
    dm = cp.asarray(dm)
    output = cp.zeros_like(dm)
    gpu_ext.kernel_v2_mapped(
        ctypes.c_void_p(output.data.ptr),
        eri.ctypes,
        ctypes.c_void_p(dm.data.ptr), ctypes.c_int(n))
    return output

if __name__ == '__main__':
    n = 160
    rng = np.random.default_rng(seed=1)
    eri = empty_mapped((n, n, n, n))
    rng.random((n, n, n, n), out=eri)
    dm = np.random.rand(n, n)
    output = get_j_zero_copy(eri, dm)
```

To assess the performance of different computational kernels, we conducted benchmarks on `kernel_v1`, `kernel_v2`, and `kernel_v3`, along with their zero-copy variants, using a V100 GPU card with the system size set to $n = 160$. The results are detailed in Table 11.1. `kernel_v2` outperforms `kernel_v1` since it ensures coalesced memory access [32,33] for the `eri` tensor and eliminates the use of `atomicAdd`. Coalesced memory transaction is a process where consecutive threads access consecutive memory addresses simultaneously. `kernel_v3` achieves slightly better performance

Table 11.1 Performance of various `get_j` CUDA kernels.

	standard	zero-copy
<code>kernel_v1</code>	1.134	5.438
<code>kernel_v2</code>	1.088	0.480
<code>kernel_v3</code>	1.081	0.478

than `kernel_v2` due to its use of shared memory, which reduces the frequency of memory access. With the zero-copy optimization, `kernel_v2` and `kernel_v3` exhibit significant speed improvements over their non-zero-copy counterparts. This enhancement can be primarily attributed to the elimination of a data transfer step from the host memory to the GPU.

Zero-copy significantly decreases the performance of `kernel_v1`. In the `kernel_v1` implementation, each thread accesses a memory address that is not continuous in memory, resulting in un-coalesced memory access [33]. GPUs are optimized for coalesced memory access. When accessing data that is fully stored in GPU memory, such as the `eri` tensor, the impact of un-coalesced memory might not be that significant. In the case of zero-copy, the overhead of accessing un-coalesced memory is much higher since data needs to be transferred from the host memory, which incurs a high latency. It is particularly important to ensure coalesced memory access when utilizing the zero-copy technique.

Summary

In this chapter, we have introduced techniques for GPU programming in Python to accelerate computations. Libraries such as CuPy and PyTorch provide functionality similar to NumPy for managing arrays and accelerating linear algebra operations on the GPU. To enable GPU acceleration in a Python program, a straightforward method is to replace the NumPy code with corresponding functions or classes in CuPy or PyTorch.

If the performance of CuPy or PyTorch is insufficient, or if the desired functionalities are not available in these libraries, it may become necessary to customize GPU kernels and integrate them into Python programs. Numba JIT compilation offers a method to translate Python functions to GPU kernels. However, it is not straightforward to achieve fine control over GPU features in this approach. For more precise control of GPU capabilities, one can develop raw CUDA code and integrate it into the Python code using CuPy or PyCUDA, or utilize `ctypes` to incorporate a CUDA C++ project into a Python program.

To better utilize the computational power of GPUs, we can take advantage of the unique features of GPU architecture to enhance program performance. For instance, CUDA streams can be utilized to execute multiple tasks concurrently on the GPU. This capability enables the overlapping of computation and data transfer. Employing

page-locked memory can reduce the overhead of data movement between the host and the GPU. Additionally, the use of zero-copy technology can further minimize this data movement overhead. Proper use of shared memory within the GPU kernel can decrease the frequency of global memory accesses. The pattern of memory access significantly impacts performance, especially when accessing host memory via zero-copy technology. GPU computation benefits from coalesced memory access. This characteristic should be taken into account in the design of the data structure and algorithm.

References

- [1] NVIDIA Corporation, CUDA C programming guide - compute capabilities, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>, 2024.
- [2] NVIDIA Corporation, Your GPU compute capability, <https://developer.nvidia.com/cuda-gpus#compute>, 2024.
- [3] NVIDIA Corporation, CUDA compiler DRIVER nvcc - the CUDA compilation trajectory, <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#the-cuda-compilation-trajectory>, 2024.
- [4] NVIDIA Corporation, NVIDIA CUDA installation guide for Linux, <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/#installation>, 2024.
- [5] Amazon Web Services, Using the deep learning base AMI - configuring CUDA versions, <https://docs.aws.amazon.com/dlami/latest/devguide/tutorial-base.html>, 2024.
- [6] NVIDIA Corporation, Installing the NVIDIA container toolkit, <https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html>, 2024.
- [7] NVIDIA Corporation, CUDA binary utilities - cuobjdump, <https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#cuobjdump>, 2024.
- [8] PyTorch developers, Get started with PyTorch - start locally, <https://pytorch.org/get-started/locally/>, 2024.
- [9] CuPy Developers, Installing CuPy, <https://docs.cupy.dev/en/stable/install.html#installing-cupy>, 2024.
- [10] The JAX Authors, Installing JAX, <https://jax.readthedocs.io/en/latest/installation.html#pip-installation-gpu-cuda-installed-via-pip-easier>, 2024.
- [11] CuPy Developers, CuPy API reference - comparison table, <https://docs.cupy.dev/en/stable/reference/comparison.html>, 2024.
- [12] CuPy Developers, CuPy user guide - differences between CuPy and NumPy, https://docs.cupy.dev/en/stable/user_guide/difference.html, 2024.
- [13] CuPy Developers, CuPy user guide - memory management, https://docs.cupy.dev/en/latest/user_guide/memory.html, 2024.
- [14] CuPy Developers, CuPy user guide - performance best practices, https://docs.cupy.dev/en/stable/user_guide/performance.html#benchmarking, 2024.
- [15] NumPy Developers, NumPy 2.0.0 release notes - np.unique return_inverse shape for multi-dimensional inputs, <https://numpy.org/devdocs/release/2.0.0-notes.html#np-unique-return-inverse-shape-for-multi-dimensional-inputs>, 2024.
- [16] PyTorch Contributors, CUDA semantics - memory management, <https://pytorch.org/docs/stable/notebooks/cuda.html#memory-management>, 2024.

- [17] The JAX authors, How to think in JAX, https://jax.readthedocs.io/en/latest/notebooks/thinking_in_jax.html, 2024.
- [18] NVIDIA Corporation, CUDA C++ best practices guide - pinned memory, <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#pinned-memory>, 2024.
- [19] NVIDIA Corporation, CUDA toolkit documentation - API synchronization behavior, <https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html#api-sync-behavior>, 2024.
- [20] Numba Developers, Numba for CUDA GPUs, <https://numba.pydata.org/numba-doc/dev/cuda/index.html>, 2024.
- [21] CuPy Developers, CuPy user guide - user-defined kernels, https://docs.cupy.dev/en/stable/user_guide/kernel.html, 2024.
- [22] A. Klöckner, PyCUDA documentation, <https://documentation.tician.de/pycuda/index.html>, 2024.
- [23] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation, Parallel Computing 38 (3) (2012) 157–174, <https://doi.org/10.1016/j.parco.2011.09.001>, <https://www.sciencedirect.com/science/article/pii/S0167819111001281>.
- [24] NVIDIA Corporation, CUDA Python manual, <https://nvidia.github.io/cuda-python/>, 2024.
- [25] NVIDIA Corporation, CUDA C++ programming guide, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>, 2024.
- [26] NVIDIA Corporation, CUDA by example: an introduction to general-purpose gpu programming, <https://developer.nvidia.com/cuda-example>, 2024.
- [27] Numba Developers, Numba CUDA kernel API - dispatcher objects, <https://numba.pydata.org/numba-doc/dev/cuda-reference/kernel.html#dispatcher-objects>, 2024.
- [28] CuPy Developers, CuPy user guide - kernel arguments, https://docs.cupy.dev/en/stable/user_guide/kernel.html#kernel-arguments, 2024.
- [29] NVIDIA Corporation, CUDA C++ best practices guide - zero copy, <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#zero-copy>, 2024.
- [30] A. Li, S.L. Song, J. Chen, J. Li, X. Liu, N.R. Tallent, K.J. Barker, Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect, IEEE Transactions on Parallel and Distributed Systems 31 (1) (2020) 94–110, <https://doi.org/10.1109/TPDS.2019.2928289>.
- [31] S.W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, W.-m. Hwu, Large graph convolutional network training with GPU-oriented data communication architecture, Proceedings of the VLDB Endowment 14 (11) (2021) 2087–2100, <https://doi.org/10.14778/3476249.3476264>.
- [32] M. Harris, How to access global memory efficiently in CUDA C/C++ kernels, <https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/>, 2013.
- [33] NVIDIA Corporation, CUDA C++ programming guide - device memory accesses, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses>, 2024.

PART

Quantum chemistry applications with Python

3

Integral evaluation

12

Integral evaluation is a fundamental task in quantum chemistry calculations. To calculate integrals efficiently and accurately, Gaussian-type orbitals (GTO) are used as the basis functions in quantum chemistry programs. In the past, various algorithms have been developed for the computation of the GTO integrals, with a particular focus on computational efficiency.

Developing an efficient analytical integral program in compiled programming languages for GTO basis sets is challenging. The formula for analytical integral computation is inherently complex. Furthermore, many integral algorithms proposed in the literature incorporate various considerations to optimize efficiency, which further complicates the code implementation of the integral program. It is difficult to develop integral programs based solely on formulas and descriptions from the literature. Although there are some open-source integral libraries available for referencing, these libraries are typically intensively optimized. This optimization often makes understanding the open-source implementation quite challenging.

How can we utilize Python to simplify the development of integral programs? How can we optimize a Python integral program to approach the performance of those written in compiled languages? These questions are the main focus of this chapter.

We will begin by implementing the most basic analytical integral formulas in a straightforward manner to establish a correct reference version. Subsequently, we will gradually introduce optimization techniques to enhance the performance of the integral program. To achieve this, we will utilize techniques developed in previous chapters, including the code optimization schemes discussed in Chapter 9 and the code generation methods introduced in Chapter 5. In certain optimization procedures, we will need the knowledge of the NumPy array data structure and memoryviews, as discussed in Chapter 2, to enhance memory access efficiency. Additionally, we will explore the following technical questions in this chapter:

- Which integral formulas are easier to implement in Python? Which are more efficient?
- What performance can an integral program in Python achieve before resorting to a compiled language?
- How effective are Cython and Numba JIT compilation in integral programs? What tricks can be employed to optimize programs using these tools?
- What factors influence the efficiency of integrals? How can these factors be incrementally optimized?

Table 12.1 Notations used in GTO integral equations and their corresponding names in the Python code.

	Notations	Example Equations	Variables in code
Atom indices	A, B, C, D		
Electron coordinates	\mathbf{r}		
Nuclear coordinates	$\mathbf{R}_A, \mathbf{R}_B, \mathbf{R}_C, \mathbf{R}_D$	(12.16)	
	X_A, Y_A, Z_A, \dots		
Center of Gaussian products	$\mathbf{R}_P, \mathbf{R}_Q$	(12.23)	Rp, Rq
Relative coordinate vector	$\mathbf{R}_{AB}, \mathbf{R}_{PQ}, \mathbf{R}_{PA}$	(12.24), (12.33)	Rab, Rpq, Rpa
	X_{AB}, Y_{PQ}, Z_{PA}	(12.19), (12.28)	Xab, Ypq, Zpa
Gaussian function	G_{lm}, G_{tuv}, G_i	(12.1), (12.8), (12.9)	
Gaussian exponent	$\alpha_i, \alpha_j, \alpha_k, \alpha_l$	(12.1), (12.25)	ai, aj, ak, al
GTO angular momentum	l, l_i, l_j, \dots	(12.1)	l, li, lj
Atomic orbital	$\chi_i, \chi_j, \chi_k, \chi_l$	(12.6), (12.12)	

12.1 Analytical integral evaluation for Gaussian type orbitals

Gaussian type orbital (GTO) basis functions are the most commonly used basis functions in quantum chemistry programs. For the Hamiltonian in regular quantum chemistry molecular systems, the integrals with GTOs can be evaluated analytically. If implementing the analytical integral program in Python, computation performance is the main challenge. In this section, we will demonstrate how to implement the integral program. We will begin with the mathematical equations and iteratively optimize the program until the performance approaches that of the native C/C++ implementations.

12.1.1 Data structure for GTO basis

A primitive GTO centered at position \mathbf{R}_A and characterized by the Gaussian exponent α has the form:

$$G_{lm}(\mathbf{r}, \alpha, \mathbf{R}_A) = N(\alpha, l) Y_{lm}(\mathbf{r}_A) e^{-\alpha r_A^2}, \quad (12.1)$$

$$\mathbf{r}_A = \mathbf{r} - \mathbf{R}_A. \quad (12.2)$$

$N(\alpha, l)$ represents the normalization factor for the radial part $r^l e^{-\alpha r^2}$. Here, l is referred to as the angular momentum of the GTO function. $Y_{lm}(\mathbf{r})$ denotes the *real-valued solid harmonics*. Additional notation conventions for Gaussian basis functions and integrals, along with the variables used in the Python code throughout this chapter, are summarized in Table 12.1.

The normalization factor $N(\alpha, l)$ in Eq. (12.1) can be calculated as:

$$N(\alpha, l) = \left(\int_0^\infty (r^l e^{-\alpha r^2})^2 r^2 dr \right)^{-1/2}. \quad (12.3)$$

By comparing to the `scipy.special.gamma` function,

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt, \quad (12.4)$$

we derive the value of the normalization factor

$$N(\alpha, l) = \left(\frac{1}{2(2\alpha)^{l+3/2}} \Gamma(l + 3/2) \right)^{-1/2}. \quad (12.5)$$

The radial part of the primitive GTOs can be combined with contraction coefficients C_n to define the contracted GTO basis functions:

$$\chi(\mathbf{r}) = \sum_n C_n G_{lm}(\mathbf{r}, \alpha_n, \mathbf{R}_A). \quad (12.6)$$

The contracted GTOs that share the same radial part and angular momentum form a *shell*. In quantum chemistry software, the integral engine typically processes GTO shells as the smallest unit, rather than individual GTO basis functions within the shell. This approach is efficient because integrals for GTOs within the same shell share many common intermediates, which can be reused during computation. The most basic integral function should accept a set of GTO shells as input arguments and output all the integrals within those shells into a memory buffer.

With this background, we can design the `basis` module, which provides the basic data type for the contracted GTO basis. The minimal data structure `CGTO` can be defined using `dataclass` APIs:

```
@dataclass
class CGTO:
    angular_momentum: int
    exponents: np.ndarray
    coefficients: np.ndarray
    coordinates: np.ndarray
```

To represent the basis data for the `CGTO` class, we can employ the configuration in YAML format. For instance, the following configuration represents the 6-31G basis of a Hydrogen atom at the position (0., 0., 0.).

```
- angular_momentum: 0
exponents:
- 18.7311370
- 2.8253937
- 0.6401217
coefficients:
- 0.03349460
- 0.23472695
- 0.81375733
```

```

coordinates:
- 0.
- 0.
- 0.
- angular_momentum: 0
exponents:
- 0.1612778
coefficients:
- 1.0000000
coordinates:
- 0.
- 0.
- 0.

```

This configuration includes two shells of contracted GTOs, which can be used to define two CGTO objects. The instantiation methods can be implemented as follows:

```

@dataclass
class CGTO:
    ... # Code omitted for brevity
    @classmethod
    def from_dict(cls, dic: dict):
        angular_momentum = int(dic['angular_momentum'])
        exponents = np.asarray(dic['exponents'], dtype=float)
        coefficients = np.asarray(dic['coefficients'], dtype=float)
        coordinates = np.asarray(dic['coordinates'], dtype=float)
        assert angular_momentum >= 0
        assert all(exponents > 0)
        assert exponents.ndim == 1
        assert coefficients.ndim == 1
        assert coordinates.shape == (3,)
        return cls(angular_momentum, exponents, coefficients, coordinates)

    @classmethod
    def from_yaml(cls, conf: str) -> List[CGTO]:
        '''Creates CGTOs from yaml configuration'''
        conf = yaml.load(conf, Loader=yaml.SafeLoader)
        return [cls.from_dict(x) for x in conf]

```

The Basis Set Exchange [1] (BSE) database is a standard resource for obtaining quantum chemistry basis sets. It is useful to implement a `from_bse` method for the CGTO class based on the BSE database.

```

from functools import lru_cache
import basis_set_exchange as bse

```

```

@dataclass
class CGTO:
    ... # Code omitted for brevity
    @classmethod
    def from_bse(cls, name, element, coordinates) -> List[CGTO]:
        '''Creates CGTOs from BSE database'''
        return [cls.from_dict({'coordinates': coordinates, **basis})
                for basis in bse_basis(name, element)]

@lru_cache(1000)
def bse_basis(name: str, element: str) -> List[dict]:
    '''Reads and converts BSE GTO basis'''
    data = bse.get_basis(name, elements=[element]) # (1)
    basis = []
    for elem_basis in data['elements'].values():
        for raw_basis in elem_basis['electron_shells']:
            exps = np.array(raw_basis['exponents']).astype(float)
            coefs = np.array(raw_basis['coefficients']).astype(float)
            ls = raw_basis['angular_momentum']
            if len(ls) == len(coefs): # SP basis
                for l, c in zip(ls, coefs):
                    basis.append({
                        'angular_momentum': int(l),
                        'exponents': exps,
                        'coefficients': c,
                    })
            else:
                for c in coefs:
                    basis.append({
                        'angular_momentum': int(ls[0]),
                        'exponents': exps,
                        'coefficients': c,
                    })
    return basis

```

BSE database can be accessed via the Python package `basis-set-exchange`. In the `bse_basis` function, we convert all BSE basis data to data types that are consistent with the attributes of the `CGTO` class. The BSE API `bse.get_basis` in line (1) returns a nested Python dict object, with all data stored in string format. We then use the `CGTO.from_bse` method to create a list of `CGTO` objects, representing all GTO shells of an atom at a particular point.

For contracted GTOs, the normalization factor in Eq. (12.5) can be computed as follows:

```

def gaussian_int(n, alpha):
    r'''int_0^inf x^n exp(-alpha x^2) dx'''
    assert n >= 0
    n1 = (n + 1) * .5
    return scipy.special.gamma(n1) / (2. * alpha**n1)

def gto_norm(l, expt):
    '''Radial part normalization'''
    assert l >= 0
    norm = (gaussian_int(l*2+2, 2*expt)) ** -.5
    # Racah normalization, assuming angular part is normalized to unity
    norm *= ((2*l+1)/(4*np.pi))**.5
    return norm

@dataclass
class CGTO:
    ... # Code omitted for brevity
    @property
    def norm_coefficients(self):           # (1)
        return gto_norm(self.momentum, self.exponents) * self.coefficients

```

As indicated by the attribute `norm_coefficients` in line (1), the normalization factor is then combined with the contraction coefficients, forming a compound coefficient to scale primitive GTO integrals.

The spherical GTOs in Eq. (12.1) can be expressed using Cartesian GTO functions:

$$G_{lm}^{\text{sph}}(\mathbf{r}, \alpha, \mathbf{R}_A) = \sum_{tuv} T_{lm,tuv} G_{tuv}^{\text{cart}}(\mathbf{r}, \alpha, \mathbf{R}_A). \quad (12.7)$$

A Cartesian GTO can be factorized into the products of three Cartesian components

$$G_{tuv}(\mathbf{r}, \alpha, \mathbf{R}_A) = x_A^t y_A^u z_A^v e^{-\alpha r_A^2} = G_t(x) G_u(y) G_v(z), \quad (12.8)$$

$$G_t(x) = x_A^t e^{-\alpha x_A^2}. \quad (12.9)$$

This factorization simplifies the evaluation of integrals for Cartesian GTOs compared to spherical-harmonic GTOs. Integrals are typically computed first on Cartesian GTOs and then transformed into spherical-harmonic GTOs. The transformation coefficients $T_{lm,tuv}$ are documented in various literature sources [2,3], which can be precomputed and cached. As a linear transformation, Eq. (12.7) is not a significant challenge in the integral evaluation program. We will omit this transformation in subsequent discussions and instead focus on integrals with Cartesian GTOs.

The indices t, u, v in Eq. (12.8) represent the angular momentum l of the Cartesian GTO function:

$$t + u + v = l. \quad (12.10)$$

They are integers greater than or equal to zero. In each GTO shell, this constraint results in $\frac{1}{2}(l+1)(l+2)$ basis functions. This count is provided by the `n_cart` function in the `basis` module. The $\frac{1}{2}(l+1)(l+2)$ Cartesian GTOs in each shell are arranged in lexicographical order. In this order, the three Cartesian components (t, u, v) can be accessed programmatically, as shown by the `iter_cart_xyz` function below

```
def n_cart(l):
    return (l+1) * (l+2) // 2

def iter_cart_xyz(l):
    for t in reversed(range(l+1)):
        for u in reversed(range(l+1-t)):
            v = l - t - u
            yield t, u, v
```

For example, the 6 Cartesian functions in a d -type GTO shell are ordered as

```
In [1]: for t, u, v in iter_cart_xyz(2):
    print(f'x^{t}y^{u}z^{v}')
```

Out[1]:

```
x^2y^0z^0
x^1y^1z^0
x^1y^0z^1
x^0y^2z^0
x^0y^1z^1
x^0y^0z^2
```

To simplify the code for indexing the location of a GTO within the basis functions, we introduced the helper function `gto_offsets`:

```
def gto_offsets(gtos):
    '''Offsets are the position of the first GTO function for each GTO
    shell inside all GTO basis functions. The last element of the offsets
    is the total number of basis functions.
    ...
    dims = [n_cart(b.angular_momentum) for b in gtos]
    return np.append(0, np.cumsum(dims))
```

In addition to the fundamental class `CGTO` for integral evaluation, we introduce the `Molecule` class to manage the geometry and basis sets of a molecule. The `Molecule.assign_basis` method generates all GTO shells of the molecule for the specified basis set.

```

@dataclass
class Molecule:
    elements: List[str]
    coordinates: np.ndarray

    @classmethod
    def from_xyz(cls, xyz: str):
        elements = []
        coordinates = []
        for line in xyz.splitlines():
            line = line.strip()
            if not line or line[0] == '#':
                continue
            elements.append(line.split()[0])
            coordinates.append(line.split()[1:4])
        # In atomic unit
        coordinates = np.array(coordinates).astype(float) / 0.52917721
        assert coordinates.shape[1] == 3
        return cls(elements=elements, coordinates=coordinates)

    def assign_basis(self, basis_set: Dict[str, str]) -> List[CGTO]:
        """Creates a list of CGTOs for the molecule in xyz geometry
        ...
        gtos = []
        for elem, r in zip(self.elements, self.coordinates):
            gtos.extend(CGTO.from_bse(basis_set[elem], elem, r))
        return gtos

```

12.1.2 Basic types for analytical GTO integrals

Quantum chemistry calculations require the evaluation of various types of integrals. The following two types are particularly important, as other integrals can be derived or transformed from these foundational types.

- One-electron overlap integrals

$$\langle \chi_i | \chi_j \rangle = \int \chi_i(\mathbf{r}) \chi_j(\mathbf{r}) d^3\mathbf{r}. \quad (12.11)$$

- Two-electron Coulomb repulsion integrals (ERI)

$$(\chi_i \chi_j | \chi_k \chi_l) = \int \chi_i(\mathbf{r}_1) \chi_j(\mathbf{r}_1) \frac{1}{r_{12}} \chi_k(\mathbf{r}_2) \chi_l(\mathbf{r}_2) d^3\mathbf{r}_1 d^3\mathbf{r}_2. \quad (12.12)$$

For example, the integral for the kinetic operator can be derived from overlap integrals

$$T_{ij} = -\frac{1}{2}\langle \chi_i | \nabla^2 | \chi_j \rangle = \frac{1}{2} \sum_{t \in x,y,z} \langle \nabla_t \chi_i | \nabla_t \chi_j \rangle = T_{ij}^x + T_{ij}^y + T_{ij}^z. \quad (12.13)$$

The derivative of a primitive Cartesian GTO results in two primitive GTOs

$$\frac{\partial}{\partial x} G_{i_x, i_y, i_z}(\mathbf{r}, \alpha) = -2\alpha G_{i_x+1, i_y, i_z}(\mathbf{r}, \alpha) + i G_{i_x-1, i_y, i_z}(\mathbf{r}, \alpha). \quad (12.14)$$

Thus, each term in Eq. (12.13) leads to

$$\begin{aligned} T_{ij}^x &= 2\alpha_i \alpha_j \langle G_{i_x+1, i_y, i_z} | G_{j_x+1, j_y, j_z} \rangle - i_x \alpha_j \langle G_{i_x-1, i_y, i_z} | G_{j_x+1, j_y, j_z} \rangle \\ &\quad - j_x \alpha_i \langle G_{i_x+1, i_y, i_z} | G_{j_x-1, j_y, j_z} \rangle + \frac{i_x j_x}{2} \langle G_{i_x-1, i_y, i_z} | G_{j_x-1, j_y, j_z} \rangle. \end{aligned} \quad (12.15)$$

Another example is the integral for the nuclear attraction operator, which is defined as:

$$V_{\text{Nuc}} = \sum_A \frac{-Z_A}{|\mathbf{r} - \mathbf{R}_A|} \quad (12.16)$$

where Z_A represents the nuclear charge of atom A . By introducing two very compact s -type GTOs G_s , we can employ the ERI formula (12.12) to simulate the interaction between two GTOs and a point unit charge at \mathbf{R}_A . This approach converts the nuclear attraction integrals into

$$V_{\text{Nuc},ij} = \sum_A -Z_A (\chi_i \chi_j | G_s(\mathbf{R}_A) G_s(\mathbf{R}_A)). \quad (12.17)$$

The core of an integral program is the computation of overlap integrals and electron repulsion integrals. In the subsequent sections, we will implement the integral program using various analytical integral algorithms. How can we develop and optimize a high-performance integral program in Python? We can follow the steps outlined below:

1. Derive the necessary formulas, specifically the recurrence relations.
2. Translate these integral formulas into Python code using the dynamic programming (DP) approach in a recursive style.
3. Create tests based on the recursive DP code to ensure reproducibility.
4. Convert the recursive DP code into a version that employs nested for-loop iterations.
5. Optimize the hotspots of the program using compilation techniques.
6. Optimize the program for specific hardware or architecture.

12.1.3 Recurrence relations and the dynamic programming implementation

Analytical integrals in quantum chemistry often involve the use of recurrence relations (RR). These formulas provide an efficient method to compute integrals by reusing previously calculated values. In this section, we will focus on the step-by-step development of a program to compute overlap integrals and ERIs using RR formulas. The mathematical derivation of these RR formulas is beyond the scope of this book. For a detailed derivation of these integral formulas, readers can refer to other quantum chemistry literature, such as the book *Molecular Electronic Structure Theory* by Helgaker [4], and the paper *Efficient Electronic Integrals and their Generalized Derivatives for Object Oriented Implementations of Electronic Structure Calculations* by Flocke and Lotrich [5].

12.1.3.1 Overlap integrals between two primitive Cartesian GTOs

An overlap integral can be factorized into the products of three Cartesian components

$$\langle G_{i_x i_y i_z} | G_{j_x j_y j_z} \rangle = S_{i_x j_x} S_{i_y j_y} S_{i_z j_z}. \quad (12.18)$$

The RR formula for an individual Cartesian component is

$$S_{i_x, j_x} = X_{AB} S_{i_x, j_x - 1} + S_{i_x + 1, j_x - 1}. \quad (12.19)$$

When $j_x = 0$, a different RR can be used

$$S_{i_x, 0} = X_{PA} S_{i_x - 1, 0} + \frac{i_x - 1}{2\alpha_{ij}} S_{i_x - 2, 0}. \quad (12.20)$$

The starting point and boundary conditions for the RR are

$$S_{00} = \sqrt{\frac{\pi}{\alpha_{ij}}} e^{-\theta_{ij} X_{AB}^2}, \quad (12.21)$$

$$S_{ij} = 0, \quad \text{if } i < 0 \text{ or } j < 0. \quad (12.22)$$

In these formulas, \mathbf{R}_A and \mathbf{R}_B are the centers of the bra and ket GTOs, respectively. The *center-of-charge coordinates* \mathbf{R}_P are calculated as

$$\mathbf{R}_P = \frac{\alpha_i \mathbf{R}_A + \alpha_j \mathbf{R}_B}{\alpha_{ij}}. \quad (12.23)$$

The separation vector between two centers is obtained as

$$\mathbf{R}_{AB} = \mathbf{R}_A - \mathbf{R}_B. \quad (12.24)$$

The term X_{AB} and X_{PA} represent a Cartesian direction of the separation vectors \mathbf{R}_{AB} and \mathbf{R}_{PA} , respectively. The parameters α_{ij} and θ_{ij} are calculated as

$$\alpha_{ij} = \alpha_i + \alpha_j, \quad (12.25)$$

$$\theta_{ij} = \frac{\alpha_i \alpha_j}{\alpha_{ij}}. \quad (12.26)$$

We can straightforwardly convert the RR formula into the overlap integral code using the top-down approach of DP. First, we create a recursive function `get_S` that takes two arguments, corresponding to the RR formula (12.21).

```
def get_S(i: int, j: int):
    if j == 0:
        return Rpa * get_S(i-1, j) + (i-1)/(2*a[i]) * get_S(i-2, j)
    return get_S(i+1, j-1) + Rab * get_S(i, j-1)
```

The `get_S` function computes the three Cartesian components simultaneously. Next, we incorporate the starting point and boundary conditions into the recursive DP function.

```
@lru_cache(1000)
def get_S(i: int, j: int):
    if i < 0 or j < 0:
        return 0.
    if i == j == 0:
        return (np.pi/a[i])**.5 * np.exp(-theta_ij*Rab**2)
    if j == 0:
        return (i-1)/(2*a[i]) * get_S(i-2, j) + Rpa * get_S(i-1, j)
    return Rab * get_S(i, j-1) + get_S(i+1, j-1)
```

Within the recursion, some of the DP states may be reached multiple times. To prevent the recomputation of intermediate states, we can memoize these states using the `lru_cache` function.

Based on this DP function, the overlap integral can be calculated according to Eq. (12.18).

```
for i, (ix, iy, iz) in enumerate(iter_cart_xyz(lj)):
    for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
        S[i,j] = get_S(ix,jx)[0] * get_S(iy,jy)[1] * get_S(iz,jz)[2]
```

Here, the indices 0, 1, and 2 represent the three Cartesian directions. To iterate over all Cartesian GTOs in the bra and ket shells, we utilize the `iter_cart_xyz` function to generate their Cartesian powers (`ix, iy, iz`) and (`jx, jy, jz`).

Combining all the components mentioned above, we obtain a function that computes the overlap integral between a pair of primitive Cartesian GTOs.

```
def primitive_overlap(lj: int, lk: int, ai: float, aj: float,
                     Ra: np.ndarray, Rb: np.ndarray) -> np.ndarray:
    aij = ai + aj
    Rab = Ra - Rb
    Rp = (ai * Ra + aj * Rb) / aij
```

```

Rpa = Rp - Ra
theta_ij = ai * aj / aij

@lru_cache(1000)
def get_S(i: int, j: int):
    if i < 0 or j < 0:
        return 0
    if j > 0:
        return get_S(i+1, j-1) + Rab * get_S(i, j-1)
    if i > 1:
        return Rpa * get_S(i-1, j) + (i-1)/(2*aij) * get_S(i-2, j)
    if i == 1:
        return Rpa * get_S(i-1, j)
    return (np.pi/aij)**.5 * np.exp(-theta_ij * Rab**2)

nfi = n_cart(li)
nfj = n_cart(lj)
overlap = np.zeros((nfi, nfj))
for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
    for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
        overlap[i,j] = (get_S(ix,jx)[0] * get_S(iy,jy)[1] *
                         get_S(iz,jz)[2])
return overlap

```

12.1.3.2 ERIs with McMurchie-Davidson algorithm

Using the McMurchie-Davidson (MD) algorithm, the four-index ERI tensor can be computed through tensor contraction.

$$\langle \chi_i \chi_j | \chi_k \chi_l \rangle = \frac{2\pi^{5/2}}{\alpha_{ij}\alpha_{kl}\sqrt{\alpha_{ij} + \alpha_{kl}}} \sum_{tuv} E_{tuv}^{ij} \sum_{efg} (-1)^{e+f+g} E_{efg}^{kl} R_{t+e,u+f,v+g}^0. \quad (12.27)$$

The RR formula to compute the R tensor is given as follows:

$$R_{tuv}^n = X_{PQ} R_{t-1,u,v}^{n+1} + (t-1) R_{t-2,u,v}^{n+1}, \quad (12.28)$$

$$R_{tuv}^n = Y_{PQ} R_{t,u-1,v}^{n+1} + (u-1) R_{t,u-2,v}^{n+1}, \quad (12.29)$$

$$R_{tuv}^n = Z_{PQ} R_{t,u,v-1}^{n+1} + (v-1) R_{t,u,v-2}^{n+1}. \quad (12.30)$$

The starting point and boundary conditions are

$$R_{000}^n = (-2\theta)^n F_n(\theta R_{PQ}^2), \quad (12.31)$$

$$R_{tuv}^n = 0 \quad \text{if } t < 0 \text{ or } u < 0 \text{ or } v < 0, \quad (12.32)$$

where X_{PQ} , Y_{PQ} , and Z_{PQ} represent the three Cartesian directions of the vector \mathbf{R}_{PQ} , which is the separation between the two center-of-charge coordinates

$$\mathbf{R}_{PQ} = \mathbf{R}_P - \mathbf{R}_Q, \quad (12.33)$$

$$\mathbf{R}_P = \frac{\alpha_i \mathbf{R}_A + \alpha_j \mathbf{R}_B}{\alpha_{ij}}, \quad (12.34)$$

$$\mathbf{R}_Q = \frac{\alpha_k \mathbf{R}_C + \alpha_l \mathbf{R}_D}{\alpha_{kl}}. \quad (12.35)$$

θ in Eq. (12.31) is defined as

$$\theta = \frac{\alpha_{ij}\alpha_{kl}}{\alpha_{ij} + \alpha_{kl}}. \quad (12.36)$$

In Eq. (12.31), F_n represents the order- n Boys function

$$F_n(x) = \int_0^1 t^{2n} e^{-xt^2} dt. \quad (12.37)$$

The program for computing the Boys function will be discussed in Section 12.2.2.

Similar to the method used for overlap integrals, we can create a recursive DP function `get_R` that applies the RR formulas and boundary conditions. This recursive function takes four integers as inputs, corresponding to the subscripts t , u , v , and the order index n of the R tensor.

```
def get_R_tensor(l, theta, Rpq):
    @lru_cache(4000)
    def get_R(n, t, u, v):
        if t < 0 or u < 0 or v < 0:
            return 0.
        elif t > 0:
            return ((t-1) * get_R(n+1, t-2, u, v) +
                    Rpq[0] * get_R(n+1, t-1, u, v))
        elif u > 0:
            return ((u-1) * get_R(n+1, t, u-2, v) +
                    Rpq[1] * get_R(n+1, t, u-1, v))
        elif v > 0:
            return ((v-1) * get_R(n+1, t, u, v-2) +
                    Rpq[2] * get_R(n+1, t, u, v-1))
        # t == u == v == 0:
        return (-2*theta)**n * boys(n, theta*np.array(Rpq).dot(Rpq))

    Rt = np.zeros((l+1, l+1, l+1))
    for t in range(l+1):
        for u in range(l+1):
            for v in range(l+1):
```

```
Rt[t,u,v] = get_R(0, t, u, v)
return Rt
```

The E tensor in Eq. (12.27) is a product of three Cartesian components.

$$E_{tuv}^{ij} = E_t^{i_x j_x} E_u^{i_y j_y} E_v^{i_z j_z}. \quad (12.38)$$

To derive each Cartesian component, we can use the following RR

$$E_t^{ij} = \frac{1}{2\alpha_{ij}t} (i E_{t-1}^{i-1,j} + j E_{t-1}^{i,j-1}), \quad (12.39)$$

$$E_0^{ij} = X_{PB} E_0^{i,j-1} + E_1^{i,j-1}, \quad (12.40)$$

$$E_0^{ij} = X_{PA} E_0^{i-1,j} + E_1^{i-1,j}, \quad (12.41)$$

$$E_0^{00} = \sqrt{\frac{\pi}{\alpha_{ij}}} e^{-\theta_{ij} X_{AB}^2}, \quad (12.42)$$

$$E_t^{ij} = 0 \quad \text{if } i < 0 \text{ or } j < 0 \text{ or } t < 0 \text{ or } t > l_i + l_j. \quad (12.43)$$

These RRs can be translated into a recursive DP function.

```
def get_E_cart_components(li, lj, ai, aj, Ra, Rb):
    aij = ai + aj
    Rab = Ra - Rb
    Rp = (ai * Ra + aj * Rb) / aij
    Rpa = Rp - Ra
    Rpb = Rp - Rb
    theta_ij = ai * aj / aij

    @lru_cache(1000)
    def get_E(i, j, t):
        if t < 0 or i < 0 or j < 0 or t > li+lj:
            return 0.
        if t > 0:
            return (i*get_E(i-1,j,t-1) + j*get_E(i,j-1,t-1)) / (2*aij*t)
        if j > 0: # t = 0
            return Rpb * get_E(i,j-1,0) + get_E(i,j-1,1)
        if i > 0: # t = 0
            return Rpa * get_E(i-1,j,0) + get_E(i-1,j,1)
        # i == j == t == 0
        return np.exp(-theta_ij * Rab**2)

    E_cart = np.zeros((li+1, lj+1, li+lj+1, 3))
    for i in range(li+1):
        for j in range(lj+1):
            for t in range(li+lj+1):
```

```

    E_cart[i,j,t] = get_E(i, j, t)
    return E_cart

```

We then use the Cartesian components to construct the E tensor (12.38)

```

def get_E_tensor(li, lj, ai, aj, Ra, Rb):
    E_cart = get_E_cart_components(li, lj, ai, aj, Ra, Rb)
    Ex = E_cart[:, :, :, 0]
    Ey = E_cart[:, :, :, 1]
    Ez = E_cart[:, :, :, 2]
    ix, iy, iz = np.array(list(iter_cart_xyz(li))).T
    jx, jy, jz = np.array(list(iter_cart_xyz(lj))).T
    return np.einsum('ijx,ijy,ijz->ijxyz', Ex[ix[:,np.newaxis],jx],
                    Ey[iy[:,np.newaxis],jy], Ez[iz[:,np.newaxis],jz])

```

After deriving the E and R tensors, we can build the ERI tensor (12.27) with the tensor contraction code:

```

from itertools import product

def primitive_ERI(li, lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    aij = ai + aj
    akl = ak + al
    theta = aij * akl / (aij + akl)
    Rp = (ai * Ra + aj * Rb) / aij
    Rq = (ak * Rc + al * Rd) / akl
    Rpq = Rp - Rq
    Rt = get_R_tensor(li+lj+lk+ll, theta, Rpq)
    lij1 = li + lj + 1
    lk11 = lk + ll + 1
    Rt2 = np.zeros((lij1,lij1,lij1, lk11,lk11,lk11))
    for e, f, g in product(range(lk11), range(lk11), range(lk11)):
        Rt2[:, :, :, e, f, g] = (-1)**(e+f+g) * Rt[e:e+lij1,f:f+lij1,g:g+lij1]

    Etab = get_E_tensor(li, lj, ai, aj, Ra, Rb)
    Etcd = get_E_tensor(lk, ll, ak, al, Rc, Rd)
    eri = (2*np.pi**2.5/(aij*akl*(aij+akl)**.5) *
           np.einsum('abtuv,tuvefg,cdefg->abcd', Etab, Rt2, Etcd))
    return eri

```

In this example, the `itertools.product` function is utilized to generate a nested loop structure.

12.1.3.3 ERIs with Obara-Saika algorithm

In the Obara-Saika (OS) algorithm, the three Cartesian components of the Gartesian GTO are explicitly encoded in the four-center ERI tensor, resulting in a 12-

dimensional tensor:

$$(\chi_i \chi_j | \chi_k \chi_l) \rightarrow g_{i_x i_y i_z; j_x j_y j_z; k_x k_y k_z; l_x l_y l_z}. \quad (12.44)$$

The OS algorithm begins with the Boys function and proceeds with three steps of RRs: vertical RR (VRR), electron-transfer RR (TRR), and horizontal RR (HRR). By sequentially executing these steps, we obtain the 12-dimensional tensor, which can be then transformed into the four-index ERI tensor.

The first step is the VRR which increases the angular momentum of the first center (i_x, i_y, i_z). The equation for increasing the power i_x is

$$\begin{aligned} g_{i_x, i_y, i_z; 000; 000; 000}^n &= X_P A g_{i_x-1, i_y, i_z; 000; 000; 000}^n - \frac{\theta}{\alpha_{ij}} X_P Q g_{i_x-1, i_y, i_z; 000; 000; 000}^{n+1} \\ &+ \frac{i_x - 1}{2\alpha_{ij}} \left(g_{i_x-2, i_y, i_z; 000; 000; 000}^n - \frac{\theta}{\alpha_{ij}} g_{i_x-2, i_y, i_z; 000; 000; 000}^{n+1} \right). \end{aligned} \quad (12.45)$$

Similar VRR equations can be derived to increase i_y and i_z . The starting point for the VRR is the Boys function

$$g_{000; 000; 000; 000}^n = \exp(-\theta_{ij} R_{AB}^2 - \theta_{kl} R_{CD}^2) F_n(\theta R_{PQ}^2). \quad (12.46)$$

These VRR equations can be implemented as a DP function, which takes one input argument for the order n in Eq. (12.45), along with three integer inputs representing the three Cartesian powers of the first center. The remaining nine subscripts of g are omitted as they are all zero in the VRR equations.

```
def primitive_ERI(l1, l2, l3, l4, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    aij = ai + aj
    Rp = (ai * Ra + aj * Rb) / aij
    theta_ij = ai * aj / aij
    Rab = Ra - Rb

    akl = ak + al
    Rq = (ak * Rc + al * Rd) / akl
    theta_kl = ak * al / akl
    Rcd = Rc - Rd

    Rpq = Rp - Rq
    theta = aij * akl / (aij + akl)
    theta_r2 = theta * Rpq.dot(Rpq)
    Kabcd = 2*np.pi**2.5/(aij*akl*(aij+akl)**.5)
    Kabcd *= np.exp(-theta_ij*Rab.dot(Rab) - theta_kl*Rcd.dot(Rcd))
    Xpq, Ypq, Zpq = Rpq
    Xpa, Ypa, Zpa = Rp - Ra
```

```

@lru_cache(10000)
def vrr(n, ix, iy, iz):
    if iz > 0:
        val = Zpa*vrr(n, ix, iy, iz-1)
        val -= theta/aij*Zpq*vrr(n+1, ix, iy, iz-1)
    if iz > 1:
        val += (iz-1)*.5/aij * (vrr(n, ix, iy, iz-2) -
                                   theta/aij*vrr(n+1, ix, iy, iz-2))
    return val
    if iy > 0:
        val = Ypa*vrr(n, ix, iy-1, iz)
        val -= theta/aij*Ypq*vrr(n+1, ix, iy-1, iz)
    if iy > 1:
        val += (iy-1)*.5/aij * (vrr(n, ix, iy-2, iz) -
                                   theta/aij*vrr(n+1, ix, iy-2, iz))
    return val
    if ix > 0:
        val = Xpa*vrr(n, ix-1, iy, iz)
        val -= theta/aij*Xpq*vrr(n+1, ix-1, iy, iz)
    if ix > 1:
        val += (ix-1)*.5/aij * (vrr(n, ix-2, iy, iz) -
                                   theta/aij*vrr(n+1, ix-2, iy, iz))
    return val
return Kabcd * boys(n, theta*Rpq.dot(Rpq))

```

In the second step we transfer the Cartesian powers from the first electron (the first center) to the second electron (the third center) using the TRR. The TRR equation for the x direction is given by:

$$\begin{aligned}
& g_{i_x, i_y, i_z; 000; k_x, k_y, k_z; 000} \\
&= \frac{i_x}{2\alpha_{kl}} g_{i_x-1, i_y, i_z; 000; k_x-1, k_y, k_z; 000} + \frac{k_x - 1}{2\alpha_{kl}} g_{i_x, i_y, i_z; 000; k_x-2, k_y, k_z; 000} \\
&\quad - \frac{\alpha_{ij}}{\alpha_{kl}} g_{i_x+1, i_y, i_z; 000; k_x-1, k_y, k_z; 000} \\
&\quad - \frac{\alpha_j X_{AB} + \alpha_l X_{CD}}{\alpha_{kl}} g_{i_x, i_y, i_z; 000; k_x-1, k_y, k_z; 000}.
\end{aligned} \tag{12.47}$$

Similar TRR equations exist for the y and z directions. When the Cartesian powers reach the boundary where $k_x = k_y = k_z = 0$, we arrive at the starting point of the TRR, which corresponds to the results of the VRR formula in Eq. (12.45). The recursive DP function for the TRR equations is

```

def primitive_ERI(lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    ... # Code omitted for brevity

```

```

Xab, Yab, Zab = Rab
Xcd, Ycd, Zcd = Rcd

@lru_cache(10000)
def trr(ix, iy, iz, kx, ky, kz):
    if kz > 0:
        val = -(aj*Zab+al*Zcd)/akl * trr(ix, iy, iz, kx, ky, kz-1)
        val -= aij/akl * trr(ix, iy, iz+1, kx, ky, kz-1)
    if kz > 1:
        val += (kz-1)*.5/akl * trr(ix, iy, iz, kx, ky, kz-2)
    if iz > 0:
        val += iz*.5/akl * trr(ix, iy, iz-1, kx, ky, kz-1)
    return val
    if ky > 0:
        val = -(aj*Yab+al*Ycd)/akl * trr(ix, iy, iz, kx, ky-1, kz)
        val -= aij/akl * trr(ix, iy+1, iz, kx, ky-1, kz)
    if ky > 1:
        val += (ky-1)*.5/akl * trr(ix, iy, iz, kx, ky-2, kz)
    if iy > 0:
        val += iy*.5/akl * trr(ix, iy-1, iz, kx, ky-1, kz)
    return val
    if kx > 0:
        val = -(aj*Xab+al*Xcd)/akl * trr(ix, iy, iz, kx-1, ky, kz)
        val -= aij/akl * trr(ix+1, iy, iz, kx-1, ky, kz)
    if kx > 1:
        val += (kx-1)*.5/akl * trr(ix, iy, iz, kx-2, ky, kz)
    if ix > 0:
        val += ix*.5/akl * trr(ix-1, iy, iz, kx-1, ky, kz)
    return val
    return vrr(0, ix, iy, iz)
... # Code omitted for brevity

```

Please note that numerical stability issues may arise with the TRR formula (12.47) when the factor $\frac{\alpha_{ij}}{\alpha_{kl}}$ is large. This factor can amplify certain intermediate variables and lead to the subtraction of large numbers, resulting in round-off errors. To mitigate numerical stability issues, an alternative TRR equation can be used:

$$\begin{aligned}
& g_{i_x, i_y, i_z; 000; k_x, k_y, k_z; 000}^n \\
&= X_Q C g_{i_x, i_y, i_z; 000; k_x-1, k_y, k_z; 000}^n + \frac{\theta X_P Q}{\alpha_{kl}} g_{i_x, i_y, i_z; 000; k_x-1, k_y, k_z; 000}^{n+1} \\
&\quad + \frac{k_x - 1}{2\alpha_{kl}} \left(g_{i_x, i_y, i_z; 000; k_x-2, k_y, k_z; 000}^n - \frac{\theta}{\alpha_{kl}} g_{i_x, i_y, i_z; 000; k_x-2, k_y, k_z; 000}^{n+1} \right) \\
&\quad + \frac{i_x}{2(\alpha_{ij} + \alpha_{kl})} g_{i_x-1, i_y, i_z; 000; k_x-1, k_y, k_z; 000}^{n+1}. \tag{12.48}
\end{aligned}$$

For simplicity, we will temporarily use Eq. (12.47) in the following sections since it does not affect our demonstration of how to implement and optimize the integral program. However, we will use the more precise formula (12.48) in the final version implemented in Section 12.1.8.

Next to the TRR, we can apply the HRR to transfer the Cartesian powers between the two centers of the same electron. Below is the HRR equation for transferring the powers in the x direction for the first electron

$$\begin{aligned} & g_{i_x, i_y, i_z; j_x, j_y, j_z; k_x, k_y, k_z; l_x, l_y, l_z} \\ &= g_{i_x+1, i_y, i_z; j_x-1, j_y, j_z; \dots} + X_{AB} \cdot g_{i_x, i_y, i_z; j_x-1, j_y, j_z; \dots} \end{aligned} \quad (12.49)$$

There are five more HRR equations that transfer the remaining Cartesian directions in the ERI tensor. These six HRR equations can be translated into a new DP function, as shown below:

```
@lru_cache(100000)
def hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz, lx, ly, lz):
    if lz > 0:
        return (hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz+1, lx, ly, lz-1) +
                Zcd * hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz, lx, ly, lz-1))
    if ly > 0:
        return (hrr(ix, iy, iz, jx, jy, jz, kx, ky+1, kz, lx, ly-1, lz) +
                Ycd * hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz, lx, ly-1, lz))
    if lx > 0:
        return (hrr(ix, iy, iz, jx, jy, jz, kx+1, ky, kz, lx-1, ly, lz) +
                Xcd * hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz, lx-1, ly, lz))
    if jz > 0:
        return (hrr(ix, iy, iz+1, jx, jy, jz-1, kx, ky, kz, lx, ly, lz) +
                Zab * hrr(ix, iy, iz, jx, jy, jz-1, kx, ky, kz, lx, ly, lz))
    if jy > 0:
        return (hrr(ix, iy+1, iz, jx, jy-1, jz, kx, ky, kz, lx, ly, lz) +
                Yab * hrr(ix, iy, iz, jx, jy-1, jz, kx, ky, kz, lx, ly, lz))
    if jx > 0:
        return (hrr(ix+1, iy, iz, jx-1, jy, jz, kx, ky, kz, lx, ly, lz) +
                Xab * hrr(ix, iy, iz, jx-1, jy, jz, kx, ky, kz, lx, ly, lz))
    return trr(ix, iy, iz, kx, ky, kz)
```

Combining all the RR functions mentioned previously, we obtain the ERI function of the OS algorithm:

```
def primitive_ERI(l1, l2, l3, l4, a1, a2, a3, a4, Ra, Rb, Rc, Rd):
    ... # Code omitted for brevity
@lru_cache(10000)
def vrr(n, ix, iy, iz):
    ...
```

```

@lru_cache(10000)
def trr(ix, iy, iz, kx, ky, kz):
    ...
    return vrr(0, ix, iy, iz)

@lru_cache(100000)
def hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz, lx, ly, lz):
    ...
    return trr(ix, iy, iz, kx, ky, kz)

nfi, nfj, nfk, nfl = map(n_cart, (li, lj, lk, ll))
eri = np.empty((nfi, nfj, nfk, nfl))
for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
    for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
        for k, (kx, ky, kz) in enumerate(iter_cart_xyz(lk)):
            for l, (lx, ly, lz) in enumerate(iter_cart_xyz(ll)):
                eri[i,j,k,l] = hrr(ix,iy,iz,jx,jy,jz,kx,ky,kz,lx,ly,lz)
return eri

```

12.1.3.4 ERIs with Rys-quadrature algorithm

ERIs can be computed using the quadrature method:

$$\begin{aligned}
(\chi_i \chi_j | \chi_k \chi_l) &= \frac{2\pi^{5/2}}{\alpha_{ij} \alpha_{kl} \sqrt{\alpha_{ij} + \alpha_{kl}}} \exp(-\theta_{ij} R_{AB}^2 - \theta_{kl} R_{CD}^2) \\
&\sum_n^{N_{\text{Rys}}} w_n I_{i_x, j_x, k_x, l_x}^x(u_n) I_{i_y, j_y, k_y, l_y}^y(u_n) I_{i_z, j_z, k_z, l_z}^z(u_n).
\end{aligned} \quad (12.50)$$

Here, u_n and w_n represent the quadrature roots and weights derived from the order- N_{Rys} Rys polynomial. The range of the summation in Eq. (12.50) is equal to the number of quadrature roots

$$N_{\text{Rys}} = \left\lfloor \frac{l_i + l_j + l_k + l_l}{2} + 1 \right\rfloor. \quad (12.51)$$

The program for calculating the quadrature roots and weights will be discussed in Section 12.2.2.

To calculate the I^x tensor, we can apply the following RR equations

$$I_{0000}^x = 1. \quad (12.52)$$

$$I_{i,0,k,0}^x = (X_{PA} - \frac{\theta u_n X_{PQ}}{\alpha_{ij}}) I_{i-1,0,k,0}^x + \frac{i-1}{2\alpha_{ij}} (1 - \frac{\theta u_n}{\alpha_{ij}}) I_{i-2,0,k,0}^x$$

$$+ \frac{ku_n}{2(\alpha_{ij} + \alpha_{kl})} I_{i-1,0,k-1,0}^x, \quad (12.53)$$

$$\begin{aligned} I_{i,0,k,0}^x = & (X_{QC} + \frac{\theta u_n X_{PQ}}{\alpha_{kl}}) I_{i,0,k-1,0}^x + \frac{k-1}{2\alpha_{kl}} (1 - \frac{\theta u_n}{\alpha_{kl}}) I_{i,0,k-2,0}^x \\ & + \frac{i u_n}{2(\alpha_{ij} + \alpha_{kl})} I_{i-1,0,k-1,0}^x, \end{aligned} \quad (12.54)$$

$$I_{i,j,k,l}^x = I_{i+1,j-1,k,l}^x + X_{AB} I_{i,j-1,k,l}^x, \quad (12.55)$$

$$I_{i,j,k,l}^x = I_{i,j,k+1,l-1}^x + X_{CD} I_{i,j,k,l-1}^x. \quad (12.56)$$

Tensors I_y and I_z can also be computed using similar RR equations. By translating these equations into a computational program, we obtain the following DP function:

```
def primitive_ERI(li, lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    aij = ai + aj
    Rab = Ra - Rb
    Rp = (ai * Ra + aj * Rb) / aij
    theta_ij = ai * aj / aij

    ak1 = ak + al
    Rcd = Rc - Rd
    Rq = (ak * Rc + al * Rd) / ak1
    theta_kl = ak * al / ak1

    Rpq = Rp - Rq
    theta = aij * ak1 / (aij + ak1)
    theta_r2 = theta * Rpq.dot(Rpq)
    Kabcd = 2*np.pi**2.5/(aij*ak1*(aij+ak1)**.5)
    Kabcd *= np.exp(-theta_ij*Rab.dot(Rab) - theta_kl*Rcd.dot(Rcd))
    lij = li + lj
    lkl = lk + ll
    nroots = (lij + lkl + 2) // 2
    rt, wt = rys_roots_weights(nroots, theta_r2)
    wt *= Kabcd
    Rpq = Rpq[:,np.newaxis]
    Rpa = Rp[:,np.newaxis] - Ra[:,np.newaxis]
    Rqc = Rq[:,np.newaxis] - Rc[:,np.newaxis]

    @lru_cache(10000)
    def trr(i, k):
        if i < 0 or k < 0:
            return 0.
        if i == k == 0:
            return np.ones((3, nroots))
        if k == 0:
```

```

        return ((Rpa - Rpq*theta/aij*rt) * trr(i-1, k) +
               (i-1)*.5/aij*(1-theta/aij*rt) * trr(i-2, k))
    if i == 0:
        return ((Rqc + Rpq*theta/akl*rt) * trr(i, k-1) +
               (k-1)*.5/akl*(1-theta/akl*rt) * trr(i, k-2))
    else:
        val = (Rqc + Rpq*theta/akl*rt) * trr(i, k-1)
        val += (k-1)*.5/akl*(1-theta/akl*rt) * trr(i, k-2)
        val += i*.5/(aij+akl)*rt * trr(i-1, k-1)
    return val

@lru_cache(10000)
def hrr(i, j, k, l):
    if j > 0:
        return hrr(i+1,j-1,k,l) + Rab[:,np.newaxis] * hrr(i,j-1,k,l)
    if l > 0:
        return hrr(i,j,k+1,l-1) + Rcd[:,np.newaxis] * hrr(i,j,k,l-1)
    return trr(i, k)

nfi, nfj, nkf, nfl = map(n_cart, (li, lj, lk, ll))
eri = np.empty((nfi, nfj, nkf, nfl))
for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
    for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
        for k, (kx, ky, kz) in enumerate(iter_cart_xyz(lk)):
            for l, (lx, ly, lz) in enumerate(iter_cart_xyz(ll)):
                Ix = hrr(ix,jx,kx,lx)[0]
                Iy = hrr(iy,jy,ky,ly)[1]
                Iz = hrr(iz,jz,kz,lz)[2]
                eri[i,j,k,l] = np.einsum('n,n,n,n->', wt, Ix, Iy, Iz)
return eri

```

12.1.3.5 Integral contraction

Having derived the program to compute the ERIs of primitive GTOs, we can now proceed to compute the ERIs of the contracted basis. Given a shell-quartet (i, j, k, l) , which represents the four shells of contracted GTO functions in ERIs, the simplest function to perform integral contraction can be implemented as follows:

```

def contracted_ERI(f, bas_i, bas_j, bas_k, bas_l):
    li, lj = bas_i.angular_momentum, bas_j.angular_momentum
    lk, ll = bas_k.angular_momentum, bas_l.angular_momentum
    Ra, Rb = bas_i.coordinates, bas_j.coordinates
    Rc, Rd = bas_k.coordinates, bas_l.coordinates
    norm_ci = bas_i.norm_coefficients
    norm_cj = bas_j.norm_coefficients

```

```

norm_ck = bas_k.norm_coefficients
norm_cl = bas_l.norm_coefficients
nfi, nfj, nkf, nfl = map(n_cart, (li, lj, lk, ll))
eri = np.zeros((nfi, nfj, nkf, nfl))

for ai, ci in zip(bas_i.exponents, norm_ci):
    for aj, cj in zip(bas_j.exponents, norm_cj):
        for ak, ck in zip(bas_k.exponents, norm_ck):
            for al, cl in zip(bas_l.exponents, norm_cl):
                eri += ci*cj*ck*cl * f(li, lj, lk, ll, ai, aj, ak, al,
                                         Ra, Rb, Rc, Rd)
return eri

```

It should be noted that this naive integral contraction scheme is not optimal in terms of floating-point operation (FLOP) counts. FLOPs can be reduced by employing an early contraction technique, which swaps the order of the HRR iteration and the integral contraction.

In the HRR equation, such as Eq. (12.49), the vector \mathbf{R}_{AB} remains constant for all primitive ERIs within the shell-quartet. They can be extracted as a common factor from the `primitive_ERI` function. As illustrated in Fig. 12.1, the order of applying intermediate steps such as `vrr`, `trr`, `hrr`, and `contraction` in the ERI algorithm for contracted GTO basis functions can be adjusted. Please note that the notations $[\chi_i \chi_j | \chi_k \chi_l]$ and $(\chi_i \chi_j | \chi_k \chi_l)$ are used in Fig. 12.1 to distinguish between the integrals of primitive GTOs and contracted GTOs, respectively. In quantum chemistry literature, the ERIs of primitive GTOs are typically denoted by square bracket, such as $[\chi_i \chi_j | \chi_k \chi_l]$, and the ERIs of contracted GTOs are denoted by parentheses, such as $(\chi_i \chi_j | \chi_k \chi_l)$. By considering more details of the angular momentum and degrees of contraction, the path of ERI evaluation might be further optimized for FLOP counts.

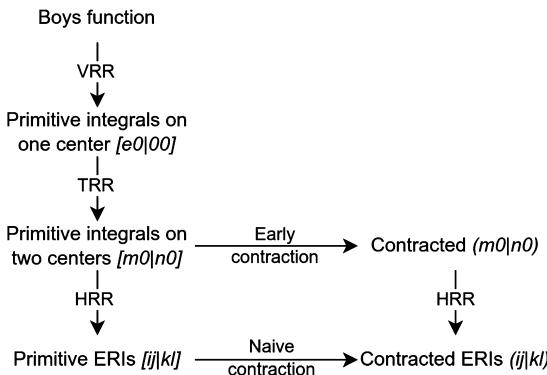


FIGURE 12.1

Differential paths for evaluating contracted ERIs.

Discussions on the optimal paths can be found in various quantum chemistry textbooks and literature [4].

Taking the OS algorithm as an example, to apply the early contraction scheme, the program described in Section 12.1.3.3 can be modified as follows:

```
def primitive_trr(li, lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    ... # Code omitted for brevity
    def vrr(n, ix, iy, iz):
        ...

        def trr(ix, iy, iz, kx, ky, kz):
            ...
            return vrr(0, ix, iy, iz)
        return trr

    def contracted_ERI(f_prim, bas_i, bas_j, bas_k, bas_l):
        li, lj = bas_i.angular_momentum, bas_j.angular_momentum
        lk, ll = bas_k.angular_momentum, bas_l.angular_momentum
        ... # Code omitted for brevity

        @lru_cache(10000)
        def trr_adapter(ai, aj, ak, al):
            return primitive_trr(li, lj, lk, ll, ai, aj, ak, al,
                                 Ra, Rb, Rc, Rd)

        def contracted_trr(ix, iy, iz, kx, ky, kz):
            eri_2c = 0.
            for ai, ci in zip(bas_i.exponents, norm_ci):
                for aj, cj in zip(bas_j.exponents, norm_cj):
                    for ak, ck in zip(bas_k.exponents, norm_ck):
                        for al, cl in zip(bas_l.exponents, norm_cl):
                            trr = trr_adapter(ai, aj, ak, al)
                            eri_2c += ci*cj*ck*cl * trr(ix, iy, iz, kx, ky, kz)
            return eri_2c

        def hrr(ix, iy, iz, jx, jy, jz, kx, ky, kz, lx, ly, lz):
            ...
            return contracted_trr(ix, iy, iz, kx, ky, kz)
        ...


```

The program structure for the early contraction algorithm couples the contraction code with the RR code. This coupling complicates the optimization procedure for the recursive DP function. To keep the code example concise, we will omit the implementations of the early contraction algorithms in this book.

12.1.3.6 Unit tests for ERIs

So far, we have developed programs for computing overlap integrals and ERIs using the top-down DP approach. Although this is not an efficient implementation for evaluating integrals, the program with the DP approach strictly follows the mathematical equations. This program is easy to develop and maintain. It can serve as a reference for future debugging and optimization. At this stage, it is crucial to establish unit tests to ensure that any future optimizations can reproduce the results of this version.

Here, we have created a test file, named `test_eri.py`, in the same directory as the source code. This setup allows us to import the modules we have just developed in the directory. If you have developed the integral code as a package (with a `__init__.py` file in the directory), the test code should import the modules either in terms of the package path or using a relative import (`from . import`).

```
from basis import CGTO, n_cart, gto_offsets
from eri_MD import contracted_ERI
import eri_MD
import eri_OS
import eri_rys

prim_gtos = '''
- angular_momentum: 0
  exponents: [0.75]
  coefficients: [1.0]
  coordinates: [0.3, 0.0, 1.0]
- angular_momentum: 1
  exponents: [0.32]
  coefficients: [1.0]
  coordinates: [0.2, 1.9, 0.0]
- angular_momentum: 2
  exponents: [0.51]
  coefficients: [1.0]
  coordinates: [0.5, 0.0, 0.5]
'''

def new_gtos():
    gtos = (CGTO.from_bse('STO-3G', 'C', np.array([0.1, 0.5, 0.8]))
            + CGTO.from_yaml(prim_gtos))
    return gtos

def get_eri_tensor(f, gtos):
    offsets = gto_offsets(gtos)
    nao = offsets[-1]
    buf = np.empty((nao, nao, nao, nao))
    for i, bas_i in enumerate(gtos):
```

```

        i0, i1 = offsets[i], offsets[i+1]
        for j, bas_j in enumerate(gtos):
            j0, j1 = offsets[j], offsets[j+1]
            for k, bas_k in enumerate(gtos):
                k0, k1 = offsets[k], offsets[k+1]
                for l, bas_l in enumerate(gtos):
                    l0, l1 = offsets[l], offsets[l+1]
                    buf[i0:i1,j0:j1,k0:k1,l0:l1] = \
                        contracted_ERI(f, bas_i, bas_j, bas_k, bas_l)

    return buf

def test_eri_MD():
    gtos = new_gtos()
    V = get_eri_tensor(eri_MD.primitive_ERI, gtos)
    assert abs(abs(V).sum() - 1277.9590013752) < 1e-8

def test_eri_OS():
    gtos = new_gtos()
    V = get_eri_tensor(eri_OS.primitive_ERI, gtos)
    assert abs(abs(V).sum() - 1277.9590013752) < 1e-8

def test_eri_rys():
    gtos = new_gtos()
    V = get_eri_tensor(eri_rys.primitive_ERI, gtos)
    assert abs(abs(V).sum() - 1277.9590013752) < 1e-8

```

We can execute the following command to run the unit tests.

```
$ pytest -v test_eri.py
```

The unit tests we demonstrated here serve as a quick regression check when optimizing the integral code. We can frequently run these tests for every change we make to the code to ensure that they do not alter the output. However, as an initial test suite, they do not address corner cases or the numerical stability issues that may arise during optimization. Designing effective tests is a challenging task, requiring significant efforts, expertise, and a deep understanding of the algorithm. More tests can be gradually added during the code development.

12.1.4 Converting recursion to iteration

Recursion is generally less efficient than iteration due to the overhead associated with function calls and the `lru_cache` lookups. A more efficient approach is to generate the intermediate tensors using nested for-loop iterations.

In Chapter 9, we discussed the process of converting recursive code to iteration code. This process involves analyzing variable dependencies, the order of for-loop

iterations, and the problem size. To facilitate this analysis, we utilize the LRU cache to capture the dependencies between DP states. As shown in the following code snippet, we customize the `lru_cache` function using an ordered dictionary.

```
def lru_cache(f):
    cache = {}
    def f_cached(*args):
        if args in cache:
            return cache[args]
        result = f(*args)
        cache[args] = result
        return result
    f_cached._cache = cache
    return f_cached
```

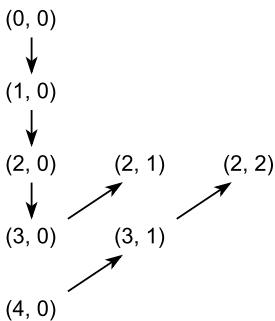
After running small test systems, the keys of the dictionary reveal the order in which the DP states are generated. From this, we can infer the iteration order. Additionally, the number of cached keys indicates the upper boundary of the DP problem.

For example, by using this `lru_cache` implementation for the overlap function, we can easily determine the required shape of the DP state array. Clearly, the shape of the DP state array is governed by (l_i+1, l_j+1) .

```
@lru_cache
def get_S(i, j):
    if i < 0 or j < 0:
        return 0.
    if i == j == 0:
        return (np.pi/aij)**.5 * np.exp(-theta_ij*Rab**2)
    if j == 0:
        if i == 1:
            return Rpa * get_S(i-1, j)
        return (i-1)/(2*aij) * get_S(i-2, j) + Rpa * get_S(i-1, j)
    return Rab * get_S(i, j-1) + get_S(i+1, j-1)

get_S(2, 2)
keys = get_S._cache.keys()
print(max(i for i, j in keys)) # 4
print(max(j for i, j in keys)) # 2
print(keys) # dict_keys([(0, 0), (1, 0), (2, 0), (3, 0), (2, 1), (4, 0),
(3, 1), (2, 2)])
```

Fig. 12.2 displays a diagram illustrating how the DP states are generated. From this diagram, we can deduce that the iteration over i should be performed first (in the innermost loop), followed by the iteration over j . The iteration range for j spans from 0 to l_j , while the iteration range for i depends on the value of j . These iterations can be structured into the following nested for-loop configuration:

**FIGURE 12.2**

The dependencies of intermediate DP states in overlap integrals.

```

for j in range(lj+1):
    for i in range(li+lj+1-j):
  
```

As a result, the overlap integrals can be computed with the following iteration code

```

def primitive_overlap(li, lj, ai, aj, Ra, Rb):
    ... # Code omitted for brevity
    lij = li + lj
    S = np.zeros((lij+1, lj+1, 3))
    for j in range(lj+1):
        for i in range(lij+1-j):
            if j > 0:
                S[i,j] = S[i+1,j-1] + Rab * S[i,j-1]
            elif i > 1:
                S[i,j] = Rpa * S[i-1,j] + (i-1)/(2*aij) * S[i-2,j]
            elif i == 1:
                S[1,0] = Rpa * S[0,0]
            else:
                S[0,0] = (np.pi/aij)**.5 * np.exp(-theta_ij * Rab**2)
    ...
    for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
        for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
            overlap[i,j] = S[ix,jx,0] * S[iy,jy,1] * S[iz,jz,2]
    return overlap
  
```

Using a similar method, we can convert the recursive DP code to the iteration code for ERI programs.

12.1.4.1 The McMurchie-Davidson algorithm

In the MD algorithm, we need to implement the iteration code for the R and the E tensors.

The R tensor is a three-dimensional tensor. However, a four-index array is required to store the intermediate DP states for R : one index n for the order, and three indices t , u , and v for the three Cartesian components. Within the recursive code, the order index n is incremented by one whenever each of the Cartesian indices is decreased by one. The maximum value that n can attain is equal to the sum of the three Cartesian indices. Therefore, the size of the DP state array is $(3l+1, l+1, l+1, l+1)$, given the upper limit l for t , u , and v . There are no dependencies between t , u , and v . The three nested loops for t , u , and v can be placed in any order. Here is the iteration code to compute the R tensor:

```
def get_R_tensor(l, theta, Rpq):
    Rt = np.zeros((l*3+1, l+1, l+1, l+1))
    for t in range(l+1):
        for u in range(l+1):
            for v in range(l+1):
                for n in range(l*3):
                    if t > 1:
                        Rt[n,t,u,v] = (t-1) * Rt[n+1,t-2,u,v]
                        Rt[n,t,u,v] += Rpq[0] * Rt[n+1,t-1,u,v]
                    elif t == 1:
                        Rt[n,1,u,v] = Rpq[0] * Rt[n+1,0,u,v]
                    elif u > 1:
                        Rt[n,t,u,v] = (u-1) * Rt[n+1,t,u-2,v]
                        Rt[n,t,u,v] += Rpq[1] * Rt[n+1,t,u-1,v]
                    elif u == 1:
                        Rt[n,t,1,v] = Rpq[1] * Rt[n+1,t,0,v]
                    elif v > 1:
                        Rt[n,t,u,v] = (v-1) * Rt[n+1,t,u,v-2]
                        Rt[n,t,u,v] += Rpq[2] * Rt[n+1,t,u,v-1]
                    elif v == 1:
                        Rt[n,t,u,1] = Rpq[2] * Rt[n+1,t,u,0]
                    else: # t == u == v == 0
                        Rt[n,0,0,0] = (-2*theta)**n \
                                      * boys(n, theta*np.array(Rpq).dot(Rpq))
    return Rt[0]
```

In the recursive `get_E_cart_components` function, the DP state is indexed by three integers: the auxiliary index t , and the Cartesian power indices i and j . There are no dependencies between i and j . However, both i and j depend on t . The maximum value for t is the sum of the largest i and j values. We can then derive the iteration code for computing each Cartesian component of the E tensor.

```

def get_E_cart_components(l1, l2, a1, a2, R1, R2):
    ... # Code omitted for brevity
    l12 = l1 + l2
    E_cart = np.zeros((l1+1, l2+1, l12+1, 3))
    for j in range(l2+1):
        for i in range(l1+1):
            for t in range(l12+1):
                if t > 0:
                    val = 0
                    if i > 0:
                        val += i*E_cart[i-1,j,t-1] / (2*a1j*t)
                    if j > 0:
                        val += j*E_cart[i,j-1,t-1] / (2*a1j*t)
                    E_cart[i,j,t] = val
                elif j > 0:
                    E_cart[i,j,0] = Rpb * E_cart[i,j-1,0] + E_cart[i,j-1,1]
                elif i > 0:
                    E_cart[i,j,0] = Rpa * E_cart[i-1,j,0] + E_cart[i-1,j,1]
                else: # i == j == t == 0
                    E_cart[0,0,0] = np.exp(-theta_ij * Rab**2)
    return E_cart

```

12.1.4.2 The Obara-Saika algorithm

In the OS algorithm, we need to rewrite the recursive DP functions for the VRR, TRR, and HRR tensors.

For the VRR tensor, the size of the DP state array and the variable dependencies are similar to those of the R tensor in the MD algorithm. We can create the iteration code for the VRR tensor as follows.

```

Kabcd = 2*np.pi**2.5/(a1j*a1k*(a1j+a1k)**.5)
Kabcd *= np.exp(-theta_ij*Rab.dot(Rab) - theta_kl*Rcd.dot(Rcd))
l12 = l1 + l2
lkl = lk + ll
l4 = l12 + lkl
vrr = np.empty((l4*3+1, l4+1, l4+1, l4+1))
for ix in range(l4+1):
    for iy in range(l4+1):
        for iz in range(l4+1):
            for ni in range(l4*3):
                if ix > 0:
                    val = Xpa*vrr[ni, ix-1, iy, iz]
                    val -= theta/a1j*Xpq*vrr[ni+1, ix-1, iy, iz]
                    if ix > 1:

```

```

        val += (ix-1)*.5/aij * (vrr[ni, ix-2, iy, iz] -
                                theta/aij*vrr[ni+1, ix-2, iy, iz])
        vrr[ni,ix,iy,iz] = val
    elif iy > 0:
        val = Ypa*vrr[ni, ix, iy-1, iz]
        val -= theta/aij*Ypq*vrr[ni+1, ix, iy-1, iz]
        if iy > 1:
            val += (iy-1)*.5/aij * (vrr[ni, ix, iy-2, iz] -
                                theta/aij*vrr[ni+1, ix, iy-2, iz])
        vrr[ni,ix,iy,iz] = val
    elif iz > 0:
        val = Zpa*vrr[ni, ix, iy, iz-1]
        val -= theta/aij*Zpq*vrr[ni+1, ix, iy, iz-1]
        if iz > 1:
            val += (iz-1)*.5/aij * (vrr[ni, ix, iy, iz-2] -
                                theta/aij*vrr[ni+1, ix, iy, iz-2])
        vrr[ni,ix,iy,iz] = val
    else:
        vrr[ni,0,0,0] = Kabcd * boys(ni, theta*Rpq.dot(Rpq))

```

When it comes to the TRR tensor, the DP state array is characterized by six indices: two in each of the three Cartesian components. Dependencies can be found between the two indices within each Cartesian component, while no dependencies exist between different Cartesian components. Here, we derive the iteration code for the TRR tensor:

```

from itertools import product
l4 = l1j + lkl
Xjl, Yjl, Zjl = -(aj*Rab+al*Rcd)/akl
trr = np.empty((l4+1,l4+1,l4+1, lkl+1,lkl+1,lkl+1))
for kx, ky, kz in product(range(lkl+1), range(lkl+1), range(lkl+1)): # (1)
    for ix in range(l4+1-kx): # (2)
        for iy in range(l4+1-ky): # (3)
            for iz in range(l4+1-kz): # (4)
                if kx > 0:
                    val = Xjl * trr[ix,iy,iz,kx-1,ky,kz]
                    val -= aij/akl * trr[ix+1,iy,iz,kx-1,ky,kz]
                    if kx > 1:
                        val += (kx-1)*.5/akl * trr[ix,iy,iz,kx-2,ky,kz]
                    if ix > 0:
                        val += ix*.5/akl * trr[ix-1,iy,iz,kx-1,ky,kz]
                    trr[ix,iy,iz,kx,ky,kz] = val
                elif ky > 0:
                    val = Yjl * trr[ix,iy,iz,kx,ky-1,kz]
                    val -= aij/akl * trr[ix,iy+1,iz,kx,ky-1,kz]

```

```

    if ky > 1:
        val += (ky-1)*.5/akl * trr[ix, iy, iz, kx, ky-2, kz]
    if iy > 0:
        val += iy*.5/akl * trr[ix, iy-1, iz, kx, ky-1, kz]
    trr[ix, iy, iz, kx, ky, kz] = val
elif kz > 0:
    val = Zjl * trr[ix, iy, iz, kx, ky, kz-1]
    val -= aij/akl * trr[ix, iy, iz+1, kx, ky, kz-1]
    if kz > 1:
        val += (kz-1)*.5/akl * trr[ix, iy, iz, kx, ky, kz-2]
    if iz > 0:
        val += iz*.5/akl * trr[ix, iy, iz-1, kx, ky, kz-1]
    trr[ix, iy, iz, kx, ky, kz] = val
else:
    trr[ix, iy, iz, 0, 0, 0] = vrr[0, ix, iy, iz]

```

In line (1), the order of the loops for the three Cartesian components is flexible. However, within each Cartesian component, the loop for the first index must be nested within the loop for the second index. As shown in lines (2) to (4), the iteration ranges of the inner and outer indices are coupled.

In the case of the HRR tensor, dependencies only occur within the indices of each electron. The HRR iterations for the two electrons can be performed separately, leading to the following iteration code:

```

hrr = np.zeros((lij+1, lij+1, lij+1, lk1+1, lk1+1, lk1+1, ll+1, ll+1, ll+1))
hrr[:, :, :, :, :, 0, 0, 0] = trr[:, :, :, :, :, 0, 0, 0]
for lx, ly, lz in product(range(ll+1), range(ll+1), range(ll+1)):
    for kx in range(lk1+1-lx):
        for ky in range(lk1+1-ly):
            for kz in range(lk1+1-lz):
                if lx > 0:
                    hrr[:, :, :, kx, ky, kz, lx, ly, lz] = \
                        (      hrr[:, :, :, kx+1, ky, kz, lx-1, ly, lz] +
                        Xcd * hrr[:, :, :, kx, ky, kz, lx-1, ly, lz])
                elif ly > 0:
                    hrr[:, :, :, kx, ky, kz, lx, ly, lz] = \
                        (      hrr[:, :, :, kx, ky+1, kz, lx, ly-1, lz] +
                        Ycd * hrr[:, :, :, kx, ky, kz, lx, ly-1, lz])
                elif lz > 0:
                    hrr[:, :, :, kx, ky, kz, lx, ly, lz] = \
                        (      hrr[:, :, :, kx, ky, kz+1, lx, ly, lz-1] +
                        Zcd * hrr[:, :, :, kx, ky, kz, lx, ly, lz-1])

erj = np.zeros((lij+1, lij+1, lij+1, lj+1, lj+1, lj+1,
                lk1+1, lk1+1, lk1+1, ll+1, ll+1, ll+1))

```

```

eri[:, :, :, 0, 0] = hrr[:, :, :, :lkl+1, :lkl+1, :lkl+1]
for jx, jy, jz in product(range(lj+1), range(lj+1), range(lj+1)):
    for ix in range(lij+1-jx):
        for iy in range(lij+1-jy):
            for iz in range(lij+1-jz):
                if jx > 0:
                    eri[ix, iy, iz, jx, jy, jz] = (eri[ix+1, iy, iz, jx-1, jy, jz] +
                        Xab * eri[ix , iy, iz, jx-1, jy, jz])
                elif jy > 0:
                    eri[ix, iy, iz, jx, jy, jz] = (eri[ix, iy+1, iz, jx, jy-1, jz] +
                        Yab * eri[ix, iy , iz, jx, jy-1, jz])
                elif jz > 0:
                    eri[ix, iy, iz, jx, jy, jz] = (eri[ix, iy, iz+1, jx, jy, jz-1] +
                        Zab * eri[ix, iy, iz , jx, jy, jz-1])

```

12.1.4.3 The Rys-quadrature algorithm

The Rys ERI algorithm involves constructing the TRR and HRR tensors separately for each of the three Cartesian components. The problem size and variable dependencies are similar to those of the TRR and HRR tensors in the OS scheme. We can derive the iteration code for the Rys ERI algorithm as follows.

```

lj = lij + lkl
trr = np.zeros((lj+1, lkl+1, 3, nroots))
for k in range(lkl+1):
    for i in range(lj+1):
        if i == k == 0:
            trr[0, 0] = np.ones((3, nroots))
        elif k == 0:
            trr[i, k] = (Rpa - Rpq*theta/aij*rt) * trr[i-1, k]
            trr[i, k] += (i-1)*.5/aij*(1-theta/aij*rt) * trr[i-2, k]
        elif i == 0:
            trr[i, k] = (Rqc + Rpq*theta/akl*rt) * trr[i, k-1]
            trr[i, k] += (k-1)*.5/akl*(1-theta/akl*rt) * trr[i, k-2]
        else:
            trr[i, k] = (Rqc + Rpq*theta/akl*rt) * trr[i, k-1]
            trr[i, k] += (k-1)*.5/akl*(1-theta/akl*rt) * trr[i, k-2]
            trr[i, k] += i*.5/(aij+akl)*rt * trr[i-1, k-1]

hrr = np.zeros((lij+1, lj+1, lkl+1, ll+1, 3, nroots))
for j in range(lj+1):
    for i in range(lij+1-j):
        for l in range(ll+1):
            for k in range(lkl+1-l):
                if j > 0:

```

```

        hrr[i,j,k,l] = hrr[i+1,j-1,k,l] + Rab*hrr[i,j-1,k,l]
    elif l > 0:
        hrr[i,j,k,l] = hrr[i,j,k+1,l-1] + Rcd*hrr[i,j,k,l-1]
    else:
        hrr[i,0,k,0] = trr[i,k]

```

12.1.5 Eliminating branches and adjusting iterations

By converting the recursive function to the iterative code, we eliminate the overhead of function calls, hashing, and cache lookups. However, the resulting iteration code is likely still a preliminary implementation. Before considering compilation optimization, several issues need to be addressed, including:

- If-else branches in the inner-most loop. This issue leads to additional computational efforts and disruption of the execution pipelines.
- Wasted intermediate states. Some intermediate states in the DP state array are unnecessary. To avoid wasting computational efforts, they can be omitted by setting appropriate iteration ranges.
- Inefficient array accessing patterns. The array accessing patterns in the DP state array were not optimized, which can lead to cache misses and high latency in memory accesses.

To eliminate the if-else branches in the innermost loop, we can rearrange the order of the branching and for-loop iterations. This involves moving the branching to the outer loops and expanding the for-loop code block for each condition. Taking the overlap integral as an example, $S[0,0]$ and $S[1,0]$ are special cases that can be extracted from the loops. Other elements are calculated in the iteration code using RR formulas. The overlap integral function can be optimized as follows:

```

def primitive_overlap(li, lj, ai, aj, Ra, Rb):
    ... # Code omitted for brevity
    lij = li + lj
    S = np.zeros((lij+1, lj+1, 3))
    S[0,0] = (np.pi/aij)**.5 * np.exp(-theta_ij * Rab**2)
    if lij > 0:
        S[1,0] = Rpa * S[0,0]
        for i in range(1, lij):
            S[i+1,0] = Rpa * S[i,0] + i/(2*aij) * S[i-1,0]
    for j in range(1, lj+1):
        for i in range(lij+1-j):
            S[i,j] = Rab * S[i,j-1] + S[i+1,j-1]
    ...

```

Similar procedures can be applied to optimize the ERI iteration code, which we will omit here.

Next, let's examine the unnecessary intermediate states in the MD and OS ERI algorithms.

In the MD scheme, the 5-index tensor E is highly sparse. The subscripts t, u, v of E_{tuv}^{ij} are associated with the Cartesian powers of the product $\chi_i(\mathbf{r})\chi_j(\mathbf{r})$. They are subject to the constraint $t + u + v \leq l_i + l_j$, where l_i and l_j are the angular momenta of the two GTOs. The size of the compound index tuv equals to

$$\frac{(l_i + l_j + 1)(l_i + l_j + 2)(l_i + l_j + 3)}{6}.$$

By merging the t, u, v indices, the E tensor can be compressed into a three-index tensor.

```
def get_E_tensor(li, lj, ai, aj, Ra, Rb):
    Ex, Ey, Ez = get_E_cart_components(li, lj, ai, aj, Ra, Rb)
    lij = li + lj
    Et = np.empty(((lij+1)*(lij+2)//2, (lj+1)*(lj+2)//2,
                  (lij+1)*(lij+2)*(lij+3)//6))
    for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
        for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
            for n, (t, u, v) in enumerate(reduced_cart_iter(lij)):
                Et[i,j,n] = Ex[ix,jx,t] * Ey[iy,jy,u] * Ez[iz,jz,v]
    return Et
```

Here, we have introduced the following generator to incorporate the constraints of Cartesian powers into the nested iterations:

```
def reduced_cart_iter(n)
    '''Nested loops for Cartesian components, subject to x+y+z <= n'''
    for x in range(n+1):
        for y in range(n+1-x):
            for z in range(n+1-x-y):
                yield x, y, z
```

When applying the contraction equation (12.27) with the compressed E tensor, the iteration ranges for t, u, v and e, f, g are reduced. This results in the use of compound indices ij and $k1$ in the following code.

```
def primitive_ERI(li, lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    ... # Code omitted for brevity
    nf_ij = (lij+1)*(lij+2)*(lij+3)//6
    nf_kl = (lk+1)*(lk+2)*(lk+3)//6
    Rt = get_R_tensor(li+lj+lk+ll, theta, Rpq)
    Rt2 = np.empty((nf_kl, nf_ij))
    for k1, (e, f, g) in enumerate(reduced_cart_iter(lk+1)):
        phase = (-1)**(e+f+g)
```

```

        for ij, (t, u, v) in enumerate(reduced_cart_iter(lij)):
            Rt2[kl,ij] = phase * Rt[t+e,u+f,v+g]

    Etab = get_E_tensor(lj, lj, ai, aj, Ra, Rb)
    Etcd = get_E_tensor(lk, lk, ak, al, Rc, Rd)
    gcd = np.dot(Rt2.T, Etcd.reshape(nfk*nfl,nf_kl).T)
    eri = np.dot(Etab.reshape(nfi*nfj,nf_ij), gcd)
    eri *= 2*np.pi**2.5/(aij*akl*(aij+akl)**.5)
    return eri

```

The sizes of the compound indices are smaller than those of the nested `tuv` or `efg` indices. As a result, the tensor contractions are preformed with smaller tensors, thus requiring fewer FLOPs.

Similarly, when constructing the R tensor, the subscripts tuv are subject to the constraint $t + u + v \leq l$, where l represents the total angular momentum of the four GTOs in the shell-quartet, equal to $l_i + l_j + l_k + l_l$. The iteration code presented in Section 12.1.4.1 can be adjusted as follows:

```

def get_R_tensor(l, theta, Rpq):
    ...
    Rt = np.zeros((l*3+1, l+1, l+1, l+1))
    for t, u, v in reduced_cart_iter(l):
        for n in range(l*3):
            if t > 1:
                Rt[n,t,u,v] = (t-1) * Rt[n+1,t-2,u,v]
                Rt[n,t,u,v] += Rpq[0] * Rt[n+1,t-1,u,v]
            elif t == 1:
                Rt[n,1,u,v] = Rpq[0] * Rt[n+1,0,u,v]
            ...

```

In the `get_R_tensor` function, we observe that the indexing $t - 1$, $u - 1$, and $v - 1$ are exclusively performed within the iteration, and only one of them is coupled to the increment of n . Given this observation, we can conclude that the maximum value of n is l , subject to the constraint $n + t + u + v \leq l$. Consequently, the iterations for the R tensor can be refined as follows:

```

def get_R_tensor(l, theta, Rpq):
    ...
    Rt = np.zeros((l+1, l+1, l+1, l+1))
    for t, u, v in reduced_cart_iter(l):
        for n in range(l+1-t-u-v):
            if t > 1:
                Rt[n,t,u,v] = (t-1) * Rt[n+1,t-2,u,v]
                Rt[n,t,u,v] += Rpq[0] * Rt[n+1,t-1,u,v]
            elif t == 1:

```

```
Rt[n,1,u,v] = Rpq[0] * Rt[n+1,0,u,v]
```

```
...
```

The optimized iterations generate a total of

$$\frac{(l+1)(l+2)(l+3)(l+4)}{24}.$$

DP states. In comparison, the naive for-loop iterations generate $3l(l+1)^3$ DP states. The optimization reduces the computational costs by approximately 72 times.

Similar exclusive patterns can be observed in the VRR implementation of the OS algorithm. In the case of the TRR tensor `trr[ix, iy, iz, kx, ky, kz]`, there exist multiple constraints on angular momenta

$$i_x + i_y + i_z \leq l_{ij}, \quad (12.57)$$

$$k_x + k_y + k_z \leq l_{kl}, \quad (12.58)$$

$$i_x + i_y + i_z + k_x + k_y + k_z \leq l_{ij} + l_{kl}. \quad (12.59)$$

For the code implementation in Section 12.1.4.2, the iteration ranges for TRR tensors can be refined to

```
l4 = l_ij + l_kl
for kx, ky, kz in reduced_cart_iter(l_kl):
    ksum = kx + ky + kz
    for ix, iy, iz in reduced_cart_iter(l4-ksum):
        if kx > 0:
            val = -(aj*Xab+al*Xcd)/akl * trr[ix,iy,iz,kx-1,ky,kz]
            val -= aij/akl * trr[ix+1,iy,iz,kx-1,ky,kz]
            if kx > 1:
                val += (kx-1)*.5/akl * trr[ix,iy,iz,kx-2,ky,kz]
            if ix > 0:
                val += ix*.5/akl * trr[ix-1,iy,iz,kx-1,ky,kz]
            trr[ix,iy,iz,kx,ky,kz] = val
    ...
```

This optimization for iteration ranges can also be applied to the HRR tensor implementation in Section 12.1.4.2.

If integrals are computed entirely in Python, the structure and order of indices for these intermediate tensors have a minor impact on performance. However, when array indexing is optimized with the compilation techniques (Section 12.1.6), its impact can become more significant. In this context, it is beneficial to consider the layout of the array and make necessary adjustments before employing the compilation optimization.

In the integral code presented previously, tensors are allocated using `np.zeros` or `np.empty`, and are thereby arranged in C-order. To leverage this memory access

pattern, it is generally more efficient to place the iteration over the last index in the innermost loop. This can be achieved by reordering either the tensor indices or the loop order. For example, in the MD contraction code shown previously, the `Rt2` array is allocated such that the index `ij` is placed last. This allocation is aligned with the loop order.

In the function `get_R_tensor`, the first index of the R tensor is accessed in the innermost loop, which can lead to inefficient memory access. Changing the loop order can make memory access align with the structure of the R tensor. As the iteration over the index n is moved to the outermost loop, we must reverse its iteration direction due to the dependency of n on $n + 1$.

```
def get_R_tensor(l, theta, Rpq):
    ... # Code omitted for brevity
    for n in range(l-1, -1, -1):
        for t, u, v in reduced_cart_iter(l-n):
            if t > 1:
                Rt[n,t,u,v] = (t-1) * Rt[n+1,t-2,u,v]
                Rt[n,t,u,v] += Rpq[0] * Rt[n+1,t-1,u,v]
            elif t == 1:
                Rt[n,1,u,v] = Rpq[0] * Rt[n+1,0,u,v]
            ...
    ...
```

Such loop swapping optimizations can easily introduce bugs. During the optimization process, it is crucial to refine and frequently run unit tests to ensure the correctness of all changes.

In the HRR implementation in Section 12.1.4.2, a sub-tensor in the leading dimensions of the HRR tensor is operated within the innermost loop. Accessing data within this sub-tensor can be inefficient due to non-contiguous memory access. Therefore, these iterations should be moved to the outer loop.

```
... # Code omitted for brevity
hrr = np.zeros((lij+1,lij+1,lij+1, lkl+1,lkl+1,lkl+1, ll+1,ll+1,ll+1))
hrr[:, :, :, :, :, 0, 0, 0] = trr[:,lij+1,:,lij+1,:,lij+1]
for ix, iy, iz in product(range(lij+1), range(lij+1), range(lij+1)):
    lx = 0
    ly = 0
    for lz in range(1, ll+1-ly):
        lsum = lx + ly + lz
        for kx, ky, kz in reduced_cart_iter(lkl-lsum):
            hrr[ix,iy,iz,kx,ky,kz,lx,ly,lz] = \
                (      hrr[ix,iy,iz,kx,ky,kz+1,lx,ly,lz-1] +
                Zcd * hrr[ix,iy,iz,kx,ky,kz ,lx,ly,lz-1])
    for ly in range(1, ll+1-lx):
        for lz in range(ll+1-ly):
            lsum = lx + ly + lz
```

```

for kx, ky, kz in reduced_cart_iter(lkl-lsum):
    hrr[ix, iy, iz, kx, ky, kz, lx, ly, lz] = \
        (      hrr[ix, iy, iz, kx, ky+1, kz, lx, ly-1, lz] +
        Ycd * hrr[ix, iy, iz, kx, ky , kz, lx, ly-1, lz])
for lx in range(1, ll+1):
    for ly in range(ll+1-lx):
        for lz in range(ll+1-lx-ly):
            lsum = lx + ly + lz
            for kx, ky, kz in reduced_cart_iter(lkl-lsum):
                hrr[ix, iy, iz, kx, ky, kz, lx, ly, lz] = \
                    (      hrr[ix, iy, iz, kx+1, ky, kz, lx-1, ly, lz] +
                    Xcd * hrr[ix, iy, iz, kx , ky, kz, lx-1, ly, lz])

```

12.1.6 Optimization with Numba JIT compilation

At this stage, we employ compilation techniques to enhance the performance of the integral program. Numba and Cython are two popular choices for compiling Python code.

Let's begin with the just-in-time (JIT) compilation using Numba. Applying Numba's JIT to compile the integral function we presented earlier can speed up the program, but the acceleration effects might be limited. To leverage the full potential of the Numba JIT compiler, some adjustments can be made to the integral code.

Expanding NumPy ufuncs

It is advisable to replace all NumPy ufuncs and broadcasting code with explicit for-loop iterations, particularly for those operations within nested loops. For instance, in the iteration code for the overlap integral that we demonstrated previously,

```

for j in range(lj+1):
    for i in range(lij+1-j):
        S[i,j] = S[i+1,j-1] + Rab * S[i,j-1]

```

the arithmetic computation and assignment involve a broadcasting operation over the three Cartesian directions. This broadcasting code should be rewritten using a loop block.

```

for j in range(lj+1):
    for i in range(lij+1-j):
        for m in range(3):
            S[i,j,m] = S[i+1,j-1,m] + Rab[m] * S[i,j-1,m]

```

Although Numba offers automatic loop expansion for these operations, manually implemented for-loop is sometimes more efficient. In the code generated by Numba, each ufunc or broadcasting statement incurs additional overhead for retrieving array structure information, performing contiguity checks, and determining loop bounds.

The costs of these extra steps are generally insignificant. However, for small array blocks or when executed repeatedly within loops, their cumulative overhead cannot be ignored.

JIT with non-python mode

The `@numba.njit` decorator can be used to compile computationally intensive functions in `non-python` mode. In this JIT mode, warnings and errors might be generated for any statements that involve Python objects or Python functions. We need to adjust the code accordingly, including:

- Replacing Pythonic iterators such as `enumerate`, `zip`, and generators with basic iterations using the `range()` function. This action is opposite to the optimization practices for pure Python code suggested in Chapter 9.
- Replacing `np.einsum` with `np.dot`. `np.dot` can be captured by Numba and compiled to BLAS functions.
- Replacing `scipy.special` functions with equivalent functions from the NumPy or `math` libraries [6]. It should be noted that Numba has limited support for some NumPy modules, such as `np.fft` or `np.polynomials`. Workarounds or redesigns of algorithms may be necessary if the code utilizes functions from these modules.

Loop unrolling

In the Rys-quadrature algorithm, the RR formulas for the three Cartesian components are independent. The loop over the three Cartesian components can be unrolled to slightly reduce the costs of branching in the for loops. The HRR iteration code in Section 12.1.4.3 can be unrolled as follows:

```
... # Code omitted for brevity
I4dx, I4dy, I4dz = hrr = np.zeros((3, l1j+1, l1j+1, l1k1+1, l1l+1, nroots))
Xab, Yab, Zab = Rab
Xcd, Ycd, Zcd = Rcd
for i in range(l1j+1):
    for k in range(l1k1+1):
        for n in range(nroots):
            I4dx[i,0,k,0,n] = trr[i,k,0,n]
            I4dy[i,0,k,0,n] = trr[i,k,1,n]
            I4dz[i,0,k,0,n] = trr[i,k,2,n]
for i in range(l1j+1):
    for l in range(1, l1l+1):
        for k in range(l1k1+1-l):
            for n in range(nroots):
                I4dx[i,0,k,l,n] = I4dx[i,0,k+1,l-1,n] + Xcd * I4dx[i,0,k,l-1,n]
                I4dy[i,0,k,l,n] = I4dy[i,0,k+1,l-1,n] + Ycd * I4dy[i,0,k,l-1,n]
                I4dz[i,0,k,l,n] = I4dz[i,0,k+1,l-1,n] + Zcd * I4dz[i,0,k,l-1,n]
for j in range(l1j):
    for i in range(l1j-j):
```



```

        Zcd * sub[kx,ky,kz ,lx,ly,lz-1])
for ly in range(1, ll+1-lx):
    for lz in range(ll+1-lx-ly):
        ...
        k1 = 0
        for kx in range(lk, -1, -1):
            for ky in range(lk-kx, -1, -1):
                kz = lk - kx - ky
                for lx in range(ll, -1, -1):
                    for ly in range(ll-lx, -1, -1):
                        lz = ll - lx - ly
                        hrr[ix,iy,iz,0,0,0,k1] = sub[kx,ky,kz,rx,ly,lz]
                        k1 += 1
...

```

Explicit data types in the function signature

Unlike the data types declaration in Cython, providing explicit type information can enhance JIT compilation but not the runtime performance. The automated type inference in the *non-python* mode can accurately determine the types, which are the same as those in the manually created function signatures. However, explicit data type declaration is still beneficial as it can help prevent errors caused by inconsistent types.

Eliminating Python classes

Numba has limited support for Python classes, which means that functions involving Python classes cannot be compiled with the `njit` mode. To handle Python classes, we need to redesign the class making it compatible with the Numba experimental feature `numba.experimental.jitclass`. In the JIT-decorated class, the data types of each attribute should be explicitly declared. For example, the `CGTO` class can be modified as follows:

```

@numba.experimental.jitclass([
    ('angular_momentum', numba.int64),
    ('exponents', numba.float64[:]),
    ('coefficients', numba.float64[:]),
    ('coordinates', numba.float64[:]),
    ('norm_coefficients', numba.float64[:]),
])
class jitCGTO:
    def __init__(self,
                 angular_momentum: int,
                 exponents: np.ndarray,
                 coefficients: np.ndarray,
                 coordinates: np.ndarray):

```

```

self.angular_momentum = angular_momentum
self.exponents = exponents
self.coefficients = coefficients
self.coordinates = coordinates
norm = gto_norm(angular_momentum, exponents)
self.norm_coefficients = norm * coefficients

```

12.1.7 Optimization with Cython compilation

As discussed in Chapter 9, the key strategy in optimizing Cython code is to replace the slow Python objects and Pythonic statements with C-like variables and statements. Here we create a `.pyx` file named `_tensors.pyx` which includes all the functions for compilation.

Most of the optimization techniques are similar to those used in the version compiled by Numba. Particularly, the optimizations we employed for the `pyx` code include:

- Rewriting NumPy ufuncs and broadcasting code with explicit for-loop iterations.
- Replacing Pythonic iterators with simple iterations using the `range()` function.
- Unrolling loops of the three Cartesian components in Rys-quadrature algorithm.
- Specifying data types for all variables and function signatures.
- Declaring the data type, shape, and contiguity for any NumPy arrays using the typed `memoryviews` [7]. For example, the HRR tensor and intermediate tensors in Rys quadrature algorithm are allocated as follows:

```

hrr = np.zeros((3, l1j+1, l1j+1, l1k+1, l1l+1, nroots))
cdef double[:, :, :, :, :, :] I4dx = hrr[0]
cdef double[:, :, :, :, :, :] I4dy = hrr[1]
cdef double[:, :, :, :, :, :] I4dz = hrr[2]

```

This step is an additional task in Cython compared to Numba compilation.

- Enhancing the efficiency of NumPy array indexing with Cython compiler directives:

```

#cython: boundscheck=False
#cython: wraparound=False
#cython: overflowcheck.fold=False
#cython: cdivision=True

```

In this rewriting process, we can use the command

```
cython -a _tensors.pyx
```

to annotate the `.pyx` file and examine whether the Cythonization code is properly optimized. Here, we do not explicitly show the Cythonization code. Based on the Python code developed in Section 12.1.5 and the Cython annotations, readers should be able to develop the Cython version of the integral program.

12.1.8 Optimization with meta-programming techniques

The ERI code that we have developed processes all types of GTOs using the same code path. However, this implementation incurs significant overhead when processing basis sets with low angular momentum. This overhead primarily stems from the branching within the nested loops and the operations required to manage the high-dimensional arrays. To reduce the overhead, we can unroll the code for certain basis functions, especially for those with low angular momentum.

Code unrolling can be achieved using the technique of code generation. To minimize the computation overhead, we decide to output the unrolled code in C, then compile the C code as an extension for the Python program. Here, let's demonstrate the unrolling procedure for the OS ERI scheme. Similar techniques can be applied to other algorithms as well.

The recursive DP function `vrr(n, ix, iy, iz)` in Section 12.1.3.3 is the first function to optimize. We can apply the following steps to unroll the `vrr` computation code:

1. Replacing the return statements in the recursive function.

We need to design a unique name to represent the return value of the `vrr` function. The name can be dynamically generated based on the function arguments, such as `f'vrr_{n}_{ix}{iy}{iz}'`.

2. Replacing the computation statements in the function.

Each Python computation statement should be encapsulated into an f-string. Some modifications are necessary to these statements, such as adding data types and the C-style statement terminator (`:`). These modifications should ensure that the computation statement complies with C programming syntax.

3. Replacing the function call `vrr(...)` with `((vrr...))` in the f-string.

This modification can trigger recursive calls to the `vrr` function in the code generator. Please be aware of the notation `{}` used in these statements. Executing function calls within these notations ensures that the C code for intermediates is generated in the appropriate order. Additionally, any references to `n`, `ix`, `iy`, and `iz` other than those within the call to `{vrr...}` should be encapsulated by `{}`.

4. Using the `lru_cache` for the recursive function.

Applying `lru_cache` to the `vrr` function is crucial, as it prevents the generation of the same intermediate computation code multiple times.

The output of the code generator for `vrr` is as follows:

```
@lru_cache(10000)
def vrr(n, ix, iy, iz):
    val = f'vrr_{n}_{ix}{iy}{iz}'
    if iz > 0:
        print(f'double {val} = Zpa*((vrr(n,ix,iy,iz-1)) - theta/aij*((vrr(n+1,
            ix,iy,iz-1));')
    if iz > 1:
```

```

    print(f'{val} += ({iz}-1)*.5/aij * ({vrr(n,ix,iy,iz-2)}) - theta/aij
    *({vrr(n+1,ix,iy,iz-2)});')
    return val

if iy > 0:
    print(f'double {val} = Ypa*({vrr(n,ix,iy-1,iz)}) - theta/aij*({vrr(n+1,
    ix,iy-1,iz)});')
if iy > 1:
    print(f'{val} += ({iy}-1)*.5/aij * ({vrr(n,ix,iy-2,iz)}) - theta/aij
    *({vrr(n+1,ix,iy-2,iz)});')
    return val

if ix > 0:
    print(f'double {val} = Xpa*({vrr(n,ix-1,iy,iz)}) - theta/aij*({vrr(n+1,
    ix-1,iy,iz)});')
if ix > 1:
    print(f'{val} += ({ix}-1)*.5/aij * ({vrr(n,ix-2,iy,iz)}) - theta/aij
    *({vrr(n+1,ix-2,iy,iz)});')
    return val
print(f'double {val} = Kabcd * _boys[{n}];')
return val

```

We can apply similar procedures to implement the code generator based on the recursive functions `trr` and `hrr`:

```

@lru_cache(10000)
def trr(n, ix, iy, iz, kx, ky, kz):
    val = f'e_trans{n}_{ix}{iy}{iz}_{kx}{ky}{kz}'
    if kz > 0:
        print(f'double {val} = Zqc * ({ttr(n,ix,iy,iz,kx,ky,kz-1)}) +
        Ztheta_akl * ({ttr(n+1,ix,iy,iz,kx,ky,kz-1)});')
    if kz > 1:
        print(f'{val} += {(kz-1)*.5}/akl * ({ttr(n,ix,iy,iz,kx,ky,kz-2}) -
        theta_akl*({ttr(n+1,ix,iy,iz,kx,ky,kz-2)}));')
    if iz > 0:
        print(f'{val} += {iz*.5}/aa * ({ttr(n+1,ix,iy,iz-1,kx,ky,kz-1)});')
    return val

    if ky > 0:
        print(f'double {val} = Yqc * ({ttr(n,ix,iy,iz,kx,ky-1,kz)}) +
        Ytheta_akl * ({ttr(n+1,ix,iy,iz,kx,ky-1,kz)});')
    if ky > 1:
        print(f'{val} += {((ky-1)*.5)}/akl * ({ttr(n,ix,iy,iz,kx,ky-2,kz}) -
        theta_akl*({ttr(n+1,ix,iy,iz,kx,ky-2,kz)}));')
    if iy > 0:

```

```

        print(f'{val} += {iy*.5}/aa * ({trr(n+1,ix,iy-1,iz,kx,ky-1,kz)});')
    return val

    if kx > 0:
        print(f'double {val} = Xqc * ({trr(n,ix,iy,iz,kx-1,ky,kz)}) +'
              f'Xtheta_akl * ({trr(n+1,ix,iy,iz,kx-1,ky,kz)});')
    if kx > 1:
        print(f'{val} += {(kx-1)*.5}/akl * ({trr(n,ix,iy,iz,kx-2,ky,kz)} -'
              f'theta_akl*({trr(n+1,ix,iy,iz,kx-2,ky,kz)}));')
    if ix > 0:
        print(f'{val} += {ix*.5}/aa * ({trr(n+1,ix-1,iy,iz,kx-1,ky,kz)});')
    return val
return vrr(n, ix, iy, iz)

@lru_cache(10000)
def hrr(ix, iy, iz, jx, jy, jz, ky, kz, lx, ly, lz):
    val = f'hrr_{ix}{iy}{iz}_{jx}{jy}{jz}_{ky}{kz}_{lx}{ly}{lz}'
    if lz > 0:
        print(f'{val} = ({hrr(ix,iy,iz,jx,jy,jz,kx,ky,kz+1,lx,ly,lz-1)}'
              f'+ Zcd * ({hrr(ix,iy,iz,jx,jy,jz,kx,ky,kz,lx,ly,lz-1)});')
    return val
    if ly > 0:
        print(f'{val} = ({hrr(ix,iy,iz,jx,jy,jz,kx,ky+1,kz,lx,ly-1,lz)}'
              f'+ Ycd * ({hrr(ix,iy,iz,jx,jy,jz,kx,ky,kz,lx,ly-1,lz)});')
    return val
    if lx > 0:
        print(f'{val} = ({hrr(ix,iy,iz,jx,jy,jz,kx+1,ky,kz,lx-1,ly,lz)}'
              f'+ Xcd * ({hrr(ix,iy,iz,jx,jy,jz,kx,ky,kz,lx-1,ly,lz)});')
    return val
    if jz > 0:
        print(f'{val} = ({hrr(ix,iy,iz+1,jx,jy,jz-1,kx,ky,kz,lx,ly,lz)}'
              f'+ Zab * ({hrr(ix,iy,iz,jx,jy,jz-1,kx,ky,kz,lx,ly,lz)});')
    return val
    if jy > 0:
        print(f'{val} = ({hrr(ix,iy+1,iz,jx,jy-1,jz,kx,ky,kz,lx,ly,lz)}'
              f'+ Yab * ({hrr(ix,iy,iz,jx,jy-1,jz,kx,ky,kz,lx,ly,lz)});')
    return val
    if jx > 0:
        print(f'{val} = ({hrr(ix+1,iy,iz,jx-1,jy,jz,kx,ky,kz,lx,ly,lz)}'
              f'+ Xab * ({hrr(ix,iy,iz,jx-1,jy,jz,kx,ky,kz,lx,ly,lz)});')
    return val
return trr(ix, iy, iz, kx, ky, kz)

```

The three recursive functions form the core of the code generator for the unrolling process.

Given four integers that represent the angular momentum of the four GTOs in an ERI, the unrolling process can be executed to generate C code using the `unrolling` function described below.

```
c_tpl = jinja2.Template('''
void {{ name }}(double *eri, double ai, double aj, double ak, double al,
              double *Ra, double *Rb, double *Rc, double *Rd,
              void (*boys_fn)(int, double, double [])) {
double aij = ai + aj;
double akl = ak + al;
double theta = aij * akl / (aij + akl);
double Xab = Ra[0] - Rb[0];
double Yab = Ra[1] - Rb[1];
double Zab = Ra[2] - Rb[2];
double Xcd = Rc[0] - Rd[0];
double Ycd = Rc[1] - Rd[1];
double Zcd = Rc[2] - Rd[2];
double theta_rij = ai * aj / aij * (Xab*Xab + Yab*Yab + Zab*Zab);
double theta_rkl = ak * al / akl * (Xcd*Xcd + Ycd*Ycd + Zcd*Zcd);
double fac = 34.986836655249725; // 2*pi**2.5
double Kabcd = fac / (aij*akl*sqrt(aij+akl)) * exp(-theta_rij - theta_rkl);
double Xp = (ai * Ra[0] + aj * Rb[0]) / aij;
double Yp = (ai * Ra[1] + aj * Rb[1]) / aij;
double Zp = (ai * Ra[2] + aj * Rb[2]) / aij;
double Xq = (ak * Rc[0] + al * Rd[0]) / akl;
double Yq = (ak * Rc[1] + al * Rd[1]) / akl;
double Zq = (ak * Rc[2] + al * Rd[2]) / akl;
double Xpq = Xp - Xq;
double Ypq = Yp - Yq;
double Zpq = Zp - Zq;
double Xpa = Xp - Ra[0];
double Ypa = Yp - Ra[1];
double Zpa = Zp - Ra[2];
double theta_r2 = theta * (Xpq*Xpq + Ypq*Ypq + Zpq*Zpq);
double Xab_cd = -(aj*Xab + al*Xcd) / akl;
double Yab_cd = -(aj*Yab + al*Ycd) / akl;
double Zab_cd = -(aj*Zab + al*Zcd) / akl;
double _boys[32];
boys_fn({{ order + 1 }}, theta_r2, _boys);
'''')

def unrolling(l1, l2, l3, l4):
```

```

print(c_tpl.render(name=f'run_eri_{li}{lj}{lk}{ll}', order=li+lj+lk+ll))
n = 0
for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
    for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
        for k, (kx, ky, kz) in enumerate(iter_cart_xyz(lk)):
            for l, (lx, ly, lz) in enumerate(iter_cart_xyz(ll)):
                print(f'eri[{n}] = {hrr(ix,iy,iz,jx,jy,jz,kx,ky,kz,rx,ly,lz)};')
                n += 1
print('')

if __name__ == '__main__':
    lmax = 5
    max_roots = 5
    for li in range(lmax):
        for lj in range(lmax):
            for lk in range(lmax):
                for ll in range(lmax):
                    if li + lj + lk + ll > max_roots:
                        continue
                    unrolling(li, lj, lk, ll)

```

Please note that unrolling for basis functions with high angular momentum can lead to a substantial amount of generated C code. To strike a balance between code size and computational efficiency, it is generally sufficient to perform unrolling for low angular momentum GTOs, such as $l_i + l_j + l_k + l_l < 6$.

We can create a driver function in C to select the appropriate unrolled C functions based on the angular momentum of each basis function.

```

driver_tpl = jinja2.Template('''
int run_eri_unrolled(double *eri, int li, int lj, int lk, int ll,
                      double ai, double aj, double ak, double al,
                      double *Ra, double *Rb, double *Rc, double *Rd,
                      void (*boys_fn)(int, double, double []))
{
    %- set ll = lmax + 1 %
    int ijkll = li*{{ ll**3 }} + lj*{{ ll**2 }} + lk*{{ ll }} + ll;
    switch (ijkll) {
    %- for li in range(ll) %
    %- for lj in range(ll) %
    %- for lk in range(ll) %
    %- for ll in range(ll) %
    %- if li + lj + lk + ll <= max_roots %
    %- set ijkll = li*ll**3 + lj*ll**2 + lk*ll + ll %
    case {{ ijkll }}: run_eri_{{ li }}{{ lj }}{{ lk }}{{ ll }}(eri, ai, aj,
                  ak, al, Ra, Rb, Rc, Rd, boys_fn); break;
}

```

```
{%- endif %}  
{%- endfor %}  
{%- endfor %}  
{%- endfor %}  
{%- endfor %}  
    default: return 1; }  
    return 0;  
}...)  
  
print(driver_tpl.render(lmax=5, max_roots=5))
```

The last argument of the C function is a function pointer to the Boys function, which is defined with the following signature:

```
void (*boys_fn)(int max_order, double x, double output[]);
```

In the `vrr` function, the Boys function is accessed multiple times. To enhance efficiency, all values of the Boys function up to the given order `max_order` can be computed once and stored in the `output` buffer.

However, the Boys function is currently implemented in Python. How to access the Boys function in C? There are three possible approaches to consider:

- Translating the Python implementation of the Boys function into C code and then integrating this C function into the C extension. However, this solution might be impractical for complex Python code. The code translation may require converting all dependencies from Python to C, involving a notable amount of programming effort.
- Precomputing the Boys function in Python and then passing the results to the C integral function. This method is inefficient as some intermediate variables might need to be computed in both Python and C.
- Creating a C wrapper for the Python function and passing the function pointer of the C wrapper to the C integral code.

In this example, the implementation of the Boys function is not overly complex. Translating the Boys function into C code is a feasible approach. Nevertheless, we chose to use the function pointer with the C wrapper because it requires minimal coding effort.

How can we execute these C functions within the framework of Python compilation, assuming that we want to use Python compilation techniques for the remaining parts of the code? Integrating C functions in Cython is straightforward, as Cython supports the execution of external C functions. Let's explore how this can be accomplished with Numba.

If a Python function is compiled in `numba.jit` mode, external C functions can be executed using the `ctypes` library. However, only limited functionalities of `ctypes` are supported by the `numba.njit` mode. The pointer object created by `ctypes.POINTER` is one such example. To pass a pointer to C, as a workaround, we must convert it to

`ctypes.c_void_p`, as demonstrated in line (1) and (2) - (6) of the following code example.

```
from rys_roots import boys

liberi_OS = ctypes.CDLL('liberi_OS.so')
run_eri_unrolled = liberi_OS.run_eri_unrolled
run_eri_unrolled.argtypes = [
    ctypes.c_void_p, # eri # (1)
    ctypes.c_int, # li
    ctypes.c_int, # lj
    ctypes.c_int, # lk
    ctypes.c_int, # ll
    ctypes.c_double, # ai
    ctypes.c_double, # aj
    ctypes.c_double, # ak
    ctypes.c_double, # al
    ctypes.c_void_p, # Ra # (2)
    ctypes.c_void_p, # Rb # (3)
    ctypes.c_void_p, # Rc # (4)
    ctypes.c_void_p, # Rd # (5)
    ctypes.c_void_p, # boys_fn # (6)
]

@numba.cfunc('void(int32, double, numba.types.CPointer(double))',
             nopython=True)
def boys_for_c(m, t, out):
    out_array = numba.carray(out, (m+1,)) # (7)
    out_array[:] = boys(m, t)

boys_fn = boys_for_c.address # (8)
```

To access the Boys function implemented in Python, we can create a C wrapper using the `numba.cfunc` decorator. The function pointer can be accessed via the `address` attribute of the C wrapper, as demonstrated in line (8). Please note that this wrapped Python function is invoked by a C function at runtime. Consequently, the data types within this function cannot be accurately inferred by the Numba JIT compiler. It is necessary to explicitly declare the data types of all input arguments for this function. Additionally, for the array object used in this function, we need to specify their shapes, as shown in line (7).

After applying the unrolling optimization for the `primitive_ERI` function, the overhead in the GTO contraction function (Section 12.1.3.5) becomes more pronounced. To reduce its overhead, we further optimize the `contracted_ERI` function and compile it using the `numba.njit`.


```
out[i0+i,j0+j,k0+k,l0+l] += fac * buf[i,j,k,l] # (12)
```

The JIT compilation greatly reduces the overhead of the nested loops and the cost of data movement for the code from line (3) to line(12). Additionally, to minimize memory management overhead, we pass the output ERI tensor directly to the contracted_ERI function, as indicated in line (1). This change allows the computation results to directly write to the output tensor. Code unrolling is performed for the ERIs computation that satisfies the condition $l_i + l_j + l_k + l_l < 6$, as shown in line (2). If this condition is not met, the contracted_ERI function will revert to the Numba-compiled Rys algorithm.

Following the contracted_ERI function, we need a program to construct the ERI tensor for a list of GTO bases. This program places the function call contracted_ERI() within four nested loops.

```
def pack_gto_attrs(gtos):
    '''Pack GTO attributes, making them more efficient to load'''
    offsets = gto_offsets(gtos)
    gto_params = []
    for i, bas_i in enumerate(gtos):
        i0 = offsets[i]
        li = bas_i.angular_momentum
        Ra = bas_i.coordinates
        exps_i = bas_i.exponents
        norm_ci = bas_i.norm_coefficients
        gto_params.append((i0, li, Ra, exps_i, norm_ci))
    return gto_params

def eri_OS_unrolled(gtos):
    offsets = gto_offsets(gtos)
    gto_params = pack_gto_attrs(gtos)
    nao = offsets[-1]
    out = np.zeros((nao, nao, nao, nao))
    for ((i0, li, Ra, exps_i, norm_ci),
          (j0, lj, Rb, exps_j, norm_cj),
          (k0, lk, Rc, exps_k, norm_ck),
          (l0, ll, Rd, exps_l, norm_cl)) \ 
        in itertools.product(*(gto_params,)*4):
        contracted_ERI(li, lj, lk, ll, exps_i, exps_j, exps_k, exps_l,
                      norm_ci, norm_cj, norm_ck, norm_cl, Ra, Rb, Rc, Rd,
                      out, i0, j0, k0, l0)
    return out
```

A special optimization we implemented in this program is the pack_gto_attrs function. This helper function packs the necessary attributes of the GTO instances into a five-element tuple. Unpacking variables from a single tuple is simpler and more

efficient than accessing multiple attributes via separate statements. It is slightly more efficient to use the iterator `itertools.product` along with the `pack_gto_attrs` function than employing four nested `for` loops.

12.1.9 Performance benchmark

We now proceed to benchmark the performance of the ERI program. In the benchmark tests, we generate a four-dimensional ERI tensor for a benzene molecule using the Pople basis 6-31G(*). This basis set comprises a total of 102 Cartesian basis functions. The test script is provided below.

```
import time
import numpy as np
from basis import CGTO, Molecule, n_cart, gto_offsets

def build_eri_tensor(f, gtos):
    offsets = gto_offsets(gtos)
    nao = offsets[-1]
    V = np.empty((nao, nao, nao, nao))
    for i, bas_i in enumerate(gtos):
        i0, i1 = offsets[i], offsets[i+1]
        for j, bas_j in enumerate(gtos):
            j0, j1 = offsets[j], offsets[j+1]
            for k, bas_k in enumerate(gtos):
                k0, k1 = offsets[k], offsets[k+1]
                for l, bas_l in enumerate(gtos):
                    l0, l1 = offsets[l], offsets[l+1]
                    V[i0:i1,j0:j1,k0:k1,l0:l1] = contracted_ERI(
                        f, bas_i, bas_j, bas_k, bas_l)
    return V

def timing(f, gtos):
    build_eri_tensor(f, gtos[:1]) # warm up
    t0 = time.perf_counter()
    v = build_eri_tensor(f, gtos)
    t1 = time.perf_counter()
    return t1 - t0

benzene = '''
C      0.0000    1.3990    0.0000
C     -1.2095    0.7442    0.0000
C     -1.2095   -0.7442    0.0000
C      0.0000   -1.3990    0.0000
C      1.2095   -0.7442    0.0000
C      1.2095    0.7442    0.0000
```

```

H      0.0000    2.4939    0.0000
H     -2.1994    1.3083    0.0000
H     -2.1994   -1.3083    0.0000
H      0.0000   -2.4939    0.0000
H      2.1994   -1.3083    0.0000
H      2.1994    1.3083    0.0000
...
mol = Molecule.from_xyz(benzene)
gtos = mol.assign_basis({'C': '6-31G*', 'H': '6-31G'})

import eri_OS, eri_MD_numba, eri_MD_cython, ...
for imp in (eri_OS, eri_MD_numba, ...):
    print(f'{imp} timing:', timing(imp.primitive_ERI, gtos))

```

The performance is gauged by the CPU time using the `time.perf_counter` function in this example. To obtain a more accurate measurement of the actual time expended, including the time spent in kernel space, wall time can be measured using the `time.time` function.

The following code implementations are evaluated in the benchmark tests: the pure Python implementation, the Numba compiled program, the Cython compiled program, and the unrolled implementation. We exclude the first two Python versions (the recursive DP code in Section 12.1.3 and the naive iteration code in Section 12.1.4) from the test because they are too slow. The implementation in Section 12.1.5, which features an optimized iteration structure, represents the pure Python approach in the benchmark. As a reference, we also benchmark the integral computation using the PySCF package,

```
pyscf.M(atom=benzene,
        basis={'C': '6-31g*', 'H': '6-31g'}).intor('int2e_cart')
```

Please note that the values of the ERIs computed by PySCF package differ from the ERIs demonstrated in this book due to different basis function normalization. PySCF applies the Racah normalization for *s*-type and *p*-type basis functions only. If you wish to match the PySCF output, you can modify the `gto_norm` function in Section 12.1.1 to

```

def gto_norm(l, expnt):
    '''Radial part normalization'''
    norm = (gaussian_int(l*2+2, 2*expnt)) ** -.5
    if l < 2:
        norm *= ((2*l+1)/(4*np.pi))**.5
    return norm

```

The benchmark results are summarized in Table 12.2. The compilation technique provides a significant acceleration, yielding a speed-up factor of about 20 times compared to the pure Python implementation. The code unrolling optimization further

Table 12.2 ERI performance benchmark.

Implementation	Section	CPU time (seconds)
OS w/o compilation	12.1.5	18752
Numba MD	12.1.6	402
Numba OS	12.1.6	1055
Numba Rys	12.1.6	432
Cython MD	12.1.7	1973
Cython OS	12.1.7	845
Cython Rys	12.1.7	1283
Numba Rys, unrolled OS	12.1.8	33.1
Numba Rys, unrolled OS, jitclass	12.1.8	22.8
PySCF		9.2

enhances performance, resulting in around 30 times speed-up. In the unrolled integral code, the presence of the Python class `CGTO` introduces non-negligible overhead. By replacing the `CGTO` class with the experimental `jitclass` from Numba, an additional 40% speed-up is achieved on top of the unrolled version. When compared to the PySCF integral program, which is aggressively optimized in C, the concise Python integral program is 2.5 times slower.

For the Rys-quadrature and MD algorithms, the Numba JIT-compiled code operates significantly faster than the Cython-generated C code. The Numba versions are 3-6 times faster than their Cython counterparts. Several NumPy functions, such as `np.dot`, `np.reshape`, `np.linalg.inv` and `np.roots`, are employed in the two algorithms. They are automatically replaced by the more efficient Numba alternatives by the Numba compiler. However, in the Cython version, these functions are still executed in the Python mode, resulting in high overhead.

In the OS scheme, The performance of Numba version is slower than the Cython compilation. In this scenario, except for a few calls to the `numpy.empty` function, there are no more Python elements in the program. The overhead from Python code execution is minimal. Since the OS algorithm involves extensive operations on high-dimensional arrays, the main computational costs are contributed by the calculation of addresses for high-dimensional arrays. The indexing code generated by the Numba JIT compiler involves more operations to compute the address, and thereby is less efficient than the code produced by the Cython transpiler.

12.1.10 Other performance factors

From the previous benchmark tests, we observed a 2.5 times performance difference between the Python integral program and the aggressively optimized integral program in PySCF. The performance differences can be partially attributed to the numerous Python objects utilized in the example program, while the integral code in PySCF is fully developed in the C programming language. Besides the overhead of

manipulating Python objects, are there other factors impacting the performance of the integral program? What optimizations can be applied to the Python code without a complete rewrite in C? We need a good understanding of the additional factors before proceeding with future optimizations.

Integral screening

Integral screening can effectively reduce the number of integrals that need to be computed. One method to filter integrals is through the ERI upper bound estimation, which is based on the Schwartz inequality:

$$|(\chi_i \chi_j | \chi_k \chi_l)| \leq \sqrt{|(\chi_i \chi_j | \chi_i \chi_j)| |(\chi_k \chi_l | \chi_k \chi_l)|} \leq Q_{ij} Q_{kl}. \quad (12.60)$$

Here, Q_{ij} represents the maximum value of $\sqrt{|(\chi_i \chi_j | \chi_i \chi_j)|}$ among all integrals within the GTO shells i and j . The Schwartz inequality estimator can be applied within the `get_eri_tensor` function to filter out ERIs when iterating over shells. If the value of $Q_{ij} Q_{kl}$ is smaller than predefined threshold, all ERIs within the shell-quartet (i, j, k, l) can be discarded. Additionally, the Schwartz estimator can be utilized within the `contracted_ERI` function to filter the primitive GTO shell-quartets.

Loop orders

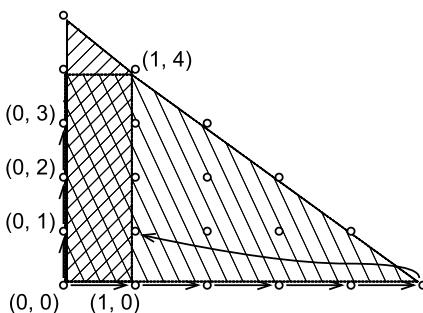
Based on the angular momentum of the four GTO shells in the ERIs, computational costs can be minimized by optimizing the order of nested loops in VRR, TRR, and HRR iterations.

In the VRR iteration, the Cartesian powers are first raised to $l_i + l_j + l_k + l_l$ on the first electron, and then transferred to the second electron until the Cartesian power reaches $l_k + l_l$ on the second electron. During this process, our current implementation has to generate all intermediates with orders greater than $l_i + l_j$ for the first electron and less than $l_k + l_l$ for the second electron. If $l_i + l_j < l_k + l_l$, it would be more efficient to raise the Cartesian powers on the second electron and then transfer them to the first electron (Fig. 12.3).

Similar optimization of the loop order can be applied in the TRR and HRR iterations. For instance, when $l_k < l_l$, the program would require fewer computational costs if the Cartesian powers are transferred to the fourth center in the TRR, and the Cartesian powers at the third center are filled in the HRR.

Polynomial approximation to the intermediates

To compute the intermediates such as the Boys function and the roots and weights of the Rys-quadrature, the code involves some computationally intensive functions including `erf`, eigen solvers, and linear equation solvers. These intermediates can be precomputed and tabulated. Their values can then be approximated using polynomial or spline fitting techniques. These techniques will be discussed in more details in Section 12.2.3.

**FIGURE 12.3**

Optimizing the Order of Nested Loops in RR Iterations. The computational cost of RR iterations is proportional to the area of a trapezoid. The cost can be effectively minimized by selecting the optimal iteration direction, either vertical or horizontal, depending on the target coordinates.

Early contraction and contraction paths

As described in Section 12.1.3.5, the integral contraction can be applied prior to invoking TRR or HRR. Implementing an early contraction scheme can reduce FLOP counts for ERIs of contracted GTOs. To optimize the contraction path, various ERI contraction schemes can be considered, such as the Head-Gordon-Pople (HGP) algorithm [8] and the PRISM algorithm [9].

Sparsity in contraction coefficients

In the integration program demonstrated previously, we only considered segment contracted basis sets. In quantum chemistry calculations, the generally contracted basis sets are also widely used. In such basis sets, like cc-pVDZ, the same primitive GTOs are shared among multiple contracted GTOs. In the generally contracted basis, some contraction coefficients are strictly zeros. This sparsity in contraction coefficients can be leveraged to reduce computational costs.

Precomputing and memoization

Some intermediates for the pairs of primitive GTOs are repeatedly computed across different shell quartets. Some of these intermediates, such as $e^{-\theta_{ij}} R_{AB}^2$ and $e^{-\theta_{kl}} R_{CD}^2$, are expensive arithmetic functions. To improve efficiency, these intermediates can be evaluated and cached in advance.

Bypassing intermediate arrays

In the current Python implementation, we create multiple temporary arrays for storing intermediates. The storage of some intermediates can be optimized. By properly organizing the structure of intermediates and directly writing the data to the result array, we may avoid unnecessary data movement.

For instance, the TRR tensor in the OS algorithm, as developed in Section 12.1.4.2, can be embedded within the HRR tensor. By appropriately designing the shape of the HRR tensor, the TRR tensor can be treated as a sub-block of the HRR tensor. Consequently, when the TRR tensor is evaluated, its results automatically transfer to the corresponding part of the HRR tensor:

```

l4 = l1j + lk1
hrr = np.zeros((l4+1,l4+1,l4+1, lk1+1,lk1+1,lk1+1, ll+1,ll+1,ll+1))
trr = hrr[:,::,::,::,0,0,0]
for kx, ky, kz in product(range(lk1+1), range(lk1+1), range(lk1+1)):
    for ix in range(l4+1-kx):
        for iy in range(l4+1-ky):
            for iz in range(l4+1-kz):
                trr[ix,iy,iz,kx,ky,kz] = ...

for lx, ly, lz in product(range(ll+1), range(ll+1), range(ll+1)):
    for kx in range(lk1+1-lx):
        for ky in range(lk1+1-ly):
            for kz in range(lk1+1-lz):
                if lx > 0:
                    hrr[:,::,kx,ky,kz,lx,ly,lz] = \
                        (      hrr[:,::,kx+1,ky,kz,lx-1,ly,lz] +
                            Xcd * hrr[:,::,kx ,ky,kz,lx-1,ly,lz])
...

```

Another example is the primitive ERIs returned by the `primitive_ERI` function, which act as intermediates for the integral contraction program. By fusing the integral contraction and the computation of primitive integrals, we can eliminate the need to first generate primitive ERIs in a buffer, then multiply these by the contraction coefficients, and finally add them to the output. This optimization can save at least N_p^4 operations for each integral, assuming N_p represents the number of primitive GTOs.

The above factors primarily optimize the integral program by reducing the FLOP counts. In addition to the FLOP counts, it is also necessary to consider the characteristics of computer architecture when developing high-performance integral programs. Several typical factors include the latency of memory accessing, the cache hit rates, and the effectiveness of branch prediction. To optimize these factors, the following programming techniques can be employed.

Unrolling

The inner dot product in the Rys ERI algorithm and the Cartesian to spherical GTO transformation can be unrolled. Additionally, the basis contraction code can also be unrolled, given that the contraction transformations are limited to a few patterns.

Unrolling is very effective when applied to integrals involving low angular momentum GTOs, as previously demonstrated. However, the benefits of a complete

unrolling for high angular momentum basis functions may be limited, or even harmful to efficiency. This is because excessive unrolling can significantly increase the size of the binary code, which in turn imposes a greater demand on the instruction cache (I-cache). The pressure on the I-cache may negatively affect the performance, due to a higher likelihood of cache misses and the increased latency in decoding instructions.

Instruction level parallelism

The SIMD parallelization might be applicable in various parts of the integral code, such as integral contraction, HRR iterations, and the inner dot in the Rys ERI algorithm. To assist the compiler in identifying code that is SIMD parallelizable, the `restrict` keyword [10], the compiler directive `#pragma GCC ivdep` [11], and compiler intrinsics [12] for explicit vectorization can be utilized. To achieve efficient data loading and storing through SIMD operations, memory alignment needs to be considered.

Caching the addresses of tensors for RR iterations

Iterating over the elements in the RR iterations may involve non-sequential access to the elements of a high-dimensional array. The iteration range in this process is typically quite short, leading to a high number of branches due to iteration boundaries. Branch prediction often fails when dealing with boundaries of these short iterations. Additionally, indexing high-dimensional array consumes a significant amount of CPU resources to compute memory addresses. One strategy to reduce these overheads is to transform the high-dimensional array into a one-dimensional vector and cache all memory addresses referenced by the RR iterations. By doing so, accessing an element in the high-dimensional array can be simplified to a few indexing operations. Furthermore, this transformation simplifies the loop structure, thereby reducing the penalty of branching mispredictions within the loop.

For example, the HRR iterations in the OS algorithm exhibit the following loop structure:

```
hrr = np.empty((lkl+1, lkl+1, lkl+1, ll+1, ll+1, ll+1))
hrr[:, :, :, 0, 0, 0] = trr
for lx in range(1, ll+1):
    for ly in range(ll+1-lx):
        for lz in range(ll+1-lx-ly):
            lsum = lx + ly + lz
            for kx, ky, kz in reduced_cart_iter(lkl-lsum):
                hrr[kx, ky, kz, lx, ly, lz] = (hrr[kx+1, ky, kz, lx-1, ly, lz] +
                                                Xcd * hrr[kx-1, ky, kz, lx-1, ly, lz])
```

This iteration code can be optimized to

```
def hrr_address(lk, ll):
    lkl = lk + ll
    def addr(kx, ky, kz, lx, ly, lz):
```

```

        k_idx = (kx * (lkl+1) + ky) * (lkl+1) + kz
        return ((k_idx * (ll+1) + lx) * (ll+1) + ly) * (ll+1) + lz

    adr0, adr1, adr2 = [], [], []
    for lx in range(1, ll+1):
        for ly in range(ll+1-lx):
            for lz in range(ll+1-lx-ly):
                lsum = lx + ly + lz
                for kx, ky, kz in reduced_cart_iter(lkl-lsum):
                    adr0.append(addr(kx , ky, kz, lx , ly, lz))
                    adr1.append(addr(kx+1, ky, kz, lx-1, ly, lz))
                    adr2.append(addr(kx , ky, kz, lx-1, ly, lz))
    return adr0, adr1, adr2

# cache the addresses for all possible angular momentum combinations
lmax = 5
address_cache = {(lk, ll): hrr_address(lk, ll)
                  for lk in range(lmax) for ll in range(lmax)}

def primitive_ERI(lj, ll, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    ...
    hrr = np.empty((lkl+1,lkl+1,lkl+1, ll+1,ll+1,ll+1))
    hrr[:, :, :, 0, 0, 0] = trr
    hrr_view = hrr.ravel()
    adr0, adr1, adr2 = address_cache[lk, ll]
    for i in range(len(adr0)):
        hrr_view[adr0[i]] = hrr_view[adr1[i]] + Xcd * hrr_view[adr2[i]]
    ...

```

Reusing registers to reduce the costs of memory access

Even when data are available in the high-speed L1 cache, the latency of memory access cannot be neglected. In the RR iterations, it is possible to reduce memory access costs by reusing data stored in registers. For example, the overlap integral function contains the following HRR iterations:

```

for j in range(1, lj+1):
    for i in range(lij+1-j):
        S[i,j] = Xab * S[i,j-1] + S[i+1,j-1]

```

To evaluate one element $S[i,j]$, this code requires two memory accesses to load $S[i,j-1]$ and $S[i+1,j-1]$. $S[i+1,j-1]$ can be reused in the adjacent iterations, which reduces the memory access counts by 50%. This optimization transforms the code into

```

for j in range(1, ljj+1):
    s0 = S[0,j-1]
    for i in range(ljj+1-j):
        s1 = S[i+1,j-1]
        S[i,j] = Xab * s0 + s1
        s0 = s1

```

Custom memory allocator

The operation of memory allocation for intermediate arrays also has a non-negligible impact on performance. Calling the `malloc` function provided by the `libc` library in C can noticeably slow down performance. Allocating new memory not only consumes many CPU cycles but also leads to CPU cache misses and page faults. To minimize the overhead of memory allocation, custom memory allocators can be employed. For instance, the integral code in PySCF actually employs a stack allocator [13] to manage memory.

Apart from the contraction path optimization, all the mentioned optimization factors have been utilized in the PySCF integral code, which is written in C language. They are the cause of the 2.5 times performance difference compared to the current Python integral program.

In the Python version of the integral program, implementing explicit SIMD parallelization and custom memory allocators is challenging. These techniques require interactions with low-level system operations. Nonetheless, other optimization factors can be applied within the Python code. However, these aggressive optimizations will significantly increase the complexity and reduce the readability of the Python code. We will not delve further into the implementation details of these optimizations.

12.2 Numerical integration

Numerical integration is a common technique in quantum chemistry programs for calculating integrals. Given integration grids \mathbf{r}_n and the associated weights (volumes) w_n , an integral in 3D space can be evaluated as:

$$\int f(x)dx = \sum_n f(\mathbf{r}_n)w_n. \quad (12.61)$$

Implementing a numerical integration scheme is straightforward. It just requires evaluating the function at the integration grids and then performing a dot product with the weights to obtain the integral value. The integration grids can be either uniformly spaced or sampled through quadrature methods.

12.2.1 Gaussian quadrature

By selecting appropriate weights and abscissas, numerical integration can be made exact for a specific class of integrands. These integrands can generally be written as the product of a polynomial and a well-established weight function $W(x)$.

For instance, the overlap integral can be effectively computed using Gauss-Hermite quadrature. This is achieved by evaluating the polynomial part of the Gaussian orbital product at the roots of the Gauss-Hermite quadrature:

$$\int (x_P + X_{PA})^{i_x} (x_P + X_{PB})^{j_x} e^{-\alpha_{ij} x_P^2} dx = \sum_n w_n (x_n + X_{PA})^{i_x} (x_n + X_{PB})^{j_x}. \quad (12.62)$$

To align with the weight function e^{-x^2} in Gauss-Hermite quadrature, the weights and abscissas are scaled by $1/\sqrt{\alpha_{ij}}$, as illustrated in lines (1) and (2) of the following code snippet. The integral computation, in terms of the quadrature formula, is implemented as follows:

```
def primitive_overlap(li, lj, ai, aj, Ra, Rb):
    aij = ai + aj
    Rab = Ra - Rb
    Rp = (ai * Ra + aj * Rb) / aij
    Rpa = Rp - Ra
    Rpb = Rp - Rb
    theta_ij = ai * aj / aij
    Kab = np.exp(-theta_ij * Rab**2)

    r, w = roots_hermite((li+lj+2)//2)
    rt = r / aij**.5
    wt = w / aij**.5 # (1)
    # (2)
    poly_i = (Rpa[:,None,None] + rt)**(np.arange(li+1)[:,None])
    poly_j = (Rpb[:,None,None] + rt)**(np.arange(lj+1)[:,None])
    I2d = np.einsum('x,xin,xjn->xijn', Kab, poly_i, poly_j)
    Ix, Iy, Iz = np.einsum('n,xijn->xij', wt, I2d)

    nfi = n_cart(li)
    nfj = n_cart(lj)
    S = np.zeros((nfi, nfj))
    for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
        for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
            S[i,j] = Ix[ix,jx] * Iy[iy,jy] * Iz[iz,jz]
    return S
```

12.2.2 Generating quadrature roots and weights

The `scipy.special` module provides a series of commonly used quadratures:

- `roots_legendre` for the weight function 1 over the interval $[-1, 1]$.
- `roots_hermite` for the weight function e^{-x^2} over $(-\infty, \infty)$.
- `roots_laguerre` for the weight function e^{-x} over $[0, \infty)$.
- `roots_chebyt` for the weight function $\frac{1}{\sqrt{1-x^2}}$ over $[-1, 1]$.
- `roots_chebyu` for the weight function $\sqrt{1-x^2}$ over $[-1, 1]$.

In certain scenarios, we may need to compute integrals for a new weight function. A systematic approach can be employed to construct the quadrature roots and weights for the orthogonal polynomials of any given weight function. To demonstrate this process, we will take Rys quadrature as an example, generating the roots and weights of the Boys polynomials.

12.2.2.1 Boys function

The first step in constructing an integral quadrature is to evaluate the moments of the weight function

$$\int_a^b x^n W(x) dx, \quad (12.63)$$

where n is a non-negative integer and $W(x)$ is the weight function. In the Rys quadrature, the weight function is e^{-tx^2} and the integral interval is $[0, 1]$,

$$\int_0^1 e^{-tx^2} f(x^2) dx = \sum_n w_n f(x_n^2). \quad (12.64)$$

The moments of this weight function are given by the Boys function.

$$F_n(t) = \int_0^1 x^{2n} e^{-tx^2} dx. \quad (12.65)$$

The Boys function can be computed using an upward recursion formula

$$F_n(t) = \frac{-e^{-t}}{2t} + \frac{2n-1}{2t} F_{n-1}(t), \quad (12.66)$$

$$F_0(t) = \frac{1}{2} \sqrt{\frac{\pi}{t}} \operatorname{erf}(\sqrt{t}). \quad (12.67)$$

However, repeatedly applying this formula N times can amplify the numerical errors in F_0 by a factor of $\frac{(2N-1)!!}{(2t)^N}$. If t is a small value, it is not appropriate to use this upward recursion formula to compute high-order Boys functions from F_0 . Instead, we should reformulate the recursion formula and adopt a downward recursion approach:

$$F_{n-1}(t) = \frac{e^{-t}}{2n-1} + \frac{2t}{2n-1} F_n(t). \quad (12.68)$$

The starting point of this recursion is

$$F_n(t) = \sum_{m=n}^{\infty} \frac{(2n-1)!!(2t)^{m-n}e^{-t}}{(2m+1)!!}. \quad (12.69)$$

This series sum can be truncated at a sufficiently large m . The downward recursion is generally more accurate than the upward recursion. However, its performance is slower than the upward version when t is a large number. In situations where the accuracy of upward recursion is acceptable, the upward recursion is often preferred. The turnover point, at which the accuracy of the upward recursion becomes unacceptable, can be tested and tabulated beforehand.

Approximately, when t is near $n + 1.5$, the accuracy of the upward and downward recursions is comparable. Consequently, we create a program that combines the two versions of the recursion formulas.

```
def boys(n, t):
    assert n >= 0
    assert t >= 0
    if t < n + 1.5:
        return downward(n, t)
    else:
        return upward(n, t)

def upward(n, t):
    tt = math.sqrt(t)
    f = math.sqrt(math.pi)/2 / tt * math.erf(tt)
    e = math.exp(-t)
    out = [f]
    for i in range(n):
        f = ((2*i+1) * f - e) / (2*t)
        out.append(f)
    return out

def downward(n, t, prec=1e-15):
    b = n + .5
    e = .5 * math.exp(-t)
    x = e
    f = e
    while x > prec * e:
        b += 1
        x *= t / b
        f += x

    b = n + .5
    f /= b
```

```

out = [f]
for i in range(n):
    b -= 1
    f = (e + t * f) / b
    out.append(f)
return out[::-1]

```

12.2.2 General routine to solve quadrature roots and weights

Rys polynomials, denoted as $p_i(x) = \sum c_{ik}x^k$, form orthogonal sets with respect to the weight function:

$$\langle p_i, p_j \rangle = \int_0^1 p_i(x) p_j(x) e^{-tx^2} dx = \delta_{ij}. \quad (12.70)$$

This orthogonality property can be translated into a linear algebra problem for the coefficients \mathbf{c} with respect to the Boys function:

$$\sum_{kl} c_{ik} c_{jl} F_{k+l} = \delta_{ij}. \quad (12.71)$$

To find the roots of the Rys polynomial of order N , we can use the `numpy.roots` function to solve the coefficients in the $(N+1)$ -th column of the coefficients matrix \mathbf{c} .

The integrals between two Rys polynomials, p_i and p_j , can be computed using the roots x_n and the weights w_n of the N -th Rys polynomial:

$$\langle p_i, p_j \rangle = \sum_n w_n p_i(x_n) p_j(x_n) = \sum_{nkl} w_n c_{ik} x_n^k c_{jl} x_n^l = \delta_{ij}. \quad (12.72)$$

The solutions to this linear equation are the quadrature weights w_n , given by:

$$w_n = \frac{1}{\sum_{ik} (c_{ik} x_n^k)^2}. \quad (12.73)$$

Based on these equations, we obtain the `rys_roots_weights` function, which computes the roots and weights of Rys polynomials as follows:

```

def schmidt_orth(moments, nroots):
    ...
    Returns the coefficients of the orthogonal polynomials

    \sum_{kl} c_{i,k} c_{j,l} moments_{k+1} = \delta_{ij}
    ...
    s = np.zeros((nroots+1, nroots+1))
    for j in range(nroots+1):

```

```

        for i in range(nroots+1):
            s[i,j] = moments[i+j]
    return np.linalg.inv(np.linalg.cholesky(s))

def find_polyroots(cs, nroots):
    return np.roots(cs[::-1])

def rys_roots_weights(nroots, t):
    moments = boys(nroots*2, t)
    cs = schmidt_orth(moments)
    roots = find_polyroots(cs[nroots])

    rtp = roots[:,np.newaxis]**np.arange(nroots+1)
    weights = 1. / (rtp.dot(cs.T)**2).sum(axis=1)
    return roots, weights

```

The `rys_roots_weights` code demonstrated above is actually a general function that can also be used to calculate quadratures for other types of weight functions. The only change required is to replace the `Boys` function with the desired `moments` function. The remaining part of the function, which computes the coefficients of the orthogonal polynomials and determines the roots and weights, remains the same.

However, this implementation may encounter severe numerical stability issues when solving for high-order quadrature roots. The numerical issues arise from the highly oscillated polynomial coefficients in the high-order terms. In practice, when the quadrature order exceeds 12, the accuracy provided by double precision, which has 16 significant digits, becomes insufficient. As a result, the program might fail during the orthogonalization procedure or the root finding function. Even if no errors occur, significant errors in the computed quadrature roots and weights may still be encountered.

There are some alternative algorithms to improve the numerical accuracy. The rough idea is to use well-known orthogonal polynomials as the basis functions to represent the target polynomials. The auxiliary polynomials can effectively sample the oscillatory polynomial coefficients, replacing them with mild coefficients, thereby reducing numerical instability. For instance, using Laguerre polynomials or Jacobi polynomials to represent Rys polynomials can yield higher-precision results [5].

12.2.2.3 Improved accuracy with the `mpmath` library

We can adopt another approach to improve numerical accuracy in Python. By using the arbitrary precision floating-point library `mpmath`, we can compute high-precision results without resorting to the alternative quadrature roots computation algorithms. To ensure that we do not accidentally lose precision, we need to make the following adjustments to the code implemented in Section 12.2.2.1 and Section 12.2.2.2:

- In the `boys` function, we should replace all `math` functions with the equivalent functions provided by `mpmath`.

- In the `rys_roots_weights` function, we need to reimplement certain NumPy functions, such as the root-finding function and the Schmidt orthogonalization, using the corresponding functions from the `mpmath` library. For the algorithm of the root-finding function, please refer to the underlying theory in Chapter 9 of the book *Numerical Recipes* [14]. Additionally, we utilize custom data types for NumPy arrays to enable vectorized operations on `mpf` data. This approach is discussed in detail in Section 2.2.4 of Chapter 2.
- Finally we should replace all floating-point constants with the multiple precision numbers using the `mpmath.mpf` function. For instance, replace `0.5` with `mpmath.mpf('0.5')`.

The outcome of these modifications is displayed in the following code example:

```

import mpmath as mp
mp.mp.dps = 30
mp.mp.pretty = True

def schmidt_orth(moments, nroots):
    """
    Returns the coefficients of the orthogonal polynomials

    \sum_{k\geq 0} c_{i,k} c_{j,l} moments_{k+l} = \delta_{ij}
    """

    cs = np.full((nroots+1, nroots+1), mp.mpf(0))
    for j in range(nroots+1):
        fac = moments[j+j]
        for k in range(j):
            dot = cs[k,:j+1].dot(moments[j:j+j+1])
            cs[j,:j] -= dot * cs[k,:j]
            fac -= dot * dot

        if fac <= 0:
            raise RuntimeError(f'schmidt_orth fail. (nroots={nroots}, fac={fac})')
        fac = fac**mp.mpf('-.5')
        cs[j,j] = fac
        cs[j,:j] *= fac
    return cs

def find_polyroots(cs, nroots):
    assert len(cs) == nroots + 1
    if nroots == 1:
        return np.array([-cs[0] / cs[1]])

    A = mp.matrix(nroots)
    for m in range(nroots-1):

```

```

A[m+1,m] = mp.mpf(1)
for m in range(nroots):
    A[0,m] = -cs[nroots-1-m] / cs[nroots]
roots = mp.eig(A, left=False, right=False)
return np.array([x.real for x in roots])

def rys_roots_weights(nroots, t):
    moments = boys(nroots*2, mp.mpf(t))
    cs = schmidt_orth(moments, nroots)
    roots = find_polyroots(cs[nroots], nroots)

    rtp = roots[:,np.newaxis]**np.arange(nroots+1)
    # Solve rtp.T.dot(diag(weights)).dot(rtp) = identity
    weights = 1. / (rtp.dot(cs.T)**2).sum(axis=1)
    return roots, weights

```

12.2.3 Approximating quadratures

Using the `mpmath` library to compute quadrature is very time-consuming. It is impractical to employ this method for on-the-fly quadrature computations. A more feasible approach is to use the `mpmath` library to calculate accurate roots and weights, and then employ polynomials to approximate the quadrature. In real applications, we can efficiently evaluate the quadrature by assessing the approximate polynomials.

First, let's consider the special cases when t approaches 0 or ∞ . In the region where $t \rightarrow 0$, the weight function of Rys quadrature is nearly equal to 1. We can use the Legendre quadrature to approximate the Rys quadrature:

$$\int_0^1 f(x)e^{-tx^2} dx \approx \sum w_n^L (1 - t(x_n^L)^2) f((x_n^L)^2). \quad (12.74)$$

When t is sufficiently large, we can approximate the integral as follows:

$$\int_0^1 f(x)e^{-tx^2} dx \approx \int_0^\infty f(x)e^{-tx^2} dx = 1/\sqrt{t} \int_0^\infty f\left(\frac{x'}{\sqrt{t}}\right)e^{-x'^2} dx'. \quad (12.75)$$

By performing the substitution $x' = \sqrt{t}x$, we can relate this integral to the Hermite quadrature. When $t = 50$, the difference between $\int_0^1 e^{-tx^2} dx$ and $\int_0^\infty e^{-tx^2} dx$ is only $7.49e-18$. Therefore, we have the following code to approximate Rys quadrature:

```

import scipy.special
def rys_roots_weights(nroots, t):
    if t < 1e-8:
        leg_r, leg_w = scipy.special.roots_legendre(nroots*2)
        roots = leg_r[nroots:]**2

```

```

weights = leg_w[nroots:] * (1-t*roots)
elif t > 50:
    hermit_r, hermit_w = scipy.special.roots_hermite(nroots*2)
    roots = hermit_r[nroots:]*2 / t
    weights = hermit_w[nroots:] / sqrt(t)
else:
    # polynomial approximation
    roots, weights = polynomial_approx(nroots, t)
return roots, weights

```

Both the Legendre quadrature and Hermite quadrature for the possible values of `nroots`, which is typically less than 15, are tabulated. In both cases, the approximation of Rys quadrature can be efficiently obtained by consulting the tables of Legendre and Hermite quadrature.

In the remaining regions between the interval (1e-8, 50), we develop the `polynomial_approx` function to approximate Rys quadrature. There are various fitting methods for approximating the quadrature roots and weights.

B-spline interpolation

In the remaining regions within the interval $(10^{-8}, 50)$, we develop the `polynomial_approx` function to approximate Rys quadrature. To approximate the value of a complicated function, one common choice is B-spline interpolation. The program for B-spline interpolation is straightforward using the `bsplines` module provided by the `scipy` library.

```

from scipy.interpolate import make_interp_spline
MAX_RYS_ROOTS = 12
NODES = 2000

def tabulate_bspline():
    xs = np.linspace(0, 50, NODES)
    bs = []
    for nrays in range(1, MAX_RYS_ROOTS+1):
        rws = np.empty((len(xs),2,nrays))
        for i, x in enumerate(xs):
            r, w = rys_roots_weights(nrays, x)
            rws[i,0] = r
            rws[i,1] = w
        bs.append(make_interp_spline(xs, rws.reshape(len(xs),2*nrays)))
    return bs

with open('bspline_tab.pkl', 'wb') as f:
    pickle.dump(tabulate_bspline(), f)

```

In the `polynomial_approx` function, the cached B-spline data can be loaded for evaluation.

```
with open('bspline_tab.pkl', 'rb') as f:
    bspline_tab = pickle.load(f)

def polynomial_approx(nroots, t):
    bs = bspline_tab[nroots-1]
    rws = bs(t)
    return rws.reshape(2, nroots)
```

Chebyshev approximation

Chebyshev polynomials [14] are effective for approximating smooth functions within the interval $[-1, 1]$. To determine the optimal fitting coefficients that yield the most accurate fitting functions, the Chebyshev nodes for polynomials up to degree N are selected using the formula:

$$\cos\left(\frac{\pi(k - \frac{1}{2})}{N}\right), \quad k = 1, \dots, N. \quad (12.76)$$

For the input parameter of Rys quadrature within the range $(10^{-8}, 50)$, we can apply piecewise fitting using the following formula to map an interval $[a, b]$ to the interval $[-1, 1]$:

$$\frac{x - (b + a)/2}{(b - a)/2}. \quad (12.77)$$

Subsequently, we apply the `np.polynomial.chebyshev.chebfit` function to compute the fitting coefficients.

```
MAX_RYS_ROOTS = 12
INTERVAL = 3.
DEGREE = 11

def tabulate_chebfit():
    n = DEGREE + 1
    cheb_nodes = np.cos(np.pi / n * (np.arange(n) + .5))

    intervals = [(a, a+INTERVAL) for a in np.arange(0., 50., INTERVAL)]
    n_intervals = len(intervals)
    cs = []
    for nrays in range(1, MAX_RYS_ROOTS+1):
        cs_i = []
        for a, b in intervals:
            # Map chebyshev nodes between (-1, 1) to the sample points
```

```

# between interval (a, b)
xs = cheb_nodes*(b-a)/2 + (b+a)/2
rws = np.empty((n,2,nrys))
for i, x in enumerate(xs):
    r, w = rys_roots.rys_roots_weights(nrys, x)
    rws[i,0] = r
    rws[i,1] = w
    cs_i.append(
        np.polynomial.chebyshev.chebfit(
            cheb_nodes, rws.reshape(n,2*nrys), DEGREE))
cs.append(np.array(cs_i))
return cs

with open('chebfit_tab.pkl', 'wb') as f:
    pickle.dump(tabulate_chebfit(), f)

```

In the `polynomial_approx` function, it is necessary to transform the input variable according to the interval transformation given in Eq. (12.77). Then, we call `np.polynomial.chebyshev.chebval` to evaluate the quadrature roots and weights.

```

with open('chebfit_tab.pkl', 'rb') as f:
    chebfit_tab = pickle.load(f)

def polynomial_approx(nroots, t):
    interval_id = int(t // INTERVAL)
    c = chebfit_tab[nroots-1][interval_id]
    a = interval_id * INTERVAL
    b = a + INTERVAL
    x = (t - (a+b)/2) / ((b-a)/2)
    rws = np.polynomial.chebyshev.chebval(x, c)
    return rws.reshape(2,nroots)

```

Other orthogonal polynomials can also be used to approximate smooth functions. For example, to utilize Legendre polynomials, we can make the following replacements in the code provided above:

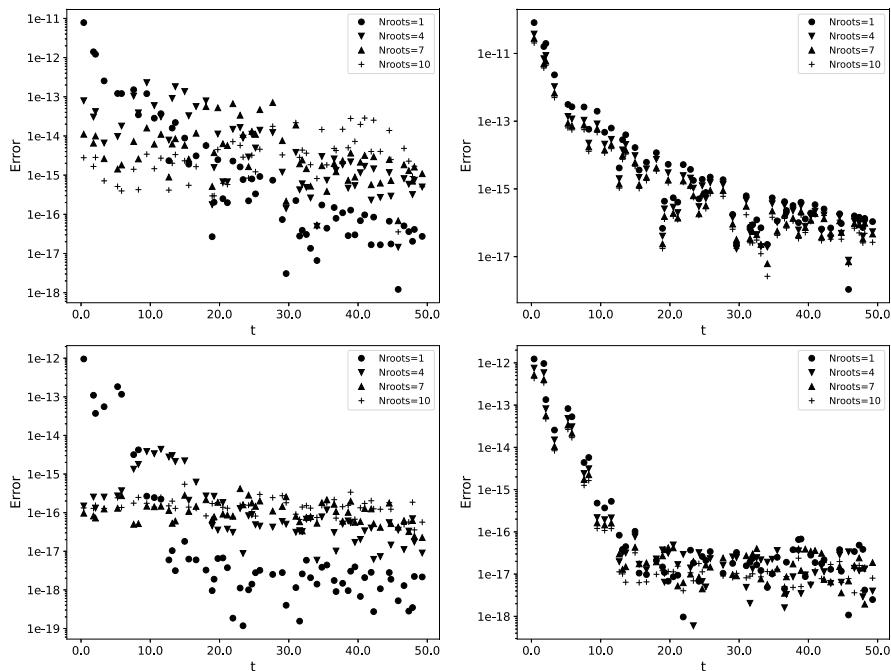
```

np.polynomial.chebyshev.chebfit -> np.polynomial.legendre.legfit
np.polynomial.chebyshev.chebval -> np.polynomial.legendre.legval

```

Nevertheless, Chebyshev polynomials are preferred in most scenarios because they can be efficiently computed using the Clenshaw recursion algorithm. The implementation of Clenshaw recursion can be found in the source code of the `np.polynomial.chebyshev.chebval` function. We can use the Numba compiler to optimize the Clenshaw recursion program.

```
@numba.njit
def chebval(x, c):
    x2 = 2*x
    c0 = c[-2]
    c1 = c[-1]
    for i in range(3, len(c) + 1):
        tmp = c0
        c0 = c[-i] - c1
        c1 = tmp + c1*x2
    return c0 + c1*x
```

**FIGURE 12.4**

Errors in Rys roots and weights using different approximations. The first row shows errors in roots and weights from B-spline fitting. The second row shows errors from Chebyshev polynomial fitting.

The largest errors between the polynomial approximation and the analytical evaluation are displayed in Fig. 12.4. Generally, both the B-spline and Chebyshev polynomial methods provide good approximations to the Rys roots and weights. Please note that the fitting parameters, `NODES = 2000` in the B-spline method and

Table 12.3 Fourier transform of two-electron interactions.

Operator	Fourier transform
$\frac{1}{r_{12}}$	$\frac{1}{2\pi^2 k^2}$
$\frac{\text{erf}(\omega r_{12})}{r_{12}}$	$\frac{1}{2\pi^2 k^2} e^{-\frac{k^2}{4\omega^2}}$
$e^{-\gamma r_{12}}$	$\frac{\gamma}{\pi^2(k^2 + \gamma^2)}$
$\frac{e^{-\gamma r_{12}}}{r_{12}}$	$\frac{1}{2\pi^2(k^2 + \gamma^2)}$
$e^{-\eta r_{12}^2}$	$\frac{1}{8(\pi\eta)^{3/2}} e^{-\frac{k^2}{4\eta}}$
$\frac{e^{-\eta r_{12}^2}}{r_{12}}$	$\frac{1}{2\pi^2 k \sqrt{\eta}} \text{dawson}(\frac{k}{2\sqrt{\eta}})$

INTERVAL = 3; DEGREE = 11 in the Chebyshev method, are not particularly optimized.

The Chebyshev approximation yields smaller errors. When $t > 10$, the errors of approximate roots and weights are smaller than 10^{-15} . In this region, the Chebyshev polynomials of degree 11 are already over-complete. A lower order may be sufficient to achieve the required accuracy. In the region where $t < 10$, the errors are relatively larger, indicating that higher polynomial orders or smaller intervals might be necessary to improve accuracy. The optimal fitting parameters can be derived through additional numerical experiments.

12.2.4 Numerical integration with Fourier transform

The integral of two-electron Coulomb-type interactions can be computed within the reciprocal space, which can be expressed as an integral over single-particle operator:

$$\int f(\mathbf{r}_1) \frac{1}{r_{12}} g(\mathbf{r}_2) d^3 \mathbf{r}_1 d^3 \mathbf{r}_2 = (\frac{1}{2\pi})^3 \int \frac{4\pi}{k^2} F(\mathbf{k}) G(-\mathbf{k}) d^3 \mathbf{k} \quad (12.78)$$

where $F(\mathbf{k})$ and $G(\mathbf{k})$ represent the Fourier transforms of the functions $f(\mathbf{r})$ and $g(\mathbf{r})$, respectively. $\frac{4\pi}{k^2}$ is the Fourier transform of the Coulomb operator. Some Coulomb-type two-electron operators and their Fourier transforms are listed in Table 12.3. Eq. (12.78) enables the numerical computation of the Coulomb-type integrals in reciprocal space. It can be used as a reference to verify the correctness of the analytical integral program.

For the three-dimensional integrals in reciprocal space, we can use spherical coordinates with an appropriate quadrature scheme to generate the integral grids:

$$\begin{aligned} \int f(\mathbf{r}) d^3 \mathbf{r} &= \int_0^\infty r^2 dr \int_0^\pi \sin \varphi d\varphi \int_0^{2\pi} d\theta f(r, \theta, \varphi) \\ &= \int_0^\infty r^2 dr \int_{-1}^1 dt \int_0^{2\pi} d\theta f(r, \theta, \arccos(t)). \end{aligned} \quad (12.79)$$

Legendre quadrature is a suitable choice for generating radial grids. A logarithmic transformation can be applied to map its interval $[-1, 1]$ to the domain $(0, \infty)$. The integrals for angular components are two-dimensional. One possible choice for the angular grids is the Lebedev grids on the spherical surface [15]. Here, we demonstrate another option, using the Legendre quadrature to generate 2D grid points on the spherical surface. The grids for t in Eq. (12.79), which lie in the interval $[-1, 1]$, can be directly obtained using the Legendre quadrature. To generate the grids for θ , a linear mapping

$$\theta = \pi x + \pi$$

can be applied to map the Legendre quadrature to the interval $(0, 2\pi)$.

The final grid points, denoted by kv , and the corresponding integration weights are generated using the program below:

```
def pw_3d_grids(n_r, n_theta, n_phi):
    x, w_x = scipy.special.roots_legendre(n_r)
    t, w_t = scipy.special.roots_legendre(n_phi)
    theta, w_theta = scipy.special.roots_legendre(n_theta)
    r = np.log(2/(1-x)) / np.log(2)
    w_r = w_x / np.log(2) / (1-x)
    theta = theta * np.pi + np.pi
    w_theta = w_theta * np.pi
    phi = np.arccos(t)

    kv = np.zeros((w_r.size, w_t.size, w_theta.size, 3))
    kv[:, :, :, 0] = einsum('i,j,k->ijk', r, np.sin(phi), np.cos(theta)) # x
    kv[:, :, :, 1] = einsum('i,j,k->ijk', r, np.sin(phi), np.sin(theta)) # y
    kv[:, :, :, 2] = einsum('i,j->ij', r, np.cos(phi))[:, :, np.newaxis] # z

    weights = einsum('i,j,k->ijk', r**2 * w_r, w_t, w_theta)
    return kv.reshape(-1, 3), weights.ravel()
```

In reciprocal space, each integral grid \mathbf{k} corresponds to a plane-wave function. We can calculate the Fourier transform of the orbital products for each grid:

$$\int e^{-i\mathbf{k}\cdot\mathbf{r}} \chi_i(\mathbf{r}) \chi_j(\mathbf{r}) d^3\mathbf{r} = S_{i_x j_x}(k_x) S_{i_y j_y}(k_y) S_{i_z j_z}(k_z). \quad (12.80)$$

The RR formulas for each individual Cartesian component are similar to those for overlap integrals we presented in Section 12.1.3.1:

$$S_{i_x, j_x}(k_x) = X_{AB} S_{i_x, j_x-1}(k_x) + S_{i_x+1, j_x-1}(k_x), \quad (12.81)$$

$$S_{i_x, 0}(k_x) = (X_{PA} - \frac{ik_x}{2\alpha_{ij}}) S_{i_x-1, j_x}(k_x) + \frac{i_x - 1}{2\alpha_{ij}} S_{i_x-2, 0}(k_x), \quad (12.82)$$

$$S_{00}(k_x) = \sqrt{\frac{\pi}{\alpha_{ij}}} e^{-\theta_{ij} X_{AB}^2 - \frac{k_x^2}{4\alpha_{ij}} - ik_x X_P}. \quad (12.83)$$

The Fourier transform program can be developed as follows:

```
from basis import n_cart, iter_cart_xyz

def ft_primitive_overlap(li, lj, ai, aj, Ra, Rb, kv):
    aij = ai + aj
    Rab = Ra - Rb
    Rp = (ai * Ra + aj * Rb) / aij
    Rpa = Rp - Ra - lj/(2*aij) * kv
    theta_ij = ai * aj / aij

    @lru_cache(1000)
    def get_S(i: int, j: int):
        if i < 0 or j < 0:
            return 0
        if i == j == 0:
            e = -theta_ij*Rab**2 - kv**2/(4*aij) - lj*kv*Rp
            return (np.pi/aij)**.5 * np.exp(e)
        if j == 0:
            return Rpa * get_S(i-1, j) + (i-1)/(2*aij) * get_S(i-2, j)
        return get_S(i+1, j-1) + Rab * get_S(i, j-1)

    nfi = n_cart(li)
    nfj = n_cart(lj)
    nk = kv.shape[0]
    ft = np.zeros((nfi, nfj, nk))
    for i, (ix, iy, iz) in enumerate(iter_cart_xyz(li)):
        for j, (jx, jy, jz) in enumerate(iter_cart_xyz(lj)):
            ft[i,j] = (get_S(ix,jx)[ :,0] * get_S(iy,jy)[ :,1] *
                       get_S(iz,jz)[ :,2])
    return ft
```

Now, we can develop a numerical integration program to approximate the ERIs.

```
def primitive_ERI_with_ft(li, lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    kv, weights = pw_3d_grids(30, 20, 20)
    Fij = ft_primitive_overlap(li, lj, ai, aj, Ra, Rb, kv)
    Fkl = ft_primitive_overlap(lk, ll, ak, al, Rc, Rd, -kv)
    coul_ft = .5/np.pi**2 / einsum('gx,gx->g', kv, kv)
    return einsum('g,g,ijg,klg->ijkl', weights, coul_ft, Fij, Fkl)
```

12.3 Integral transformation

Integral transformation in quantum chemistry typically refers to the process that transforms the matrix representation of an operator from atomic orbitals (AOs), which are described by GTOs, to molecular orbitals (MOs).

It is generally straightforward to implement the transformation correctly using the `np.einsum` function. For instance, given the four-index ERI tensor generated by the analytical integral program in Section 12.1, the integral transformation can be performed using the following `einsum` code:

```
mo_eri = np.einsum('pqrs,pi,qj,rk,s1->ijkl', eri, mo, mo, mo, mo,
                    optimize=True)
```

The `optimize=True` parameter is necessary as it breaks the five-tensor contraction into four steps, each transforming one index. This results in a computational complexity scaling of N^5 and a memory consumption of N^4 for the intermediate states. The transformation is performed in memory, typically referred to as *incore* transformation.

When more AO functions are involved in the system, the memory might become insufficient to accommodate the entire ERI tensor, to store the transformed integrals, or to hold any intermediates generated by the `einsum` function. In such cases, we need to use the so-called *outcore* method to perform the integral transformation, which leverages the disk storage to handle any intermediate tensors.

The bottleneck in outcore methods is the disk I/O performance. The I/O optimization techniques discussed in Chapter 9 can be utilized to enhance I/O efficiency. Additionally, we can implement the following optimization strategies when designing the outcore integral transform algorithm:

- Reduce the amount of data transferred by I/O operations.
- Use memory buffers to store a bulk of data before invoking I/O, so as to decrease the number of I/O operations.
- Ensure sequential reading and writing to reduce the overhead of each I/O operation.
- Execute I/O operations asynchronously, overlapping computation and I/O operations.

Reducing the amount of I/O operations

The four-index integral transformation requires four steps of tensor contractions, involving three intermediate tensors. If all three intermediate tensors are stored on disk, it would require reading and writing the N^4 -sized tensors three times. Some of these contractions can be combined to reduce the necessary storage. For example, we can apply a two-step transformation algorithm, which only requires storing one intermediate tensor. In the first step, we transform two indices, such as `ij` in `ijkl`, and save the resulting intermediate tensor to disk. In the second step, we read the intermediate tensor, transform the remaining two indices, and then obtain the fully transformed integrals.

To reduce I/O volume, we can also utilize the symmetry within the ERI tensor. In the two-step transformation algorithm, both the ERI tensor in the AO basis and the partially transformed intermedates exhibit the four-fold permutation symmetry

$$(ij|kl) = (ji|kl) = (ij|lk). \quad (12.84)$$

By incorporating this symmetry, we not only reduce the computational efforts but also decrease the I/O volume to $\frac{1}{4}N^4$.

Storage format for sequential reading and writing

In the two-step integral transformation algorithm, due to the limitations of available memory, the integral computation and tensor contraction often need to be processed in several batches. When performing the integral transform for the first electron, the memory is only sufficient to store a portion of the AO indices for the second electron. The entire list of AOs on the second electron needs to be divided into multiple batches for processing. Similarly, when performing the integral transform for the second electron, we may need to read in the indices for the first electron in batches.

This process implicitly introduces a tensor transpose on the disk. If the first step stores the indices for the first electron (`ij` in `ijk1`) continuously on disk, the indices `k1` cannot be made continuously in the storage. When reading a portion of the `ij` indices along with the entire set of `k1` indices in the second step, continuous data access is not achievable. If the storage is designed to keep the `k1` indices continuous for the second step, it interrupts the continuous storage of the `ij` indices in the first step. Therefore, we cannot effectively utilize the `np.memmap` to store the ERI tensor or intermediate tensors. Discontinuous reading or writing severely impacts the performance of `np.memmap`.

The HDF5 format with chunks is a more suitable option for this scenario. Although chunking does not favor any particular data access pattern, it mitigates the issues of data discontinuity when accessing a small portion of rows or columns.

When configuring the chunk size, it is preferable to align the chunks with the block size of the file system. The operating system reads and writes data in blocks. Theoretically, aligned chunk sizes would provide us the minimal overhead in I/O operations. Most file systems use a block size of 4096 bytes (4 KB), which is equivalent to 512 elements in double precision. The optimal size of a chunk can be selected to a multiple of 512.

Memory buffer

In the first step of the integral transform, we need a memory buffer to temporarily hold integrals. Once enough data has been accumulated, an HDF5 data storing API is called. This approach helps avoid repeated calls to I/O APIs, thereby reducing the overhead of I/O operations.

Is the memory buffer the larger the better? The answer is no, even though it is generally preferable to have a relatively large memory buffer. Considering the `einsum` contraction process, which might generate additional temporary data, very large intermediate tensors can lead to overhead in memory management, and may even exceed

the available memory, causing an out-of-memory error. Additionally, a large memory buffer can increase page faults, which significantly impacts efficiency.

Asynchronous I/O operations

By utilizing Python multithreading, I/O can be placed in a separate thread to overlap computation and I/O operations. Data writing in the first step can be executed asynchronously. However, asynchronous writing might not significantly enhance performance because the operating system can automatically use main memory to cache the writing operation, implicitly applying asynchronous data writing. A more effective method for asynchronous I/O operations is to prefetch data during the read operations in the second step. The function `iterate_with_prefetch` developed in Chapter 9 can be used to achieve data prefetching.

After considering the key points mentioned above, we can implement the integral transform program. The following code demonstrates the first step that transforms the AOs of the first electron:

```
import tempfile, itertools, h5py, numpy as np
from contextlib import contextmanager
from basis import gto_offsets
from eri_OS_rys import contracted_ERI, pack_gto_attrs

IO_BUF_SIZE = 1e7 # 800 MB

def transform_outcore(gtos, mo):
    '''Outcore integral transformation for two-electron integrals:
    einsum('pqrs,pi,qj,rk,sl->ijkl', eri, mo, mo, mo, mo)
    ...
    n_shells = len(gtos)
    offsets = gto_offsets(gtos)
    gto_params = pack_gto_attrs(gtos)
    nao, nmo = mo.shape
    ao_idx1, ao_idx2 = np.tril_indices(nao)
    mo_idx1, mo_idx2 = np.tril_indices(nmo)
    nao_pair = len(ao_idx1)
    nmo_pair = len(mo_idx1)

    @contextmanager
    def temp_h5dat():
        with tempfile.TemporaryDirectory(dir='.') as tmpdir:
            with h5py.File(f'{tmpdir}/eri.h5', 'w') as f:
                yield f.create_dataset('eri', (nmo_pair, nao_pair),
                                      dtype='f8', chunks=(320,320)) # (1)

    with temp_h5dat() as dataset:
```

```

p1 = 0
def save_data(buf):
    nonlocal p1
    dat = np.hstack(buf)
    p0, p1 = p1, p1 + dat.shape[1]
    dataset[:,p0:p1] = dat

with background(save_data) as write:                                # (2)
    buf = []
    for shell_k in range(n_shells):
        k0, lk, Rc, exps_k, norm_ck = gto_params[shell_k]
        k1 = offsets[shell_k+1]
        eri = np.empty((nao, nao, k1-k0, k1))                      # (3)
        for ((i0, li, Ra, exps_i, norm_ci),
              (j0, lj, Rb, exps_j, norm_cj)) \
            in itertools.product(*gto_params,*2):
            if i0 < j0: # symmetry between ij in (ij|kl)           # (4)
                continue
            for l0, ll, Rd, exps_l, norm_cl in gto_params[:shell_k+1]: # (5)
                contracted_ERI(li, lj, lk, ll, exps_i, exps_j, exps_k, exps_l,
                                   norm_ci, norm_cj, norm_ck, norm_cl,
                                   Ra, Rb, Rc, Rd, eri, i0, j0, 0, 10)
            eri[ao_idx2, ao_idx1] = eri[ao_idx1, ao_idx2]               # (6)
            for k, l0 in enumerate(range(k0, k1)):
                dat = eri[:, :, k, :l0+1]                               # (7)
                dat = np.einsum('pq$,pi,jq->ijs', dat, mo, mo, optimize=True)
                dat = dat[mo_idx1, mo_idx2] # symmetry between ij in (ij|kl)
                buf.append(dat)
            if sum(x.size for x in buf) > IO_BUF_SIZE:
                write(buf)
                buf = []
    if buf: # sync the remaining data in buf
        write(buf)

```

In line (1), we choose a chunk size of (320, 320), which corresponds to 200 blocks with each block being 4 KB in size. The four-fold permutation symmetry ($i \geq j$ and $k \geq l$ for $(ij|kl)$) is enabled when generating the ERIs in AO basis. In lines (4) and (6), the permutation symmetry for the AOs of the first electron is utilized. In lines (3), (5), and (7), the permutation symmetry for the AOs of the second electron is considered. Therefore, we only allocate a temporary array of necessary size in line (3) and only access the necessary elements in line (7). When the memory buffer reaches approximately 800 MB, the program triggers the operation of data writing and reset the memory buffer. A thread dedicated to the data writing operation is initialized in line (2). The asynchronous function is implemented as follows:

```

@contextmanager
def background(fn):
    with ThreadPoolExecutor(max_workers=1) as ex:
        future = None
        def bg_worker(*args, **kwargs):
            nonlocal future
            # Block the IO operation to avoid using too many memory
            if future is not None:
                future.result()
            future = ex.submit(fn, *args, **kwargs)

        try:
            yield bg_worker
        finally:
            if future is not None:
                future.result()

```

Then we can implement the integral transform for the second electron.

```

from pychem_book.chap09.io_utils import iterate_with_prefetch

def transform_outcore(gtos, mo):
    ...

    with temp_h5dat() as dataset:
        ...

        block_size = 320                                         # (1)
        tasks = range(0, nmo_pair, block_size)
        def loader(i0):
            i1 = min(i0 + block_size, nmo_pair)
            return dataset[i0:i1]

        out = np.empty((nmo_pair, nmo_pair))                      # (2)
        for i0, buf in iterate_with_prefetch(tasks, loader):
            di = buf.shape[0]
            dat = np.empty((di, nao, nao))
            dat[:, ao_idx1, ao_idx2] = buf # fill lower triangular part
            dat[:, ao_idx2, ao_idx1] = buf # fill upper triangular part
            dat = np.einsum('xrs,rk,sl->xkl', dat, mo, mo, optimize=True)
            out[i0:i0+di] = dat[:, mo_idx1, mo_idx2]

    return out

```

It is preferable to align the `block_size` with the chunks configuration of the HDF5 file, as shown in line (1). In line (2), we allocated a NumPy array as the output container to store the transformed integrals for simplicity. Alternatively, the output could be an `np.memmap`, or a regular HDF5 dataset.

Summary

In this chapter, we explored analytical integral computation algorithms for Gaussian type orbitals, including the McMurchie-Davidson, Obara-Saika, and Rys quadrature methods. Additionally, we presented the method for evaluating integral quadratures, the use of polynomial approximation to reduce the computational overhead of Rys quadrature algorithm, the approximation algorithm for two-electron integrals using Fourier transform, and the integral transformation program for computing integrals in the molecular orbital basis.

We demonstrated how integral algorithms are progressively optimized, starting from the mathematical formulas. When implementing integration programs using Python, it is not necessary to prioritize program efficiency at beginning. We illustrated how to quickly develop a correct version using dynamic programming techniques. Subsequently, we utilized the LRU cache to analyze the recursive program and rewrote it into iterative code. We then introduced Python compilation techniques to optimize the iteration code. Finally, we employed meta-programming techniques to unroll the program, minimizing various overheads. Through this series of program optimization steps, the Python integral program achieved performance comparable to native C/C++ implementations.

The integral program is a classical example of Python compilation. In this example, we demonstrated the techniques of Numba JIT compilation and Cython compilation in handling complex numerical computation problems. The main trick in Python compilation is to remove pythonic elements as much as possible. In addition, it is also advisable to avoid object-oriented code for Numba JIT compilation. Numba and Cython have different advantages. Numba JIT compilation can automatically optimize NumPy operations and functions, effectively reducing the overhead of Python function calls. Cython compilation, on the other hand, is more efficient when handling high-dimensional arrays. To best utilize the strengths of each tool, these compilation techniques, along with native C/C++ programs, can be integrated using `ctypes` to achieve optimal efficiency.

References

- [1] B.P. Pritchard, D. Altarawy, B. Didier, T.D. Gibson, T.L. Windus, New basis set exchange: an open, up-to-date resource for the molecular sciences community, *Journal of Chemical Information and Modeling* 59 (11) (2019) 4814–4820, <https://doi.org/10.1021/acs.jcim.9b00725>, pMID: 31600445.

- [2] H.B. Schlegel, M.J. Frisch, Transformation between Cartesian and pure spherical harmonic Gaussians, International Journal of Quantum Chemistry 54 (2) (1995) 83–87, <https://doi.org/10.1002/qua.560540202>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.560540202>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.560540202>.
- [3] R. Flores-Moreno, R.J. Alvarez-Mendez, A. Vela, A.M. Köster, Half-numerical evaluation of pseudopotential integrals, Journal of Computational Chemistry 27 (9) (2006) 1009–1019, <https://doi.org/10.1002/jcc.20410>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.20410>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20410>.
- [4] T. Helgaker, P. Jørgensen, J. Olsen, Molecular Integral Evaluation, John Wiley & Sons, Ltd, 2000, Ch. 9, pp. 336–432, <https://doi.org/10.1002/9781119019572.ch9>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119019572.ch9>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119019572.ch9>.
- [5] N. Flocke, V. Lotrich, Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations, Journal of Computational Chemistry 29 (16) (2008) 2722–2736, <https://doi.org/10.1002/jcc.21018>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.21018>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21018>.
- [6] Numba Developers, Numba reference manual - supported NumPy features, <https://numba.readthedocs.io/en/stable/reference/numpysupported.html>, 2024.
- [7] S. Behnel, R. Bradshaw, D.S. Seljebotn, G. Ewing, W. Stein, G. Gellner, et al., Cython for NumPy users - efficient indexing with memoryviews, https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html#efficient-indexing-with-memoryviews, 2024.
- [8] M. Head-Gordon, J.A. Pople, A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations, Journal of Chemical Physics 89 (9) (1988) 5777–5786, <https://doi.org/10.1063/1.455553>, https://pubs.aip.org/aip/jcp/article-pdf/89/9/5777/18972946/5777_1_online.pdf.
- [9] P.M. Gill, Molecular integrals over Gaussian basis functions, in: J.R. Sabin, M.C. Zerner (Eds.), Advances in Quantum Chemistry, in: Advances in Quantum Chemistry, vol. 25, Academic Press, 1994, pp. 141–205, [https://doi.org/10.1016/S0065-3276\(08\)60019-2](https://doi.org/10.1016/S0065-3276(08)60019-2), <https://www.sciencedirect.com/science/article/pii/S0065327608600192>.
- [10] GNU Compiler Collection Team, GCC documentation - restricting pointer aliasing, <https://gcc.gnu.org/onlinedocs/gcc/Restricted-Pointers.html>, 2024.
- [11] GNU Compiler Collection Team, GCC documentation - loop-specific pragmas, <https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html>, 2024.
- [12] Intel Corporation, Intel intrinsics guide, <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>, 2024.
- [13] G. Bill, Memory allocation strategies, <https://www.gingerbill.org/series/memory-allocation-strategies/>, 2024.
- [14] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd edition, Cambridge University Press, USA, 2007.
- [15] V.I. Lebedev, D.N. Laikov, A quadrature formula for the sphere of the 131st algebraic order of accuracy, Doklady. Mathematics 59 (1999) 477–481, <https://api.semanticscholar.org/CorpusID:118893131>.

Mean-field methods

13

Mean-field calculations are foundational in quantum chemistry applications. Nearly all quantum chemistry methods begin with a mean-field calculation as the initial step. This chapter is dedicated to the implementation of mean-field methods, specifically Hartree-Fock (HF) and Kohn-Sham Density Functional Theory (KS-DFT).

The computation of integrals is the most challenging and time-consuming task in mean-field calculations. The mean-field programs can be developed using the integral programs described in Chapter 12. In addition to integral computation, implementing a mean-field program may also involve the following questions:

- How to effectively utilize the permutation symmetry in the electron repulsion integrals when constructing the Coulomb and exchange matrices?
- How to numerically compute the integrals for the exchange-correlation potential in KS-DFT methods?
- How to enable cross-function optimization between the integral evaluation code and the Coulomb matrix construction code?
- How to design and organize the program to reuse functions for different mean-field methods, given that many mean-field methods share a very similar structure?
- How to manipulate the initial guess to accelerate the convergence of the mean-field calculations?

In this chapter, we will not delve into the detailed theory of mean-field methods. The theoretical foundations of HF and KS-DFT can be found in standard quantum chemistry textbooks. Two highly recommended references for understanding these theories are:

- *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory* by Attila Szabo and Neil S. Ostlund [1];
- *Density Functional Theory of Atoms and Molecules* by Robert G. Parr and Weitao Yang [2].

This chapter builds upon several techniques developed in previous chapters. To manage the numerous mean-field variants, we will refer to the Python modular design principles developed in Chapter 1 (Python Programming Environment) and the meta-programming techniques discussed in Chapter 5 (Meta-programming and Non-numerical Computation). To save intermediate data for mean-field calculations and

facilitate the restarting of these calculations, we will utilize the techniques developed in Chapter 6 (Input and Output).

13.1 The self-consistency iteration program

In mean-field methods, the wave function is represented by a single Slater determinant. By employing a set of one-particle orbitals, ϕ_i , we can construct the Slater determinant and derive the total energy expressions for mean-field methods:

$$E_{\text{HF}} = \sum_{i \in \text{ooc}} \langle \phi_i | h_{\text{core}} | \phi_i \rangle + E_{\text{coul}} + E_x + E_{\text{nn}}, \quad (13.1)$$

$$E_{\text{KS}} = \sum_{i \in \text{ooc}} \langle \phi_i | h_{\text{core}} | \phi_i \rangle + E_{\text{coul}} + E_{\text{xc}} + E_{\text{nn}}. \quad (13.2)$$

Here, h_{core} represents the one-electron core Hamiltonian:

$$h_{\text{core}} = \sum_i -\frac{1}{2} \nabla_i^2 \sum_{iA} \frac{-Z_A}{|\mathbf{r}_i - \mathbf{R}_A|}. \quad (13.3)$$

E_{coul} , E_x , and E_{xc} correspond to the *Coulomb energy*, *exact exchange energy*, and *exchange-correlation (XC) energy* associated with the two-electron interactions, respectively. E_{nn} represents the nuclear repulsion energy.

Applying the variational principle to the total energy leads to the eigenvalue equation for the orbitals

$$\mathbf{F}\mathbf{C} = \mathbf{S}\mathbf{E}, \quad (13.4)$$

where \mathbf{F} is the Fock matrix, \mathbf{S} is the overlap matrix, \mathbf{C} represents the orbital coefficients, and \mathbf{E} denotes the orbital energies. The Fock matrix consists of two terms:

$$\mathbf{F} = \mathbf{h}_{\text{core}} + \mathbf{V}_{\text{eff}}. \quad (13.5)$$

The effective potential matrix \mathbf{V}_{eff} arises from the interactions between electrons. This matrix needs to be continually updated during the self-consistent field iteration process.

Generally, \mathbf{V}_{eff} can be expressed as a combination of several components:

$$\mathbf{V}_{\text{eff}} = \mathbf{J} - a\mathbf{K} + b\mathbf{V}_{\text{xc}}. \quad (13.6)$$

The \mathbf{J} matrix originates from the Coulomb energy E_{coul} , while \mathbf{K} and \mathbf{V}_{xc} are derived from the exact exchange energy E_x and the exchange-correlation energy E_{xc} . \mathbf{V}_{eff} is a key quantity that distinguishes different mean-field models such as Hartree-Fock (HF) and Kohn-Sham (KS) for closed-shell methods, open-shell methods, and generalized spin-orbital methods. For instance, in the closed-shell HF method, \mathbf{V}_{eff}

corresponds to $a = \frac{1}{2}$ and $b = 0$.

$$\mathbf{V}_{\text{eff}} = \mathbf{J} - \frac{1}{2}\mathbf{K} \quad (13.7)$$

with

$$J_{\mu\nu} = \sum_{\kappa\lambda} (\mu\nu|\kappa\lambda) \gamma_{\lambda\kappa}, \quad (13.8)$$

$$K_{\mu\nu} = \sum_{\kappa\lambda} (\mu\lambda|\kappa\nu) \gamma_{\lambda\kappa}, \quad (13.9)$$

where the density matrix γ is given by

$$\gamma_{\mu\nu} = \sum_{i \in \text{occ}} 2C_{\mu i} C_{\nu i}^*. \quad (13.10)$$

The computation of \mathbf{V}_{eff} involves the analytical evaluation of electron repulsion integrals and the numerical integration of DFT exchange-correlation (XC) functionals. They are the main sources of computational cost in mean-field methods. We will develop computational programs for these processes in the next two sections.

The mean-field wavefunction is essentially a set of orthonormal orbitals. To calculate these mean-field orbitals, we need to construct the Fock matrix. However, the construction of Fock matrix relies on the orbitals themselves, resulting in a recursive relationship. To address this, we can employ the fixed-point iteration method to solve the mean-field problem. This iteration method provides the most straightforward algorithm for mean-field calculations, known as the self-consistent field (SCF) iteration. The SCF iteration begins with an initial guess for the orbitals (or the density matrix), which is used to construct the Fock matrix. We then solve the eigenvalue problem for the Fock matrix to obtain a new set of orbitals. This process is repeated until the solution converges to a stable point.

Using the `einsum` function and diagonalization functions provided by NumPy and SciPy, along with the integral program developed in Chapter 12, it is not difficult to implement a simple SCF iteration for the restricted HF method. However, if we aim to develop a more general SCF program to accommodate various mean-field methods, the SCF program would need to be designed with sufficient flexibility. The SCF process should be broken down into several intermediate steps, with each step corresponding to a separate function call. These functions should be exposed as interfaces, which allows different mean-field models to override, making it easy to reuse the SCF program. The interfaces can be organized into an SCF class, which includes the following functionalities:

- A method to construct the \mathbf{h}_{core} and the overlap matrix, which remain unchanged throughout the SCF iterations.
- A method to construct an initial guess, which could be the orbitals or the density matrix.

- The `get_veff` method to compute the effective potential matrix for the given mean-field wavefunction.
- The `get_fock` method to assemble the Fock matrix from the \mathbf{h}_{core} matrix and the \mathbf{V}_{eff} matrix.
- An eigenvalue solver to generate molecular orbitals and orbital energies.
- A method to construct the new wavefunction in terms of orbitals and orbital energies.
- The `check_convergence` method, where additional quantities such as the density matrices, orbital gradients, and the mean-field energy, are computed to assess the convergence of the SCF iteration.

Below is a minimal implementation of the SCF program that incorporates the considerations mentioned above.

```
class SCFWavefunction:
    gtos = None
    orbitals = None
    energies = None
    occupancies = None
    density_matrices = None

    def scf_iter(model, wfn: SCFWavefunction = None) -> SCFWavefunction:
        hcore, s = model.get_hcore_s()
        if wfn is not None:
            wfn = model.get_initial_guess()
        converged = False
        while not converged:
            f = model.get_fock(wfn, hcore, s)
            mo_orbitals, mo_energies = model.eigen(f, s, wfn)
            wfn, wfn_old = model.make_wfn(mo_orbitals, mo_energies), wfn
            if model.check_convergence(wfn, wfn_old):
                converged = True
        return wfn
```

A restricted closed-shell HF class can be implemented according to the interfaces defined in this SCF program:

```
class RHF:
    def __init__(self, mol, gtos):
        self.mol = mol
        self.gtos = gtos
        self.threshold = 1e-6

    @lru_cache
    def get_hcore_s(self):
        mol = self.mol
```

```
s = overlap_MD.get_matrix(self.gtos, overlap_MD.primitive_overlap)
t = overlap_MD.get_matrix(self.gtos, overlap_MD.primitive_kinetic)
v = sum(-z * coulomb_le_MD.get_matrix(self.gtos, Rc)
        for z, Rc in zip(mol.nuclear_charges, mol.coordinates))
return t+v, s

def get_initial_guess(self):
    h, s = self.get_hcore_s()
    return self.make_wfn(*self.eigen(h, s))

@property
@lru_cache
def eri_tensor(self):
    return eri_OS_unrolled(self.gtos)

@lru_cache(2)                                     # (1)
def get_jk(self, wfn):
    dm = wfn.density_matrices
    j = einsum('ijkl,lk->ij', self.eri_tensor, dm)
    k = einsum('ijkl,jk->il', self.eri_tensor, dm)
    return j, k

def get_veff(self, wfn):
    j, k = self.get_jk(wfn)
    return j - k * .5

def get_fock(self, wfn, hcore, s):
    veff = self.get_veff(wfn)
    return hcore + veff

def eigen(self, fock, overlap):
    e, c = scipy.linalg.eigh(fock, overlap)
    return c, e

def make_wfn(self, orbitals, energies):
    return RestrictedCloseShell(self, orbitals, energies)

def check_convergence(self, wfn, wfn_old):
    t1 = t2 = self.threshold * 1e3
    t3 = self.threshold
    dm = wfn.density_matrices
    dm_old = wfn_old.density_matrices
    return (np.linalg.norm(dm - dm_old) < t1
            and np.linalg.norm(self.orbital_gradients(wfn)) < t2)
```

```

        and abs(self.total_energy(wfn)
                  - self.total_energy(wfn_old)) < t3)

    def total_energy(self, wfn):
        hcore, s = self.get_hcore_s()
        j, k = self.get_jk(wfn)
        dm = wfn.density_matrices
        e = einsum('ij,ji', hcore, dm)
        e += einsum('ij,ji', j, dm) * .5 - einsum('ij,ji', k, dm) * .25
        e += self.mol.nuclear_repulsion_energy()
        return e

    def orbital_gradients(self, wfn):
        hcore, s = self.get_hcore_s()
        fock = self.get_fock(wfn, hcore, s)
        fock_mo = wfn.orbitals.T.dot(fock).dot(wfn.orbitals)
        occ = wfn.occupancies
        return fock_mo[occ!=0, occ[:,None]==0].ravel() * 2.

    class RestrictedCloseShell(SCFWavefunction):
        def __init__(self, mf, orbitals, energies):
            self.gtos = mf.gtos
            self.orbitals = orbitals
            self.energies = energies
            nocc = mf.mol.nelectron // 2
            self.occupancies = np.zeros_like(energies)
            self.occupancies[:nocc] = 2.

        @property
        @lru_cache
        def density_matrices(self):
            c = self.orbitals
            return (c * self.occupancies).dot(c.T)

```

Integral functions, including `primitive_overlap`, `primitive_kinetic`, `coulomb_1e_MD`, and `eri_OS_unrolled`, were developed in Chapter 12. These integrals are cached to avoid recompilation during the SCF iterations. Some functions, such as `get_jk` and `get_veff`, are called by several methods of the class, such as `orbital_gradients`, `total_energy`, and `convergence_check`. In the SCF iterations, it is very likely that these functions are evaluated multiple times for the same wavefunction objects. To optimize performance, it is useful to cache the results of the most recent call to these functions, as shown in line (1). To ensure the correctness of caching, the `SCFWavefunction` object should be immutable. This means that whenever we want to modify the attributes of the wavefunction object, such as the orbital occupancies, we

should create a new wavefunction object instead of making changes in-place to the attributes.

By overriding specific methods of the RHF class, we can derive various mean-field models. For instance, to implement the restricted Kohn-Sham (RKS) method, we only need to modify the `get_veff` and `total_energy` methods of the RHF class, while keeping all other components unchanged. The computation of the exchange-correlation (XC) energy `exc` and the XC matrix `vxc` within these two methods will be discussed in Section 13.3.

```
class RKS(RHF):
    def get_veff(self, wfn):
        j, k = self.get_jk(wfn)
        vxc = get_vxc(wfn)
        return j - hybrid_factor * k + vxc

    def total_energy(self, wfn):
        hcore, s = self.get_hcore_s()
        j, k = self.get_jk(wfn)
        exc = get_exc(wfn)
        dm = wfn.density_matrices
        return (einsum('ij,ji', hcore, dm) + einsum('ij,ji', j, dm) * .5
                - einsum('ij,ji', k, dm) * .25 * hybrid_factor
                + exc + self.mol.nuclear_repulsion_energy())
```

Other mean-field models can be adapted to the same framework by overriding specific functions of the base SCF class. Below, we will outline some mean-field models along with the functions that need to be overridden for each model. The specific implementation details will be omitted.

- Unrestricted HF (UHF): This method involves constructing two \mathbf{V}_{eff} matrices and two Fock matrices for spin-up and spin-down orbitals. The two sets of orbitals lead to modifications in `get_veff`, `total_energy`, `orbital_gradients`, `eigen`, and `make_wfn`.
- Restricted open-shell HF (ROHF): This method also requires two \mathbf{V}_{eff} matrices corresponding to the spin-up and spin-down density matrices. The Fock matrix is constructed differently, not merely by the addition of $\mathbf{h} + \mathbf{V}_{\text{eff}}$. The affected functions are `get_veff`, `get_fock`, `total_energy`, `orbital_gradients`, and `make_wfn`.
- Generalized HF (GHF) without spin constraints on orbitals: This method results in a larger dimension of the overlap, \mathbf{h} , and \mathbf{V}_{eff} matrices. The affected functions are `get_hcore_s`, `get_veff`, and `make_wfn`.
- UKS: As a DFT method, the UKS method requires the inclusion of the XC functionals in \mathbf{V}_{eff} and the total energy calculations, similar to the RKS method. Functions `get_veff` and `total_energy` should be overridden on top of the UHF method.
- ROKS: Similar to UKS, the ROKS method requires the overriding of `get_veff` and `total_energy` functions, building on the ROHF method.

- GKS: Similar to UKS, the GKS method requires the overriding of `get_veff` and `total_energy` functions, building on the GHF method.
- The 4-component Dirac-Hartree-Fock (DHF): This relativistic method uses a different Hamiltonian and basis functions compared to the non-relativistic methods. It impacts functions that involve integral computation, including `get_hcore_s`, `get_veff`, and `make_wfn`.
- Dirac-Kohn-Sham (DKS): The relativistic DFT method, DKS, requires the overriding of `get_veff` and `total_energy` functions, building on the DHF method.

On top of these HF and KS classes, one can incorporate additional treatments and derive more variants of mean-field models:

- Scalar relativistic correction: Scalar relativistic corrections can be applied to non-relativistic methods. Only the `get_hcore_s` function needs modification.
- Spin-orbit coupling: The inclusion of one-electron spin-orbit coupling effects introduces an additional matrix in `get_hcore_s` for GHF and GKS methods.
- Two-component relativistic theory [3–6]: This relativistic approach extends the GHF or GKS methods and requires modification of the `get_hcore_s` function.
- Density fitting (DF) approximation [7,8]: The DF technique can be utilized to accelerate the calculation of electron repulsion integrals (ERIs). To implement this approximation, the `get_jk` function needs to be overridden.
- Pseudo-spectral methods [9]: Similar to the DF technique, this method can accelerate the calculation of the exchange matrix. Consequently, the `get_jk` function needs to be overridden.
- QM/MM methods [10,11]: Quantum mechanical (QM) calculations are performed with the electrostatic potential of the classical molecular mechanics (MM) particles. The `get_hcore_s` function can be overridden to incorporate the electrostatic potential, and the `total_energy` function can be adjusted to include the energy contribution from the MM component.
- Implicit solvent models: The effects of the solvent are represented by an external potential which depends on the density matrix. This potential needs to be updated during the SCF iteration. The relevant functions are `get_veff` and `total_energy`.
- Classical dispersion corrections: These corrections do not alter the quantum Hamiltonian. Only the `total_energy` function of the underlying classes needs to be adjusted.
- Point-group symmetry adaptation: The symmetry information can be utilized to block-diagonalize the Fock matrix. A straightforward method to employ this symmetry information is to override the `eigen` function. Additionally, symmetry information may be used in constructing the wave function and in the calculation of integrals. In such cases, `make_wfn` and `get_veff` need to be overridden.
- Thermal smearing method: The smearing method can improve SCF convergence by modifying the orbital occupancies. Functions `make_wfn`, `total_energy`, and `orbital_gradients` are associated with this method.
- Constraint DFT: By adjusting the effective potential with `get_veff`, this method influences the distribution of electron density and the wave function.

- Periodic boundary condition (PBC) at the Γ point: Since orbitals at the Γ point can be made real, the mean-field PBC class for the Γ point can reuse most of the functionality of the molecular mean-field class. The use of periodicity directly influences the calculation of integrals in the functions `get_hcore_s` and `get_veff`. The `total_energy` function also requires modification, as the calculation of the total energy with PBC differs from that in a molecule.
- k -point adapted PBC: The k -point PBC methods introduce an additional index to label the k -point symmetry for the orbitals and integrals. This results in the data structure of the underlying variables being different from that of the molecular mean-field classes. All methods have to be rewritten to accommodate the additional k -point index. However, the PBC mean-field class can still be adapted to the existing SCF iteration program.

These mean-field characteristics can be combined to form new mean-field models. This flexibility complicates the code structure and the hierarchy of mean-field methods. We will discuss this issue further in Section 13.5.

13.2 Coulomb and exchange matrices

In Section 13.1, we utilized the `einsum` function to compute the Coulomb matrix **J** and the exchange matrix **K**. This functionality can be abstracted in the following code snippet:

```
from eri_OS_rys import get_eri_tensor
eri_tensor = eri_OS_unrolled(gtos)
jmat = einsum('ijkl,ji->kl', eri, dm)
kmat = einsum('ijkl,jk->il', eri, dm)
```

In this process, the computation of the integral tensor `eri_tensor` is the most computationally demanding step. Here, we evaluate the entire ERI tensor, which includes a large amount of redundant integrals. To enhance performance, we can take advantage of the 8-fold permutation symmetry of the ERIs,

$$\begin{aligned} (ij|kl) &= (ji|kl) = (ij|lk) = (ji|lk) \\ &= (kl|ij) = (kl|ji) = (lk|ij) = (lk|ji). \end{aligned} \quad (13.11)$$

By considering this symmetry, we only need to compute the unique integrals that satisfy the conditions: $i \geq j$, $k \geq l$, and the compound indices $(ij) \geq (kl)$.

One approach to incorporate the 8-fold symmetry is to modify the `get_eri_tensor` function. We can compute the necessary integrals in the ERI tensor and then fill up the remaining parts of the tensor according to the permutation relation (13.11). However, this algorithm requires a substantial amount of memory to store the ERI tensor and incurs a significant memory access overhead to fill the entire tensor.

An improved approach is to only compute the unique integrals and consume them with eight types of contractions according to the 8-fold symmetry. Taking the exchange matrix as an example, the eight contraction patterns are:

```
einsum('ijkl,jk->il', eri, dm)
einsum('ijkl,ik->jl', eri, dm)
einsum('ijkl,jl->ik', eri, dm)
einsum('ijkl,il->jk', eri, dm)
einsum('ijkl,kj->li', eri, dm)
einsum('ijkl,ki->lj', eri, dm)
einsum('ijkl,lj->ki', eri, dm)
einsum('ijkl,li->kj', eri, dm)
```

Please note that the integral program developed in Chapter 12 treats individual GTO shells as the basic unit. Given the indices of four GTO shells (a shell-quartet), we can compute a small four-index ERI tensor and immediately process it with the eight contraction patterns. This approach is more memory-efficient.

To implement this algorithm, we begin with a function that does not incorporate any symmetry considerations.

```
def dispatch(gto_params, shell_idx):
    '''This function gives all positions of the four GTO shells in the
    shell-quartet'''
    i, j, k, l = shell_idx
    i0, i1 = gto_params[i][:2]
    j0, j1 = gto_params[j][:2]
    k0, k1 = gto_params[k][:2]
    l0, l1 = gto_params[l][:2]
    return i0, i1, j0, j1, k0, k1, l0, l1

def contract_k_s1(kmat, gto_params, dm, shell_idx):
    i0, i1, j0, j1, k0, k1, l0, l1 = dispatch(gto_params, shell_idx)
    sub_eri = get_eri_sub_tensor(gto_params, shell_idx)
    kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1, k0:k1])
    return

def build_k(gtos, dm):
    nao = num_functions(gtos)
    kmat = np.zeros((nao, nao))
    gto_params = pack_gto_attrs(gtos)
    ngtos = len(gtos)
    for i, j, k, l in itertools.product(*[range(ngtos)]*4):
        contract_k_s1(kmat, gto_params, dm, (i, j, k, l))
    return kmat
```

Based on this implementation, it is easy to apply the four-fold permutation symmetry

$$(ij|kl) = (ji|kl) = (ij|lk) = (ji|lk), \quad (13.12)$$

for GTO functions with $i \geq j$ and $k \geq l$. However, when the IDs of two GTO shells are identical, the GTO functions within them may not necessarily satisfy the conditions $i \geq j$ or $k \geq l$. To handle this special case, certain contraction patterns should be excluded.

```
def contract_k_s4(kmat, gto_params, dm, shell_idx):
    '''i>=j and k>=l'''
    i, j, k, l = shell_idx
    if i < j or k < l:
        return
    i0, i1, j0, j1, k0, k1, l0, l1 = dispatch(gto_params, shell_idx)
    sub_eri = get_eri_sub_tensor(gto_params, shell_idx)
    if i > j and k > l:
        kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1, k0:k1])
        kmat[j0:j1, l0:l1] += einsum('ijkl,ik->jl', sub_eri, dm[i0:i1, k0:k1])
        kmat[i0:i1, k0:k1] += einsum('ijkl,jl->ik', sub_eri, dm[j0:j1, l0:l1])
        kmat[j0:j1, k0:k1] += einsum('ijkl,il->jk', sub_eri, dm[i0:i1, l0:l1])
    elif i > j: # k == l
        kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1, k0:k1])
        kmat[j0:j1, l0:l1] += einsum('ijkl,ik->jl', sub_eri, dm[i0:i1, k0:k1])
    elif k > l: # i == j
        kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1, k0:k1])
        kmat[i0:i1, k0:k1] += einsum('ijkl,jl->ik', sub_eri, dm[j0:j1, l0:l1])
    else: # i == j and k == l
        kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1, k0:k1])
    return
```

The condition $(ij) \geq (kl)$ needs to be handled carefully as it is more complex to apply. When $i > k$, the condition $(ij) > (kl)$ is naturally satisfied. This leads to the 8-fold symmetry version `contract_k_s8`, where we can swap ij and kl on top of the `contract_k_s4` implementation. When $i = k$, imposing symmetry between ij and kl is still possible for certain values of j and l . However, correctly managing the symmetry between ij and kl in this scenario is more challenging. In the current implementation, we ignore their symmetry and directly call the `contract_k_s4` function, although this approach may slightly sacrifice performance.

```
def contract_k_s8(kmat, gto_params, dm, shell_idx):
    '''i>=j, k>=l, ij >= kl'''
    i, j, k, l = shell_idx
    if i < j or k < l or i < k:
        return
```

```

if i == k:
    return contract_k_s4(kmat, gto_params, dm, shell_idx)

i0, i1, j0, j1, k0, k1, l0, l1 = dispatch(gto_params, shell_idx)
sub_eri = get_eri_sub_tensor(gto_params, shell_idx)
if i > j and k > l:
    kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1,k0:k1])
    kmat[j0:j1, l0:l1] += einsum('ijkl,ik->jl', sub_eri, dm[i0:i1,k0:k1])
    kmat[i0:i1, k0:k1] += einsum('ijkl,jl->ik', sub_eri, dm[j0:j1,l0:l1])
    kmat[j0:j1, k0:k1] += einsum('ijkl,il->jk', sub_eri, dm[i0:i1,l0:l1])
    kmat[l0:l1, i0:i1] += einsum('ijkl,kj->li', sub_eri, dm[k0:k1,j0:j1])
    kmat[l0:l1, j0:j1] += einsum('ijkl,ki->lj', sub_eri, dm[k0:k1,i0:i1])
    kmat[k0:k1, i0:i1] += einsum('ijkl,lj->ki', sub_eri, dm[l0:l1,j0:j1])
    kmat[k0:k1, j0:j1] += einsum('ijkl,li->kj', sub_eri, dm[l0:l1,i0:i1])
elif i > j: # k == l
    kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1,k0:k1])
    kmat[j0:j1, l0:l1] += einsum('ijkl,ik->jl', sub_eri, dm[i0:i1,k0:k1])
    kmat[l0:l1, i0:i1] += einsum('ijkl,kj->li', sub_eri, dm[k0:k1,j0:j1])
    kmat[l0:l1, j0:j1] += einsum('ijkl,ki->lj', sub_eri, dm[k0:k1,i0:i1])
elif k > l: # i == j
    kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1,k0:k1])
    kmat[i0:i1, k0:k1] += einsum('ijkl,jl->ik', sub_eri, dm[j0:j1,l0:l1])
    kmat[l0:l1, i0:i1] += einsum('ijkl,kj->li', sub_eri, dm[k0:k1,j0:j1])
    kmat[k0:k1, i0:i1] += einsum('ijkl,lj->ki', sub_eri, dm[l0:l1,j0:j1])
else: # i == j and k == l
    kmat[i0:i1, l0:l1] += einsum('ijkl,jk->il', sub_eri, dm[j0:j1,k0:k1])
    kmat[l0:l1, i0:i1] += einsum('ijkl,kj->li', sub_eri, dm[k0:k1,j0:j1])
return

```

When benchmarking the symmetry-encoded J/K builder, we observed that it runs slower than the plain `einsum` implementation, which generates the full ERI tensor and then applies the `einsum` contraction on the full tensor. A simple profiling shows that 80% of the computation time in `contract_k_s8` is spent on the `einsum` function. This overhead arises from the frequent calls to the `einsum` function. For N GTO shells, `einsum` is called N^4 times. To reduce the overhead of `einsum`, two strategies can be considered:

- *JIT compilation.*

By expanding the `einsum` contraction code into nested for-loops, one can use the Numba JIT to compile the for-loop code. The compiled code will have smaller overhead from function calls.

Additionally, we can fuse different contraction patterns within the same loop. This allows each element of the `sub_eri` tensor to be accessed once and used multiple

Table 13.1 J/K builder performance.

Implementations	CPU time (seconds)
Plain contraction (w/o symmetry)	24.3
8-fold symmetry based on GTO shells	30.2
8-fold symmetry based on blocks	4.87

times, thus minimizing the overhead of repeatedly accessing the `sub_eri` tensor in separate `einsum` functions.

- *Consolidating GTO shells to blocks.*

Another approach is to consolidate several GTO shells into a single block, utilizing the block as the basic unit for the loops in the `build_k` driver. This GTO-consolidated method enlarges the dimensions of the objects involved in the `einsum` operation and reduces the number of `einsum` calls. As the basic unit increases in size, slightly more integrals need to be evaluated in each block. This treatment leads to a modest increase in the total computational costs of integral evaluation. However, the GTO-consolidated algorithm is more cache-friendly compared to the version without blocks. This advantage can offset the increased costs in integral computation. When the system contains more than 1000 orbitals, the density matrix and the resulting **J**, **K** matrices may not entirely fit into the CPU cache. By employing the consolidating algorithm, an individual sub-block of these matrices is frequently accessed within the contraction code, thereby increasing the likelihood of cache hits.

Another advantage of the GTO-consolidated algorithm is that it can be easily adapted for distributed computation. By decomposing the J/K build problem into subsystems with appropriate block sizes, parallelization can be achieved by distributing these subsystems across different computing nodes. This distributed computation can be managed using the MPI executor, as developed in Chapter 10, or the job launcher tools introduced in Chapter 7.

The current implementation of the `build_k` function can be readily adapted to the GTO-consolidated algorithm with minimal modifications. It only needs to process more GTO shells within the given block, thereby building a larger sized `sub_eri` tensor. No modifications are required for the remaining contraction code.

In Table 13.1, we evaluated the efficiency of various J/K builders using the benzene molecule as a test case. By consolidating four GTO shells into a single block, the GTO block version reduces the number of `einsum` calls to $1/4^4 = 1/256$ of the original count. Consequently, the computational cost of the `einsum` components is reduced by more than 90%. The overall performance of the GTO-consolidated algorithm is five times faster than the plain J/K contraction code. The speedup can potentially reach up to eight times as the size of the system increases. However, due to the corner cases that break the 8-fold symmetry, achieving the ideal eightfold speedup is not feasible.

13.2.1 Integral screening for Coulomb and exchange matrices

You might have noticed that the values of ERIs can vary greatly. In a medium-sized molecule, most ERIs have values that are very close to zero, and their contribution to the J/K matrices is negligible. Similarly, if the values of the density matrix elements are very small, the computation of related ERIs and tensor contractions can be omitted. Based on the magnitudes of the ERIs and the density matrix elements, we can incorporate a screening process to exclude calculations that do not contribute significantly to the final result.

To estimate the value of an ERI, we can use the Schwarz inequality to find an upper bound for the ERI:

$$(ij|kl) \leq \sqrt{(ij|ij)}\sqrt{(kl|kl)}. \quad (13.13)$$

$\sqrt{(ij|ij)}$ is calculated using the following function:

```
def schwarz_halfcond(gtos):
    '''sqrt((ij|ij)) for all GTOs'''
    gto_params = pack_gto_attrs(gtos)
    nao = num_functions(gtos)
    v = np.zeros((nao, nao))
    for params_i, params_j in itertools.product(gto_params, gto_params):
        i0, i1, li, Ra, exps_i, norm_ci = params_i
        j0, j1, lj, Rb, exps_j, norm_cj = params_j
        out = np.zeros((i1-i0, j1-j0, i1-i0, j1-j0))
        contracted_ERI(li, lj, i1, j1, exps_i, exps_j, exps_i, exps_j,
                        norm_ci, norm_cj, norm_ci, norm_cj,
                        Ra, Rb, Ra, Rb, out, 0, 0, 0, 0)
        v[i0:i1,j0:j1] = einsum('ijij->ij', out) ** .5
    return v
```

To add the functionality of integral screening to the `contract_k` function, we can apply the following `screen_k` decorator. This screening function takes into account the 8-fold permutation symmetry within the ERI tensor and the hermitian symmetry of the mean-field density matrix.

```
def screen_k(contract_k, gtos, dm, threshold=1e-12):
    sc_half = schwarz_halfcond(gtos)
    offsets = gto_offsets(gtos)
    ngtos = len(gtos)
    sc = np.zeros((ngtos, ngtos))
    dc = np.zeros((ngtos, ngtos))
    for i, (i0, i1) in enumerate(itertools.pairwise(offsets)):
        for j, (j0, j1) in enumerate(itertools.pairwise(offsets)):
            sc[i,j] = sc_half[i0:i1,j0:j1].max()
            dc[i,j] = dm[i0:i1,j0:j1].max()
```

```

def screen_contract_k(kmat, gto_params, dm, shell_idx):
    i, j, k, l = shell_idx
    dm_max = max(dc[j,k], dc[i,k], dc[j,l], dc[i,l])
    if threshold > sc[i,j]*sc[k,l] * dm_max:
        return
    return contract_k(kmat, gto_params, dm, shell_idx)
return screen_contract_k

def build_k_8fold(gtos, dm):
    contract_k = screen_k(contract_k_s8, gtos, dm)
    nao = num_functions(gtos)
    kmat = np.zeros((nao, nao))
    gto_params = pack_gto_attrs(gtos)
    ngtos = len(gtos)
    for i in range(ngtos):
        for j in range(i+1):
            for k in range(j+1):
                for l in range(k+1):
                    contract_k(kmat, gto_params, dm, (i,j,k,l))
    return kmat

```

For more refined screening of ERIs at the level of individual shell-quartets, similar decorators can be developed for the `contracted_ERI` or `primitive_ERI` functions. We will not delve into the specific code details of these decorators, as their implementation should be straightforward.

The J/K matrices are linear functions of the density matrix. This property enables us to incrementally construct the J/K matrices within the SCF iterations. By utilizing the J/K matrices from the previous iteration along with the changes in the density matrices relative to the previous iteration, we can compute the J/K matrices for the current step.

```
kmat = build_k(gtos, dm-dm_prev) + kmat_prev
```

As the SCF iterations progress, the changes in the density matrices decrease. By employing the previously mentioned integral screening technique, more integral computations can be omitted as the SCF approaches convergence. Please note that errors may accumulate if J/K matrices are constructed incrementally. The integral screening threshold should be appropriately tightened.

13.2.2 J-engine

In the McMurchie-Davidson ERI algorithm (Section 12.1.3.2 in Chapter 12), an individual ERI block `sub_eri` for a specific GTO shell-quartet is constructed through the tensor contraction of the intermediate R and E tensors:

```
sub_eri = einsum('ijp,pq,qkl->ijkl', E, R, E)
```

Here, we temporarily disregard the issue of basis contraction. When computing its contribution to the J matrix, we can obtain the following contraction operations:

```
jmat[k0:k1, l0:l1] = einsum(
    'ji,ijp,pq,qk1->k1', dm[j0:j1,i0:i1], E, R, E, optimize=True)
```

Optimizing the order of contractions can reduce the computational effort required for integral calculations. The contraction between the density matrix and one of the E tensors can be performed first. Following this, the contractions with the R tensor and the other E tensor are carried out. We can apply this procedure to each ERI block, and implement the function to compute J matrix as follows:

```
from pychem_book.chap12.analytical_integrals.v5.coulomb_1e_MD import (
    get_R_tensor, get_E_tensor)

def build_j(gtos, dm):
    nao = num_functions(gtos)
    jmat = np.zeros((nao, nao))
    gto_params = pack_gtoAttrs(gtos)
    ngtos = len(gtos)

    @lru_cache(ngtos*ngtos)
    def contract_Et_dm(i, j):
        i0, i1, li, Ra, exps_i, norm_ci = gto_params[i]
        j0, j1, lj, Rb, exps_j, norm_cj = gto_params[j]
        dm_block = dm[j0:j1, i0:i1]
        Et_dm = []
        for ai, ci in zip(exps_i, norm_ci):
            for aj, cj in zip(exps_j, norm_cj):
                Et = get_E_tensor(li, lj, ai, aj, Ra, Rb)
                Et_dm.append(ci * cj * einsum('ijt,ji->t', Et, dm_block))
        return Et_dm

    for i in range(ngtos):
        for j in range(ngtos):
            Rt_dm = 0.
            for k in range(ngtos):
                for l in range(ngtos):
                    Et_dm = contract_Et_dm(k, l) # (1)
                    Rt_dm += contract_dm_R(
                        gto_params, (i,j,k,l), Et_dm) # (2)

    i0, i1, li, Ra, exps_i, norm_ci = gto_params[i]
    j0, j1, lj, Rb, exps_j, norm_cj = gto_params[j]
    ij = 0
```

```

jblock = 0.
for ai, ci in zip(exps_i, norm_ci):
    for aj, cj in zip(exps_j, norm_cj):
        Et = get_E_tensor(li, lj, ai, aj, Ra, Rb)
        R_cicj = ci * cj * Rt_dm[ij]
        jblock += einsum('ijt,t->ij', Et, R_cicj)      # (3)
        ij += 1
    jmat[i0:i1,j0:j1] = jblock
return jmat

```

The intermediates from the contraction between the density matrix and the E tensors can be precomputed and cached, as shown in line (1). At line (2), the `contract_dm_R` function applies the contraction between the intermediates and the R tensor. Finally, we use another E tensor to transform the R -intermediates into the J matrix in line (3). For simplicity, the 8-fold permutation symmetry is not utilized in this demonstration. The 8-fold symmetry can be enabled by following a similar methodology as described in previous sections.

The R tensor in the MD algorithm is constructed for primitive GTOs only. By modifying the MD integral program in Chapter 12, we can embed the R tensor program into the contraction code and derive the function `contract_dm_R` as follows:

```

def contract_dm_R(gto_params, shell_idx, Et_dm):
    i, j, k, l = shell_idx
    i0, i1, li, Ra, exps_i, norm_ci = gto_params[i]
    j0, j1, lj, Rb, exps_j, norm_cj = gto_params[j]
    k0, k1, lk, Rc, exps_k, norm_ck = gto_params[k]
    l0, l1, ll, Rd, exps_l, norm_cl = gto_params[l]
    out = []
    for ai in exps_i:
        for aj in exps_j:
            Rt_dm = 0.
            k1 = 0
            for ak in exps_k:
                for al in exps_l:
                    Rt = primitive_R_tensor(li, lj, lk, ll, ai, aj, ak, al,
                                           Ra, Rb, Rc, Rd)
                    Rt_dm += einsum('pq,q->p', Rt, Et_dm[k1])
                    k1 += 1
            out.append(Rt_dm)
    return np.array(out)

def primitive_R_tensor(li, lj, lk, ll, ai, aj, ak, al, Ra, Rb, Rc, Rd):
    lij = li + lj
    aij = ai + aj
    Rp = (ai * Ra + aj * Rb) / aij

```

```

lkl = lk + ll
akl = ak + al
Rq = (ak * Rc + al * Rd) / akl
Rpq = Rp - Rq
theta = aij * akl / (aij + akl)
Rt = get_R_tensor(lij + lkl, theta, Rpq)
nf_ij = (lij+1)*(lij+2)*(lij+3)//6
nf_kl = (lkl+1)*(lkl+2)*(lkl+3)//6
Rt2 = np.empty((nf_ij, nf_kl))
for ij, (e, f, g) in enumerate(reduced_cart_iter(lij)):
    phase = (-1)**(e+f+g)
    for kl, (t, u, v) in enumerate(reduced_cart_iter(lkl)):
        Rt2[ij,kl] = phase * Rt[t+e,u+f,v+g]
Rt2 *= 2*np.pi**2.5/(aij*akl*(aij+akl)**.5)
return Rt2

@lru_cache(20)
def reduced_cart_iter(n):
    '''Nested loops for Cartesian components, subject to x+y+z <= n'''
    return [(x, y, z) for x in range(n+1)
            for y in range(n+1-x) for z in range(n+1-x-y)]

```

Compared to the approach that constructs the four-index integral tensor and then applies the `einsum` contraction, the three-step contraction strategy described above can reduce the FLOP counts and the memory footprint of intermediates, thereby enhancing the performance of the J matrix construction. This optimization technique for the computation of J matrix is known as the J-engine algorithm [12].

13.3 Integrals for exchange-correlation functionals

KS-DFT methods require the computation of the exchange-correlation (XC) potential V_{xc} in the `get_veff` function. This involves two computational tasks: calculating the derivatives of the XC functional ϵ_{xc} and performing the numerical integration to obtain the XC matrix \mathbf{V}_{xc} .

In some literature, the XC functional is defined as the XC energy per particle, $e_{xc}[\rho]$, leading to the following integral for the XC energy:

$$E_{xc}[\rho] = \int e_{xc}[\rho] \rho(r) d^3\mathbf{r}. \quad (13.14)$$

However, for simplicity, in this section, we will define the XC functional $\epsilon_{xc}[\rho]$ as the integrand of E_{xc} :

$$E_{xc}[\rho] = \int \epsilon_{xc}[\rho] d^3\mathbf{r}, \quad (13.15)$$

where

$$\epsilon_{xc} = e_{xc}\rho. \quad (13.16)$$

This convention is utilized by popular XC functional libraries such as LibXC [13] and XCFun [14]. The electron density in this integral is computed using the density matrix, denoted as γ , and AO functions, $\chi(\mathbf{r})$. For instance, the density of spin-up electrons can be expressed as

$$\rho_{\uparrow} = \sum_{\mu\nu} \gamma_{\nu\mu}^{\uparrow} \chi_{\mu}(\mathbf{r}) \chi_{\nu}^{*}(\mathbf{r}). \quad (13.17)$$

For the Local Density Approximation (LDA) functional $\epsilon_{xc}[\rho_{\uparrow}, \rho_{\downarrow}]$, the XC matrix of spin-up orbitals can be expressed as

$$(V_{xc}^{\uparrow})_{\mu\nu} = \frac{\partial E_{xc}}{\partial \gamma_{\nu\mu}^{\uparrow}} = \int \chi_{\mu} \chi_{\nu} \frac{\partial \epsilon_{xc}}{\partial \rho_{\uparrow}} d^3 \mathbf{r}. \quad (13.18)$$

By employing an appropriate set of integration grids \mathbf{r}_n , and weights ω_n , we can numerically compute the LDA matrix elements.

$$(V_{xc}^{\uparrow})_{\mu\nu} = \sum_n \omega_n \chi_{\mu}(\mathbf{r}_n) \chi_{\nu}(\mathbf{r}_n) \frac{\partial \epsilon_{xc}}{\partial \rho_{\uparrow}}. \quad (13.19)$$

In the case of the Generalized Gradient Approximation (GGA) functional $\epsilon_{xc}[\rho, \sigma]$, it additionally depends on the contracted density gradients

$$\sigma_{\uparrow\uparrow} = \nabla \rho_{\uparrow} \cdot \nabla \rho_{\uparrow}, \quad \sigma_{\uparrow\downarrow} = \nabla \rho_{\uparrow} \cdot \nabla \rho_{\downarrow}, \quad \sigma_{\downarrow\downarrow} = \nabla \rho_{\downarrow} \cdot \nabla \rho_{\downarrow}. \quad (13.20)$$

The XC matrix elements for spin-up orbitals can be calculated

$$\begin{aligned} (V_{xc}^{\uparrow})_{\mu\nu} &= \int \chi_{\mu} \chi_{\nu} \left(\frac{\partial \epsilon_{xc}}{\partial \rho_{\uparrow}} + \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow}} \frac{\partial \sigma_{\uparrow\uparrow}}{\partial \gamma_{\nu\mu}^{\uparrow}} + \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\downarrow}} \frac{\partial \sigma_{\uparrow\downarrow}}{\partial \gamma_{\nu\mu}^{\uparrow}} \right) d^3 \mathbf{r} \\ &= \int \chi_{\mu} \chi_{\nu} \frac{\partial \epsilon_{xc}}{\partial \rho_{\uparrow}} + \nabla(\chi_{\mu} \chi_{\nu}) \cdot \left(2 \nabla \rho_{\uparrow} \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow}} + \nabla \rho_{\downarrow} \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\downarrow}} \right) d^3 \mathbf{r}. \end{aligned} \quad (13.21)$$

This integral can be translated into the numerical integration program as:

$$(V_{xc}^{\uparrow})_{\mu\nu} = \sum_n \omega_n \left(\chi_{\mu}(\mathbf{r}_n) \chi_{\nu}(\mathbf{r}_n) v_{\rho\uparrow} + \nabla(\chi_{\mu}(\mathbf{r}_n) \chi_{\nu}(\mathbf{r}_n)) \cdot v_{\nabla\rho\uparrow} \right), \quad (13.22)$$

where

$$v_{\nabla\rho\uparrow} = 2 \nabla \rho_{\uparrow} \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow}} + \nabla \rho_{\downarrow} \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\downarrow}}. \quad (13.23)$$

For the meta-GGA functional $\epsilon_{xc}[\rho, \sigma, \tau]$, the kinetic energy density τ is defined as

$$\tau = \frac{1}{2} \sum_{\mu\nu} \nabla \chi_\mu \cdot \nabla \chi_\nu \gamma_{\nu\mu}^\uparrow. \quad (13.24)$$

We can perform the numerical integration for the XC matrix:

$$(V_{xc}^\uparrow)_{\mu\nu} = \sum_n \omega_n (\chi_\mu(\mathbf{r}_n) \chi_\nu(\mathbf{r}_n) \frac{\partial \epsilon_{xc}}{\partial \rho_\uparrow} + \nabla(\chi_\mu(\mathbf{r}_n) \chi_\nu(\mathbf{r}_n)) \cdot v \nabla \rho_\uparrow + \frac{1}{2} \nabla \chi_\mu(\mathbf{r}_n) \cdot \nabla \chi_\nu(\mathbf{r}_n) \frac{\partial \epsilon_{xc}}{\partial \tau_\uparrow}). \quad (13.25)$$

Some mean-field based applications, such as stability analysis and time-dependent DFT, may require the higher order derivatives of XC functionals. The computation of integrals for these higher-order XC derivatives is more complex. Let's consider the second-order derivatives of the GGA functional as an example. The integral of the $f_{xc}^{\uparrow\uparrow}$ component is given by:

$$(f_{xc}^{\uparrow\uparrow})_{\mu\nu,\kappa\lambda} = \frac{\partial (V_{xc})_{\mu\nu}}{\partial \gamma_{\kappa\lambda}^\uparrow} = \int q d^3\mathbf{r}, \quad (13.26)$$

where the integrand q is expressed as:

$$\begin{aligned} q = & \chi_\mu \chi_\nu \chi_\kappa \chi_\lambda \frac{\partial^2 \epsilon_{xc}}{\partial \rho_\uparrow \partial \rho_\uparrow} \\ & + \chi_\mu \chi_\nu \nabla(\chi_\kappa \chi_\lambda) \cdot \left(2 \nabla \rho_\uparrow \frac{\partial^2 \epsilon_{xc}}{\partial \rho_\uparrow \partial \sigma_{\uparrow\uparrow}} + \nabla \rho_\downarrow \frac{\partial^2 \epsilon_{xc}}{\partial \rho_\uparrow \partial \sigma_{\uparrow\downarrow}} \right) \\ & + \chi_\kappa \chi_\lambda \nabla(\chi_\mu \chi_\nu) \cdot \left(2 \nabla \rho_\uparrow \frac{\partial^2 \epsilon_{xc}}{\partial \rho_\uparrow \partial \sigma_{\uparrow\uparrow}} + \nabla \rho_\downarrow \frac{\partial^2 \epsilon_{xc}}{\partial \rho_\uparrow \partial \sigma_{\uparrow\downarrow}} \right) \\ & + 2 \nabla(\chi_\mu \chi_\nu) \cdot \left(2 \nabla \rho_\uparrow \frac{\partial^2 \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow} \partial \sigma_{\uparrow\uparrow}} + \nabla \rho_\downarrow \frac{\partial^2 \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow} \partial \sigma_{\uparrow\downarrow}} \right) \nabla(\chi_\kappa \chi_\lambda) \cdot \nabla \rho_\uparrow \\ & + \nabla(\chi_\mu \chi_\nu) \cdot \left(2 \nabla \rho_\uparrow \frac{\partial^2 \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow} \partial \sigma_{\uparrow\downarrow}} + \nabla \rho_\downarrow \frac{\partial^2 \epsilon_{xc}}{\partial \sigma_{\uparrow\downarrow} \partial \sigma_{\uparrow\downarrow}} \right) \nabla(\chi_\kappa \chi_\lambda) \cdot \nabla \rho_\downarrow \\ & + 2 \nabla(\chi_\mu \chi_\nu) \cdot \nabla(\chi_\kappa \chi_\lambda) \frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\uparrow}}. \end{aligned} \quad (13.27)$$

Various intermediates would need to be generated if the integrals are computed using this formula. This can significantly increase the complexity of the numerical integration program, making it more prone to errors. Actually, the numerical integration for XC functionals can be simplified by transforming this formula into a process of tensor contractions.

Let's treat the density variables (ρ , $\nabla\rho$, and τ) as general variables x

$$x = (\rho_{\uparrow}, \nabla_x \rho_{\uparrow}, \nabla_y \rho_{\uparrow}, \dots, \tau_{\uparrow}, \rho_{\downarrow}, \dots).$$

The partial derivatives of ϵ_{xc} with respect to these variables can be written as an n -dimensional tensor v

$$v^i = \frac{\partial \epsilon_{xc}}{\partial x^i}, \quad (13.28)$$

$$v^{ij} = \frac{\partial^2 \epsilon_{xc}}{\partial x^i \partial x^j}. \quad (13.29)$$

By using these derivative tensors, the computation of XC matrix can be simplified to the tensor contraction:

$$(V_{xc}^{\uparrow})_{\mu\nu} = \sum_n \sum_{s \text{ has } \uparrow} \omega_n x_{\mu\nu}^s[\mathbf{r}_n] v^s[\mathbf{r}_n]. \quad (13.30)$$

Here, the summation over s is restricted to the variables associated with the \uparrow spin. Similarly, the f_{xc} integrals can be computed with the tensor contraction

$$(f_{xc}^{\uparrow\uparrow})_{\mu\nu,\kappa\lambda} = \sum_n \sum_{s \text{ has } \uparrow} \sum_{t \text{ has } \uparrow} \omega_n x_{\mu\nu}^s[\mathbf{r}_n] v^{st}[\mathbf{r}_n] x_{\kappa\lambda}^t[\mathbf{r}_n]. \quad (13.31)$$

Using the well-established integration grids, such as Becke grids [15] or uniform grids, we can calculate the XC matrix using the XC derivative tensor and the `einsum` contraction function:

```
def get_vxc(gtos, dm, xc_code, grids, weights):
    ao = eval_gtos(gtos, grids) # (1)
    nao, ngrids = ao.shape[1:]
    rho_ij = np.zeros((5, nao, nao, ngrids))
    xctype = xc_family(xc_code) # (2)
    xc_tensor = xc_deriv_tensor1(density, xc_code) # (3)
    match xctype:
        case 'LDA':
            rho_ij[0] = einsum('xig,xjg->ijg', ao[:1], ao[:1])
        case 'GGA':
            rho_ij[0:4] = einsum('xig,jg->xijg', ao[:4], ao[0])
            rho_ij[1:4] += einsum('ig,xjg->xijg', ao[0], ao[1:4])
        case 'MGGA':
            rho_ij[0:4] = einsum('xig,jg->xijg', ao[:4], ao[0])
            rho_ij[1:4] += einsum('ig,xjg->xijg', ao[0], ao[1:4])
            rho_ij[4] = einsum('xig,xjg->ijg', ao[1:4], ao[1:4]) * .5
    density = einsum('xijg,sji->sxg', rho_ij, dm)
    vxc = einsum('g,xijg,axg->aij', weights, rho_ij, xc_tensor)
    return vxc
```

The `ao` tensor in line (1) stores the values of GTO functions and their derivatives on grids. The `xc_code` in line (2) is an abbreviation for the XC functional, such as '`PBE`' or '`B3LYP`'. The XC derivative tensor is generated by the `xc_deriv_tensor1` function in line (3), which we will discuss later.

The `ao` tensor has four entries in the first dimension, corresponding to the GTO value and the x , y , and z components of their first-order derivatives. Given the simple form of GTO functions, it is straightforward to compute the `ao` tensor. For instance, the derivative of a Cartesian GTO can be evaluated as follows:

$$\nabla_x G_{ijk}(\mathbf{r}, \alpha, \mathbf{R}_A) = 2\alpha G_i(x)G_j(y)G_k(z) + iG_{i-1}(x)G_j(y)G_k(z). \quad (13.32)$$

Below is the implementation for computing the GTO values up to their first-order derivatives, which are sufficient for the computation of the XC matrix.

```
def eval_gto(gto, grids):
    l = gto.angular_momentum
    dx, dy, dz = (grids - gto.coordinates).T
    r2 = dx * dx + dy * dy + dz * dz
    ao = np.zeros((4, n_cart(l), ngrids))

    for e, c in zip(gto.exponents, gto.coefficients):
        fac = c * np.exp(-e * r2)
        gtox = dx**np.arange(0, l+2)[:,np.newaxis]
        gtoy = dy**np.arange(0, l+2)[:,np.newaxis]
        gtoz = dz**np.arange(0, l+2)[:,np.newaxis]
        for n, (lx, ly, lz) in enumerate(iter_cart_xyz(l)):
            ao[0,n] += fac * gtox[lx] * gtoy[ly] * gtoz[lz]

            ao[1,n] -= 2 * e * fac * gtox[lx+1] * gtoy[ly] * gtoz[lz]
            if lx > 0:
                ao[1,n] += fac * lx * gtox[lx-1] * gtoy[ly] * gtoz[lz]
            ao[2,n] -= 2 * e * fac * gtox[lx] * gtoy[ly+1] * gtoz[lz]
            if ly > 0:
                ao[2,n] += fac * ly * gtox[lx] * gtoy[ly-1] * gtoz[lz]
            ao[3,n] -= 2 * e * fac * gtox[lx] * gtoy[ly] * gtoz[lz+1]
            if lz > 0:
                ao[3,n] += fac * lx * gtox[lx] * gtoy[ly] * gtoz[lz-1]
    return ao

def eval_gtos(gtos, grids):
    return np.concatenate([eval_gto(gto, grids) for gto in gtos], axis=1)
```

In the tensor contraction scheme mentioned above, the primary challenge is the calculation of the XC derivative tensors v^s and v^{st} . Normally, XC libraries such as

LibXC or XCfun only provide partial derivatives with respect to ρ , σ , and τ , such as

$$\frac{\partial^2 \epsilon_{xc}}{\partial \rho_{\uparrow} \partial \sigma_{\uparrow\downarrow}}, \frac{\partial^2 \epsilon_{xc}}{\partial \sigma_{\uparrow\downarrow} \partial \sigma_{\uparrow\downarrow}}.$$

Furthermore, these libraries omit certain derivatives due to the symmetry in partial derivatives

$$\frac{\partial^2 \epsilon_{xc}}{\partial x \partial y} = \frac{\partial^2 \epsilon_{xc}}{\partial y \partial x},$$

and the symmetry between $\sigma_{\uparrow\downarrow}$ and $\sigma_{\downarrow\uparrow}$

$$\frac{\partial \epsilon_{xc}}{\partial \sigma_{\uparrow\downarrow}} = \frac{\partial \epsilon_{xc}}{\partial \sigma_{\downarrow\uparrow}}.$$

To construct the XC derivative tensor v in Eqs. (13.28) and (13.29), we need to restore the omitted derivatives. This involves two types of conversions:

- Type I conversion restores the symmetry in second-order partial derivatives

$$\left(\frac{\partial^2 \epsilon}{\partial x \partial x}, \frac{\partial^2 \epsilon}{\partial x \partial y}, \frac{\partial^2 \epsilon}{\partial y \partial y} \right) \rightarrow \begin{pmatrix} \frac{\partial^2 \epsilon}{\partial x \partial x} & \frac{\partial^2 \epsilon}{\partial x \partial y} \\ \frac{\partial^2 \epsilon}{\partial x \partial y} & \frac{\partial^2 \epsilon}{\partial y \partial y} \end{pmatrix}. \quad (13.33)$$

```
def conversion_1(v, axis=0):
    '''[u_u, u_d, d_d] -> [[u_u, u_d], [d_u, d_d]]'''
    assert v.shape[axis] == 3
    return v[(slice(None),) * axis + ([0,1],[1,2]),:]
```

- Type II conversion handles the different spin components of σ

$$\left(\frac{\partial \epsilon}{\partial \sigma_{\uparrow\uparrow}}, \frac{\partial \epsilon}{\partial \sigma_{\uparrow\downarrow}}, \frac{\partial \epsilon}{\partial \sigma_{\downarrow\downarrow}} \right) \rightarrow \begin{pmatrix} 2 \frac{\partial \epsilon}{\partial \sigma_{\uparrow\uparrow}} & \frac{\partial \epsilon}{\partial \sigma_{\uparrow\downarrow}} \\ \frac{\partial \epsilon}{\partial \sigma_{\uparrow\downarrow}} & 2 \frac{\partial \epsilon}{\partial \sigma_{\downarrow\downarrow}} \end{pmatrix}. \quad (13.34)$$

```
def conversion_2(v, axis=0):
    '''[uu, ud, dd] -> [[uu*2, ud], [du, dd*2]]'''
    assert v.shape[axis] == 3
    v = v[(slice(None),) * axis + ([0,1],[1,2]),:]
    v[(slice(None),)*axis + (0,0)] *= 2 # (1)
    v[(slice(None),)*axis + (1,1)] *= 2
    return v
```

In the two conversion functions, the NumPy fancy indexing $[[0,1],[1,2]]$ is applied to reorder a 3-element vector into a 2×2 array. The index to be transformed may not always be the leading index. To accommodate this, we use the

`slice(None)` notation for the missing indices. For example, the complex indexing code `v[(slice(None),)*axis + (0,0)]` in line (1) is equivalent to the array indexing `v[:, :, 0, 0]` when `axis=2`.

If we consider the spin indices for the density variables ρ , $\nabla\rho$, and τ , the LDA functional has 2 independent variables, GGA has 2×4 variables, and meta-GGA has 2×5 variables. These variables determine the shapes of the first-order and the second-order XC derivative tensors. For LDA, the shapes are $(2, 1)$ and $(2, 1, 2, 1)$, respectively. For GGA, the shapes are $(2, 4)$ and $(2, 4, 2, 4)$, respectively. For meta-GGA, the shapes are $(2, 5)$ and $(2, 5, 2, 5)$, respectively.

When constructing XC derivative tensors in the program, we can utilize type I conversion to restore symmetry for the derivatives of ρ and τ . For the derivatives of $\nabla\rho$, Type II conversion is necessary to transform the derivatives of σ into a symmetric form. Below, we present the code for the first-order and second-order XC derivative tensors.

```

def xc_deriv_tensor1(density, xc_code):
    raw_xc = eval_xc(xc_code, density, spin=1, deriv=1) # (1)
    xtype = xc_family(xc_code)
    ngrids = density.shape[-1]
    if xtype == 'LDA':
        vp = raw_xc['vrho'].reshape(2, 1, ngrids)
    else:
        if xtype == 'GGA':
            vp = np.empty((2, 4, ngrids))
        else:
            vp = np.empty((2, 5, ngrids))
        vp[:,0] = raw_xc['vrho']
        vp[:,1:4] = einsum('abg,bxg->axg',
                            conversion_2(raw_xc['vsigma']), density[:,1:4])
    if xtype == 'MGGA':
        vp[:,4] = raw_xc['vtau']
    return vp

def xc_deriv_tensor2(density, xc_code):
    raw_xc = eval_xc(xc_code, density, spin=1, deriv=2)
    xtype = xc_family(xc_code)
    ngrids = density.shape[-1]
    if xtype == 'LDA':
        vp = conversion_1(raw_xc['v2rho2']).reshape(2,1,2,1, ngrids)
    elif xtype == 'GGA':
        vp = np.empty((2, 4, 2, 4, ngrids))
        vp[:,0,:,:0] = conversion_1(raw_xc['v2rho2'])
        # First transforms to
        # [[uu_uu, ud_ud, ud_dd],
        # [ud_uu, ud_ud, ud_dd],
```

```

# [dd_uu, dd_ud, dd_dd]
qgg = raw_xc['v2sigma2'][[[0, 1, 2],
                           [1, 3, 4],
                           [2, 4, 5]]]
# Expands to
#[[uu_uu*2, ud_ud], [ud_ud, ud_dd*2]],
#[[ud_uu*2, ud_ud], [ud_ud, ud_dd*2]],
#[[dd_uu*2, dd_ud], [dd_ud, dd_dd*2]]]
qgg = conversion_2(qgg, axis=1)
# Expands to
#[[[uu_uu*2*2, ud_ud*2], [ud_ud*2, ud_dd*2*2]],
#[[ud_uu*2 , ud_ud ], [ud_ud , ud_dd*2 ]]],
#[[[ud_uu*2 , ud_ud ], [ud_ud , ud_dd*2 ]],
#[[dd_uu*2*2, dd_ud*2], [dd_ud*2, dd_dd*2*2]]]
qgg = conversion_2(qgg, axis=0)
vp[:,1:4,:,:1:4] = einsum('abcdg,bxg,dyg->axcyg',
                           qgg, density[:,1:4], density[:,1:4])
qg = conversion_2(raw_xc['vsigma'])
for i in range(1, 4):
    vp[:,i,:,:i] += qg
qrg = conversion_2(raw_xc['v2rhosigma'].reshape(2,3,ngrids), 1)
qrg = einsum('rabg,bxg->raxg', qrg, density[:,1:4])
vp[:,0,:,:1:4] = qrg
vp[:,1:4,:,:0] = qrg.transpose(1,2,0,3)
elif xctype == 'MGGA':
    ...
return vp

```

The helper function `eval_xc` in line (1) is used to evaluate XC functionals in the LibXC library. For simplicity, we utilize the interface provided by the PySCF package to access the functionalities of LibXC.

```

def eval_xc(density, xc_code, order=0):
    '''Returns the raw derivatives from XC libraries

    exc: the energy
    vrho: first partial derivative in terms of the density
    vsigma: first partial derivative in terms of sigma
    vtau: first partial derivative in terms of the kinetic energy density
    v2rho2, v2rhosigma, v2sigma2, v2rhoau, v2sigmatau, v2tau2:
        second order partial derivatives
    ...
    # Use pyscf libxc module to mimic the LibXC functionality
    from pyscf.dft import libxc
    assert density.ndim == 3

```

```

exc, vxc, fxc, kxc = libxc.eval_xc(density, xc_code, deriv=order)
xctype = xc_family(xc_code)

results = {}
results['exc'] = exc * density[0]
if order > 0:
    results['vrho'] = vxc[0].T
    if 'GGA' in xctype:
        results['vsigma'] = vxc[1].T
    if 'MGGA' in xctype:
        results['vtau'] = vxc[3].T
if order > 1:
    results['v2rho2'] = fxc[0].T
    if 'GGA' in xctype:
        results['v2rhosigma'] = fxc[1].T
        results['v2sigma2'] = fxc[2].T
    if 'MGGA' in xctype:
        results['v2rhotau'] = fxc[6].T
        results['v2sigmatau'] = fxc[9].T
        results['v2tau2'] = fxc[4].T
if order > 2:
    raise NotImplementedError
return results

def xc_family(xc_code):
    '''Returns the family (LDA, GGA, MGGA) of a XC functional.
    ...
    from pyscf.dft import libxc
    return libxc.xc_type(xc_code)

```

The approach for constructing higher-order tensors is similar, though it is more complex. Following this methodology, the PySCF package offers a general transformation function to construct derivative tensors of arbitrary orders. Readers interested in the detailed implementation can refer to the PySCF source code, which we will not explore in this discussion.

13.4 DIIS

DIIS (Direct Inversion in the Iterative Subspace) extrapolation method was originally developed by Peter Pulay to accelerate and stabilize the convergence of Hartree-Fock self-consistent field (SCF) calculations. The DIIS extrapolation method is commonly used to accelerate fixed-point iterations. The fundamental concept of DIIS is to approximate the solution using a set of trial vectors generated during the fixed-point

iteration. Given a set of trial vectors \mathbf{t} and their associated error vectors \mathbf{e} , the solution vector is approximated by a linear combination of the trial vectors with coefficients \mathbf{c} :

$$x = \sum_i c_i \mathbf{t}_i. \quad (13.35)$$

The coefficients are determined by minimizing the linear combinations of the error vectors

$$L = \left| \sum_i c_i \mathbf{e}_i \right|^2 + 2\lambda \left(\sum_i c_i - 1 \right), \quad (13.36)$$

subject to the constraints $\sum_i c_i = 1$. The stationary condition of the Lagrangian yields the linear equation for the coefficients

$$\begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1n} & 1 \\ B_{21} & B_{22} & \cdots & B_{2n} & 1 \\ \vdots & \vdots & \ddots & \vdots & 1 \\ B_{n1} & B_{n2} & \cdots & B_{nn} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ \lambda \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}, \quad (13.37)$$

where $B_{ij} = \mathbf{e}_i \cdot \mathbf{e}_j$.

It is straightforward to translate these equations to a Python program. Given a list of error vectors and trail vectors as the input, the `extrapolate` function below generates a solution vector.

```
def extrapolate(errvecs, tvecs):
    space = len(tvecs)
    B = np.zeros((space+1, space+1))
    B[-1, :-1] = B[:-1, -1] = 1.
    g = np.zeros(space+1)
    g[-1] = 1
    for i, e1 in enumerate(errvecs):
        for j, e2 in enumerate(errvecs):
            if j < i:
                continue
            B[i, j] = B[j, i] = e1.dot(e2)

    c = scipy.linalg.solve(B, g, assume_a='sym')[:-1]
    sol = tvecs[0] * c[0]
    for v, x in zip(tvecs[1:], c[1:]):
        sol += v * x
    return sol
```

In a fixed-point iteration process, the error vector and trial vector from each iteration are added to the DIIS program to expand the space for error vectors and trial

vectors. These vectors are passed to the `extrapolate` function to construct a new solution, which is then utilized in the subsequent iteration of the fixed-point problem.

How can we book-keep these states across separate function calls? There are three common strategies to achieve this in Python:

- Using a *class*. DIIS states are stored as attributes of a class. Specific methods of the class can be executed to update the states within the SCF iteration program.
- Using a *higher-order function*. DIIS states are stored in local variables of a higher-order function. This high-order function can create a *closure*, which retains access to the local variables of the high-order function. The closure can be invoked by the SCF iteration solver to access and update these states.
- Using a *coroutine*. DIIS states are preserved within a coroutine. A coroutine is a special type of function that can be paused and resumed. When the coroutine is resumed by the SCF iteration solver, the DIIS can continue the extrapolation process using the states that were created earlier.

Here, we will utilize a class-based approach to develop the DIIS program. This choice is convenient for extending the DIIS functionalities, given the existence of multiple DIIS variants.

```
from collections import deque
class DIIS:
    def __init__(self, max_space=8):
        self.errvecs = deque(maxlen=max_space)
        self.tvecs = deque(maxlen=max_space)

    def update(self, errvec, tvec):
        self.errvecs.append(_HashableVector(errvec.ravel()))
        self.tvecs.append(_HashableVector(tvec))
        return extrapolate(self.errvecs, self.tvecs)

class _HashableVector:
    def __init__(self, vec):
        self.data = vec

    @lru_cache
    def dot(self, other):
        assert self.data.ndim == other.data.ndim == 1
        return self.data.dot(other.data)

    def __mul__(self, val):
        return self.data * val
```

In quantum chemistry applications, it is often not necessary to retain all DIIS states. The DIIS program can discard older states and approximates the solution with the latest n trial vectors. The Python built-in data structure `collections.deque` is a

good choice in this scenario as it offers the functionality to limit the size of the DIIS space.

You might have noticed that every time the `extrapolate` function is called, the elements of the `B` matrix, which are the dot product between two error vectors, are recomputed. If we only update one error vector at a time, most of the elements in the `B` matrix do not change. To avoid redundant computations, we can cache the dot product results. However, the input NumPy array object is not hashable, which means the `dot` function cannot be cached using the `lru_cache` function. To overcome this limitation, we create a temporary class `_HashableVector` to wrap the NumPy array object, since Python allows hashing of instances of regular classes. When adding new error vectors and trial vectors, we can wrap them under the `_HashableVector` class. Please note that older states may be recursively referenced by the `lru_cache`, which breaks the DIIS space size control provided by the `deque` object. To avoid recursive dependency on the older states, we should assign the newly created vectors to the argument `other` when calling the `dot` method.

In the `extrapolate` function, we implemented explicit for-loops to construct the `B` matrix and perform the linear combination $\sum_i c_i \mathbf{t}_i$. One might ask, why not use `einsum` to compute these quantities? The reason is that DIIS is often applied to problems involving large-sized trial vectors (such as the Coupled cluster amplitudes in Chapter 15), error vectors, or both. These vectors can be stored on disk to reduce the memory footprint. If `einsum` was used, all vectors would need to be loaded into memory, which may exceed the available memory. By using explicit for-loops, we only need to keep up to two vectors in memory at any given time.

When managing large-sized vectors during the DIIS extrapolation process, we may encounter problems related to storage and serialization. Due to limited memory capacity, it becomes necessary to store the states of a DIIS object (error vectors and trial vectors) on disk. Additionally, there may be situations where we need to resume a crashed calculation from the DIIS states. These scenarios all involve the serialization and deserialization of the DIIS object. Here, the Python built-in `pickle` module is not ideal for this purpose. This is because pickling an object tends to be slow and consumes a considerable amount of memory during the serialization and deserialization processes.

To store the DIIS states, available options include the HDF5 storage, and NumPy `memmap` objects. The advantage of using NumPy `memmap` is its simplicity. One only needs to convert the trial vector or error vector into NumPy `memmap` objects within the `_HashableVector` class, without requiring any modifications to the DIIS code itself. However, if we also consider the need to serialize the DIIS object, including all its attributes, the HDF5 storage becomes a more suitable choice. It offers the convenience of handling various types of arrays within a single file. To integrate HDF5 storage, we can modify the DIIS class and the `_HashableVector` class as below.

```
class DIIS:
    def __init__(self, filename, max_space=8):
        self.filename = filename
```

```

# keys records the active dataset names in HDF5 storage
self.keys = deque(maxlen=max_space)
# head points to the next available dataset ID
self.head = 0

@property
def max_space(self):
    return self.keys maxlen

def update(self, errvec, tvec):
    with h5py.File(self.filename, mode='a') as f:
        head, self.head = self.head, (self.head + 1) % self.max_space
        self.keys.append(head)
        if f'e{head}' in f:
            # Reuse existing datasets
            f[f'e{head}'][...] = errvec.ravel()
            f[f't{head}'][...] = tvec
        else:
            f[f'e{head}'] = errvec.ravel()
            f[f't{head}'] = tvec
        if 'metadata' in f:
            del f['metadata']
        f['metadata'] = self.dumps()
        f.flush()

    errvecs = [_HashableVector(f[f'e{key}']) for key in self.keys]
    tvecs = [_HashableVector(f[f't{key}']) for key in self.keys]
    return extrapolate(errvecs, tvecs)

def dumps(self):
    """Only JSON-serializes a few necessary attributes"""
    return json.dumps({
        'max_space': self.max_space,
        'keys': self.keys,
        'head': self.head})

@classmethod
def restore(cls, filename):
    with h5py.File(filename, mode='r') as f:
        attrs = json.loads(f['metadata'][()])
        obj = cls(filename, attrs['max_space'])
        obj.keys = attrs['keys']
        obj.head = attrs['head']
    return obj

```

```
class _HashableVector:
    def __init__(self, vec: Union[np.ndarray, h5py.Dataset]):
        self.data = vec

    @lru_cache
    def dot(self, other):
        assert self.data.ndim == other.data.ndim == 1
        return np.asarray(self.data).dot(np.asarray(other.data))

    def __mul__(self, val):
        return np.asarray(self.data) * val
```

The DIIS method requires only a few of the most recent vectors, allowing us to reuse the HDF5 datasets and reduce the size of the HDF5 file. Consequently, the `keys` attribute is introduced to monitor the actual datasets referenced by the error vectors and trial vectors. It should be noted that the `errvecs` and `tvec` lists provided to the `extrapolate` function contain only references to `h5py` Datasets. These references are only valid within the context of an opened HDF5 file. To access the data within these vectors, the `np.asarray` function must be used to load the data into memory.

In addition to the error vectors and trial vectors, we also need to store some necessary metadata, which is generated by the `dumps` method. This metadata includes the space size of the DIIS solver, and a list of keys associated with the datasets created in the HDF5 file. This information is necessary to restore the DIIS object from an HDF5 file. We employ JSON serialization to convert these elements into a string, which is then stored under the key `metadata`. With this metadata in place, reconstructing a DIIS object is straightforward, as demonstrated by the classmethod `_HashableVector.restore`.

13.5 Design of Python classes for mean-field methods

In Section 13.1, we briefly described the design of the SCF iteration program. The SCF program exposes a series of interfaces. By binding these interfaces to different functions, we can perform calculations for different mean-field models. This strategy eliminates the need for cumbersome nested if-else conditions to determine which code to execute for different mean-field methods.

In this design, it seems natural to utilize the Python class to represent a mean-field model. We can define a new mean-field model by implementing a class with specific attributes and methods. There are multiple design approaches to organizing mean-field classes. In the following discussion, we will explore four possible strategies for designing these classes, each with its own advantages and drawbacks. If you encounter similar class design challenges in applications, you can refer to the approaches discussed here for class management.

13.5.1 New mean-field classes via class inheritances

This strategy creates new mean-field classes through class inheritance and method overloading from an existing mean-field class. Many functions are closely related across mean-field models and can be reused via class inheritance to minimize code duplication. For instance, the primary distinction between the RKS DFT class and the RHF class lies in the `get_veff` and `total_energy` methods. In the RKS class, we can only rewrite the two methods, while the other methods can be inherited from the RHF class.

This approach offers clear advantages. The use of class inheritance, in terms of *Object-Oriented Programming* (OOP), is straightforward and intuitive. The hierarchy among the different mean-field methods can be seamlessly integrated into the class hierarchy structure. When implementing a new mean-field class, one simply needs to identify a suitable parent class to begin with and incorporate the relevant methods into the new class. This process does not require sophisticated designs.

However, a mean-field calculation may involve a combination of several features. For instance, one might perform an unrestricted KS-DFT calculation that incorporates a density fitting approximation, point-group symmetry, relativistic corrections, and solvation effects. If we treated every possible combination as a distinct mean-field model and created separate classes for each, we could end up with an excessively large number of classes. Additionally, within these classes, we might have to repeatedly implement very similar functions, as certain feature combinations may not have clear inheritance relationships. To introduce a new feature to mean-field methods, we would need to review and possibly extend all existing classes to incorporate this new feature. This approach is unscalable.

13.5.2 Mean-field classes via the visitor pattern

The previous class inheritance approach requires overriding the functionality of interface functions such as `get_hcore_s`, `get_veff`, and `total_energy`. A different strategy to provide these functionalities is to serve these interfaces using independent clients. When the interface functions are invoked, requests are forwarded to the underlying clients. Each interface is associated with several interchangeable client instances. By combining different interface clients, the SCF class can accommodate the functionality of various mean-field methods. In some OOP design categorizations, this strategy is referred to as the *visitor pattern*.

For example, suppose we have already implemented a general mean-field class along with the necessary feature clients. To implement a density-fitting RKS DFT method, one simply needs a factory function to assemble the density-fitting client and other necessary clients based on specific configurations.

```
class RHFVeffClient(VeffClient):
    def accept(self, mf):
        j, k = mf.get_jk()
        return j - .5 * k
```

```
class KSClient(VeffClient):
    def accept(self, mf):
        ...
        return j + vxc

class DensityFittingClient(JKClient):
    def accept(self, mf, dm):
        ...
        return j, k
    ...

class MeanField:
    hcore_client: HcoreClient = None
    jk_client: JKClient = None
    veff_client: VeffClient = None
    ...

    def get_hcore_s(self):
        return self.hcore_client.accept(self)

    def get_veff(self):
        return self.veff_client.accept(self)
    ...

    def mean_field_factory(**configuration):
        model = MeanField()
        model.set_basic_clients()
        if 'KS' in configuration and configuration['KS']:
            model.set_veff_client(KSClient())
            model.set_total_energy_client(KSEnergyClient())
        if 'DF' in configuration and configuration['DF']:
            model.set_jk_client(DensityFittingClient())
        ...
        return model

# Perform a density-fitting RKS-DFT calculation
df_rks_model = mean_field_factory(KS=True, DF=True)
scf_iter(df_rks_model)
```

This visitor pattern approach offers several advantages:

- It decouples the operations from the mean-field class, providing the flexibility to create new mean-field methods through combinations of clients.
- It eliminates ambiguity in the program execution flow. It is easy to track the functionality of the clients and understand their role in the mean-field calculations.

On the other hand, applying this approach presents some challenges. One such challenge is the coordination between different clients, as exchanging information between clients is not as straightforward as in the OOP classes. The coordination between the DIIS algorithm and the convergence status is one such example, where DIIS may be adjusted according to the convergence conditions. Another challenge arises when a calculation requires to bind multiple feature clients to the same interface. For instance, the `hcore_client` for QM/MM (Quantum Mechanical/Molecular Mechanics) and scalar relativistic correction may conflict as they both change the core Hamiltonian. If one introduces the if-else chains in the client to handle multiple mean-field features, the code in the client class would become complicated.

13.5.3 Dynamic patches to mean-field objects

In the visitor pattern approach, client attributes are introduced to customize the mean-field functionality. In fact, Python allows us to directly overwrite the methods of a mean-field object without the need for using client attributes. By overwriting a method of the mean-field object with a function that has the same signature, we can achieve effects similar to that of the visitor pattern approach. When the method is invoked, Python will proceed with the call to the overridden function.

For instance, the QM/MM method can be integrated with various HF or KS DFT models. To include QM/MM contributions, one can implement a function to override the `get_hcore` and `total_energy` functions of a mean-field object. This is illustrated in the following code example:

```
def apply_qmmm(mf, MM_charges, MM_coordinates):
    original_hcore_s = mf.get_hcore_s
    original_total_energy = mf.total_energy

    def custom_hcore_s():
        hcore, s = original_hcore_s()
        # integrals to compute < i | MM-electrostatic potential | j >
        v_mm = sum(-z * coulomb_1e_MD.get_matrix(self.gtos, Rc)
                   for z, Rc in zip(MM_charges, MM_coordinates))
        return hcore + v_mm, s

    def custom_total_energy():
        e_qm = original_total_energy()
        e_mm = ...          # 1/2 sum_{ij} Qi * Qj / |Ri-Rj|
        return e_qm + e_mm

    mf.get_hcore_s = custom_hcore_s                         # (1)
    mf.total_energy = custom_total_energy                   # (2)
```

In this code snippet, we create two functions to mimic the calls to `mf.get_hcore()` and `mf.total_energy()`. We then dynamically patch the `mf` object with these custom

functions in lines (1) and (2). The two custom functions need to invoke the original functions to compute the contributions from the QM region. It is important to avoid calling `mf.get_hcore_s()` and `mf.total_energy()` within the custom functions. These calls would lead to a recursive call to themselves, resulting in an infinite calling loop. To prevent recursive invocation, we introduced two temporary variables, `original_hcore` and `original_total_energy`, which hold references to the original methods. Within the custom functions, we utilize these variables to invoke the original methods.

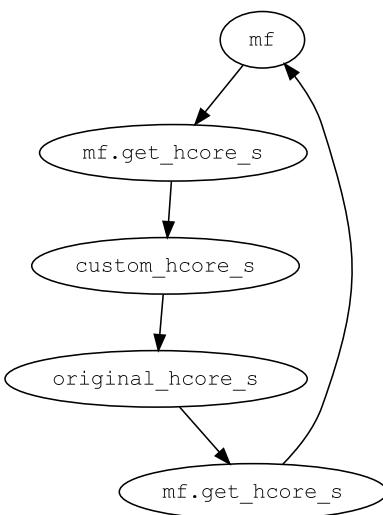
For additional mean-field features, more patching functions can be applied in a similar manner. However, this approach is only suitable for making temporary changes to an existing object. For the frequently used mean-field methods, it is recommended to use more structured and explicit approaches, like subclassing, to extend the functionality. While dynamic patching is simple to apply, we should be aware of its drawbacks:

- Dynamically patching an object can cause confusion when reading or debugging a program. When analyzing a program using Python debuggers or code editors (such as PyCharm, Vscode), the dynamic patches applied to an object are invisible until runtime.
- The dynamic patching approach may create references to the object itself, resulting in circular references. Circular references are a major cause of memory leaks in Python. The Python garbage collector is unable to automatically release the memory occupied by objects involved in circular references. In the previous QM/MM example, the variable `original_hcore_s` is bound to the `mf` object. When called by the `custom_hcore_s` function, the custom function indirectly references the `mf` object. Assigning the `custom_hcore_s` function to `mf.get_hcore` creates a circular reference, as illustrated in Fig. 13.1. Due to these circular references, all attributes of the `mf` object will persist in memory, never being released until the program terminates.

You might be wondering why we only dynamically patch the method attribute of an instance. Is it possible to directly override the methods of a class instead? In terms of syntax, Python does allow us to do this. This is essentially a technique known as monkey patching. For instance, customizing the `MeanField` class as shown below would provide us with a mean-field method that includes similar QM/MM features:

```
original_hcore_s = mf.__class__.get_hcore_s
def custom_hcore_s(self):
    hcore = original_hcore_s(self)
    ...
    return hcore + v_mm, s
mf.__class__.get_hcore_s = custom_hcore_s
```

However, this patch not only permanently changes the functionality of the class referenced by `mf.__class__`, it also affects the behavior of all classes that derive from this base class. This can lead to unintended consequences.

**FIGURE 13.1**

Circular reference in the dynamic patch method.

13.5.4 Dynamic mean-field classes

Compared to the dynamic patch to objects, using a Python class to represent a mean-field method is an intuitive approach. The primary issue with this strategy, as discussed in Section 13.5.1, is the need to define numerous classes to manage various combinations of features. Fortunately, due to the dynamic nature of Python classes, we do not have to explicitly implement a class for each possible mean-field method in advance. Instead, we can dynamically compose the derived mean-field classes for different mean-field features at runtime.

Let's continue using the QM/MM example to illustrate the dynamic class approach. Within this dynamically defined `QMMM_MeanField` class, we can override the `get_hcore` and `total_energy` methods using the standard subclassing syntax.

```

def apply_qmmm(mf, MM_charges, MM_coordinates):
    class QMMM_MeanField(mf.__class__):
        def get_hcore_s(self):
            hcore, s = super().get_hcore_s()
            ...
            return hcore + v_mm, s

        def total_energy(self):
            e_qm = super().get_energy()
            ...
            return e_qm + e_mm
  
```

```
return QMMM_MeanField()
```

Suppose we have a theoretical model which is a variant of the QM/MM method, and we wish to reuse methods from the QM/MM class. If the QM/MM dynamic class is defined within a function, it is inconvenient to inherit or reuse its functionalities. To address this issue, we can group the core functionality of the QM/MM class into a separate class, `QMMM`, within the module namespace. This class can be considered a Mixin class. By inheriting from the `QMMM` Mixin, we can develop new Mixins that introduce additional features. In the code used to generate the dynamic class, we can create a subclass that combines the `QMMM` Mixin with the mean-field class through multiple inheritance, as demonstrated in the following example.

```
class QMMM:
    def __init__(self, MM_charges, MM_coordinates):
        self.MM_charges = MM_charges
        self.MM_coordinates = MM_coordinates

    def get_hcore_s(self):
        hcore, s = super().get_hcore_s() # (1)
        ...
        return hcore + v, s

    def total_energy(self):
        e_qm = super().get_energy()
        ...
        return e_qm + e_mm

    def apply_qmmm(mf, MM_charges, MM_coordinates):
        class QMMM_MeanField(QMMM, mf.__class__): # (2)
            def __init__(self):
                super().__init__(MM_charges, MM_coordinates) # (3)
                self.__dict__.update(mf.__dict__)
        return QMMM_MeanField()
```

Please note the effects of the `super()` function in this context. The call to `super()` within the `QMMM_MeanField` class at line (3) behaves similarly to the `super()` in a regular subclass. This `super()` call is equivalent to `super(QMMM_MeanField, self)`. It searches the Method Resolution Order (MRO) for the first class that contains an `__init__` method, starting from the `QMMM_MeanField` class. The MRO for the `QMMM_MeanField` class is like

`(QMMM_MeanField, QMMM, mf.__class__, ...)`

followed by the base classes inherited by `mf.__class__`. The `super()` in line (3) finally resolves to the `QMMM` class because an `__init__` method is defined there.

In the `get_hcore_s` method of the `QMMM` class, the `super()` in line (1) resolves as `super(QMMM, self).get_hcore_s`. The starting point for this `super()` is the `QMMM` class, which might be in the middle of the MRO list, depending on how the `QMMM` class is inherited. The order of multiple inheritance is crucial in this context. To ensure the `super()` in line (1) resolves to a mean-field method, the Mixin `QMMM` must be placed before the mean-field class when creating a subclass, as shown in line (2). If `mf.__class__` is positioned before the `QMMM` class, the `super()` function within the `QMMM` class may not find an appropriate class candidate that provides the `get_hcore_s` method, leading to an `AttributeError`. Even if this statement does not raise any errors, the resolution may be incorrect. This is a critical consideration when dynamically creating classes using the design of Mixins.

The code for the dynamic class can be enhanced by using a general class composition function:

```
@lru_cache(None)
def composed_class(name, bases):
    return type(name, bases, {})

def compose(features, name):
    bases = tuple(feature.__class__ for feature in features)
    new_cls = composed_class(name, bases)
    new_obj = new_cls.__new__(new_cls)
    for feature in reversed(features):
        new_obj.__dict__.update(feature.__dict__)
    return new_obj

def apply_qmmm(mf, MM_charges, MM_coordinates):
    qmmm = QMMM(MM_charges, MM_charges)
    return compose((qmmm, mf), f'QMMM_{mf.__class__.__name__}' )
```

The `compose` function takes a list of objects as a template and uses the `type` function to dynamically create a new class. As discussed in Section 5.1.3 of Chapter 5, the `type` function requires the name, the base classes, and the class attributes to compose a new class. The newly composed class is then cached to ensure that only one class is defined for the same mean-field model.

Please note that the `__init__` method is not explicitly defined in the composed class. The instance for the newly composed class only needs to collect the attributes from each `feature` object. A different instance creation approach is employed in this example. We use the `__new__` method to create an empty instance, `new_obj`, of the newly composed class. The attributes of the input objects are then transferred to the new instance by calling `new_obj.__dict__.update()`.

How do we integrate various features in mean-field calculations using this approach? We can create independent Mixin classes for different features. Each feature class focuses on a limited range of functionalities. There is no need for nested if-else conditions for irrelevant features. When constructing a mean-field class that incor-

porates multiple feature combinations, we can apply the `compose` function to add features in a specific order. This method enables us to manage a large number of mean-field classes programmatically. For instance, the PySCF package employs this design to handle various derived mean-field classes.

This class composition approach presents its own technical challenges, including:

- This code is difficult to read, debug, and test. The new class is only available after the `compose` functions are executed. Many type errors and variable initialization issues can only be discovered during runtime, making them hard to catch in advance by the editor or tests.
- The complexities of multiple inheritance. Similar to the drawbacks of multiple inheritance, the use of multiple inheritance in this approach introduces ambiguity in method resolution order. It is difficult to determine the program's execution flow intuitively.
- The application order of different Mixin classes. The behavior of the composed class may vary depending on the order in which the features are added. For example, consider the scenario where we want to combine two Mixins: the QM/MM class and a two-component relativistic method (such as X2C [5]). The QM/MM Mixin simply adds an external MM potential to the `hcore` of the QM molecule. The X2C method involves a transformation (such as Foldy-Wouthuysen [16] transformation) to modify the Hamiltonian. If the X2C method is applied first, followed by the QM/MM Mixin, the resulting model would add the non-relativistic MM electrostatic potential to the X2C relativistic Hamiltonian. Conversely, if the application order starts with the QM/MM Mixin and then the X2C method, the relativistic correction might also be applied to the MM potential. The different application orders result in two distinct mean-field models.
- The class serialization challenge. Pickling the instance of a composed class is not feasible. This poses a challenge in the context of multiprocessing parallelization. To address this issue, one might need to use `cloudpickle` or customize serialization protocols, as discussed in Chapter 6.

13.6 Speeding up mean-field calculations

In this section, we explore three approaches to accelerate mean-field calculations:

- Optimizing the computational efficiency of ERI evaluation and the construction of J/K matrices. These are typically the most time-consuming components in mean-field methods.
- Enhancing the convergence of the SCF iteration. Several strategies can improve the convergence performance of the SCF iteration, such as the use of DIIS, level-shifting, and second-order SCF methods.
- Improving the SCF initial guess. This can potentially reduce the number of iterations required to achieve convergence.

13.6.1 Cholesky decomposition and resolution of identity methods

Apart from optimizing the integral algorithm, we can apply the Cholesky decomposition (CD) or the resolution of identity (RI) method to accelerate the ERI calculation. These methods transform the ERI calculation into tensor contraction problems, which are generally more computationally efficient.

Cholesky decomposition enables us to decompose the ERI tensor into a product of two three-dimensional tensors:

$$(ij|kl) = \sum_p T_{pij} T_{pkl}. \quad (13.38)$$

In practice, the RI method, also known as the *density fitting* (DF) approximation, is often used to decompose the ERI tensor. In this approximation, a set of auxiliary basis functions (ABF) ϕ_p is introduced to expand the tensor:

$$T_{pij} \approx \sum_q (\mathbf{M}^{-1})_{pq} (q|ij), \quad (13.39)$$

where $(q|ij)$ represents the three-center two-electron (3c2e) repulsion integrals, and the matrix \mathbf{M} is the CD of the two-center two-electron (2c2e) integral tensor

$$(p|q) = \sum_r M_{pr} M_{qr}. \quad (13.40)$$

To compute the 2c2e and 3c2e integrals, we can adapt the four-center integral program developed in Chapter 12. This can be achieved by introducing a fictitious *s*-type GTO to replace certain GTO functions in the shell-quartet. The fictitious *s*-type GTO function is configured with an exponent of 0 and is positioned at the same location as one of the GTO functions in the shell-quartet. Below is an example code to construct the 3c2e integral tensor. The 2c2e integral tensor can be constructed in a similar manner.

```
def eval_int3c2e(aux_gtos, gtos):
    ...
    for i, k, l in itertools.product(
        range(len(aux_gtos)), range(len(gtos)), range(len(gtos))):
        i0, i1, k0, k1, l0, l1 = dispatch_offsets(i, k, l)
        li, Ra = aux_gtos[i].angular_momentum, aux_gtos[i].coordinates
        lk, Rc = gtos[k].angular_momentum, gtos[k].coordinates
        ll, Rd = gtos[l].angular_momentum, gtos[l].coordinates
        # A fictitious s-type GTO on the second index
        lj, exps_j, norm_cj, Rb = 0, np.zeros(1), np.ones(1), Ra
        eri[i0:i1,k0:k1,l0:l1] = contracted_ERI(...)[ :,0]
```

Once we obtain the Cholesky-decomposed ERI (CDERI) tensor as shown in Eq. (13.39), we can proceed to construct the J and K matrices. These matrices are

computed using the `einsum` function:

```
jmat = np.einsum('pij,pkl,ji->k1', cderi, cderi, dm, optimize=True)
kmat = np.einsum('pij,pkl,jk->i1', cderi, cderi, dm, optimize=True)
```

The `optimize=True` option for `einsum` optimizes the order of tensor contractions. If the contractions were performed in the order specified in the input, it would lead to a contraction scaling of N^5 between the two `cderi` tensors. With this optimization, the tensor contraction is first carried out between the `cderi` tensor and the density matrix `dm`, generating an intermediate tensor. This intermediate tensor is then contracted with the other `cderi` tensor. As a result, the computational complexity of the J and K matrices scales as $N_{AO}^2 N_{ABF}$ and $N_{AO}^3 N_{ABF}$, respectively, where N_{AO} denotes the number of orbital basis functions, and N_{ABF} denotes the number of auxiliary basis functions.

When constructing the K-matrix, the `optimize=True` option introduces a memory overhead of two N^3 -sized intermediate tensors. The first intermediate tensor arises from the contraction

```
tmp = einsum('pij,jk->pik', cderi, dm)
```

If the `cderi` and `tmp` tensors are stored in C-order, to meet the memory contiguity requirements of matrix multiplication in BLAS library, the subsequent contraction `einsum('pik,pkl->i1', tmp, cderi)` requires the indices of the `tmp` tensor to be transposed from `pik` to `ipk`. This transposition results in the creation of another intermediate tensor.

To minimize the memory footprint and reduce the overhead of tensor transpose operations, an alternative arrangement for the tensors can be employed. For instance, when constructing the `cderi` tensor, we can place the index of auxiliary functions between the two AO indices. By using the symmetry between the two AO bases, the contractions for the K matrix

```
einsum('ipj,jk,lpk->i1', cderi, dm, cderi)
```

produce only one intermediate tensor, thereby eliminating the need for tensor transpositions.

```
tmp = einsum('ipj,jk->ipk', cderi, dm)
kamt = einsum('ipk,lpk->i1', tmp, cderi)
```

However, these optimizations exceed the capabilities of the `optimize=True` setting in `einsum`.

In mean-field methods, the density matrix is often computed by the product of occupied molecular orbitals. When computing the K matrix, we can first contract the `cderi` tensor with the occupied orbitals. The resultant tensors are then used to form the K matrix. For example, in terms of the RHF density matrix

```
mo_occupied = mo[:, :nocc]
dm = einsum('it,jt->ij', mo_occupied, mo_occupied) * 2
```

the exchange matrix can be computed as

```
tmp = einsum('jt,pij->tpi', mo_occupied, cderi)
kmat = einsum('tpi,tpj->ij', tmp, tmp) * 2
```

The MO-based contraction algorithm can significantly reduce the computational cost from $N_{AO}^3 N_{ABF}$ to $N_{AO}^2 N_{occ} N_{ABF}$, where N_{occ} represents the number of occupied orbitals. Given that the number of orbital basis functions is typically 5 to 10 times greater than the number of occupied orbitals, this optimization can lead to a speed improvement of 5 to 10 times.

The K matrix in mean-field methods is symmetric. Theoretically, this symmetry allows for the computation of only the lower triangular or the upper triangular part of the matrix. However, `einsum` and `tensordot` cannot directly exploit this symmetry. Only the NumPy `dot` function can recognize such symmetry and utilize the BLAS `*syrk` function to perform the inner or outer product of two identical arrays. Since the `tensordot` function eventually calls the NumPy `dot` function, it can utilize this symmetry if the tensor contraction can be represented as the inner or outer product of two identical arrays. More specifically, the two tensor operands involved in the `einsum` must be identical, and the contraction should not require any transposition to rearrange the indices. The contraction

```
np.einsum('tpi,tpj->ij', tmp, tmp, optimize=True)
```

in the MO-based contraction algorithm can meet this requirement, assuming that the tensor `tmp` is in C-contiguous storage. This is an additional computational benefit in the MO-based contraction algorithm. Moreover, even with a general positive-definite `dm` matrix, which is not the product of occupied orbitals, the K-matrix computation can be adapted to the MO-based contraction algorithm by performing a Cholesky decomposition on `dm`. Although this additional Cholesky decomposition step increases the computational effort, it has the potential to enhance the overall performance.

Unless the ERI tensor exhibits significant sparsity, computing J/K matrices with the CDERI tensor has a clear advantage over the construction method using analytical four-center integrals, as described in Section 13.2. However, the storage of the CDERI tensor and the data access efficiency are the challenges faced by the Cholesky decomposition method. The size of the CDERI tensor scales as N^3 with respect to the system size N . As the system size increases, the size of the tensor grows rapidly and can easily exceed the feasible memory capacity. Although there are techniques to distribute a large tensor across different nodes, the centralized scheme with disk storage remains the most common approach. If the CDERI tensor is stored on disk, loading the tensor can become a major bottleneck that limits the performance of the J/K matrix computation. In this regard, the I/O optimization techniques in Section 9.6 of Chapter 9 can be applied to optimize performance. Below are some of the technical considerations.

Data layout

The CDERI tensor may be too large to fit in memory. It is more practical to load and process the CDERI tensor in batches. A natural choice for batch processing is to divide the auxiliary dimension into smaller segments. Segmenting the auxiliary dimension offers several advantages:

- Each segment needs to be loaded only once when computing the J/K matrices, thereby minimizing the required I/O transfer.
- Permutation symmetry and sparsity are inherent among the AO pairs. We can leverage these properties to reduce the storage size.

To ensure efficient I/O access, the CDERI tensor should be stored in C-order format, and the auxiliary index should be placed in the leading dimension.

Storage format

Loading a segment of the CDERI tensor involves sequentially reading a large volume of data from a temporary file. The NumPy `memmap` is a suitable option in this scenario. Once the calculation is completed, the temporary file for the CDERI tensor can be safely deleted. We can use the `tempfile.NamedTemporaryFile` function to automatically delete the temporary files.

Chunk storage

If we only consider the scenario of reading the CDERI tensor from disk, chunk storage is not necessary as there is no need to access sub-blocks of the tensor. However, when generating this three-index tensor, it is not feasible to process the entire tensor in memory simultaneously. We need to partition the AO indices into chunks and generate the CDERI tensor for each chunk separately. The `partition_gtos` function below can divide the GTO bases into groups such that each group contains an appropriate number of GTO shells.

```
def partition_gtos(gtos, block_size):
    """Partition GT0s and return the shell offsets of each block
    ...
    def get_block_dims(dims):
        if not dims:
            return []
        s = 0
        for i, d in enumerate(dims):
            s += d
            if s >= block_size:
                break
        sub_blocks = get_block_dims(dims[i+1:])
        sub_blocks.append(i+1)
        return sub_blocks
```

```

dims = [n_cart(b.angular_momentum) for b in gtos]
block_dims = np.array(get_block_dims(dims))[:-1]
return np.append(0, np.cumsum(block_dims))

```

Since `np.memmap` does not support chunk storage, we must store different chunks in separate `mmap` files. When loading the CDERI tensor, we need to read a specific block of data from each `mmap` file and concatenate them to form the complete CDERI tensor.

Data compression

The permutation symmetry $T_{pij} = T_{pji}$ between AO indices and the sparsity within the CDERI tensor can be used to compress the CDERI tensor. The compression is specifically applied to the AO indices i and j , while the auxiliary index p remains uncompressed.

Schwarz inequality is an effective way to identify sparsity among AO pairs. By using the `schwarz_halfcond` function which we developed in Section 13.2.1, we can generate a `mask` array to filter out negligible AO pairs.

```

mask = np.tril(schwarz_halfcond(gtos) > 1e-15)
idx = np.where(mask)

```

The `idx` array tracks the significant elements within the AO dimensions.

The symmetry and sparsity used in compressing the CDERI tensor can also be applied to the 3c2e integrals. The `eval_compress_int3c2e` function below compresses the 3c2e integrals, rather than the CDERI tensor.

```

def eval_compress_int3c2e(gtos, aux_gtos, gto_pair, mask):
    offsets = gto_offsets(gtos)
    gto0, gto1 = gto_pair
    j3c = []
    for shell0, shell1 in pairwise(gto0, gto1, 10):
        i0, i1 = offsets[shell0], offsets[shell1]
        dat = eval_int3c2e_block(aux_gtos, gtos[shell0:shell1], gtos)
        j3c.append(dat[:, mask[i0:i1]])
    return np.hstack(j3c)

```

For better readability, we calculate all integrals before filtering out the significant ones. However, to improve efficiency, the calculation of insignificant integrals should be skipped.

To reconstruct the three-index CDERI tensor, we can use the following function to load the compressed data chunks and write them to the output tensor.

```

def decompress(cderi_blocks, idx, out):
    naux = cderi_blocks[0].shape[0]
    p0 = p1 = 0

```

```
for d in cderi_blocks:
    p0, p1 = p1, p1 + d.shape[1]
    idx0 = idx[0][p0:p1]
    idx1 = idx[1][p0:p1]
    out[:naux, idx0, idx1] = out[:naux, idx1, idx0] = d
return out[:naux]
```

Prefetch

When processing tensor contractions for each CDERI tensor segment, we can utilize the `iterate_with_prefetch` function from Chapter 9 to prefetch the next segment, enabling the overlap of tensor contraction and I/O operations.

Optimizing for file system cache

During the SCF iteration, the CDERI tensor is traversed multiple times. Optimizing the traversal direction can enhance the file-system cache hit rate. File systems typically employ the Least Recently Used (LRU) algorithm to decide which data to cache. If we traverse the CDERI tensor in the same directions across all scenarios, each traversal will refresh the cache. If we flip the traversal direction after each traversal of the CDERI tensor, there is a high probability that the cached data will be reused. A global boolean variable can be used to toggle the traversal direction each time the CDERI is traversed.

Taking all these factors into account, we can develop a class dedicated to computing and managing the CDERI tensor.

```
class CDERI:
    segment_max_size = 500_000_000 # 4 GB
    forward = True

    def __init__(self, gtos, aux_gtos):
        self.gtos = gtos
        self.aux_gtos = aux_gtos
        self.nao = num_functions(gtos)
        self.naux = num_functions(aux_gtos)
        self.blocksize = 200

        self.mask = np.tril(schwarz_halfcond(gtos) > 1e-15)

        c = scipy.linalg.cholesky(eval_int2c2e(aux_gtos))
        self.datafile = []
        block_size = CDERI.segment_max_size // self.nao
        for gto_pair in itertools.pairwise(
            partition_gtos(gtos, block_size)):
            int3c2e = eval_compress_int3c2e(gtos, aux_gtos, gto_pair, mask)
```

```

# Using NamedTemporaryFile for automatic deletion
chunk = np.memmap(NamedTemporaryFile(), mode='w+',
                   shape=int3c2e.shape, dtype=float)
chunk[:] = scipy.linalg.solve_triangular(c, int3c2e,
                                         lower=True)
self.datafile.append(chunk)

def __iter__(self):                                     # (1)
    blocks = pairwise(0, self.naux, self.blocksize)
    if not CDERI.forward:
        blocks = reversed(list(blocks))
    # Flip the iteration direction to maximize FS cache utilization
    CDERI.forward = not CDERI.forward

    # buf0 and buf1 are reused to reduce memory allocation overhead
    buf0, buf1 = np.zeros((2, self.blocksize, self.nao, self.nao))
    def unpack(block):
        start, end = block
        dat = [d[start:end] for d in self.datafile]
        return decompress(dat, self.mask, buf0)

    for _, data in iterate_with_prefetch(blocks):
        buf0, buf1 = buf1, buf0
        yield data

def pairwise(start, stop, step):
    return itertools.pairwise(np.append(range(start, stop, step), stop))

```

The `__iter__` method in line (1) defines an iterator for this CDERI class. This iterator enables us to compute the J/K matrices with straightforward tensor contraction code.

```

cderi = CDERI(gtos, aux_gtos)
nao = num_functions(gtos)
jmat, kmat = np.zeros((nao,nao)), np.zeros((nao,nao))
for sub_cderi in iter(cderi):
    tmp = einsum('jt,pij->tpi', mo_occupied, sub_cderi)
    kmat += einsum('tpi,tpj->ij', tmp, tmp) * 2
    jmat += einsum('pij,ji.pkl->pkl', sub_cderi, dm, sub_cderi)

```

13.6.2 Improving convergence using DIIS

DIIS leverages a collection of error vectors to stabilize the solution vector. There are several variants of DIIS, such as Pulay's DIIS [17], ADIIS (augmented Roothaan-

Hall DIIS) [18], and EDIIS (energy DIIS) [19]. There are also many possible choices for the error vectors and solution vectors. The error vector can be the orbital gradients, the change in the Fock matrix between iterations, the change in the density matrix between iterations, and so forth. The solution vector can include the density matrix and the Fock matrix. Here, we adopt Pulay’s original work, which employs the error term **FDS – SDF** (corresponding to the orbital gradients) to stabilize the Fock matrix. When developing this variant of DIIS, several technical aspects must be considered:

- DIIS extrapolation can be applied at different stages of the SCF iterations, depending on the chosen error vectors and solution vectors. To stabilize the Fock matrix, it is suitable to incorporate DIIS extrapolation within the `get_fock` function. Additionally, when creating a mean-field instance, we can initialize a DIIS instance as an attribute of the mean-field object.
- The error vector **FDS – SDF** should be represented in orthonormal bases. The orthonormal bases can be formed using the eigenvectors of the overlap matrix or the eigenvectors from the first SCF iteration. Without using orthonormal bases, one might encounter numerical stability issues, particularly when the orbital bases approach linear dependence.
- As the SCF calculation approaches convergence, the error vectors may approach zero. This can lead to singularity issues when solving the linear equations in the DIIS extrapolation function. To address this, several strategies can be employed, such as discarding the states generated in the previous iterations, damping the diagonal elements [20], or adopting different error minimization schemes [21].

To implement the CDIIS class for mean-fields, we can inherit it from the DIIS class developed in Section 13.4. In addition to the existing attributes, we need to include orthonormal bases as an additional attribute. These orthonormal bases should be stored in the DIIS serialization data and restored during deserialization. To address the issue of ill-conditioning when solving linear equations in DIIS extrapolation, we progressively reduce the DIIS space until the condition number falls below the desired threshold of 10^{12} .

```
class CDIIS(DIIS):
    def __init__(self, filename, max_space=8):
        super().__init__(filename, max_space)
        self.c = None # The orthonormal basis for error vectors

    def update(self, f, d, s):
        with h5py.File(self.filename, mode='a') as h5f:
            if self.c is None:
                _, self.c = scipy.linalg.eigh(s)
                h5f['c'] = self.c
            errvec = f.dot(d).dot(s)
            errvec = errvec - errvec.T # FDS - SDF
            # Transforms error vector to orthonormal basis
```

```

errvec = self.c.T.dot(errvec).dot(self.c).ravel()

head, self.head = self.head, (self.head + 1) % self.max_space
self.keys.append(head)
if f'e{head}' in h5f:
    # Reuse existing datasets
    h5f[f'e{head}'][...] = errvec
    h5f[f't{head}'][...] = f
else:
    h5f[f'e{head}'] = errvec
    h5f[f't{head}'] = f
if 'metadata' in h5f:
    del h5f['metadata']
h5f['metadata'] = self.dumps()
h5f.flush()

errvecs = [_HashableVector(h5f[f'e{k}']) for k in self.keys]
space = len(self.keys)
B = np.zeros((space+1, space+1))
B[-1,:-1] = B[:,-1,-1] = 1.
for i, e1 in enumerate(errvecs):
    for j, e2 in enumerate(errvecs):
        if j < i:
            continue
        B[i,j] = B[j,i] = e1.dot(e2)
while np.linalg.cond(B) > 1e12:
    B = B[1:,1:]
    self.keys.pop(0)

g = np.zeros(len(self.keys)+1)
g[-1] = 1
c = scipy.linalg.solve(B, g, assume_a='sym')[:-1]

sol = np.zeros_like(f)
for key, x in zip(self.keys, c):
    sol += h5f[f't{key}'][()] * x
return sol

@classmethod
def restore(cls, filename):
    with h5py.File(filename, mode='r') as f:
        attrs = json.loads(f['metadata'][()])
        obj = cls(filename)
        obj.keys = deque(attrs['keys'], maxlen=attrs['max_space'])

```

```

    obj.head = attrs['head']
    obj.c = f['c'][()]
    return obj

```

13.6.3 Improving initial guess

The initial guess greatly impacts the convergence and efficiency of the SCF program. A well-chosen initial guess can significantly enhance the speed of convergence. In the previous SCF example code, the initial guess is derived by solving the core Hamiltonian. However, the solution derived from such a single-electron problem is not an ideal initial guess. A good initial guess should be close to the solution of the problem we are attempting to solve.

Superposition of atomic results

We can represent an SCF initial guess using a block-diagonal matrix formed by the superposition of atomic densities. These atomic densities can be derived either from atomic calculations using the same basis sets or from an atomic calculation database. For instance, the ANO-RCC base set is a candidate of such databases, which comprises the natural orbitals from atomic calculations. By specifying the ground state atomic occupancies for the ANO-RCC bases, we can approximate atomic density matrices $\gamma_A, \gamma_B, \dots$. Subsequently, we can compute the overlap matrix S_{TA} between the target bases (T) and the reference bases (A) to transform the block-diagonal density matrix.

$$S_T^{-1} S_{TA} \begin{pmatrix} \gamma_A & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \gamma_B & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \ddots & \vdots \\ \vdots & \vdots & & \ddots \end{pmatrix} S_{AT} S_T^{-1} \quad (13.41)$$

Empirical models, such as the Hückel model and the SAP (superposition of atomic potentials) model [22], can also be employed to generate the initial guess by using the superposition of atomic data.

Initial guess from similar calculations

In tasks such as potential energy surface scans or geometry optimizations, the wave function from the previous step is often very close to that of the current step. We can utilize the density matrix from the previous step as the initial guess for the current step. Alternatively, we can perform a mean-field calculation with a smaller basis set and then project the resulting density matrix onto a larger basis set.

If the molecular geometry of the previous calculation differs from the geometry of the current calculation, do we need to adjust the initial guess with respect to the changes in geometry? It is not necessary in most scenarios. If the basis sets are identical, the density matrix from the previous calculation is a good initial guess, even if

it might not correspond to the correct number of electrons in the new geometry. If the two calculations employ different basis sets, the projection between the two basis sets should be performed under the same geometry. The resulting density matrix can then be used directly in the new geometry.

In some initial guess methods, we may only obtain the density matrix rather than the orbitals. To adapt the density matrix initial guess to the wave function object required by the SCF program, we can create a pseudo `SCFWavefunction` class that only provides the density matrix.

```
class InitialGuess(SCFWavefunction):
    def __init__(self, dm):
        self.density_matrices = dm

    @property
    def orbitals(self):
        raise RuntimeError('Orbitals not available in initial guess')
```

To ensure that the orbital information is not accidentally used, we override the `orbitals` attribute of this pseudo `SCFWavefunction` object.

Next, let's consider the process of restarting a calculation, an important aspect that is closely related to the treatment of the initial guess. A calculation may terminate early or fail to successfully converge. In such situations, we can use the results of these incomplete calculations as the initial guess to continue the calculation.

To enable this feature, it is necessary to save the state of the wave function during the SCF process. We can serialize the wave function object and save it in a checkpoint file within the `make_wfn` function. To continue a terminated calculation, we can deserialize the checkpoint file and pass it as the initial guess to the `scf_iter` function.

Using the checkpoint file, an unfinished calculation can be fully restored. Since the SCF iteration depends on DIIS, fully restoring an SCF calculation requires restoring the state of DIIS as well. Restoring a DIIS object was discussed in Chapter 13.4. Taking the RHF method as an example, the restore method can be implemented as follows:

```
class RHF:
    def __init__(self, mol, gtos):
        self.mol = mol
        self.gtos = gtos
        self.threshold = 1e-6
        self.chkfile = tempfile.mktemp()
        diisfile = tempfile.mktemp()
        print(f'Checkpoint file is {self.chkfile}.')
        print(f'DIIS is saved in {diisfile}')
        self.diis = CDIIS(diisfile)

    ...
```

```

def make_wfn(self, orbitals, energies):
    wfn = RestrictedCloseShell(self, orbitals, energies)
    with open(self.chkfile, 'wb') as f:
        pickle.dump({
            'wfn': wfn, 'mol': self.mol, 'gtos': self.gtos
        }, f)
    return wfn

@classmethod
def restore(cls, chkfile, diisfile=None):
    with open(chkfile, 'rb') as f:
        attrs = pickle.load(f)
    obj = cls(attrs['mol'], attrs['gtos'])
    obj.wfn = attrs['wfn']
    if diisfile is not None:
        obj.diis = CDIIS.restore(diisfile)
    return obj

# A fully restart from a previous calculation.
model = RHF.restore(chkfile='/tmp/tmpw5t7yfgo',
                     diisfile='/tmp/tmpc7_dzxad')
scf_iter(model, model.wfn)

```

Summary

Implementing a basic self-consistent field (SCF) program, such as a restricted Hartree-Fock program, is straightforward. However, developing an SCF program that is efficient, straightforward to maintain, and easy to extend involves various technical challenges.

An efficient SCF program must employ effective methods for calculating Coulomb and exchange matrices. We investigated the algorithm that utilizes the eight-fold permutation symmetry and the sparsity of electron-repulsion integrals to accelerate these calculations. By applying the Cholesky decomposition to factorize the electron repulsion integrals, we transform the computation of Coulomb and exchange matrices into a tensor contraction problem. The Cholesky decomposition can be approximated using the density fitting method. We presented several optimization tips for the density fitting approximation, focusing on enhancing storage efficiency, compressing tensors, and optimizing both memory usage and computational costs associated with tensor operations. In density functional theory, the SCF program requires the computation of the exchange-correlation (XC) matrix. This matrix is

typically evaluated through numerical integration. We explored a simplified method that transforms the numerical integration process into a tensor contraction program.

Beyond the efficiency of integral computations, the convergence of the SCF program also critically impacts performance. We discussed the use of DIIS extrapolation to accelerate convergence, the construction of initial guesses from atomic calculations, and strategies for restarting a failed SCF calculation.

Considering the numerous variants of mean-field methods, which differ only slightly from one another, we explored four design strategies to organize mean-field models effectively. These strategies are: simple class inheritance, the visitor pattern, dynamic patches to mean-field objects, and the use of dynamic classes.

References

- [1] A. Szabo, N.S. Ostlund, Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory, 1st edition, Dover Publications, Inc., Mineola, 1996.
- [2] R.G. Parr, Y. Weitao, Density-Functional Theory of Atoms and Molecules, Oxford University Press, USA, 1994, <http://www.amazon.com/Density-Functional-Molecules-International-Monographs-Chemistry/dp/0195092767/>.
- [3] T. Saue, Relativistic Hamiltonians for chemistry: a primer, *ChemPhysChem* 12 (17) (2011) 3077–3094, <https://doi.org/10.1002/cphc.201100682>, <https://chemistry-europe.onlinelibrary.wiley.com/doi/pdf/10.1002/cphc.201100682>, <https://chemistry-europe.onlinelibrary.wiley.com/doi/abs/10.1002/cphc.201100682>.
- [4] Z. Li, Y. Xiao, W. Liu, On the spin separation of algebraic two-component relativistic Hamiltonians, *Journal of Chemical Physics* 137 (15) (2012) 154114, <https://doi.org/10.1063/1.4758987>, https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4758987/13274957/154114_1_online.pdf.
- [5] W. Liu, D. Peng, Exact two-component Hamiltonians revisited, *Journal of Chemical Physics* 131 (3) (2009) 031104, <https://doi.org/10.1063/1.3159445>, https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.3159445/15671680/031104_1_online.pdf.
- [6] M. Reiher, Relativistic Douglas–Kroll–Hess theory, *WIREs Computational Molecular Science* 2 (1) (2012) 139–149, <https://doi.org/10.1002/wcms.67>, <https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.67>, <https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wcms.67>.
- [7] B.I. Dunlap, Robust and variational fitting, *Physical Chemistry Chemical Physics* 2 (2000) 2113–2116, <https://doi.org/10.1039/B000027M>.
- [8] F. Weigend, A fully direct RI-HF algorithm: implementation, optimised auxiliary basis sets, demonstration of accuracy and efficiency, *Physical Chemistry Chemical Physics* 4 (2002) 4285–4291, <https://doi.org/10.1039/B204199P>.
- [9] F. Neese, F. Wennmohs, A. Hansen, U. Becker, Efficient, approximate and parallel Hartree–Fock and hybrid DFT calculations. A ‘chain-of-spheres’ algorithm for the Hartree–Fock exchange, in: *Moving Frontiers in Quantum Chemistry*, *Chemical Physics* 356 (1) (2009) 98–109, <https://doi.org/10.1016/j.chemphys.2008.10.036>, <https://www.sciencedirect.com/science/article/pii/S0301010408005089>.
- [10] H. Lin, D.G. Truhlar, QM/MM: what have we learned, where are we, and where do we go from here?, *Theoretical Chemistry Accounts* 117 (2) (2007) 185–199, <https://doi.org/10.1007/s00214-006-0143-z>.

- [11] E. Brunk, U. Rothlisberger, Mixed quantum mechanical/molecular mechanical molecular dynamics simulations of biological systems in ground and electronically excited states, <https://doi.org/10.1021/cr500628b>, <https://infoscience.epfl.ch/handle/20500.14299/113676>, 2015.
- [12] C.A. White, M. Head-Gordon, A J matrix engine for density functional theory calculations, *Journal of Chemical Physics* 104 (7) (1996) 2620–2629, <https://doi.org/10.1063/1.470986>, https://pubs.aip.org/aip/jcp/article-pdf/104/7/2620/19069983/2620_1_online.pdf.
- [13] S. Lehtola, C. Steigemann, M.J. Oliveira, M.A. Marques, Recent developments in LIBXC — a comprehensive library of functionals for density functional theory, *SoftwareX* 7 (2018) 1–5, <https://doi.org/10.1016/j.softx.2017.11.002>, <https://www.sciencedirect.com/science/article/pii/S2352711017300602>.
- [14] U. Ekström, L. Visscher, R. Bast, A.J. Thorvaldsen, K. Ruud, Arbitrary-order density functional response theory from automatic differentiation, *Journal of Chemical Theory and Computation* 6 (7) (2010) 1971–1980, <https://doi.org/10.1021/ct100117s>, <http://pubs.acs.org/doi/pdf/10.1021/ct100117s>, <http://pubs.acs.org/doi/abs/10.1021/ct100117s>.
- [15] A.D. Becke, A multicenter numerical integration scheme for polyatomic molecules, *Journal of Chemical Physics* 88 (4) (1988) 2547–2553, <https://doi.org/10.1063/1.454033>, https://pubs.aip.org/aip/jcp/article-pdf/88/4/2547/18968948/2547_1_online.pdf.
- [16] A.J. Silenko, General properties of the Foldy-Wouthuysen transformation and applicability of the corrected original Foldy-Wouthuysen method, *Physical Review A* 93 (2016) 022108, <https://doi.org/10.1103/PhysRevA.93.022108>, <https://link.aps.org/doi/10.1103/PhysRevA.93.022108>.
- [17] P. Pulay, Improved SCF convergence acceleration, *Journal of Computational Chemistry* 3 (4) (1982) 556–560, <https://doi.org/10.1002/jcc.540030413>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.540030413>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.540030413>.
- [18] X. Hu, W. Yang, Accelerating self-consistent field convergence with the augmented Roothaan–Hall energy function, *Journal of Chemical Physics* 132 (5) (2010) 054109, <https://doi.org/10.1063/1.3304922>, <http://scitation.aip.org/content/aip/journal/jcp/132/5/10.1063/1.3304922>.
- [19] K.N. Kudin, G.E. Scuseria, E. Cancès, A black-box self-consistent field convergence algorithm: one step closer, *Journal of Chemical Physics* 116 (19) (2002) 8255–8261, <https://doi.org/10.1063/1.1470195>, <http://scitation.aip.org/content/aip/journal/jcp/116/19/10.1063/1.1470195>.
- [20] T.P. Hamilton, P. Pulay, Direct inversion in the iterative subspace (DIIS) optimization of open-shell, excited-state, and small multiconfiguration SCF wave functions, *Journal of Chemical Physics* 84 (10) (1986) 5728–5734, <https://doi.org/10.1063/1.449880>, https://pubs.aip.org/aip/jcp/article-pdf/84/10/5728/18959418/5728_1_online.pdf.
- [21] H. Sellers, The C2-DIIS convergence acceleration algorithm, *International Journal of Quantum Chemistry* 45 (1) (1993) 31–41, <https://doi.org/10.1002/qua.560450106>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.560450106>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.560450106>.
- [22] S. Lehtola, L. Visscher, E. Engel, Efficient implementation of the superposition of atomic potentials initial guess for electronic structure calculations in Gaussian basis sets, *Journal of Chemical Physics* 152 (14) (2020) 144105, <https://doi.org/10.1063/5.0004046>, https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0004046/13478079/144105_1_online.pdf.

Post Hartree-Fock I: full configuration interaction

14

Although the Hartree-Fock method accounts for the main portion of molecular energy, it is insufficient for studying the chemical properties and characteristics of chemical reactions. Chemistry is driven by the effects stemming from electron correlation, which is a core aspect of quantum chemistry. Numerous post-Hartree-Fock (post-HF) methods have been developed to characterize the effects of electron correlation.

Developing programs for post-HF methods involves various technical challenges. Post-HF methods introduce many-body wavefunctions, which typically involve a large amount of parameters. Manipulating the wavefunction can lead to intensive floating-point operations. The intermediate variables generated during the computation process often consume substantial amounts of memory space, and sometimes require extensive disk space for storage. This results in significant data transfer between the CPU core, memory, and disk. The structure of data storage and the memory management strategy can greatly impact the efficiency of post-HF method programs. How to efficiently utilize the floating-point computation capabilities of the computer? How to effectively properly design data structure to minimize data transfer overhead? These are critical aspects in the development of post-HF methods.

Configuration Interaction (CI) is an effective post-HF method for describing the effects of electron correlation. Among its variants, Full Configuration Interaction (FCI) is the most comprehensive form, where all possible electron configurations (or determinants) generated from a given set of orbitals are considered in the wavefunction. This method offers the most accurate description of electron correlation. However, it is computationally extremely demanding due to the exponential increase in the size of the wavefunction parameters as the system size grows.

From a programming perspective, FCI presents many typical technical challenges encountered in post-HF methods. It is a typical example to demonstrate how to utilize various techniques to optimize programs.

- The number of parameters in the FCI wavefunction is enormous. It is only feasible to solve the FCI wavefunction using algorithms for sparse matrices.
- Applying the Hamiltonian to the FCI wavefunction requires a significant number of floating-point operations.
- It is beneficial to accelerate FCI programs with parallel computation. The efficiency of parallel computation heavily depends on the program design and implementation.

- The FCI method generates extremely large intermediate variables, which poses a challenge for memory management efficiency.

In this chapter, we will employ the techniques of performance analysis and code optimization discussed in Chapter 9. Additionally, we will demonstrate how to effectively parallelize dependent computation tasks using the pipeline executor developed in Chapter 10.

14.1 Theory of full configuration interaction

In the Configuration Interaction (CI) method, the electronic wave function is expressed as a linear combination of Slater determinants, which are also referred to as configurations:

$$|\Psi\rangle = \sum_I C_I |\Phi_I\rangle. \quad (14.1)$$

A Slater determinant $|\Phi\rangle$ represents the occupation of electrons in specific orbitals. The expansion coefficients \mathbf{C} are obtained by variational optimization of the CI energy, as described by the eigenvalue equation:

$$\mathbf{H}\mathbf{C} = \mathbf{C}E, \quad (14.2)$$

where the element of the Hamiltonian matrix is

$$H_{IJ} = \langle \Phi_I | \hat{H} | \Phi_J \rangle. \quad (14.3)$$

The FCI method requires the inclusion of all determinants in the wave function expansion. In non-relativistic calculations, the numbers of electrons in α spin and β spin are conserved. For a system with n spatial orbitals and $k_\alpha + k_\beta$ electrons, the FCI wave function comprises $\binom{n}{k_\alpha} \binom{n}{k_\beta}$ determinants. Each determinant can be expressed as the product of configurations for electrons with α and β spins:

$$|\Phi_I\rangle = |I_\alpha I_\beta\rangle = \hat{I}_\alpha \hat{I}_\beta |\text{vacuum}\rangle. \quad (14.4)$$

I_α and I_β represent the configurations of creation operators for α and β electrons, respectively:

$$I_\alpha = \hat{a}_{i_1\alpha}^\dagger \hat{a}_{i_2\alpha}^\dagger \cdots, \quad (14.5)$$

$$I_\beta = \hat{a}_{i_1\beta}^\dagger \hat{a}_{i_2\beta}^\dagger \cdots. \quad (14.6)$$

In many contexts, I_α and I_β are referred to as the α and β strings, respectively.

Since the number of determinants increases exponentially, the FCI method is only applicable for relatively small systems. Consider a half-filled wave function with 20 spatial orbitals, comprising 10 alpha electrons and 10 beta electrons. The FCI

expansion would then include approximately $\binom{20}{10}^2 \approx 34$ billion determinants. Storing this wave function in double precision would require approximately 250 GB of space. This requirement approaches the practical upper limit of the FCI method.

Nevertheless, the implementation of the FCI program is a good example of how to apply Python programming techniques to handle practical quantum chemistry problems. The development of the FCI program involves several practical techniques, such as:

- How to compute partial eigenvalues of a large matrix without fully constructing the entire matrix.
- How to maximize floating-point computations by reformulating the problem as a tensor contraction problem.
- How to optimize memory efficiency and performance.
- How to utilize parallelization to accelerate computations.

14.2 The string representation

To represent the α and β strings of a determinant, we utilize the `String` class to store the indices of their creation operators. The `String` class is similar to the `Determinant` class in Section 9.7.3 of Chapter 9. Additionally, we implement the `__hash__` and `__eq__` methods in the `String` class. These methods enable `String` objects to be used as dictionary keys. This functionality will be utilized later in this chapter to simplify the computation of addresses.

```
class String:  
    def __init__(self, occupied_orbitals: List):  
        self.occupied_orbitals = set(occupied_orbitals)  
  
    def __repr__(self):  
        return f'Det({self.occupied_orbitals})'  
  
    @classmethod  
    def vacuum(cls):  
        return cls(set())  
  
    @classmethod  
    def fully_occupied(cls, n):  
        return cls(set(range(n)))  
  
    def add_occupancy(self, orbital_id):  
        assert orbital_id not in self.occupied_orbitals  
        return String(self.occupied_orbitals.union([orbital_id]))  
  
    def __hash__(self):
```

```

        return hash(tuple(self.occupied_orbitals))

    def __eq__(self, other):
        return self.occupied_orbitals == other.occupied_orbitals

    def annihilate(self, orbital_id):
        """Apply an annihilation operator. Returns the sign and a new
        string."""
        if orbital_id not in self.occupied_orbitals:
            return 0, String.vacuum()
        sign = (-1) ** sum(i > orbital_id for i in self.occupied_orbitals)
        return sign, String(self.occupied_orbitals.difference([orbital_id]))
)

    def create(self, orbital_id):
        """Apply a creation operator. Returns the sign and a new string."""
        if orbital_id in self.occupied_orbitals:
            return 0., String.vacuum()
        sign = (-1) ** sum(i > orbital_id for i in self.occupied_orbitals)
        return sign, String(self.occupied_orbitals.union([orbital_id]))
)

```

When applying second quantized operators to a `String` object, it can produce either a positive or a negative sign. In the `String` representation, we adopt the convention that operators are applied from orbitals with lower indices to those with higher indices, such as

$$\hat{a}_{4\alpha}^\dagger \hat{a}_{3\alpha}^\dagger \hat{a}_{1\alpha}^\dagger |\text{vacuum}\rangle.$$

To determine the sign, we simply count the number of operators whose indices are greater than the index of the target orbital. This count represents the number of permutations, or equivalently, the number of sign flips, required by the application of a second quantization operator. The necessary sign flips are handled in the `annihilate` and `create` methods.

Given a specified number of orbitals and occupancies, we can programmatically generate all possible CI strings. This is the functionality of the `make_strings` function below, which is essentially identical to the `dets` function introduced in Chapter 9.

```

@lru_cache(200)
def make_strings(norb: int, noccupied: int):
    assert norb >= noccupied
    if norb == 0:
        return [String.vacuum()]
    elif noccupied == 0:
        return [String.vacuum()]
    elif norb == noccupied:
        return [String.fully_occupied(norb)]
)

```

```

    return (make_strings(norb-1, nooccupied) +
        [s.add_occupancy(norb-1)
            for s in make_strings(norb-1, nooccupied-1)])

```

14.3 Davidson diagonalization

Typically, the FCI method only requires the calculation of the lowest few eigenstates of the Hamiltonian, corresponding to the ground state wavefunction and the low-lying excited states. In quantum chemistry, the Davidson diagonalization algorithm [1,2] is used for computing the lowest eigenvalues and eigenvectors.

Given a set of basis vectors $\{v_0, v_1, \dots, v_{n-1}\}$, we can represent the matrix A and the eigenvector x as follows:

$$Ax = \lambda x, \quad (14.7)$$

$$h_{ij} = v_i^\dagger A v_j, \quad (14.8)$$

$$x = \sum_i c_i v_i. \quad (14.9)$$

The eigenvalue problem is reduced to a smaller problem that can be efficiently solved:

$$\mathbf{hc} = e\mathbf{c}. \quad (14.10)$$

For an approximate eigenvector x , a residual can be constructed for the lowest eigenvalue e_0 :

$$\sigma = Ax - e_0 x = \sum_i (Av_i - e_0 v_i) c_{i0}. \quad (14.11)$$

To minimize the computational expense of the matrix-vector multiplication, the residual is constructed using the precomputed product Av_i . This residual is then used to generate a new basis vector:

$$v_n = \frac{\sigma}{\theta - \text{diag}(A)}, \quad (14.12)$$

where $\text{diag}(A)$ denotes the diagonal elements of the matrix A .

The Davidson iteration begins with an initial guess, v_0 , and employs the above steps to refine the subspace basis vectors until the residual, as defined in (14.11), is smaller than the required convergence threshold. Below is a simplified implementation for the lowest eigenvalue of a real symmetric matrix:

```

def davidson(A, A_diag, x0=None, tol=1e-5, maxiter=50, space=15):
    def precond(r, e):
        return r / (A_diag - e)

```

```

# The initial guess
if x0 is None:
    x0 = 1. / (A_diag - A_diag.argmin() + 1e-2)
x0 = x0 / np.linalg.norm(x0)

vs = []
hv = []
h = np.empty((space, space))

for cycle in range(maxiter):
    vs.append(x0)
    hv.append(A(x0))
    n = len(vs)
    for i in range(n):
        h[i,n-1] = np.dot(vs[i], hv[n-1])
        h[n-1,i] = h[i,n-1].conj()

    e, c = np.linalg.eigh(h[:n,:n])
    e0 = e[0]
    c0 = c[:,0]

    x = np.zeros_like(x0)
    for i, ci in enumerate(c0):
        x += vs[i] * ci
    residual = -e0 * x
    for i, ci in enumerate(c0):
        residual += hv[i] * ci
    norm = np.linalg.norm(residual)
    print(f'davidson {cycle=} {e0=} residual={norm:.5g}')
    if norm < tol:
        break

    if len(vs) >= space:
        # Restart the calculation, the last x is used as initial guess
        x0, vs, hv = x, [], []
        continue

    x0 = precond(residual, e0)
    # orthogonalize the basis
    for v in vs:
        x0 -= v * np.dot(v, x0)
    norm = np.linalg.norm(x0)
    assert norm < 1e-10, 'Linearly dependent basis vectors'

```

```

        x0 /= norm
    else:
        raise RuntimeError('davidson not converged')

    return e0, x

```

During the iteration, the new basis vector must be orthogonalized against the existing basis vectors before it is added to the subspace. Without this orthogonalization step, the subspace eigenvalue problem would become a general eigenvalue problem,

$$Hc = Sce. \quad (14.13)$$

Although the general eigenvalue problem with the non-orthogonal basis vectors should theoretically yield the same eigenvalues as the original problem, practical experience has shown that non-orthogonal basis vectors tend to introduce more errors and numerical issues.

Please note the parameter `space`, which limits the size of the subspace. When the subspace size reaches this limit, the eigenvalue solver clears the subspace and restarts the diagonalization iteration. This strategy not only reduces the storage requirements for the basis vectors and $A(x)$ vectors but also helps in controlling numerical errors.

The residual vector is the difference between two large-valued vectors. During the diagonalization iteration, the residuals generally decrease in magnitude. Consequently, the iteration process can accumulate noticeable numerical errors. By restarting the iteration every 10 to 15 iterations, the program can effectively reduce these numerical errors [3]. This restarting mechanism is crucial for maintaining accuracy in quantum chemistry applications. The eigenvalue solvers provided by the `scipy.sparse.linalg` module do not offer the restarting feature. This is one reason why we need to use a custom diagonalization program in quantum chemistry software.

To perform the Davidson diagonalization for FCI Hamiltonian, we need a function to compute the matrix-vector product between the Hamiltonian and the CI coefficients $\sigma = HC$:

```

def matvec(x)
    return compute_hc(h1, eri, x, norb, nelec_a, nelec_b)

```

The argument `h1` is the one-electron integral matrix and `eri` is the two-electron integral tensor. Additional parameters include the number of orbitals `norb`, the number of operators in the α string (`nelec_a`), and in the β string (`nelec_b`). `compute_hc` is the most computationally intensive part in FCI method. It can be evaluated using an efficient high-dimensional tensor contraction algorithm, which will be discussed in Section 14.4.

Another argument required by the Davidson algorithm is the diagonal elements of the Hamiltonian. These elements can be computed using the Slater-Condon rules [4].

$$\langle I_\alpha I_\beta | H | I_\alpha I_\beta \rangle = \sum_{i \in \{I_\alpha, I_\beta\}} h_{ii} + \frac{1}{2} \sum_{ij \in \{I_\alpha, I_\beta\}} (ii|jj) - \frac{1}{2} \sum_{ij \in I_\alpha} (ij|ji) - \frac{1}{2} \sum_{ij \in I_\beta} (ij|ji). \quad (14.14)$$

Thanks to the fancy indexing feature of NumPy, the program to compute the diagonal elements is straightforward to implement.

```
def make_hdiag(h1, eri, norb, nelec_a, nelec_b):
    occs_a = make_strings(norb, nelec_a).occupied_orbitals
    occs_b = make_strings(norb, nelec_b).occupied_orbitals
    eri = ao2mo.restore(1, eri, norb)
    diagj = np.einsum('iijj->ij', eri)
    diagh = np.einsum('ijji->ij', eri)
    hdiag = []
    for aocc in occs_a:
        for bocc in occs_b:
            e1 = h1[aocc,aocc].sum() + h1[bocc,bocc].sum()
            e2 =(diagj[aocc][:,aocc].sum() + diagj[aocc][:,bocc].sum() +
                  diagj[bocc][:,aocc].sum() + diagj[bocc][:,bocc].sum() -
                  diagh[aocc][:,aocc].sum() - diagh[bocc][:,bocc].sum())
            hdiag.append(e1 + e2*.5)
    return np.array(hdiag)
```

The `FCI_solve` function for the ground state FCI wavefunction can be implemented as follows:

```
def FCI_solve(h1, eri, norb, nelec_a, nelec_b, x0=None):
    # matrix-vector product
    def matvec(x):
        hc = compute_hc(h1, eri, x, norb, nelec_a, nelec_b)
        return hc.ravel()

    # Diagonal elements
    h_diag = make_hdiag(h1, eri, norb, nelec_a, nelec_b)

    e, wfn = davidson(matvec, h_diag, x0=x0)
    return e, wfn
```

In practice, the Davidson diagonalization program is often significantly more complex than the example provided above. Additional technical details that must be

considered. We will not implement code examples for each of these technical aspects within this book. Some of the key technical points are briefly discussed below.

Symmetry of the initial guess

Davidson diagonalization may not always yield the lowest eigenstate, especially in systems with spin symmetry or high-order point group symmetry. The diagonalization process can only produce the lowest eigenstates for a particular symmetry subspace, which is associated with the symmetry of the initial guess. The Hamiltonian matrix is often a totally symmetric operator. If the initial guess is confined to a particular symmetry subspace, the basis generation code can only produce basis vectors that are restricted to that subspace. This is one of the most common issues encountered in quantum chemistry applications of Davidson diagonalization.

Therefore, constructing an appropriate initial guess of the desired symmetry is crucial in the Davidson iteration solver. If the symmetry of the lowest eigenstate is unknown, a practical strategy is to solve for multiple eigenvalues of the problem. This approach increases the likelihood of capturing the correct symmetry subspace, thereby enhancing the effectiveness of the Davidson diagonalization program in finding the lowest eigenstates.

Singularity in the preconditioner

The denominator of the preconditioner in Eq. (14.12) consists of the difference between the diagonal elements of matrix A and the approximate eigenvalues. This difference can become very small (less than 10^{-6}), which may lead to singularity or numerical errors in the preconditioner.

It is important to ensure that the basis vectors generated by the preconditioner are appropriately scaled. Large numerical values in the basis vectors can silently introduce numerical instability. To avoid numerical issues, it is essential to filter out singular (or excessively large) elements in the preconditioner.

Multiple eigenstates

Davidson diagonalization can be employed to compute multiple eigenstates at the lower end of the eigenvalue spectrum. During the Davidson iteration, multiple eigenvector candidates are utilized to generate new basis vectors. However, these eigenvectors may not converge at the same rate. It is common that some eigenstates have been converged, while others are not. If the program continues to generate basis vectors using the converged eigenvectors, it can introduce errors into the basis space and potentially cause numerical instability in the solver. In such scenarios, we need special treatments to the converged eigenstates, such as:

- Excluding the converged eigenstates from basis generation. The newly generated basis vectors can be made orthogonal to the converged eigenstates. By doing so, the resulting basis vectors will only be able to improve the unconverged eigenstates.

- Adding a penalty term to the converged eigenstates. When applying the matrix-vector operation $A(x)$, this method incorporates a penalty term s for the converged eigenvectors t

$$Ax + \sum_i st_i(t_i \cdot x). \quad (14.15)$$

The un converged eigenvectors, which is now the low-lying states, will be gradually improved in subsequent iterations.

The process of solving for multiple eigenstates may encounter various additional issues, such as:

- The order of the eigenstates may vary across different iterations, posing challenges in identifying the converged eigenstates.
- Certain states may fluctuate near the convergence threshold in different iterations, making it difficult to determine their convergence states.
- The higher eigenstates are not variational, their convergence heavily relies on the accuracy of the lower eigenstates.

In these situations, simply improving the program implementation might not resolve all issues. We might need to employ various strategies to address the eigenvalue problems effectively. For example, we could calculate more eigenstates but only select a few lowest ones. Alternatively, we could appropriately relax the convergence criteria for eigenstates that are difficult to converge.

Non-Hermitian matrix

Although the Davidson diagonalization algorithm was originally developed for symmetric matrices, it can be adapted to diagonalize non-Hermitian matrices with minor modifications. However, for a non-Hermitian matrix, the diagonalization process tends to converge more slowly.

Non-Hermitian matrices feature distinct left and right eigenvectors. Consequently, the residuals associated with these eigenvectors also differ. Therefore, the basis vectors generated by the left and right residuals must be incorporated into the same subspace. In other words, it is inappropriate to use one subspace for left eigenstates and another for right eigenstates. Utilizing separate subspaces for left and right eigenstates would lead to convergence issues.

Moreover, diagonalizing a non-Hermitian matrix can result in complex-valued eigenvalues. In some cases, even though the solution eigenvalues are expected to be real, the intermediate eigenvalues during the iteration process might exhibit an imaginary part due to the incompleteness of the basis subspace. Certain eigenvectors, if their eigenvalues display a reasonably small imaginary part, can be retained in the subspace.

14.4 Direct CI algorithm

The second quantized spin-free Hamiltonian can be represented as [5]

$$\hat{H} = \sum_{pq} h_{pq} \hat{E}_{pq} + \frac{1}{2} \sum_{pqrs} (pq|rs)(\hat{E}_{pq} \hat{E}_{rs} - \delta_{rq} \hat{E}_{ps}), \quad (14.16)$$

where

$$\hat{E}_{pq} = \hat{a}_{p\alpha}^\dagger \hat{a}_{q\alpha} + \hat{a}_{p\beta}^\dagger \hat{a}_{q\beta}. \quad (14.17)$$

The matrix elements are

$$\begin{aligned} H_{I_\alpha I_\beta, J_\alpha J_\beta} &= \sum_{pq} \tilde{h}_{pq} \langle I_\alpha I_\beta | \hat{E}_{pq} | J_\alpha J_\beta \rangle \\ &\quad + \frac{1}{2} \sum_{pqrs} (pq|rs) \langle I_\alpha I_\beta | \hat{E}_{pq} \hat{E}_{rs} | J_\alpha J_\beta \rangle, \end{aligned} \quad (14.18)$$

where \tilde{h} is the dressed one-electron operator

$$\tilde{h}_{pq} = h_{pq} - \frac{1}{2} \sum_r (pr|rq). \quad (14.19)$$

Please note that this one-electron operator is not related to the Fock operator. The summation over index r involves all orbitals.

Given the particle number operator

$$\hat{N} = \sum_p \hat{E}_{pp} = \sum_{pq} \hat{E}_{pq} \delta_{pq}, \quad (14.20)$$

we can construct a diagonal matrix within the determinant bases

$$\sum_{pq} \langle I_\alpha I_\beta | \hat{E}_{pq} \delta_{pq} | J_\alpha J_\beta \rangle = N_{elec} \delta_{I_\alpha J_\alpha} \delta_{I_\beta J_\beta}. \quad (14.21)$$

Using this diagonal matrix, we integrate the one-electron part of the Hamiltonian into the two-electron integrals, thereby simplifying the Hamiltonian matrix:

$$H_{I_\alpha I_\beta, J_\alpha J_\beta} = \sum_{pqrs} V_{pqrs} \langle I_\alpha I_\beta | \hat{E}_{pq} \hat{E}_{rs} | J_\alpha J_\beta \rangle, \quad (14.22)$$

$$V_{pqrs} = \frac{1}{2} (pq|rs) + \frac{1}{2N_{elec}} (\delta_{rs} \tilde{h}_{pq} + \delta_{pq} \tilde{h}_{rs}). \quad (14.23)$$

By inserting the resolution of identity between \hat{E}_{pq} and \hat{E}_{rs} , the Hamiltonian matrix is factorized into a tensor contraction

$$H_{I_\alpha I_\beta, J_\alpha J_\beta} = \sum_{pqrs, K_\alpha, K_\beta} E_{pq, I_\alpha I_\beta K_\alpha K_\beta} V_{pqrs} E_{rs, K_\alpha K_\beta J_\alpha J_\beta}. \quad (14.24)$$

The E tensors are defined as

$$E_{pq, I_\alpha I_\beta J_\alpha J_\beta} = \langle I_\alpha I_\beta | \hat{E}_{pq} | J_\alpha J_\beta \rangle = E_{pq, I_\alpha J_\alpha}^\alpha \delta_{I_\beta J_\beta} + E_{pq, I_\beta J_\beta}^\beta \delta_{I_\alpha J_\alpha}, \quad (14.25)$$

$$E_{pq, I_\alpha}^\alpha = \langle I_\alpha | \hat{a}_{pq}^\dagger \hat{a}_{pq} | J_\alpha \rangle. \quad (14.26)$$

The elements of the E tensor have only three possible values: 0, +1, and -1. We can determine their values using the algebra of second quantization, as demonstrated in the following code example.

```
def Etensor_value(create_p: int, annihilate_q: int,
                  strI: String, strJ: String):
    '''The value of <stringI|p^\dagger q|stringJ>'''
    sign1, strK1 = strJ.annihilate(annihilate_q)
    sign2, strK2 = strK1.create(create_p)
    if strK2 == strI:
        return sign1 * sign2
    else:
        return 0

def make_Etensor(norb, nelec):
    strings = make_strings(norb, nelec)
    na = len(strings)
    Et = np.zeros((norb, norb, na, na))
    for p in range(norb):
        for q in range(norb):
            for i, strI in enumerate(strings):
                for j, strJ in enumerate(strings):
                    Et[p, q, i, j] = Etensor_value(p, q, strI, strJ)
    return Et
```

Given the structure of determinants in Eq. (14.4), we can structure the FCI wave function coefficients into a matrix $C_{J_\alpha J_\beta}$. The rows and columns of this matrix correspond to the α and β strings, respectively. The matrix-vector product, $\mathbf{H}\mathbf{C}$, can be decomposed into three steps of tensor contractions:

$$D_{rs, K_\alpha K_\beta} = \sum_{J_\alpha} E_{rs, K_\alpha J_\alpha}^\alpha C_{J_\alpha K_\beta} + \sum_{J_\beta} E_{rs, K_\beta J_\beta}^\beta C_{K_\alpha J_\beta}, \quad (14.27)$$

$$G_{pq, K_\alpha K_\beta} = \sum_{pqrs} V_{pqrs} D_{rs, K_\alpha K_\beta}, \quad (14.28)$$

$$(\mathbf{H}\mathbf{C})_{I_\alpha I_\beta} = \sigma_{I_\alpha I_\beta} = \sum_{pq, K_\alpha} E_{pq, I_\alpha K_\alpha}^\alpha G_{pq, K_\alpha I_\beta} + \sum_{pq, K_\beta} E_{pq, I_\beta K_\beta}^\beta G_{pq, I_\alpha K_\beta}. \quad (14.29)$$

These tensor contractions can be rapidly implemented using the `einsum` function.

```

def compute_hc(h1, eri, fciwfn, norb, nelec_a, nelec_b):
    Etensor_a = make_Etensor(norb, nelec_a)
    Etensor_b = make_Etensor(norb, nelec_b)

    d = einsum('pqKI,IJ->pqKJ', Etensor_a, fciwfn, optimize=True)
    d += einsum('pqKJ,IJ->pqIK', Etensor_b, fciwfn, optimize=True)

    v = merge_h1_eri(h1, eri, nelec_a + nelec_b)
    g = einsum('pqrs,rsIJ->pqIJ', v, d)

    sigma = einsum('pqKI,pqIJ->KJ', Etensor_a, g, optimize=True)
    sigma += einsum('pqKJ,pqIJ->IK', Etensor_b, g, optimize=True)
    return sigma

```

14.5 Optimizing tensor contractions

Eqs. (14.27) and (14.29) involve computationally intensive tensor contractions. However, the E tensor in these equations is very sparse. This sparsity can be leveraged to reduce the computational cost.

The non-zero elements of the E tensor can be efficiently stored in a lookup table:

- Each string corresponds to one entry of the lookup table.
- Each entry of the table stores several 4-element tuples, $(p, q, \text{address}, \text{value})$. Given the table entry J_α , this tuple represents the orbital indices p, q , the address of string I_α , and the value of the tensor element in $E_{pq, I_\alpha J_\alpha}^\alpha$.

The following code demonstrates how the lookup table is constructed:

```

def make_Elt(norb, nelec):
    '''The lookup table for non-zero elements of E tensor'''
    strings = make_strings(norb, nelec)
    # To setup the map from string to address,
    # __hash__ and __eq__ methods must be created for String object.
    strings_address = {s: i for i, s in enumerate(strings)}
    Elt = []
    for k, strI in enumerate(strings):
        table_k = []
        occs = strI.occupied_orbitals
        uoccs = [p for p in range(norb) if p not in occs]
        for q in uoccs:
            sign1, strK1 = strI.annihilate(q)
            for p in uoccs:
                sign2, strJ = strK1.create(p)
                Elt.append((strI, strJ, strings_address[strI], sign1 * sign2))

```

```

        # Applying a^\dagger_p a_q on address of strI leads to
        # to address of the output string (strJ)
        table_k.append([p, q, strings_address[strJ], sign1*sign2])
    for p in occs:
        table_k.append([p, p, k, 1])
    Elt.append(table_k)
return Elt

```

To simplify the conversion between the `String` object and its corresponding address in the CI coefficients matrix, we employ a dictionary to index the string and calculate its address.

By using the lookup table, we eliminate the E tensor and all tensor contractions associated with the E tensor. Consequently, we achieve an optimized implementation for **HC**:

```

def compute_hc(h1, eri, fciwfn, norb, nelec_a, nelec_b):
    Elt_a = make_Elt(norb, nelec_a)
    Elt_b = make_Elt(norb, nelec_b)
    na = len(Elt_a)
    nb = len(Elt_b)

    d = np.zeros((norb,norb,na,nb))
    for I, tab in enumerate(Elt_a):
        for a, i, J, sign in tab:
            d[a,i,J] += sign * fciwfn[I]
    for I, tab in enumerate(Elt_b):
        for a, i, J, sign in tab:
            d[a,i,:,J] += sign * fciwfn[:,I]

    v = absorb_h1(h1, eri, nelec_a + nelec_b)
    g = einsum('pqrs,rsIJ->pqIJ', v, d)

    sigma = np.zeros_like(fciwfn)
    for I, tab in enumerate(Elt_a):
        for a, i, J, sign in tab:
            sigma[J] += sign * g[a,i,I]
    for I, tab in enumerate(Elt_b):
        for a, i, J, sign in tab:
            sigma[:,J] += sign * g[a,i,:,I]
    return sigma

```

In this function, the call to `make_Elt` represents a primary bottleneck. The most time-consuming operations within `make_Elt` include determining the sign for $\hat{a}_p^\dagger \hat{a}_q$ and performing the address lookup for the resulting string. To enhance the efficiency of this function, we employ the following optimizations:

- Convert the `String` object to a binary representation. This optimization can accelerate the lookup speed in the dictionary.
- Cache the sign for each second quantized operator. This enables the rapid computation of the sign for the operator pair $\hat{a}_p^\dagger \hat{a}_q$ using the cached signs. When $p > q$, the number of orbitals to be swapped for \hat{a}_p^\dagger and \hat{a}_p remains the same as the cases in the cache. When $p < q$, an extra minus sign is required because the application of \hat{a}_q removes one orbital, which affects \hat{a}_p^\dagger .

The optimized `make_Elt` function is implemented as follows:

```
def as_bin(occupied_orbitals):
    binstr = 0
    for i in occupied_orbitals:
        binstr |= (1 << i)
    return binstr

def make_Elt(norb, nelec):
    '''The lookup table for non-zero elements of E tensor'''
    strings = make_strings(norb, nelec)
    strings_address = {as_bin(s): i for i, s in enumerate(strings)}

    Elt = []
    for k, binI in enumerate(bin_strings):
        table_k = []
        occs = []
        uocc = []
        sign_cache = []
        sign = 1
        for i in reversed(range(norb)):
            sign_cache.append(sign)
            if (1 << i) & binI:
                occs.append(i)
                sign = -sign
            else:
                uocc.append(i)
        sign_cache = sign_cache[::-1]
        occs = occs[::-1]
        uocc = uocc[::-1]

        for p in occs:
            table_k.append([p, p, k, 1])
        for q, p in product(occs, uocc):
            binJ = binI ^ (1 << q) ^ (1 << p)
            if p > q:
                table_k.append([p, q, k, 1])
            else:
                table_k.append([q, p, k, 1])
    return table_k
```

```

        sign = sign_cache[p] * sign_cache[q]
    else:
        sign = -sign_cache[p] * sign_cache[q]
    table_k.append([p, q, strings_address[binJ], sign])
    Elt.append(table_k)
return np.array(Elt, dtype=int)

```

14.6 Optimization for memory efficiency

The `compute_hc` function in Section 14.5 allocates two temporary 4-index tensors, `d` and `g`. These temporary variables consume a substantial amount of memory. For instance, in a system with 14 orbitals and $7\alpha + 7\beta$ electrons, the FCI wave function includes $\binom{14}{7}^2 = 11778624$ float64 coefficients, which occupy approximately 100 MB of memory. The actual memory footprint observed during program execution is close to 40 GB. The primary contributors are the `d` tensor and the `g` tensor, both of which scale as $\binom{14}{7}^2 \times 14^2 = 2.31 \times 10^9$.

To reduce memory usage, we should process the `d` tensor and `g` tensor in blocks. In this algorithm, care must be taken with the address provided by the `Elt` lookup table because it may exceed the boundary of the blocked `d` tensor. For instance, in the statement

```
d[a,i,J] += sign * fciwfn[I]
```

an arbitrary index `J` is very likely not located within the current sub-block of the `d` tensor.

Due to the permutation symmetry of the E tensor:

$$E_{pq, I_\alpha J_\alpha}^\alpha = E_{qp, J_\alpha I_\alpha}^\alpha, \quad (14.30)$$

both the statement

```
d[a,i,J] += sign * fciwfn[I]
```

and the statement

```
d[i,a,I] += sign * fciwfn[J]
```

will produce the same intermediate `d` tensor. Therefore, we can change the `compute_hc` function to a block-style implementation:

```

@numba.njit
def _compute_hc(v, fciwfn, norb, Elt, blocksize=60):
    Elt_a, Elt_b = Elt
    na, nb = fciwfn.shape

```

```

sigma = np.zeros_like(fciwfn)

for Ka0 in range(0, na, blocksize):
    ma = min(na - Ka0, blocksize)
    Ka1 = Ka0 + ma
    for Kb0 in range(0, nb, blocksize):
        mb = min(nb - Kb0, blocksize)
        Kb1 = Kb0 + mb
        d = np.zeros((norb,norb,ma,mb))

        for I, tab in enumerate(Elt_a[Ka0:Ka1]):
            for a, i, J, sign in tab:
                for K in range(mb):
                    d[i,a,I,K] += sign * fciwfn[J,Kb0+K]
        for I, tab in enumerate(Elt_b[Kb0:Kb1]):
            for a, i, J, sign in tab:
                for K in range(ma):
                    d[i,a,K,I] += sign * fciwfn[Ka0+K,J]

        g = v.dot(d.reshape(norb**2,-1)).reshape(norb,norb,ma,mb)

        for I, tab in enumerate(Elt_a[Ka0:Ka1]):
            for a, i, J, sign in tab:
                for K in range(mb):
                    sigma[J,Kb0+K] += sign * g[a,i,I,K]
        for I, tab in enumerate(Elt_b[Kb0:Kb1]):
            for a, i, J, sign in tab:
                for K in range(ma):
                    sigma[Ka0+K,J] += sign * g[a,i,K,I]
    return sigma

def compute_hc(h1, eri, fciwfn, norb, nelec_a, nelec_b, blocksize=60):
    Elt_a = make_Elt(norb, nelec_a)
    Elt_b = make_Elt(norb, nelec_b)
    v = absorb_h1(h1, eri, nelec_a + nelec_b).reshape(norb**2,-1)
    return _compute_hc(v, fciwfn, norb, (Elt_a, Elt_b), blocksize)

```

Here is a new question for the block computation algorithm: how should we choose the block size? Different block sizes offer distinct advantages:

- A larger block size may reduce the overhead of various operations incurred in Python code and enhance the efficiency of the matrix multiplication

```
g = v.dot(d.reshape(norb**2,-1))
```

- A relative small block size can be beneficial for memory management. Accessing small-sized arrays has a lower possibility of page faults.
- When considering the utilization of the CPU cache, the block size might need to be adjusted based on the cache size. If the intermediate variables can be completely loaded within the CPU cache, this can significantly enhance computational efficiency.

Another issue in the `compute_hc` function is the use of the `np.zeros` function to create the buffer for the `d` tensor inside the loop. This operation is not memory-efficient as it increases the overhead of memory allocation. Additionally, the newly allocated memory is likely to refer to unmapped virtual memory, which can result in more page faults and a higher likelihood of cache misses.

To address this issue, we can move the memory allocation operation outside the loop and allow the `d` tensor to reuse the same memory as a buffer. Additionally, we can pass a buffer to the `np.dot` function to reuse the memory for the output of matrix multiplication. After calling the `np.dot` function, we swap the buffers for the `d` tensor and the `g` tensor. This allows the program to use the `g` buffer for the next `d` tensor. It can increase the cache hit rates when processing the next blocks, as the memory for the `g` tensor remains hot.

```
@numba.njit
def _compute_hc(v, fciwfn, norb, Elt, blocksize=60):
    Elt_a, Elt_b = Elt
    na, nb = fciwfn.shape
    sigma = np.zeros_like(fciwfn)
    d_buf = np.empty(norb**2*blocksize**2)
    g_buf = np.empty(norb**2*blocksize**2)

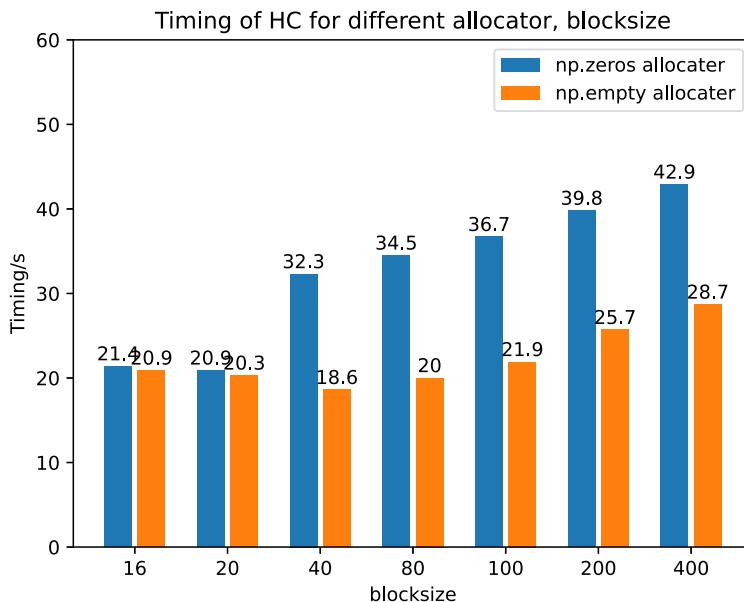
    for Ka0 in range(0, na, blocksize):
        ma = min(na - Ka0, blocksize)
        Ka1 = Ka0 + ma
        for Kb0 in range(0, nb, blocksize):
            mb = min(nb - Kb0, blocksize)
            Kb1 = Kb0 + mb
            d = d_buf[:norb*norb*ma*mb].reshape(norb,norb,ma,mb)
            d[:] = 0.

            ...

            g = g_buf[:norb*norb*ma*mb].reshape(norb*norb,ma*mb)
            g = np.dot(v, d.reshape(norb**2,-1), out=g)
            g = g.reshape(norb,norb,ma,mb)
            d_buf, g_buf = g_buf, d_buf

            ...

            g = g_buf[:norb*norb*ma*mb].reshape(norb*norb,ma*mb)
            g = np.dot(v, d.reshape(norb**2,-1), out=g)
            g = g.reshape(norb,norb,ma,mb)
            d_buf, g_buf = g_buf, d_buf
```

**FIGURE 14.1**

FCI `compute_hc` performance for 14 orbitals and $7\alpha + 7\beta$ electrons.

We tested the performance of the `compute_hc` function for a system with 14 orbitals and $7\alpha + 7\beta$ electrons using various block sizes. This test included a comparison between the straightforward implementation with the `np.zeros` allocator and the memory access optimized version (labeled as `np.empty`). The results are displayed in Fig. 14.1.

This test clearly demonstrates the impact of different block sizes on performance. Generally, smaller block sizes yield better performance than larger ones, suggesting that memory management plays a more significant role in performance than the overhead in Python interpretation execution. For the `np.empty` allocator, optimal performance is achieved with a block size of around 40.

The testing platform features a 1 MB L2 cache on each CPU core and a 20 MB L3 cache shared among all CPU cores. For this 14-orbital system, to accommodate both the `d` tensor and the `g` tensor into the L2 cache, the block size should not exceed 18, since

$$14^2 \times 18^2 \times 2 \times 8 \text{ B} \approx 0.97 \text{ MB}.$$

Regarding the L3 cache, the maximum block size might be chosen to approximately 80:

$$14^2 \times 80^2 \times 2 \times 8 \text{ B} \approx 19.14 \text{ MB}.$$

However, setting the block size to 16 to fit the L2 cache can lead to worse performance. This is because the problem is decomposed into 46000 blocks at this size, resulting in significant overhead from Python bytecode interpretation.

The performance of the `np.empty` allocator is significantly faster than that of the `np.zeros` version for larger block sizes. When the block size is 20, the difference between the two implementations is small. However, when the block size is increased to 40, the memory-optimized version is almost twice as fast as the `np.zeros` version. Why does optimizing memory management lead to such a significant difference? To understand this, let's use the `perf` profiler to examine the effects of memory efficiency optimization when the block size is set to 40.

The output of the performance profiling for the `np.zeros` version is shown below:

```
$ perf python fci_zeros_allocator.py
...
      8,681,115    page-faults          # 0.245 M/sec
  109,270,271,406    cycles          # 3.083 GHz
 187,988,136,856   instructions        # 1.72 insn per cycle
 11,589,167,229    branches         # 326.946 M/sec
     97,273,047   branch-misses      # 0.84% of all
branches

35.466817095 seconds time elapsed

25.798432000 seconds user
 9.648909000 seconds sys
```

After optimizing the memory efficiency, the profiling output is as follows:

```
$ perf python fci_empty_allocator.py
...
      62,376    page-faults          # 0.003 M/sec
  65,411,436,911    cycles          # 3.065 GHz
 153,148,554,394   instructions        # 2.34 insn per cycle
  5,683,433,297    branches         # 266.341 M/sec
     69,155,915   branch-misses      # 1.22% of all
branches

21.360130211 seconds time elapsed

21.199824000 seconds user
 0.139998000 seconds sys
```

The most notable difference between the two versions is the number of page faults. In the version which eagerly allocates new arrays by `np.zeros`, there are 8.7 million minor page faults. Each minor page fault takes about one microsecond to process in kernel space. Consequently, more than 9 seconds (`sys` time) are spent in kernel

space to manage memory page-related operations. Additionally, there is a difference of nearly 4.5 seconds in `user` time between the two versions. This difference primarily reflects the impact of CPU cache misses due to the newly allocated memory.

We can make a quick estimate based on the system size to identify the source of the page faults. The total number of newly allocated `d` tensors and `g` tensors is

$$2 \times \left(\binom{14}{7} / 40 \right)^2 \approx 14792.$$

Each `d` tensor or `g` tensor occupies

$$14^2 \times 40^2 \times 8 \text{ B} = 2450 \text{ KB}.$$

For arrays of this size, the memory allocator utilizes 4 KB memory pages. The total number of 4 KB pages requested during the computation is

$$14792 \times 2450 / 4 \approx 9 \text{ million}.$$

This calculation precisely matches the 8.7 million minor page faults.

14.7 Parallel computation

The `compute_hc` function exhibits low parallel efficiency, as illustrated in Fig. 14.2. The speedup is even less than 2 when using 8 threads. It is not difficult to identify the reason for the poor parallel efficiency. Inside the `compute_hc` function, the `np.dot` function is the only operation that utilizes parallelization (through the underlying BLAS library), while the remaining operations are not parallelized. The serial code quickly becomes the dominant factor in computation time, negating the benefits of parallelization.

To enhance parallel efficiency, one approach is to parallelize the for-loop iterations in the serial code using the automatic threading parallelization techniques, such as the OpenMP threads offered by the Cython `prange` function or the Numba threads provided by the Numba `prange` function. Before applying the automatic threading parallelization, let's further analyze the `compute_hc` function to determine suitable parallelization schemes.

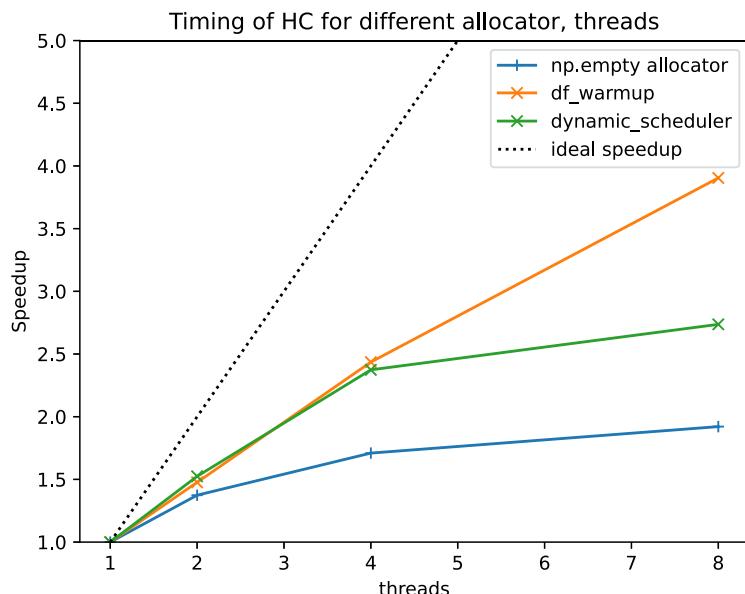
It is challenging to parallelize the outer loops for the code implemented in Section 14.6, such as the loop

```
for Kb0 in range(0, nb, blocksize)
```

or

```
for Ka0 in range(0, na, blocksize)
```

This is because the `np.dot` function, which is called inside these loops, can also spawn threads, leading to nested threads. Nested threads may oversubscribe CPU resources and dramatically increase the overhead of context switches.

**FIGURE 14.2**

The speed up of the matrix-vector multiplication in various FCI implementations.

To achieve threading parallelization for the outer loops, one solution is to utilize different threading controllers to manage the different code blocks separately. For example, the `np.dot` function typically relies on a specific threading controller provided by BLAS libraries. Certain environment variables, such as `OPENBLAS_NUM_THREADS`, `MKL_NUM_THREADS`, can be used to control the threads of the BLAS library. For OpenMP and Numba threads, the environment variables `OMP_NUM_THREADS` and `NUMBA_NUM_THREADS` can be configured to manage the threads. Please note that most threading controllers will utilize all available CPU cores for parallelization by default, unless specific environment variables are explicitly set. To address the challenges of threading management, one can use the `threadpoolctl` library to explicitly manage threads for various threading controllers within the code.

Automatic threading parallelization for constructing the `d` tensor and distributing the `g` tensor faces different challenges. This difficulty arises because different threads might access the same memory addresses in some operations. For instance, in the following code for constructing the `d` tensor, parallelizing the index `I` can result in the last index of the `d` tensor being accessed by multiple threads.

```
for I, tab in enumerate(Elt_b[Kb0:Kb1]):
    for a, i, J, sign in tab:
        for K in range(ma):
            d[i,a,K,I] += sign * fciwfn[Ka0+K,J]
```

Even without race conditions, accessing the contiguous memory address via different processors can cause *false cache sharing contention* among different CPU cores. This results in high cache coherence overhead, as discussed in Chapter 9.

To parallelize the code for the `d` and `g` tensor, we can restructure the two tensors to avoid inefficient memory access patterns. For instance, by swapping the orbital indices with the `String` indices (changing the layout from `d[i,a,K,I]` to `d[K,I,i,a]`), we can construct the `d` tensor as follows:

```
for I, tab in enumerate(Elt_b[Kb0:Kb1]):
    for a, i, J, sign in tab:
        for K in range(ma):
            d[K,I,i,a] += sign * fciwfn[Ka0+K,J] # (1)
```

This modification will reduce the occurrence of *memory coherence* issues among different threads. However, this change may slightly decrease the effectiveness of the CPU cache, as the assignments in line (1) involve non-contiguous data access.

It is possible to further optimize the efficiency of the `compute_hc` function through automatic threading parallelization. By using features like critical sessions and thread synchronization provided by OpenMP, race conditions can be addressed and more loops can be parallelized. This reduces the portion of the program that executes serially, thereby enhancing the parallel performance. However, these features are accessible through the C/C++ level APIs of OpenMP. The FCI code would have to be written in C/C++ and then imported into Python using the Python/C interface method described in Chapter 8. Here, we omit the details of the code written in C/C++.

In contrast to the automatic threading parallelization, we can use the pipeline executor (Section 10.4 of Chapter 10) to enable the parallel computation. To integrate the pipeline executor into the `compute_hc` function, it is necessary to divide the `compute_hc` function into three distinct operations: `build_d`, `dot_v`, and `assemble_g`. Each operation corresponds to one step outlined from Eqs. (14.27) to (14.29).

```
@numba.njit(nogil=True)
def build_d(task, fciwfn, norb, Elt):
    Ka0, Ka1, Kb0, Kb1 = task
    ma = Ka1 - Ka0
    mb = Kb1 - Kb0
    Elt_a, Elt_b = Elt
    d = np.zeros((norb, norb, ma, mb))
    for I, tab in enumerate(Elt_a[Ka0:Ka1]):
        for a, i, J, sign in tab:
            for K in range(mb):
                d[i,a,I,K] += sign * fciwfn[J,Kb0+K]
    for I, tab in enumerate(Elt_b[Kb0:Kb1]):
        for a, i, J, sign in tab:
            for K in range(ma):
                d[i,a,K,I] += sign * fciwfn[Ka0+K,J]
```

```

        return d, task                                # (1)

@numba.njit(nogil=True)
def dot_v(d_task, v):
    d, task = d_task
    norb, ma, mb = d.shape[1:]
    g = v.reshape(norb**2,-1).dot(d.reshape(norb**2,ma*mb))
    return g.reshape(norb,norb,ma,mb), task          # (2)

def assemble_g(g_task, norb, Elt, sigma_pool):
    thread_id = threading.get_ident()
    if thread_id in sigma_pool:
        sigma = sigma_pool[thread_id]
    else:
        Elt_a, Elt_b = Elt
        na = len(Elt_a)
        nb = len(Elt_b)
        sigma = np.zeros((na, nb))
        sigma_pool[thread_id] = sigma                # (3)
    _assemble_g(g_task, norb, Elt, sigma)

@numba.njit(nogil=True)
def _assemble_g(g_task, norb, Elt, sigma):
    g, task = g_task
    Ka0, Ka1, Kb0, Kb1 = task
    ma = Ka1 - Ka0
    mb = Kb1 - Kb0
    Elt_a, Elt_b = Elt
    for I, tab in enumerate(Elt_a[Ka0:Ka1]):
        for a, i, J, sign in tab:
            for K in range(mb):
                sigma[J,Kb0+K] += sign * g[a,i,I,K]
    for I, tab in enumerate(Elt_b[Kb0:Kb1]):
        for a, i, J, sign in tab:
            for K in range(ma):
                sigma[Ka0+K,J] += sign * g[a,i,K,I]

```

Operations `build_d`, `dot_v`, and `assemble_g` need to be executed sequentially in the pipeline. The output of each operation serves as the input for the next operation. Therefore, in addition to returning the necessary tensor as the result of each operation, we also output the task ID of current job, as shown in line (1) and (2). This task ID is used to determine which task to process in the subsequent operations.

The function `assemble_g` requires access to the output variable `sigma`. To avoid race condition between threads, a private `sigma` object is allocated for each thread

worker. These objects are cached in the `sigma_pool` dictionary, as shown in line (3). After the pipeline processing is complete, we can aggregate the private `sigma` objects from each thread to obtain the final result.

To minimize the overhead of data transfer, we utilize the `ThreadPoolExecutor` within the `pipeline` function, which enables us to share the output of each operation in the same memory space. However, the Python Global Interpreter Lock (GIL) may severely affect the efficiency of thread parallelization. To mitigate this issue, the `nogil=True` option is enabled in the Numba JIT compiler to remove the GIL for each compiled function.

The optimized `compute_hc` function is implemented in a completely different manner, as follows:

```
def create_tasks(na, nb, blocksize):
    for Ka0 in range(0, na, blocksize):
        Ka1 = min(na, Ka0 + blocksize)
        for Kb0 in range(0, nb, blocksize):
            Kb1 = min(nb, Kb0 + blocksize)
            yield Ka0, Ka1, Kb0, Kb1

def compute_hc(h1, eri, fciwfn, norb, nelec_a, nelec_b, blocksize=40,
              threads=2):
    from threadpoolctl import ThreadpoolController
    Elt_a = make_Elt(norb, nelec_a)
    Elt_b = make_Elt(norb, nelec_b)
    Elt = (Elt_a, Elt_b)
    v = merge_h1_eri(h1, eri, nelec_a + nelec_b).reshape(norb**2, -1)
    na = len(Elt_a)
    nb = len(Elt_b)
    tasks = create_tasks(na, nb, blocksize)

    sigma_pool = {}
    blas_cpus = (threads+1)//2 # (1)
    remaining_cores = threads - blas_cpus
    with ThreadpoolController().limit(limits=blas_cpus, user_api='blas'):
        with ThreadPoolExecutor(max_workers=remaining_cpus) as q1: # (2)
            with ThreadPoolExecutor(max_workers=1) as q2: # (3)
                sigmas = pipeline([
                    (q1, build_d, (fciwfn, norb, Elt)), # (4)
                    (q2, dot_v, (v,)), # (5)
                    (q1, assemble_g, (norb, Elt, sigma_pool)), # (6)
                ], tasks, dynamic_scheduler)
    return sum(sigma_pool.values())
```

The computational costs for executing `build_d`, `dot_v`, and `assemble_g` operations are different. How to properly allocate resources for each operation and achieve a

balance among the various operations and tasks? After some attempts of experiments and profiling, we observed that the total time consumed by `build_d` and `assemble_g` is roughly equivalent to that of `dot_v`. Consequently, we allocate half of the available threads to `dot_v`, as indicated in line (1). The `np.dot` function in `dot_v` can be executed in parallel through the underlying BLAS library. We utilize the `threadpoolctl` library to limit the number of threads that `np.dot` can use. To avoid nested multithreading issues, we restrict the `dot_v` function to use only one thread, as shown in line (3). The `dot_v` operation is executed in this dedicated queue, `q2`, as indicated in line (5). The remaining threads are allocated to `build_d` and `assemble_g`, as shown in line (2). To utilize resources more efficiently, `build_d` and `assemble_g` share the same queue, `q1`, as indicated in lines (4) and (6).

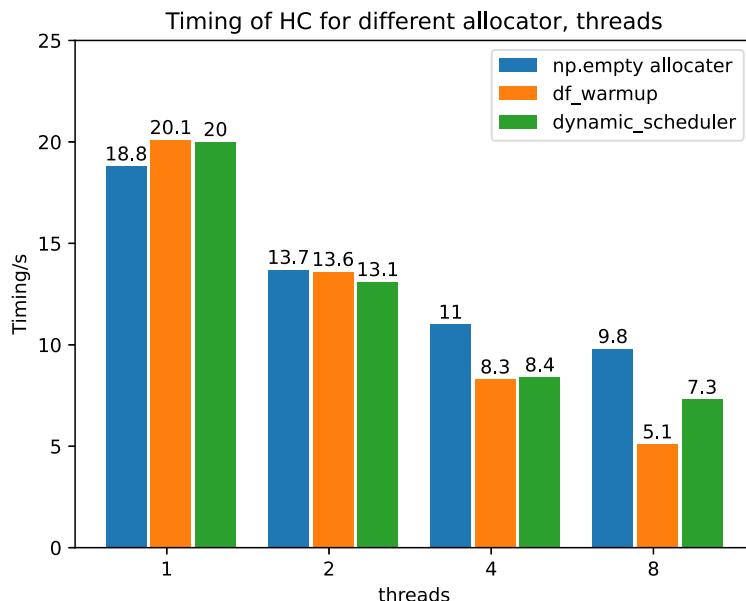


FIGURE 14.3

The parallel efficiency of various FCI `compute_hc` implementations.

This version invokes the `dynamic_scheduler` (see Chapter 10) for the pipeline executor. Other job schedulers like the `df_warmup_scheduler` can be utilized with the pipeline executor as well. The performance of these schedulers under different numbers of threads is illustrated in Fig. 14.3.

The pipeline execution of the `compute_hc` function achieves a speedup of approximately 4 times with 8 threads, corresponding to about 50% parallel efficiency. Although this speedup is still far from the ideal parallel efficiency, it already shows a significant improvement over the regular implementations in Section 14.6.

An interesting observation in these tests is that, as more threads are employed, `df_warmup_scheduler` becomes faster than `dynamic_scheduler`. This is because in `dynamic_scheduler`, the `build_d` function processes tasks freely and produces an excessive number of `d_tensor` objects in the task queue. These `d_tensor` objects are not consumed timely by the downstream consumer, the `dot_v` function. Before `dot_v` triggers the subsequent consumer `assemble_g`, the pending tasks generated by `build_d` already occupy a significant amount of memory, which affects the memory management efficiency.

In the case of `df_warmup_scheduler`, the task producer is blocked after the `build_d` operation creates several initial tasks. Once the `dot_v` function consumes an output from `build_d`, a new `build_d` task can be added to the task queue. The `dot_v` tasks are similarly blocked after producing the initial tasks for the `assemble_g` operation. The size of the task queue generated by `build_d` is limited to a certain number of pending tasks. No additional tasks are produced until the pending tasks are consumed by the `dot_v` operation. This operation itself generates tasks and then is blocked until the tasks are consumed by the `assemble_g` function. As a result, the memory consumption is limited, and the overhead of memory management is also minimized.

Summary

Solving the Full Configuration Interaction (FCI) wavefunction involves the use of the Davidson diagonalization algorithm. A matrix-vector operation between the Hamiltonian and the FCI wavefunction is required for the Davidson solver. We have demonstrated how to implement and optimize the matrix-vector function.

The matrix-vector operation is the most demanding task in the FCI program. This operation can be formulated using tensor contractions. Some tensors are sparse. This property can be leveraged to decrease the computational costs of tensor contractions. We employed techniques such as dynamic programming, LRU cache, and lookup tables to store and index the non-zero values of the sparse tensor.

The FCI program is not only computationally intensive, but memory efficiency is also a crucial factor. To enhance memory efficiency, the tensor contractions within the matrix-vector function are restructured into a block-based algorithm, leading to multiple tensor operations with smaller blocks. This optimization reduces the memory footprint, decreases page faults, and enhances CPU cache utilization.

Another challenge we discussed is the parallel execution of the FCI program. In addition to the parallel execution of tensor contractions in the BLAS library, we can also utilize Numba and Cython compilation to automatically implement threading parallelization for certain for-loops. However, some computational problems do not have suitable loops for automatic threading parallelization. In such cases, different components of the program can be computed in parallel using the pipeline executor we developed in Chapter 10. Taking the FCI matrix-vector operation as an example, we decomposed the matrix-vector function into three independent operations. These operations have different demands on memory and CPU. We then tested several job

schedulers in conjunction with the pipeline executor to optimize the load balance, CPU utilization, and memory footprint.

References

- [1] E.R. Davidson, The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, *Journal of Computational Physics* 17 (1) (1975) 87–94, [https://doi.org/10.1016/0021-9991\(75\)90065-0](https://doi.org/10.1016/0021-9991(75)90065-0), <https://www.sciencedirect.com/science/article/pii/0021999175900650>.
- [2] Y. Saad, 8. Preconditioning Techniques, pp. 193–218, <https://epubs.siam.org/doi/pdf/10.1137/1.9781611970739.ch8>, <https://doi.org/10.1137/1.9781611970739.ch8>, <https://epubs.siam.org/doi/abs/10.1137/1.9781611970739.ch8>.
- [3] Y. Saad, 7. Filtering and Restarting Techniques, pp. 163–191, <https://epubs.siam.org/doi/pdf/10.1137/1.9781611970739.ch7>, <https://doi.org/10.1137/1.9781611970739.ch7>, <https://epubs.siam.org/doi/abs/10.1137/1.9781611970739.ch7>.
- [4] A. Szabo, N.S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, 1st edition, Dover Publications, Inc., Mineola, 1996.
- [5] T. Helgaker, P. Jørgensen, J. Olsen, *Configuration-Interaction Theory*, John Wiley & Sons, Ltd, 2000, Ch. 11, pp. 523–597, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119019572.ch11>, <https://doi.org/10.1002/9781119019572.ch11>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119019572.ch11>.

Post Hartree-Fock II: coupled cluster 15

Coupled cluster (CC) theory is a very successful post-Hartree-Fock method for describing electron correlation effects. The results of CC calculations are often used as benchmarks to assess the accuracy of other theoretical methods. Despite the complex equations in coupled cluster theory, developing a basic CC program that includes single and double excitations (CCSD) is quite manageable, thanks to the NumPy library and various tensor contraction tools in Python.

From a technical perspective, the CC program represents a typical post-HF implementation that requires the use of various techniques to achieve optimal performance. The techniques used in CC programs are quite different from those utilized by the full CI program discussed in Chapter 14. We will use the Coupled Cluster with double excitation (CCD) program as a case study to explore some distinctive techniques utilized in CC programs. We will demonstrate the use of mutexes and condition variables, which were introduced in Chapter 10, to protect shared resources during asynchronous execution. The code generation technique from Chapter 5 will be employed to develop a just-in-time (JIT) compiler, which compiles the CCD code to enable the functionality of data prefetching. Additionally, we will develop a symbolic programming approach to automatically derive CC equations and programs, extending the discussions on symbolic computation from Chapter 5.

This chapter focuses on the techniques applicable in coupled cluster programs. We will briefly introduce the basic formulas and theoretical background of coupled cluster theory, but we do not intend to delve deeply into its theoretical framework. There are several excellent textbooks that document coupled cluster theory in more details. Readers may refer to the book *Many-Body Methods in Chemistry and Physics, MBPT and Coupled-Cluster Theory* by Shavitt and Bartlett [1], as well as the introductory article *An Introduction to Coupled Cluster Theory for Computational Chemists* by Crawford and Schaefer [2].

Low-excitation coupled cluster methods, such as CCSD, CCSD(T) (CCSD with perturbative triples), CC2 (second-order linear-response Coupled Cluster), and EOM-CCSD (Equation of Motion CCSD), are standard methods in quantum chemistry. If you are interested in the Python implementations of these CC methods, you can also explore the implementations in the open-source quantum chemistry software package, PySCF [3]. The Python implementation of these methods primarily involves the use of the `einsum` function, which is straightforward to understand.

15.1 Coupled cluster theory

In coupled cluster theory, the wave function is described using an exponential ansatz. This ansatz provides a systematic way to describe electron correlation effects beyond the mean-field approximation and approach the Full Configuration Interaction wave function, which is expressed as

$$\Psi_{\text{CC}} = e^{\hat{T}} \Psi_0, \quad (15.1)$$

where the reference state Ψ_0 is typically chosen to be the Hartree-Fock determinant. The cluster operator \hat{T} is a sum of excitation operators up to infinity orders

$$\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \dots \quad (15.2)$$

The terms $\hat{T}_1, \hat{T}_2, \hat{T}_3, \dots$ correspond to single, double, triple, and high-order excitations, respectively:

$$\hat{T}_1 = \sum_{ai} t_i^a \hat{a}_d^\dagger \hat{a}_i, \quad (15.3)$$

$$\hat{T}_2 = \sum_{a>b, i>j} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i, \quad (15.4)$$

$$\hat{T}_3 = \sum_{\substack{a>b>c \\ i>j>k}} t_{ijk}^{abc} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \hat{a}_k \hat{a}_j \hat{a}_i. \quad (15.5)$$

Here, the indices i, j, k, \dots represent the occupied molecular orbitals, while a, b, c, \dots represent the virtual orbitals. \hat{a}^\dagger and a are the creation and annihilation operators associated to these orbitals, respectively. The tensors $t_i^a, t_{ij}^{ab}, t_{ijk}^{abc}$ are commonly referred to as the cluster *amplitudes* in the literature of coupled cluster methods.

By applying a similarity transformation to the Hamiltonian using the exponential of the cluster operator, we obtain the effective Hamiltonian:

$$\hat{H}_{\text{eff}} = e^{-\hat{T}} \hat{H} e^{\hat{T}}. \quad (15.6)$$

To determine the coupled cluster energy E_{CC} and the amplitudes, the effective Hamiltonian is projected onto the reference state Ψ_0 and the excited state projectors μ , resulting in the equations:

$$E_{\text{CC}} = \langle \Psi_0 | \hat{H}_{\text{eff}} | \Psi_0 \rangle, \quad (15.7)$$

$$0 = \langle \mu | \hat{H}_{\text{eff}} | \Psi_0 \rangle. \quad (15.8)$$

If we exclude the single excitation amplitudes \hat{T}_1 from CCSD, we obtain the CCD (coupled cluster with doubles) method. Although the computational costs of CCD are similar to those of CCSD, its accuracy is lower due to the absence of \hat{T}_1 amplitudes.

Consequently, the CCD method is not as widely used in practical chemical applications. However, from a programming perspective, the CCD method adequately illustrates the technical challenges encountered in CC programs. For the sake of simplicity and to demonstrate computational techniques, we will focus on the CCD method in the following context.

15.2 CCD program

The equations for CCD and CCSD have been extensively discussed in the literature [4–6] [1,2]. Given the abundance of resources available, we will omit the derivation process and directly list the relevant equations below. The CCD correlation energy is given by:

$$E_{\text{corr}} = \sum_{i>j, a>b} \langle ij || ab \rangle t_{ij}^{ab}. \quad (15.9)$$

The equation for \hat{T}_2 amplitudes is

$$\begin{aligned} 0 = & \langle ab || ij \rangle + \sum_{k>l} (\langle ij || kl \rangle + \sum_{c>d} \langle kl || cd \rangle t_{ij}^{cd}) t_{kl}^{ab} + \sum_{c>d} \langle ab || cd \rangle t_{ij}^{cd} \\ & + P(ab) \sum_c (f_{bc} - \sum_{k>l,d} \langle kl || cd \rangle t_{kl}^{bd}) t_{ij}^{ac} \\ & - P(ij) \sum_k (f_{kj} + \sum_{l,c>d} \langle kl || cd \rangle t_{jl}^{cd}) t_{ik}^{ab} \\ & - P(ij)P(ab) \sum_{ck} \langle bk || cj \rangle t_{ik}^{ac} + 2P(ij)P(ab) \sum_{k>l,c>d} \langle kl || cd \rangle t_{ik}^{ac} t_{jl}^{bd}. \end{aligned} \quad (15.10)$$

Here, $\langle pq || rs \rangle$ is an abbreviation for the anti-symmetric two electron integrals

$$\langle pq || rs \rangle = (pr | qs) - (ps | qr). \quad (15.11)$$

f_{pq} represents the Fock matrix

$$f_{pq} = h_{pq} + \sum_i \langle pi || qi \rangle. \quad (15.12)$$

The notation $P(pq)$ indicates an exchange of two indices in an anti-symmetrized manner

$$P(pq)f(p, q) = f(p, q) - f(q, p). \quad (15.13)$$

The T_2 equation is a nonlinear equation that can be solved using fixed-point iteration:

$$(t_{ij}^{ab})_{n+1} = \frac{\sigma((t_{ij}^{ab})_n) + \Delta_{ab}^{ij}(t_{ij}^{ab})_n}{\Delta_{ab}^{ij}}, \quad (15.14)$$

where, $\sigma(t_{ij}^{ab})$ represents the residual, which is computed from the right-hand side of Eq. (15.10). The denominator term is given by

$$\Delta_{ab}^{ij} = f_{ii} + f_{jj} - f_{aa} - f_{bb} + v, \quad (15.15)$$

where v is a level shift constant. The fixed-point iteration (15.14) may encounter convergence issues, particularly when the energy gap between occupied and virtual orbitals is small. A suitable level shift term v can stabilize the iteration. The convergence of the fixed-point iteration can be accelerated using the DIIS method. To stabilize the T_2 amplitudes, one choice for the DIIS error vector is the difference between the T_2 amplitudes of two iterations.

Let's now examine the Python program for CCD. First, we define functions for calculating the CCD correlation energy and updating the T_2 amplitudes according to Eq. (15.14).

```
def get_CCD_corr_energy(H, t2):
    return .25 * einsum('ijab,abij->', H['oovv'], t2)

def update_CCD_amplitudes(H: h5py.Group, t2, level_shift=0):
    nvir, nocc = t2.shape[1:3]
    fock = H['fock']
    foo = fock[:nocc,:nocc]
    fvv = fock[nocc:,:nocc:]
    e_o = foo.diagonal()
    e_v = fvv.diagonal() + level_shift

    Fvv = fvv - 0.5*einsum('k lcd,bdkl->bc', H['oovv'], t2)
    Foo = foo + 0.5*einsum('k lcd,cdjl->kj', H['oovv'], t2)
    Fvv[np.diag_indices(nvir)] -= e_v
    Foo[np.diag_indices(nocc)] -= e_o

    t2out = .25 * np.asarray(H['vvo'])
    t2out -= einsum('bkcj,acik->abij', H['vovo'], t2)
    t2out += .5 * einsum('bc,acij->abij', Fvv, t2)
    t2out -= .5 * einsum('kj,abik->abij', Foo, t2)
    t2out += .5*einsum('k lcd,acik,bdj1->abij', H['oovv'], t2, t2)
    t2out = t2out - t2out.transpose(0,1,3,2)
    t2out = t2out - t2out.transpose(1,0,2,3)
    oooo = 0.5*einsum('k lcd,cdij->ijkl', H['oovv'], t2)
```

```

oooo += np.asarray(H['oooo'])
t2out += 0.5*einsum('ijkl,abkl->abij', oooo, t2)
t2out += 0.5*einsum('abcd,cdij->abij', H['vvvv'], t2)

t2out /= e_o + e_o[:,None] - e_v[:,None,None] - e_v[:,None,None,None]
return t2out

```

The CCD equation involves a large number of index summations that can be effectively computed using tensor contraction programs. Here, we utilize the `einsum` function to perform these tensor contractions. To enhance the readability of the example program, we leverage the anti-symmetric property of the T_2 amplitudes and omit the constraints (such as $i > j$ and $a > b$) between different indices in these tensors. This converts the constrained summation into an unconstrained one, for example:

$$\sum_{i>j,a>b} \langle ij||ab \rangle t_{ij}^{ab} \rightarrow \frac{1}{4} \sum_{ijab} \langle ij||ab \rangle t_{ij}^{ab}. \quad (15.16)$$

However, this modification significantly increases the computational workload. CCD is a computationally intensive application, which includes several tensor contractions with a complexity of N^6 . In real applications, it is necessary to consider the constraints between the indices to reduce the required FLOPs.

The CCD program involves five types of MO integrals, denoted as `vvoo`, `oovv`, `vovo`, `oooo`, and `vvvv`. In these notations, `o` and `v` represent the occupied and virtual orbital indices, respectively, within the four indices of $\langle pq||rs \rangle$. The computation of these integrals is demonstrated in the following functions.

```

import pyscf
def mo_integrals(mol: pyscf.gto.Mole, orbitals, Hfile=None):
    '''MO integrals in physists notation <pq||rs>'''
    no = mol.nelectron
    nmo = orbitals.shape[1]
    eri = np.zeros([nmo*2]*4)
    eri[ ::2, ::2, ::2, ::2] = eri[ ::2, ::2,1::2,1::2] = \
    eri[1::2,1::2, ::2, ::2] = eri[1::2,1::2,1::2,1::2] = \
        pyscf.ao2mo.kernel(mol, orbitals, compact=False).reshape([nmo]*4)
    eri = eri.transpose(0,2,1,3) - eri.transpose(2,0,1,3)

    if Hfile is None:
        Hfile = tempfile.mktemp()
    with h5py.File(Hfile, 'w') as H:
        H['vvoo'] = vvoo = eri[no:,no,:,:no]
        H['oovv'] = vovo.conj().transpose(2,3,0,1)
        H['vovo'] = eri[no,:,:no,no,:,:no]
        H['oooo'] = eri[:no,:no,:,:no]
        H['vvvv'] = eri[no,:,:no,no,:,:]

```

```

        hcore = pyscf.scf.hf.get_hcore(mol)
        hcore = einsum('pq,pi,qj->ij', hcore, orbitals, orbitals)
        hcore_mo = np.zeros([nmo*2]*2)
        hcore_mo[::2,::2] = hcore_mo[1::2,1::2] = hcore
        H['fock'] = hcore_mo + einsum('ipiq->pq', eri[:no,:,:no,:])
    return Hfile

```

To simplify the process of MO integral computation, we have employed the PySCF package to compute the entire MO integral tensor and then extract the required sub-blocks. The integral computation program can be made more efficient by utilizing permutation symmetry in the AO integrals, and by overlapping the I/O operations with computation. We have already covered these technical topics in Chapter 12 and will not repeat here.

The `ao2mo` module in PySCF only produces MO integrals in spatial orbitals, following the chemistry notation ($pr|qs$). The `mo_integrals` function presented above converts the MO integrals from the PySCF convention to the physicist's notation in spin-orbital bases:

$$(pr|qs) - (qr|ps) \rightarrow (pr||qs) \rightarrow \langle pq||rs \rangle.$$

In this conversion, each spatial orbital is mapped to two spin-orbitals

$$\psi_1, \psi_2, \dots \rightarrow \psi_{1\alpha}, \psi_{1\beta}, \psi_{2\alpha}, \psi_{2\beta}, \dots$$

Array indices `0::2` and `1::2` are utilized to access the alpha and beta orbitals in the integral tensor, respectively.

Eventually, we can implement the fixed-point iteration to solve the T_2 amplitudes.

```

import tempfile
import pyscf
from pychem_book.chap13.diis import DIIS

def mp2(H):
    nocc = H['oooo'].shape[0]
    fock = np.asarray(H['fock'])
    e_o = fock.diagonal()[:nocc]
    e_v = fock.diagonal()[nocc:]
    eijab = e_o + e_o[:,None] - e_v[:,None,None] - e_v[:,None,None,None]
    t2 = np.asarray(H['vvoo']) / eijab
    e = get_CCD_corr_energy(H, t2)
    return e, t2

def CCD_solve(mf: pyscf.scf.hf.RHF, conv_tol=1e-6, max_cycle=100):
    '''A fixed-point iteration solver for spin-orbital CCD'''
    mol = mf.mol

```

```

orbitals = mf.mo_coeff
e_hf = mf.e_tot

with tempfile.TemporaryDirectory() as tmpdir:
    Hfile = mo_integrals(mol, orbitals, f'{tmpdir}/H')
    diis = DIIS(f'{tmpdir}/diis')
    e_ccd = e_hf
    with h5py.File(Hfile, 'r') as H:
        e_corr, t2 = mp2(H) # initial guess
        e_ccd = e_hf + e_corr
        print(f'E(MP2)={e_ccd}')

    for cycle in range(max_cycle):
        t2, t2_prev = update_CCD_amplitudes(H, t2), t2
        e_ccd, e_prev = get_CCD_energy(H, t2) + e_hf, e_ccd
        print(f'{cycle}, E(CCD)={e_ccd}, dE={e_ccd-e_prev}')
        if abs(t2 - t2_prev).max() < conv_tol:
            break
        t2 = diis.update(t2 - t2_prev, t2)
    return e_ccd

if __name__ == '__main__':
    mol = pyscf.M(atom='N 0. 0 0; N 1.5 0 0', basis='cc-pvdz')
    mf = mol.RHF().run()
    e_ccd = CCD_solve(mf)

```

To utilize the CCD solver, a mean-field wave function is required as the starting point. For simplicity, we use the PySCF package to generate the mean-field object. Typically, the initial guess for the CCD solver is derived from the MP2 wave function. This initial guess is equivalent to setting the CCD T_2 amplitudes to zero and performing one step of the fixed-point iteration.

15.3 Optimizing data transferring

The `update_CCD_amplitudes` function is the most time-consuming component of the CCD program, primarily due to three bottlenecks:

1. Tensor contractions that scale as the sixth power of the system size. Using n_v to represent the number of virtual orbitals and n_o to represent the number of occupied orbitals, the tensor contraction

```
einsum('abcd,cdij->abij', H['vvvv'], t2)
```

scales as $n_v^4 n_o^2$. Given that $n_v \gg n_o$ in most molecular CCD calculations, this contraction is the most demanding step. Other sixth-order contractions are less costly, including three with $n_o^3 n_v^3$ scaling and two with $n_v^2 n_o^4$ scaling.

2. Reading MO integrals from the disk. The largest integral tensor, `vvvv`, requires reading $8n_v^4$ bytes of data, assuming that the data is stored in double precision.
3. Memory efficiency for transposing tensors in the `einsum` function and other data movement operations. This factor is typically less pronounced compared to the previous two bottlenecks.

The N^6 scaling in tensor contractions might seem impressive. However, in practice, the computational bottleneck often arises from the I/O of reading the `vvvv` tensor. The largest tensor contraction requires a FLOP count of $2n_v^4 n_o^2$ and a data transfer of $8n_v^4$ bytes. As a result, the ratio of computational load to data transfer (FLOPs/bytes) is $\frac{n_o^2}{4}$. On modern multi-core processors, the computational power can easily exceed 1 TFLOPs (Tera Floating-point Operations per second), while a typical Linux file system with SSD storage can provide a data transfer bandwidth of several GB per second. This yields approximately 500 FLOPs per byte transferred. For a system with 50 occupied orbitals, the time required for I/O and tensor contraction is very close ($\frac{50^2}{4} \times \frac{1}{500} = 1.25$). If the tensor contraction is offloaded to a GPU, which can deliver performance exceeding 10 TFLOPs, the I/O time will be 10 times that of the tensor contraction.

Prefetching data before computation is a particularly effective strategy for performance optimization in the CCD program. Regarding the functionality of data prefetching, you may consider using the `iterate_with_prefetch` function developed in Chapter 9. However, this function is not suitable in this context because it is designed to prefetch only one types of data within a for-loop iteration. In the current scenario, the code needs to prefetch five distinct types of data from the disk.

An effective strategy would be to prefetch each of the five data types in advance, ensuring that all of them are readily available in memory when needed. To apply this strategy, the following functionalities need to be implemented:

- A background daemon that continuously reads data, which will be used in the near future, into a memory cache.
- A data loader that retrieves the required data from the cache.
- A monitor to manage the cache size, preventing excessive memory consumption.

This data prefetch scheme is somewhat similar to the functionality of the Python first-in-first-out (FIFO) `Queue` class. Inspired by the `Queue` model, we can design the FIFO cache after addressing the following considerations:

- Which data structure should the cache adopt? The cache is a dictionary that indexes the data by its name, rather than by the order in which the data is passed to the FIFO queue.

- How can we ensure the data is available in the cache before it is accessed by the data loader? We can use a condition variable (`not_empty`) to monitor the status of the cache.
- Given the possibility for the prefetch daemon to use substantial memory, how can we block this daemon to limit the cache size? We can use another condition variable (`not_full`) to pause the prefetch daemon until some data in the cache has been consumed.

The first three points lead to the basic functionality as follows:

```
not_empty = Condition() # condition variables shared by two threads
not_full = Condition()
# the prefetch daemon in thread 1
def prefetch():
    for key in data_names:
        cache[key] = h5obj[key]
        not_empty.notify()
    while len(cache) >= maxsize:
        not_full.wait()

# data loader in thread 2
def loader(key):
    not_empty.wait()
    data = cache.pop(key)
    not_full.notify()
    return data
```

- There is a potential race condition between the prefetch daemon and the loader, as both need to modify the cache. To address this, we need a mutex lock to safeguard the cache dictionary.
- A deadlock scenario might occur if the condition variables in the prefetch daemon and the data loader are waiting for signals from each other at the same time. A wait mechanism must be carefully designed to ensure that the wait methods of the two condition variables are executed exclusively.
- To gracefully terminate the prefetch daemon, it is necessary to execute cleanup code then inform the daemon to terminate. We can use the `contextmanager` in conjunction with the `with` statement to ensure the cleanup code is executed properly.

Here is an example code that incorporates all the aforementioned technical considerations. We have named the function `readahead` to distinguish it from the `iterate_with_prefetch` function discussed in Chapter 9.

```
import threading
from contextlib import contextmanager

#Implement the readahead function using the producer-consumer model.
@contextmanager
```

```
def readahead(h5obj, tasks, maxsize=3):
    assert maxsize > 0
    cache = {}
    # This mutex lock is shared by the prefetch function and loader.
    # It must be held whenever mutating the cache.
    mutex = threading.RLock()                                # (1)
    not_full = threading.Condition(mutex)
    not_empty = threading.Condition(mutex)
    terminate = False

    def prefetch(tasks):
        for task in tasks:
            if terminate:
                break
            # Use a condition variable to block the producer.
            # To prevent the cache storing too many items.
            data = np.asarray(h5obj[task])
            with not_full:
                cache[task] = data
                not_empty.notify()
                while len(cache) >= maxsize:
                    not_full.wait()
        daemon = threading.Thread(target=prefetch, args=(tasks,))
        daemon.start()

    def loader(key):
        with not_empty:
            # wait until the required task has been prepared
            if not cache:
                not_empty.wait()
            data = cache.pop(key)
            not_full.notify()
        return data

    yield loader

    terminate = True
    with not_full:
        # release any locks in prefetch, then the terminate condition in
        # prefetch function will be triggered
        not_full.notify()
    daemon.join()
```

It should be noted that the mutex lock in line (1) is shared by the two condition variables `not_empty` and `not_full`. When they are acquired (by the `with` context), any resources inside the scope of the `with` context are protected by the same mutex lock. Therefore, there is no need to introduce an additional mutex to guard the `cache` object. The `cache` dictionary is accessed by only one thread at a time, until the `wait` method of the condition variable is called. The `wait` method then yields control to other threads. When the `notify` method in other threads is executed, it will awaken the `wait` condition and resume the program flow in the current thread.

The `readahead` function reads and caches data in the order specified by the data names provided in the input argument. It is preferable to prefetch the data in the same order in which it is consumed. To align the prefetch order with the data access order in the `update_CCD_amplitudes` function, we need to thoroughly analyze and organize the data access order. Subsequently, the appropriate name list is passed to the `readahead` daemon. A mismatch in the order could potentially freeze the data loader within the `readahead` daemon. As an exercise, the reader is encouraged to explore the reasons and conditions that might lead to this deadlock.

```
def update_CCD_amplitudes(H: h5py.Group, t2):
    ...
    with readahead(H, [
        'oovv', 'vvoo', 'vovo', 'oooo', 'vvvv',
    ]) as load:
        oovv = load('oovv')
        Fvv = fvv - 0.5*einsum('klcd,bdkl->bc', oovv, t2)
        Foo = foo + 0.5*einsum('klcd,cdjl->kj', oovv, t2)
        Foo[np.diag_indices(nvir)] -= e_v
        Foo[np.diag_indices(nocc)] -= e_o

        t2out = .25 * load('vvoo')
        t2out -= einsum('bkcj,acik->abij', load('vovo'), t2)
        t2out += .5 * einsum('bc,acij->abij', Fvv, t2)
        t2out -= .5 * einsum('kj,abik->abij', Foo, t2)
        t2out += .5*einsum('klcd,acik,bdj1->abij', oovv, t2, t2)
        t2out = t2out - t2out.transpose(0,1,3,2)
        t2out = t2out - t2out.transpose(1,0,2,3)
        oooo = 0.5*einsum('klcd,cdij->ijkl', oovv, t2) + load('oooo')
        t2out += 0.5*einsum('ijkl,abkl->abij', oooo, t2)
        t2out += 0.5*einsum('abcd,cdij->abij', load('vvvv'), t2)

    t2out /= e_o + e_o[:,None] - e_v[:,None,None] - e_v[:,None,None,None]
    return t2out
```

The current scheme for data prefetching has a limitation: the prefetching sequence must be manually managed. This implies that whenever the access order is changed, the task list for the `readahead` function may also need to be updated. Such manual

adjustments are not only cumbersome but also increase the risk of introducing errors. To enhance the usability of the `readahead` function, we can consider employing the Just-In-Time (JIT) compilation technique to dynamically determine the data prefetch order at runtime.

15.4 Just-in-time compilation for I/O `readahead`

The Just-In-Time (JIT) compilation technique may seem complex at first glance. In fact, it is straightforward in its principles and implementation strategy. Essentially, JIT involves optimizing or dynamically generating code paths based on the data types of the input arguments during runtime. Additionally, these optimized code paths are cached for future reuse.

To analyze or optimize the code paths, the JIT technique needs to access and modify the AST (Abstract Syntax Tree) of the function. This process involves the use of the `inspect` and `ast` modules, which were explored in Chapter 5. In the case of the CCD program with the `readahead` function, we can use the AST to search for all data access operations within the `update_CCD_amplitudes` function. We can create the following JIT compiler to implement this functionality.

```
import inspect
def readahead_jit(f):
    # modify and cache the function until the first run
    new_f = None

    @functools.wraps(f)
    def f_with_readahead(*args, **kwargs):
        for i, a in enumerate(args):
            if isinstance(a, h5py.Group):
                break
        else:                                     # (1)
            return f(*args, **kwargs)

        nonlocal new_f
        if new_f is None:
            arg_names = list(inspect.signature(f).parameters)
            new_f = _inject_readahead(f, arg_names[i])
        return new_f(*args, **kwargs)
    return f_with_readahead
```

The `for-else` statement in line (1) excludes the scenario where HDF5 I/O is not required. When a function is invoked with an `h5py` object among the input arguments, the function is forwarded to the `_inject_readahead` function to analyze data access patterns.

This JIT compiler can inject the `readahead` code into the original function. For instance, in the CCD program, the `_inject_readahead` function can prepend a `with` statement at the beginning of `update_CCD_amplitudes` and replace the `h5py` objects with the corresponding cached variables. The following code sample illustrates the rough effects achieved by the JIT compiler.

```
def update_CCD_amplitudes(...):
    with readahead(h5object, [key1, key2, ...]) as _loader:
        ...
        einsum(..., _loader[key1], ...)
        ...
```

To achieve this effect, we first scan the AST to identify I/O operations. For instance, consider the I/O operation `H['vvvv']`, which is an element of type `ast.Subscript`. Typically, the `ast.Subscript` operation contains one of three types of content: a slice, a constant, or an iterable object (such as a tuple or list). In the case of `H['vvvv']`, the subscript is a constant string. To identify code statements of this type, we can traverse the AST and filter for elements of `ast.Subscript` where the name matches the keyword (e.g., `H`), and the content is a string constant.

The `ast` module provides two primary methods for traversing the AST. One such method is using the `ast.walk` function. However, when traversing the AST via `ast.walk`, nodes may not appear in the same order as they are in the source code. Here, it is more appropriate to use the `ast.NodeVisitor` to maintain the order, as it traverses the AST in a depth-first manner.

```
def _search_tasks(f, keyword):
    tasks = []
    class IdentifyTask(ast.NodeVisitor):
        def visit_Subscript(self, node):
            if ((isinstance(node.value, ast.Name)
                and node.value.id == keyword
                and isinstance(node.slice, ast.Constant)
                and isinstance(node.slice.value, str)):
                tasks.append(node.slice.value)
            return self.generic_visit(node)
    IdentifyTask().visit(ast.parse(inspect.getsource(f)))
    print('# Readahead_jit finds tasks', tasks)
    return tasks
```

Having obtained the list of I/O operations, we proceed to inject the `readahead` function through the AST of the `update_CCD_amplitudes` function. According to the AST documentation for the `with` statement, the following code block

```
with f() as var:
    code_block
```

compiles into the AST node: `With(withitem("f()", var), body=code_block)`. We can create the `ast.With` node for the `readahead` function using this structure. Next, we need to traverse the entire AST to locate and replace the I/O operations associated with `h5py` objects. This process requires the use of the `ast.NodeTransformer` to modify the AST. After the transformation, the AST is converted back into executable Python code using the `exec` function. Below is the implementation for these processes.

```
def _inject_readahead(f, keyword):
    tasks = _search_tasks(f, keyword)
    loader = f'_{keyword}_loader'
    assert loader not in f.__code__.co_varnames
    tree = ast.parse(inspect.getsource(f))
    fn_node = tree.body[0]
    fn_node.decorator_list = [] # remove the "@readahead" decorator
    fn_node.body = [
        ast.With([
            ast.withitem(
                ast.parse(f'readahead({keyword}, {tasks})',
                          mode='eval').body,
                ast.Name(loader)), # as loader
            fn_node.body)
    ]

    class RewriteGetitem(ast.NodeTransformer):
        def visit_Subscript(self, node):
            if (isinstance(node.value, ast.Name)
                and node.value.id == keyword
                and isinstance(node.slice, ast.Constant)
                and isinstance(node.slice.value, str)):
                node.value.id = loader
            return self.generic_visit(node)

    new_tree = ast.fix_missing_locations(RewriteGetitem().visit(tree))
    print(f'# Modified {f.__name__} function')
    print(ast.unparse(new_tree))
    exec(ast.unparse(new_tree))
    return locals()[f.__name__]
```

What if the I/O operations involve duplicated objects? Duplicated I/O tasks do not need to be repeatedly prefetched. We can optimize the `readahead` function to incorporate this property.

```
@contextmanager
def readahead(h5obj, tasks, maxsize=3):
    assert maxsize > 0
```

```
cache = {}
mutex = threading.RLock()
not_full = threading.Condition(mutex)
new_slot = threading.Condition(mutex)
terminate = False

class Data:
    def __init__(self, data):
        self.data = data
        self.refcount = 1

def prefetch(tasks):
    for task in tasks:
        if terminate:
            break

        with mutex:
            if task in cache:
                cache[task].refcount += 1
                continue

            data = Data(np.asarray(h5obj[task]))
            with not_full:
                cache[task] = data
                new_slot.notify()
                if len(cache) >= maxsize:
                    not_full.wait()

    daemon = threading.Thread(target=prefetch, args=(tasks,))
    daemon.start()

class Loader:
    def __getitem__(self, key):
        with new_slot:
            while key not in cache:
                if len(cache) >= maxsize:
                    raise RuntimeError('Cache size insufficient')
                new_slot.wait()
            data = cache[key].data
            cache[key].refcount -= 1
            if cache[key].refcount <= 0:
                cache.pop(key)
                not_full.notify()

        return data
```

```

yield Loader()

terminate = True
with not_full
    not_full.notify()
daemon.join()

```

If a task is already cached, we can simply add a reference to the existing task instead of performing a fresh read, as indicated in line (1). Unlike the previous version with the FIFO cache, in this version, we only modify the reference count in the `loader` method, as shown in line (2). The task will be evicted from the cache when its reference count reaches zero, as processed in line (3).

So far, we have implemented a simple JIT compiler to improve the data access mechanism for the CCD program. To enable JIT compilation, we simply apply the `readahead_jit` decorator to the target function `update_CCD_amplitudes`. There is no need to make any modifications to the code implementation of `update_CCD_amplitudes`.

```

@readahead_jit
def update_CCD_amplitudes(H: Union[Dict, h5py.Group], t2: np.ndarray):
    ...
    Fvv = fvv - 0.5*einsum('klcd,bdkl->bc', H['oovv'], t2)
    Foo = foo + 0.5*einsum('klcd,cdjl->kj', H['oovv'], t2)
    t2out = .25 * H['vvoo']
    t2out -= einsum('bkcj,acik->abij', H['vovo'], t2)
    ...

```

15.5 Symbolic programming for coupled cluster theory

Coupled cluster theory, particularly in its higher excitation variants, involves complex equations. The systematic methods for deriving CC equations are well-established, utilizing the linked diagram method [1]. Additionally, there are programs available that automate the derivation of CC diagrams [7].

Suppose we lack an established theoretical framework. How can we derive the CC equations? To explore this question, we use the CCD as an example to demonstrate the relevant symbolic computation and code generation techniques. SymPy provides the `secondquant` module, which can also be used to derive coupled cluster equations. However, the methodology presented here offers a more general approach. By following the outlined development procedure, we can create symbolic programs to derive equations for algebras that are not available in SymPy.

Essentially, the symbolic program for the CC equations is a CAS (Computer Algebra System), designed to perform the normal ordering procedure for second

quantization operators. Generally, the development of a CAS program involves the following steps:

1. Define the basic elements of the problem.
2. Implement computation rules for operations acting on basic elements.
3. Create an *eval* function that can recursively reduce the expressions until the results are derived.

15.5.1 Defining fundamental symbolic elements

The following classes are the basic elements in the CCD symbolic program:

- A `FieldOperator` class to represent a second quantized operator. The `FieldOperator` in CCD involves both occupied and virtual orbitals. Therefore, we assign a label to `FieldOperator` that can take the values `O` (for occupied), `V` (for virtual), or `*` (indicating both). The `Annihilation` and `Creation` operators are defined as subclasses of `FieldOperator`.
- A `Delta` class to represent the Kronecker delta, which arises from the commutation of two second quantized operators.
- A `Tensor` class to represent the multi-dimensional coefficients and integrals used in the \hat{T} excitation operator and the Hamiltonian.

The code for each class is listed below.

```
class FieldOperator:
    '''Annihilation or creation operators'''
    def __init__(self, label='*'):
        assert label in 'OV*'
        self.label = label

class Annihilation(FieldOperator):
    def __repr__(self):
        return f'{self.label}'

class Creation(FieldOperator):
    def __repr__(self):
        return f'{self.label}^+'

class Delta:
    '''the Kronecker delta.'''
    def __init__(self, indices: List[FieldOpeartor]):
        self.indices = indices

    def __repr__(self):
        ops = ' '.join(op.label for op in self.indices)
        return f'Delta({ops})'
```

```

class Tensor:
    def __init__(self, indices: List[FieldOp], label=None):
        self.indices = indices
        self.label = label

    def __repr__(self):
        ops = '\n'.join(op.label for op in self.indices)
        return f'{self.label or self.__class__.__name__}({ops})'

```

15.5.2 Implementing computation rules

The second quantization operators obey the following commutation relations:

$$[\hat{a}_p^\dagger, \hat{a}_q^\dagger]_+ = 0, \quad (15.17)$$

$$[\hat{a}_p, \hat{a}_q]_+ = 0, \quad (15.18)$$

$$[\hat{a}_p^\dagger, \hat{a}_q]_+ = [\hat{a}_p, \hat{a}_q^\dagger]_+ = \delta_{pq}. \quad (15.19)$$

These relations provide the basis for the contraction operations, which transforms time-ordered operators into their normal-ordered representations:

$$\hat{f}_1 \hat{f}_2 = C(\hat{f}_1, \hat{f}_2) + \hat{f}_2 \hat{f}_1. \quad (15.20)$$

The value of $C(\hat{f}_1, \hat{f}_2)$ is zero if the `FieldOperators` are commutable, which occurs under either of the following conditions:

- The operators are of the same type. They are either both annihilation operators or both creation operators.
- The operators have different occupancy labels. One targets an occupied orbital, while the other targets a virtual orbital.

These conditions are translated into the following `commutable` function:

```

COMMUTABLE = {'V': 'O', 'O': 'V', '*': '*'}

def commutable(op1: FieldOp, op2: FieldOp):
    return (op1.__class__ == op2.__class__ or
            op1.label == COMMUTABLE[op2.label])

```

We then translate Eq. (15.20) into the `contract` function for two `FieldOperators` as follows:

```

def _contract_fop_fop(
    fop1: FieldOperator, fop2: FieldOperator) -> List[String]:
    output = []
    if not commutable(fop1, fop2):

```

```

        output.append(String([], 1, [Delta([fop1, fop2])]))
        output.append(String([fop2], -1, remaining=[fop1]))
    return output

```

To facilitate the management of multiple second quantized operators, we introduce the `String` class:

```

class String:
    ...
    Normal-ordered FieldOpearctors
    ...
    __slots__ = ['operators', 'factor', 'deltas', 'remaining']      # (1)

    def __init__(self, operators, factor=1, deltas=None, remaining=None):
        self.operators: List[FieldOperator] = operators
        self.factor = factor
        self.deltas = deltas or []
        self.remaining: List[FieldOperator] = remaining or []

    def __repr__(self):
        ops = self.operators + self.deltas + self.remaining
        ops = ' '.join(repr(op) for op in ops)
        return f'{self.__class__.__name__}({ops})'

```

Let's examine the attributes in line (1). The attributes `operators`, `deltas`, and `factor` have clear meanings. They represent a set of `FieldOperators`, a set of Kronecker `deltas` generated by contraction, and the coefficient associated with the entire string, respectively. So, what is the purpose of the `remaining` attribute?

The attribute `remaining` is used to store any dangling operators after contraction. Consider the contraction between a `FieldOperator` \hat{f}_1 and a sequence of operators:

$$\hat{f}_1 \hat{f}_2 \hat{f}_3 \dots = C(\hat{f}_1, \hat{f}_2) \hat{f}_3 \dots + \hat{f}_2 \hat{f}_1 \hat{f}_3 \dots \quad (15.21)$$

After transforming the product $\hat{f}_1 \hat{f}_2$ into the normal-ordered form as shown in Eq. (15.20), the contraction of \hat{f}_1 with subsequent operators can continue. The `remaining` attribute keeps track of the operators that are still available for further contraction. Using the `remaining` attribute, we can implement the normal ordering process described in Eq. (15.21) with the following recursive function.

```

def _contract_fop_string(
    fop: FieldOperator, string: String) -> List[String]:
    operators = string.operators
    remaining = string.remaining
    factor = string.factor
    deltas = string.deltas
    if len(operators) == 0:

```

```

        return [String(operators, factor, deltas, [fop]+remaining)]

# C(x, a1*a2*...) = C(x, a1)*a2*a3*...
#           = delta(x,a1)*a2*a3*... + C(remaining, a2*a3*...)
output = []
for s in _contract_fop_fop(fop, operators[0]):
    if not s.remaining:
        output.append(String(operators[1:], s.factor * factor,
                             s.deltas + deltas, remaining))
    else:
        substring = String(operators[1:], factor, deltas, remaining)
        for r in _contract_fop_string(s.remaining[0], substring): # (1)
            output.append(String(s.operators + r.operators,
                                 s.factor * r.factor,
                                 r.deltas, r.remaining))
return output

```

For the output string from `_contract_fop_fop`, if its `remaining` attribute is empty, it indicates that the string does not contain any operators that can be further contracted. Otherwise, we recursively call the `_contract_fop_string` function to handle the subsequent contraction for the `FieldOperator` objects within the `remaining` attribute, as shown in line (1).

On top of the `_contract_fop_string` function, we can readily implement a function to simulate the string-string contraction.

$$(\hat{f}_1 \hat{f}_2 \dots)(\hat{g}_1 \hat{g}_2 \dots) = \dots + (\hat{g}_1 \hat{g}_2 \dots)(\hat{f}_1 \hat{f}_2 \dots) \quad (15.22)$$

```

import itertools
def flatten(lst):
    return list(itertools.chain(*lst))

def _contract_string_string(string1, string2) -> List[String]:
    assert isinstance(string1, String) and not string1.remaining
    assert isinstance(string2, String) and not string2.remaining
    if not string2.operators:
        return [String(string1.operators,
                      string1.factor * string2.factor,
                      string1.deltas + string2.deltas)]
    output = [String(string2.operators,
                     string1.factor * string2.factor,
                     string1.deltas + string2.deltas)]
    for fop in reversed(string1.operators):
        # C(x1*x2*..., a1*a2*...) = C(x1, C(x2, ..., C(xn, a1*a2*...)))

```

```

        output = flatten(_contract_fop_string(fop, s) for s in output)
    return output

def contract(string1, string2):
    return [String(s.operators + s.remaining, s.factor, s.deltas)
            for s in _contract_string_string(string1, string2)]

```

So far, we have completed the necessary algebra for the `FieldOperator` and `String` classes. Next, we will define the Hamiltonian operator and the \hat{T}_2 excitation operators in CCD theory using these foundational elements. The second quantized Hamiltonian is formulated as follows:

$$H = \sum_{pq} h_{pq} q_p^\dagger \hat{a}_q + \frac{1}{4} \sum_{pqrs} \langle pq || rs \rangle \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r. \quad (15.23)$$

Please note that we have adopted the two-electron integral tensor in its anti-symmetrized form. The rationale for employing anti-symmetric tensors will become clearer in the subsequent discussion. The symbolic representation of the Hamiltonian can be easily constructed using the `String` class and the `Tensor` class.

Similar methods can be used to construct the \hat{T}_2 operator.

$$\hat{T}_2 = \frac{1}{4} \sum_{abij} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i. \quad (15.24)$$

The projector μ in CCD equation is expressed as

$$\langle \Phi_{ij}^{ab} | = \langle \Phi_0 | \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_b \hat{a}_a. \quad (15.25)$$

Below is the code implementation for these operators.

```

def _make_operator(factor, indices, label=None):
    n = len(indices)
    assert n % 2 == 0
    operators_c = [Creation(x) for x in indices[:n//2].upper()]
    operators_a = [Annihilation(x) for x in indices[n//2:].upper()]
    operators = operators_c + operators_a[::-1]
    tensor = Tensor(operators_c + operators_a, label)
    return tensor, String(operators, factor)

def hamiltonian(n, factor):
    return _make_operator(factor, '**'*n, f'h{n}')

def excitation(n):
    return _make_operator(.25, 'V'*n + '0'*n, f'T{n}')

def deexcitation(n):

```

```

    return _make_operator(.25, '0'*n + 'V'*n, f'T{n}')

def bra_projector(n):
    bra_tensor, bra = _make_operator(.25, '0'*n + 'V'*n)
    return bra_tensor.indices[::-1], bra

```

In these operators, we utilize the same set of `FieldOperator` objects for both the tensor indices and the string operators. To establish a correspondence between the `FieldOperators` and the tensor indices, each `FieldOperator` requires a unique identifier. These identifiers are then used to identify the tensor indices. Since Python inherently generates a unique ID for each distinct `FieldOperator` object, for simplicity, we use the `FieldOperator` object itself as the identifier for the tensor indices. This approach implies that each `FieldOperator` object is unique and cannot be cloned, as any `copy()` method would result in a new `FieldOperator` object with a different ID.

Having derived the symbolic representation of the Hamiltonian and the \hat{T}_2 operators, we can now proceed to compute \hat{H}_{eff} . This effective Hamiltonian can be evaluated using the Baker-Campbell-Hausdorff (BCH) expansion:

$$\hat{H}_{\text{eff}} = e^{-\hat{T}} \hat{H} e^{\hat{T}} = \hat{H} + [\hat{H}, \hat{T}] + \frac{1}{2!} [[\hat{H}, \hat{T}], \hat{T}] + \dots \quad (15.26)$$

In the framework of coupled cluster theory, the BCH expansion can be truncated at the fourth order. The BCH expansion is simulated in the following Python code:

```

def apply_BCH(truncation=4):
    hs = [hamiltonian(1, 1), hamiltonian(2, .25)]
    for order in range(truncation+1):
        factor = 1./math.factorial(order)
        for htensor, hstring in hs:
            ht_tensors = [htensor]
            ht_strings = [String(hstring.operators,
                                 hstring.factor * factor)]
            for i in range(order):
                amplitudes, t = excitation(2)
                ht_tensors.append(amplitudes)
                ht_strings = flatten(commute(ht, t) for ht in ht_strings)
            for ht_string in ht_strings:
                yield ht_tensors, ht_string

```

The `commute` function in this program evaluates the commutator between two strings.

```

def commute(string1, string2) -> List[String]:
    output = contract(string1, string2)
    n_deltas = len(string1.deltas) + len(string2.deltas)
    return [s for s in output if len(s.deltas) > n_deltas]

```

Both the Hamiltonian and the \hat{T} operators contain an even number of `FieldOperators`. Fully swapping the positions of two strings using the `contract` function requires an even number of permutations, which results in an overall factor of 1.0. This cancels out the other term in the commutator.

$$[\hat{f}_1 \hat{f}_2 \dots, \hat{g}_1 \hat{g}_2 \dots] = \dots + (\hat{g}_1 \hat{g}_2 \dots)(\hat{f}_1 \hat{f}_2 \dots) - (\hat{g}_1 \hat{g}_2 \dots)(\hat{f}_1 \hat{f}_2 \dots).$$

Therefore, we can simply scan the output of the `contract` function, searching for the item where the two input strings have been completely swapped, and remove it from the output. In this `commute` function, we identify the swapped strings based on the number of Kronecker deltas, since swapping two strings does not change the number of Kronecker deltas, whereas any contractions will increase this number.

15.5.3 Evaluating expressions symbolically

We can evaluate the value of a contracted string along with its associated `Tensor` coefficients by projecting it onto the HF reference. This is implemented in the following `on_HF_reference` function.

```
VSYMBOLS = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
OSYMBOLS = 'ijklmnopqrstuvwxyzJKLMNOPQRSTUVWXYZ'

def on_HF_reference(string, tensors, output_indices=None) -> List[str]:
    '''<HF|string|HF>'''
    h_ops = []
    for op in string.operators:
        if op.label == '*':
            h_ops.append(op)
            continue
        # Creation or annihilation operators in T are normal ordered.
        # They must be zero when applied either on |HF> or on <HF|
    return ''

deltas = string.deltas
if h_ops:
    # The remaining operators in Hamiltonian
    if (len(h_ops) == 2 and
        isinstance(h_ops[0], Creation) and
        isinstance(h_ops[1], Annihilation)):
        deltas = deltas + [Delta(h_ops)]
    elif len(h_ops) == 4:
        return f'{string.factor*2} * einsum("ijij->", h2_oooo)'
    else:
        return ''
```

```

vsymbols, osymbols = iter(VSYMBOLS),iter(OSYMBOLS)
symbol_table = {}
for delta in deltas: # (1)
    op1, op2 = delta.indices
    labell, label2 = op1.label, op2.label
    if labell == 'V' or label2 == 'V':
        symbol_table[op1] = symbol_table[op2] = next(vsymbols)
    else:
        symbol_table[op1] = symbol_table[op2] = next(osymbols)

if output_indices is None:
    output_script = ''
else:
    output_script = ''.join(symbol_table[i] for i in output_indices)
scripts = []
operands = []
for tensor in tensors:
    s = ''.join(symbol_table[i] for i in tensor.indices)
    scripts.append(s)
    if tensor.label[0] == 'h':
        operands.append(
            tensor.label + '_' +
            ''.join('v' if x in VSYMBOLS else 'o' for x in s))
    else:
        operands.append(tensor.label)
subscripts = f'",".join(scripts)}->{output_script}'
operands = ', '.join(operands)
return f'{string.factor} * einsum("{subscripts}", {operands})'

```

The key points of this function include:

- The string may contain operators `Creation('*')` and `Annihilation('*')` that originate from the Hamiltonian operator. These operators should only appear at the rightmost position of the contracted string. When applied to the HF reference, they result in a summation over the occupied indices in the integrals. If the string contains four '*'-labeled `FieldOperator` objects, they must all originate from the two-electron parts of the Hamiltonian. This term corresponds to the two-electron energy of the HF determinant, given by the expression:

$$\frac{1}{2} \sum_{ij} \langle ij || ij \rangle.$$

- If there are any `Annihilation('0')` or `Creation('V')` operators remaining in the string, the value of the string would be zero. This is because applying these operators to the HF reference state on the bra $\langle \Psi_0 |$ results in a null state.

- The code block near line (1) identifies the tensor indices that are associated with Delta objects within the given String object.
- The output can be presented in various formats. For simplicity, we choose to represent the output using the `einsum` scripts.

15.5.4 Generating CCD equations symbolically

For each string from the output of `apply_BCH`, we can invoke the `on_HF_reference` function to obtain the CCD energy expression. To derive the equations for CCD amplitudes, we first project these strings onto the double excitation bra projector and then call `on_HF_reference`, as demonstrated in the following code.

```
def CCD_energy(truncation=4):
    output = []
    for ht_tensors, ht_string in apply_BCH(truncation):
        output.append(on_HF_reference(ht_string, ht_tensors))
    return [x for x in output if x]

def CCD_equations(truncation=4):
    output_indices, bra = bra_projector(2)
    output = []
    for ht_tensors, ht_string in apply_BCH(truncation):
        for s in contract(bra, ht_string):
            output.append(on_HF_reference(s, ht_tensors, output_indices))
    return [x for x in output if x]

if __name__ == '__main__':
    for x in CCD_energy(2):
        print(f'E += {x}')

    for x in CCD_equations(2):
        print(f'F2 += {x}')
```

The output displays

```
E += 1.0 * einsum("ii->", h1_oo)
E += 0.5 * einsum("ijij->", h2_oooo)
E += 0.0625 * einsum("ijba,baj->", h2_oovv, T2)
E += -0.0625 * einsum("ijba,baji->", h2_oovv, T2)
E += -0.0625 * einsum("ijba,abij->", h2_oovv, T2)
E += 0.0625 * einsum("ijba,abji->", h2_oovv, T2)
F2 += 0.0625 * einsum("baij->abji", h2_vvoo)
F2 += -0.0625 * einsum("baji->abji", h2_vvoo)
F2 += -0.0625 * einsum("abij->abji", h2_vvoo)
... # 705 more lines for F2
```

You might notice that the symbolic program operates slowly for higher-order truncations. This is primarily due to the large number of intermediate strings generated by the program. In fact, for the majority of these intermediate strings, the leftmost operator is either `Annihilation('0')` or `Creation('V')`. Regardless of the remaining part of the string and the subsequent contractions they undergo, their expectation value on the HF reference is zero. We can detect these intermediate strings early and exclude them from the final expression.

This optimization can be achieved using *lazy evaluation*. It is not necessary to generate all strings for `on_HF_reference` to process at once. Instead, the action for each `FieldOperator` object in the string can be gradually triggered by the for-loop within the `on_HF_reference` function. Once the zero-value condition in `on_HF_reference` is met, the subsequent actions within the `contract` function can be safely skipped. The lazy evaluation technique can be implemented using generators and the `yield` statement. This technique will be extensively utilized in the subsequent code examples.

Another observation from the output is that it contains significantly more terms than those specified in the CCD equation (15.10). The extra terms arise due to the lack of permutation symmetry treatments during the contraction procedure. Next, we will enhance the symbolic program by considering the permutation symmetry for the CCD tensors.

15.5.5 Applying permutation symmetry to CCD tensors

How can we apply the permutation symmetry associated with the tensor in the contraction function? To incorporate the permutation symmetry, we need to encode additional information into the `String` object. Here is one approach to represent the symmetry among `FieldOperators`:

1. Introduce a `F0pGroup` class to group `FieldOperator` objects that are related through the permutation symmetry of the tensor indices. For the `FieldOperator` objects within the same `F0pGroup`, their indices are identical due to the symmetry.
2. Modify the `operators` attribute of the `String` class to accommodate both `FieldOperator` objects and `F0pGroup` objects.

```
class F0pGroup(tuple):
    pass

class String:
    ...
    Normal-ordered FieldOperatros
    ...
    __slots__ = ['operators', 'factor', 'deltas', 'remaining']

    def __init__(self, operators, factor=1, deltas=None, remaining=None):
        self.operators: List[Union[FieldOperator, F0pGroup]] = operators
```

```

    self.factor = factor
    self.deltas = deltas or []
    self.remaining = remaining or []

    def __repr__(self):
        ops = ''.join(repr(op) for op in self.operators + self.deltas)
        return f'{self.__class__.__name__}({ops})'

```

When contracting two strings, the `FOpGroup` object should be treated as a single entity, which can contract with other elements (either a `FieldOperator` or another `FOpGroup`). As a result, there are four contraction combinations:

- `_contract_fop_fop` for contracting two `FieldOperator` objects;
- `_contract_fop_fgroup` for contracting a `FieldOperator` with a `FOpGroup`;
- `_contract_fgroup_fop` for contracting a `FOpGroup` with a `FieldOperator`;
- `_contract_fgroup_fgroup` for contracting two `FOpGroup` objects.

We have already discussed the first scenario previously. In the following, let us focus on the latter three contraction scenarios.

If a `FieldOperator` object does not commute with the operators within the `FOpGroup` object, the `_contract_fop_fgroup` function will yield an output consisting of $m + 1$ strings. Among these $m + 1$ strings, m of them are contracted strings that include a `Delta` object. Given the permutation symmetry among the m operators within the `FOpGroup`, the m contracted strings will have the same contribution to the result. Therefore, it is sufficient to just consider one of these cases in the output, in which the contributions from the other $m - 1$ strings can be combined and included in the `factor` attribute.

```

def _contract_fop_fgroup(fop: FieldOperator, fgroup: FOpGroup) -> List[String]:
    degeneracy = len(fgroup)
    op = fgroup[0]
    if not commutable(fop, op):
        yield _fgroup2str(fgroup[1:], degeneracy, [Delta([fop, op])])
    yield String([fgroup], (-1)**degeneracy, remaining=[fop])

def _fgroup2str(fgroup, factor, deltas, remaining=None):
    if len(fgroup) > 1:
        operators = [FOpGroup(fgroup)]
    else:
        operators = list(fgroup)
    if remaining:
        if len(remaining) > 1:
            remaining = [FOpGroup(remaining)]
    else:
        remaining = list(remaining)

```

```
    return String(operators, factor, deltas, remaining)
```

It is possible that the operators in the output string contain a subset of the `FOpGroup`. The subset may still form a `FOpGroup` object. A helper function `_fgroup2str` is introduced to handle the subset of a `FOpGroup`, which constructs an appropriate `String` object based on the subset.

The `_contract_fgroup_fop` function is similar to `_contract_fop_fgroup`. Its output also includes an m -fold degenerated string, corresponding to the contraction between the m operators in `FOpGroup` and a `FieldOperator` object.

```
def _contract_fgroup_fop(fgroup: FOpGroup, fop: FieldOperator) -> List[  
    String]:  
    degeneracy = len(fgroup)  
    op = fgroup[-1]  
    if not commutable(fop, op):  
        yield _fgroup2str([], degeneracy, [Delta([op, fop])], fgroup[:-1])  
    yield String([fop], (-1)**degeneracy, remaining=[fgroup])
```

The `_contract_fgroup_fgroup` function is more complex. Due to the presence of multiple `FieldOperator` objects within the two `FOpGroups`, it may result in a variety of contracted strings. However, we can simplify the implementation by directly considering the final state of the contraction output. Thanks to the permutation symmetry within the `FOpGroup`, the output of the contraction can contain at most t different contracted strings, where t is the largest integer that does not exceed the size of the input `FOpGroup` objects. These t different contracted strings can be distinguished by the number of `Delta` objects, ranging from one `Delta` object to t objects. For each contracted string, we can construct it using the simplest contraction pattern and then assign a proper multiplier to account for the degeneracy. The level of degeneracy is equal to the number of different possible combinations to form the `Delta` objects. For instance, a contracted string containing k `Delta` objects can involve $\binom{m}{k}$ different choices of the operators from the first `FOpGroup` and $\binom{n}{k}$ choices from the second `FOpGroup`, with $k!$ combinations for each choice. This results in a degeneracy of $k! \binom{m}{k} \binom{n}{k}$.

```
def _contract_fgroup_fgroup(fgroup1: FOpGroup, fgroup2: FOpGroup) -> List[  
    String]:  
    degen1 = len(fgroup1)  
    degen2 = len(fgroup2)  
    if not commutable(fgroup1[0], fgroup2[0]):  
        for k in range(1, min(degen1, degen2)+1):  
            deltas = [Delta([x1, x2])  
                      for x1, x2 in zip(reversed(fgroup1[:k]), fgroup2[:k])]  
            factor = (math.comb(degen1, k) * math.comb(degen2, k)  
                      * math.factorial(k) * (-1)**((degen1-k)*degen2))  
            yield _fgroup2str(fgroup2[k:], factor, deltas, fgroup1[k:])
```

```
yield String([fgroup2], (-1)**(degen1*degen2), remaining=[fgroup1])
```

On top of the four contraction functions for `FieldOperator` and `FOpGroup`, we can implement the `_contract_fop_string` function.

```
def _contract_fop_string(fop: Union[FieldOperator, FOpGroup], string:
    String) -> List[String]:
    operators = string.operators
    remaining = string.remaining
    factor = string.factor
    deltas = string.deltas
    if len(operators) == 0:
        yield String(operators, factor, deltas, [fop]+remaining)
        return

    op = operators[0]
    match (isinstance(fop, FOpGroup), isinstance(op, FOpGroup)):
        case (True, True): fcontract = _contract_fgroup_fgroup
        case (True, False): fcontract = _contract_fgroup_fop
        case (False, True): fcontract = _contract_fop_fgroup
        case (False, False): fcontract = _contract_fop_fop

    for s in fcontract(fop, op):
        if not s.remaining:
            yield String(s.operators + operators[1:], s.factor * factor,
                         s.deltas + deltas, remaining)
        else:
            substring = String(operators[1:], s.factor * factor,
                               s.deltas + deltas, remaining)
            for r in _contract_fop_string(s.remaining[0], substring):
                yield String(s.operators + r.operators,
                            r.factor, r.deltas, r.remaining)
```

Moving on to the construction of the Hamiltonian, the \hat{T}_2 excitation operator, and the bra projector, we also need to utilize the `FOpGroup` class to incorporate the information regarding permutation symmetry. It should be noted that the projector μ in the CCD equation is subject to the anti-symmetric permutation symmetry. It can be represented in an anti-symmetric form as well.

$$|\Phi_{ij}^{ab}\rangle = |\Phi_{ji}^{ba}\rangle = -|\Phi_{ji}^{ab}\rangle = -|\Phi_{ij}^{ba}\rangle. \quad (15.27)$$

```
def _make_operator(factor, indices, label=None, symmetric=False):
    n = len(indices)
    assert n % 2 == 0
    operators_c = [Creation(x) for x in indices[:n//2].upper()]
```

```

operators_a = [Annihilation(x) for x in indices[n//2:].upper()]
if symmetric and n > 2:
    operators = [FOpGroup(operators_c), FOGroup(operators_a[::-1])]
else:
    operators = operators_c + operators_a[::-1]
tensor = Tensor(operators_c + operators_a, label)
return tensor, String(operators, factor)

def hamiltonian(n, factor):
    return _make_operator(factor, '**'*n, f'h{n}', symmetric=True)

def excitation(n):
    return _make_operator(.25, 'V'*n + 'O'*n, f'T{n}', symmetric=True)

def deexcitation(n):
    return _make_operator(.25, 'O'*n + 'V'*n, f'T{n}', symmetric=True)

def bra_projector(n):
    bra_tensor, bra = _make_operator(.25, 'O'*n + 'V'*n, symmetric=True)
    return bra_tensor.indices[::-1], bra

```

By running the symmetry-aware symbolic program, we can obtain a concise symbolic output. It is not hard to verify that the permutation symmetry of the Hamiltonian and \hat{T}_2 operators are properly addressed in the results.

```

E += 1.0 * einsum("ii->", h1_oo)
E += 0.5 * einsum("ijij->", h2_oooo)
E += 0.25 * einsum("jiab,abji->", h2_oovv, t2)
F2 += 0.25 * einsum("abji->baij", h2_vvoo)
F2 += -0.5 * einsum("ac,cbjj->abij", h1_vv, t2)
F2 += 0.5 * einsum("ki,abjk->baji", h1_oo, t2)
F2 += -1.0 * einsum("kaic,cbjk->abji", h2_ovov, t2)
F2 += 0.5 * einsum("akkc,cbji->abij", h2_voov, t2)
F2 += 0.125 * einsum("abcd,cdji->baij", h2_vvvv, t2)
F2 += -0.5 * einsum("klli,abjk->baji", h2_oooo, t2)
F2 += 0.125 * einsum("lkji,abl->baij", h2_oooo, t2)
F2 += 0.5 * einsum("lkcd,dail,cbjk->abji", h2_oovv, t2, t2)
F2 += -0.125 * einsum("lkcd,dalk,cbji->abij", h2_oovv, t2, t2)
F2 += -0.125 * einsum("lkcd,daji,cblk->abij", h2_oovv, t2, t2)
F2 += -0.125 * einsum("lkcd,cdil,abjk->baij", h2_oovv, t2, t2)
F2 += 0.03125 * einsum("lkcd,cdji,abl->baij", h2_oovv, t2, t2)
F2 += -0.125 * einsum("lkcd,abil,cdjk->baji", h2_oovv, t2, t2)
F2 += 0.03125 * einsum("lkcd,abl,>cbji->baij", h2_oovv, t2, t2)

```

Given that we have assumed anti-symmetry for the bra projectors, it implies that the F_2 tensor should be anti-symmetrized before using it in a real CCD program.

```
F2 = F2 - F2.transpose(0,1,3,2)
F2 = F2 - F2.transpose(1,0,2,3)
```

Additionally, we can find the trace of the normal-ordered Hamiltonian in the results, even though we did not use any coupled cluster (CC) diagram methods to parameterize the Hamiltonian in normal order. By utilizing the anti-symmetry characteristic of the two-electron tensors, the Fock matrix can be recognized in the following code:

```
F2 += -0.5 * einsum("ac,cbji->abij", h1_vv, t2)
F2 += 0.5 * einsum("akkc,cbji->abij", h2_voov, t2)
F2 += 0.5 * einsum("ki,abjk->baji", h1_oo, t2)
F2 += -0.5 * einsum("klli,abjk->baji", h2_oooo, t2)
```

In this output, we still observe duplicated tensor contractions, which stem from the symmetry between different \hat{T}_2 operators. By rearranging the t_2 tensors and merging common terms, the CCD equations (15.10) can be reproduced. Although more complexly, the symmetry between different t_2 strings can be addressed in a similar fashion to the treatment of the symmetry among `FieldOperators`. We do not intend to discuss this symmetry further in the book.

We can extend this symbolic program to accommodate the formalism for other quantum chemical methods.

- By employing the spin-traced excitation operators to define the Hamiltonian

$$\hat{H} = \sum_{pq} h_{pq} \hat{E}_q^p + \sum_{pqrs} (pq|rs) \hat{e}_{qs}^{pr}, \quad (15.28)$$

$$\hat{E}_q^p = \hat{a}_{p\alpha}^\dagger \hat{a}_{q\alpha} + \hat{a}_{p\beta}^\dagger \hat{a}_{q\beta}, \quad (15.29)$$

$$\hat{e}_{qs}^{pr} = \hat{E}_q^p \hat{E}_s^r - \delta_{rq} \hat{E}_s^p, \quad (15.30)$$

and adding the rule for the commutator of spin-traced excitation operators

$$[\hat{E}_q^p, \hat{E}_s^r]_+ = \delta_{rq} \hat{E}_s^p - \delta_{ps} \hat{E}_q^r, \quad (15.31)$$

this symbolic program can be adapted to the spin-free CC theory.

- By implementing an `on_MCSCF_reference` function to support the Multi-Configuration Self-Consistent Field (MCSCF) wavefunction reference and refining the contraction rules accordingly, the symbolic program can be extended to the multi-reference CC theory.
- The Hamiltonian operator can be recursively applied as a perturber to the reference state, leading to the symbolic program for many-body perturbation methods.

- By updating the excitation function in Section 15.5.1 to parameterize $\hat{T} - \hat{T}^\dagger$ for the effective Hamiltonian

$$\hat{H}_{\text{eff}} = e^{-(\hat{T} - \hat{T}^\dagger)} \hat{H} e^{(\hat{T} - \hat{T}^\dagger)}, \quad (15.32)$$

the symbolic code can be utilized to derive unitary coupled cluster expansions.

Summary

In this chapter, we explored various techniques for developing and optimizing coupled cluster programs. A challenge in the coupled cluster program is the time spent reading integral tensors from disk, which often dominates the computational time. Data prefetching is an effective strategy to reduce the overhead of disk reads. To implement data prefetching, one must analyze the I/O operations in the code and insert data loading statements at appropriate points to facilitate background data reading. We introduced JIT (Just-in-Time) technology to automate this process. After identifying the I/O operations suitable for prefetching, the JIT compiler creates a data loader daemon to continuously reading data in the background. The JIT compiler then modifies the source code to accommodate the prefetched data. To limit the memory usage, the prefetched data is cached in a fixed-size dictionary. Advanced techniques, such as code generation, asynchronous computation, and the use of mutex and condition variables, are employed in the development of the JIT compiler.

Additionally, we explored the method for deriving coupled cluster equations using symbolic programming. We provided a detailed tutorial on developing a symbolic program. The main steps include representing basic elements, implementing commutators for these elements, and evaluating the output using the Hartree-Fock reference state. The output from the symbolic program includes numerous redundant terms. These redundant terms can be combined and simplified by considering the permutation symmetry in coupled cluster theory. We took into account the permutation symmetry within each tensor, which yielded results consistent with those derived from diagrammatic theory.

References

- [1] I. Shavitt, R.J. Bartlett, Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory, Cambridge Molecular Science, Cambridge University Press, 2009.
- [2] T.D. Crawford, H.F. Schaefer III, An Introduction to Coupled Cluster Theory for Computational Chemists, John Wiley & Sons, Ltd, 2000, Ch. 2, pp. 33–136, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470125915.ch2>, <https://doi.org/10.1002/9780470125915.ch2>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470125915.ch2>.
- [3] The PySCF Developers, Quantum chemistry with python, <https://pyscf.org/>, 2024.

- [4] I. Purvis, D. George, R.J. Bartlett, A full coupled-cluster singles and doubles model: the inclusion of disconnected triples, *Journal of Chemical Physics* 76 (4) (1982) 1910–1918, <https://doi.org/10.1063/1.443164>, https://pubs.aip.org/aip/jcp/article-pdf/76/4/1910/18935003/1910_1_online.pdf.
- [5] G.E. Scuseria, C.L. Janssen, Henry F. Schaefer III, An efficient reformulation of the closed-shell coupled cluster single and double excitation (CCSD) equations, *Journal of Chemical Physics* 89 (12) (1988) 7382–7387, <https://doi.org/10.1063/1.455269>, https://pubs.aip.org/aip/jcp/article-pdf/89/12/7382/18973922/7382_1_online.pdf.
- [6] R.J. Bartlett, G.D. Purvis, Many-body perturbation theory, coupled-pair many-electron theory, and the importance of quadruple excitations for the correlation problem, *International Journal of Quantum Chemistry* 14 (5) (1978) 561–581, <https://doi.org/10.1002/qua.560140504>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.560140504>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.560140504>.
- [7] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. chung Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, A. Sibiryakov, Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models, *Proceedings of the IEEE* 93 (2) (2005) 276–292, <https://doi.org/10.1109/JPROC.2004.840311>.

Molecular properties

16

An important aspect of quantum chemistry calculations is the prediction of molecular properties. Generally, there are two primary types of methods for calculating molecular properties: the expectation value of certain operators based on the wavefunction, and the derivatives of molecular energy.

Given an operator, such as the spin-square operator or the momentum operator, the expectation value of the operator can be decomposed into the calculation of integrals for the operator and the density matrices of the wavefunction.

For the derivatives of molecular energy, this category includes properties such as molecular forces (nuclear gradients), Electron Spin Resonance (ESR), and Nuclear Magnetic Resonance (NMR). Molecular forces, aka nuclear gradients, are the first-order derivatives of energy with respect to the coordinates of atoms in a molecule. This property is particularly useful, as it is frequently employed in geometry optimization and molecular dynamics simulations.

There are several methods for computing analytical nuclear gradients. One approach is the analytical gradients program which utilizes perturbation theory to evaluate the derivatives of energy. Alternatively, one can employ the automatic differentiation (AD) technique to compute derivatives. In this chapter, we will use the perturbation theory for the nuclear gradients of the Hartree-Fock (HF) method. Subsequently, we will use the JAX AD technique to compute the nuclear gradients of the CCD (Coupled Cluster with double excitation) method.

Why do we use a combination of the analytical gradient program and the AD technique instead of applying AD universally? While it is possible to develop AD from scratch and use it to compute HF derivatives, we choose the hybrid approach to demonstrate how to customize and integrate the AD technique into an existing Python project. This AD customization involves several advanced features of the JAX library, which is an extension to the introduction of the JAX package in Chapter 5.

16.1 Molecular properties for single-particle operators

Computing molecular properties associated with the expectation values of single-electron operators is straightforward. These properties can be expressed as products of atomic orbital (AO) integrals and the one-particle reduced density matrix. For example, properties such as the dipole moment, electron density in real space, and the Molecular Electrostatic Potential (MEP) all fall into this category.

MEP is a valuable tool for studying molecular reactivity. Let's use the MEP to demonstrate how to compute these properties in terms of integrals and density matrices. The MEP at a specific position can be calculated through tensor contraction:

$$\begin{aligned} V_{\text{MEP}}(\mathbf{r}) &= \int \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}' - \sum_A \frac{Z_A}{|\mathbf{r} - \mathbf{R}_A|} \\ &= \sum_{\mu\nu} V_{\mu\nu}(\mathbf{r}) \gamma_{\nu\mu} - \sum_A \frac{Z_A}{|\mathbf{r} - \mathbf{R}_A|}, \end{aligned} \quad (16.1)$$

where

$$V_{\mu\nu}(\mathbf{r}) = \int \frac{\mu^*(\mathbf{r}')\nu(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}'. \quad (16.2)$$

This integral can be computed using the one-electron Coulomb integral function `coulomb_1e_MD` introduced in Chapter 12. The computational code for the MEP can be implemented as follows.

```
from pychem_book.chap12.analytical_integrals.v5.basis import Molecule, CGTO
from pychem_book.chap12.analytical_integrals.v5 import coulomb_1e_MD
from pychem_book.chap13.simple_scf import SCFWavefunction

def eval_mep(mol: Molecule, gtos: List[CGTO], wfn: SCFWavefunction, grids):
    dist = np.linalg.norm(grids[:,None,:] - mol.coordinates, axis=2)
    mep = -np.einsum('z,gz->g', np.array(mol.nuclear_charges), 1./dist)

    dm = wfn.density_matrices
    for i, r in enumerate(grids):
        v = coulomb_1e_MD.get_matrix(gtos, r)
        mep[i] += np.einsum('ij,ji->', v, dm)
    return mep
```

Using this function, we can calculate the MEP on uniform grids and subsequently employ the visualization tools introduced in Chapter 3 to display the MEP. For instance, the following script generates a cube file, which can then be visualized using JMol, VMD, or other visualization software.

```
from pychem_book.chap03.render_cube import render_cube
from pychem_book.chap13.simple_scf import RHF, simple_scf
xyz = '''
N -.75 0 0
N 0.75 0 0'''
mol = Molecule.from_xyz(xyz)
gtos = mol.assign_basis({'N': '6-31g'})
model = RHF(mol, gtos)
wfn = scf_iter(model)
```

```

boundary = [[-5., 5.],
            [-5., 5.],
            [-5., 5.]]
mesh = [20, 20, 20]
mgrids = np.mgrid[[slice(r[0], r[1], m*1j) for r, m in zip(boundary, mesh)]]
grids = mgrids.reshape(3, -1).T
rho = eval_density(mol, gtos, wfn, grids).reshape(mesh)
mep = eval_mep(mol, gtos, wfn, grids).reshape(mesh)

origin = mgrids[:,0,0,0]
voxel = np.array([mgrids[:,1,0,0] - origin,
                  mgrids[:,0,1,0] - origin,
                  mgrids[:,0,0,1] - origin])

with open('mep.cub', 'w') as f:
    render_cube(mol.elements, mol.coordinates, voxel, origin, mep)

```

16.2 Analytical nuclear gradients

16.2.1 Basic equations of Hartree-Fock derivatives

Nuclear gradients of Hartree-Fock methods can be analytically evaluated using perturbation theory. The restricted HF (RHF) energy can be expressed in terms of the occupied HF orbitals:

$$E = \sum_i h_{ii}^{\text{core}} + \frac{1}{2} \sum_{ij} (ii||jj) + E_{\text{nn}}, \quad (16.3)$$

where HF orbitals are subject to the ortho-normality constraints

$$\mathbf{C}^\dagger \mathbf{S} \mathbf{C} - \mathbf{1} = 0. \quad (16.4)$$

To optimize the HF orbitals, we can use a Lagrangian to incorporate the normality constraints

$$\mathcal{L} = E(\mathbf{C}) - \sum_{ij} e_{ij} (\mathbf{C}^\dagger \mathbf{S} \mathbf{C} - \mathbf{1})_{ij}. \quad (16.5)$$

The stationary condition for molecular orbitals reads

$$\frac{\partial \mathcal{L}}{\partial C} = 0. \quad (16.6)$$

It leads to the Hartree-Fock-Roothaan equation

$$\mathbf{FC} = \mathbf{SC}e. \quad (16.7)$$

For *canonical HF orbitals*, the Lagrangian multiplier e is a diagonal matrix.

When calculating the derivatives of the energy with respect to the nuclear coordinates, we can use the Lagrangian to replace the energy E . This leads to the following equation for energy derivatives:

$$\frac{dE(\mathbf{C})}{dX_A} = \frac{d\mathcal{L}}{dX_A} = \frac{\partial \mathcal{L}}{\partial X_A} + \frac{\partial \mathcal{L}}{\partial C} \frac{dC}{dX_A}. \quad (16.8)$$

By applying the stationary condition (16.6) and the RHF energy (16.3), the energy derivatives can be computed as follows:

$$\begin{aligned} \frac{dE(\mathbf{C})}{dX_A} &= \frac{\partial \mathcal{L}}{\partial X_A} = \sum_i \frac{\partial h_{ii}^{\text{core}}}{\partial X_A} + \frac{1}{2} \sum_{ij} \left(\frac{\partial(ii|jj)}{\partial X_A} - \frac{\partial(ij|ji)}{\partial X_A} \right) + \frac{\partial E_{\text{nn}}}{\partial X_A} \\ &\quad - \sum_i e_i (\mathbf{C}^\dagger \frac{\partial \mathbf{S}}{\partial X_A} \mathbf{C})_{ii}. \end{aligned} \quad (16.9)$$

We then utilize the density matrix \mathbf{D}

$$D_{v\mu} = 2 \sum_i^{n/2} C_{vi} C_{\mu i}, \quad (16.10)$$

and the energy-weighted density matrix \mathbf{D}^e

$$D_{v\mu}^e = 2 \sum_i^{n/2} e_i C_{vi} C_{\mu i}, \quad (16.11)$$

to rewrite the expressions for the energy derivatives as shown in Eq. (16.9):

$$\sum_i \frac{\partial h_{ii}^{\text{core}}}{\partial X_A} = \sum_{\mu\nu} D_{v\mu} \frac{\partial h_{v\mu}^{\text{core}}}{\partial X_A} \quad (16.12)$$

$$J^{(X_A)} = \sum_{ij} \frac{\partial(ii|jj)}{\partial X_A} = \sum_{\mu\nu\kappa\lambda} D_{v\mu} D_{\lambda\kappa} \frac{\partial(\mu\nu|\kappa\lambda)}{\partial X_A} \quad (16.13)$$

$$K^{(X_A)} = \sum_{ij} \frac{\partial(ij|ji)}{\partial X_A} = \sum_{\mu\nu\kappa\lambda} \frac{1}{2} D_{\lambda\mu} D_{\nu\kappa} \frac{\partial(\mu\nu|\kappa\lambda)}{\partial X_A} \quad (16.14)$$

$$\sum_i e_i (\mathbf{C}^\dagger \frac{\partial \mathbf{S}}{\partial X_A} \mathbf{C})_{ii} = \sum_{\mu\nu} D_{v\mu}^e \frac{\partial S_{\mu\nu}}{\partial X_A}. \quad (16.15)$$

For a detailed derivation of the nuclear gradients in the Hartree-Fock method, please refer to Appendix C in the book *Modern Quantum Chemistry* by Szabo.

16.2.2 Derivatives of integrals

The derivatives of integrals are related to the derivatives of basis functions with respect to electronic coordinates. Consider a Cartesian Gaussian function centered on atom A :

$$\chi_{m_x m_y m_z, A}(\mathbf{r}) = (x - X_A)^{m_x} (y - Y_A)^{m_y} (z - Z_A)^{m_z} e^{-\alpha|\mathbf{r} - \mathbf{R}_A|^2}. \quad (16.16)$$

The derivatives of this basis function with respect to the nuclear coordinates can be transformed into derivatives with respect to the electronic coordinates

$$\frac{\partial}{\partial X_B} \chi_{m_x m_y m_z, A} = \begin{cases} -\frac{\partial}{\partial x} \chi_{m_x m_y m_z, A}, & A = B \\ 0, & A \neq B \end{cases}. \quad (16.17)$$

These derivatives can be expressed as a combination of two Cartesian Gaussian functions (except for the s -type functions):

$$\frac{\partial}{\partial x} \chi_{m_x m_y m_z, A} = l \chi_{m_x-1, m_y m_z} - 2\alpha \chi_{m_x+1, m_y m_z}. \quad (16.18)$$

Using this equation, we can calculate the derivatives of integrals in terms of two standard integrals.

It is not difficult to implement the code for integral derivatives using the integral program developed in Chapter 12. For simplicity, we will not repeat the integral program in this chapter. Instead, we will utilize the PySCF package to compute these integral derivatives.

The PySCF package offers a general integral evaluator `.intor()` within its `Mole` class. By specifying the name of the integral function, `Mole.intor` can generate a tensor that represents the integrals for all basis functions in a molecule. For instance, the command `mol.intor('intle_ipovlp')` computes the overlap type integrals with derivatives of electron coordinates on the bra

$$\langle \nabla \chi_\mu | \chi_\nu \rangle.$$

Here, the notation `ip` represents the operator $i \hat{p}$, which is equal to the operator ∇ . PySCF can process the derivatives of the three Cartesian directions simultaneously, producing an array with the shape $(3, N, N)$. In addition to the derivatives for overlap integrals, Table 16.1 lists other integrals necessary for nuclear gradient calculations. You can consult the PySCF documentation [1] for a comprehensive list of supported integrals.

Please note that all integrals listed in Table 16.1 are computed as the *derivatives with respect to the electronic coordinates* on the bra function. To calculate the derivatives for nuclear coordinates with the transformation (16.17), we must identify the atom center associated with each basis function. In PySCF, basis functions associated with the same atom are grouped together. The package offers a method

Table 16.1 PySCF integral name code for nuclear gradients integrals.

Function name	Expression
intle_ipovlp	$\langle \nabla \mu v \rangle$
intle_ipkin	$\langle \nabla \mu \frac{1}{2} \nabla^2 v \rangle$
intle_ipnuc	$\langle \nabla \mu V_{\text{nuc}} v \rangle$
intle_iprinv	$\langle \nabla \mu \frac{1}{ \mathbf{r} - \mathbf{R} } v \rangle$
int2e_ip1	$(\nabla \mu, v \kappa \lambda)$

Mole-aoslice_by_atom() to determine the positions of the first and last basis functions for each atom. For instance, in the case of overlap integrals, the coordinate transformation is given by:

$$\frac{\partial S_{\mu\nu}}{\partial X_A} = -\langle \nabla_x \chi_{\mu A} | \chi_{\nu} \rangle - \langle \nabla_x \chi_{\nu A} | \chi_{\mu} \rangle^*. \quad (16.19)$$

The corresponding Python code is

```
def grad_overlap(mol, atom_id):
    s1 = mol.intor('intle_ipovlp')
    p0, p1 = mol-aoslice_by_atom()[atom_id, 2:4]
    s_A = np.zeros((3, nao, nao))
    s_A[:, p0:p1] = -s1[:, p0:p1]
    return s_A + s_A.transpose(0, 2, 1)
```

The calculation of derivatives for the four-index electron repulsion integrals is similar to that of overlap integrals. The relevant function name is int2e_ip1. This function provides the derivatives for the basis function at the first index within the four-index tensor. The four indices are arranged in accordance with the chemist's notation. To symmetrize the integral tensor, two transpositions are performed to distribute the derivatives across the remaining three indices.

```
eri1 = mol.intor('int2e_ip1')
p0, p1 = mol-aoslice_by_atom()[atom_id, 2:4]
eri_A = np.zeros((3, nao, nao, nao))
eri_A[:, :, p0:p1] = -eri1[:, :, p0:p1]
eri_A = eri_A + eri_A.transpose(0, 2, 1, 3, 4)
eri_A = eri_A + eri_A.transpose(0, 3, 4, 1, 2)
```

The h^{core} operator consists of the kinetic operator and the nuclear attraction operator. The derivatives of the kinetic integrals are similar to those of the overlap integrals. For the nuclear attraction operator V_{nuc} , the computation requires both the derivatives applied to the basis functions and the derivatives of the operator itself:

$$\frac{\partial V_{\text{nuc}}}{\partial X_A} = \frac{\partial}{\partial X_A} \frac{-Z_A}{|\mathbf{r} - \mathbf{R}_A|} = -\nabla_x \frac{-Z_A}{|\mathbf{r} - \mathbf{R}_A|}. \quad (16.20)$$

The integration-by-parts formula is then employed to transform Eq. (16.20) into derivatives of the basis functions:

$$-\langle \mu | \nabla_x \frac{-Z_A}{|\mathbf{r} - \mathbf{R}_A|} | \nu \rangle = \langle \nabla_x \mu | \frac{-Z_A}{|\mathbf{r} - \mathbf{R}_A|} | \nu \rangle + \langle \mu | \frac{-Z_A}{|\mathbf{r} - \mathbf{R}_A|} | \nabla_x \nu \rangle. \quad (16.21)$$

The program for the integrals of h^{core} can be implemented as:

```
def grad_hcore(mol, atom_id):
    # derivatives of T + V on bra
    hcore_partial = mol.intor('intle_ipkin') + mol.intor('intle_ipnuc')
    with mol.with_rinv_at_nucleus(atom_id):
        v = mol.intor('intle_iprinv') * -mol.atom_charge(atom_id)
    p0, p1 = mol-aoslice_by_atom()[atom_id, 2:4]
    v[:, p0:p1] -= hcore_partial[:, p0:p1]
    return v + v.transpose(0, 2, 1)
```

16.2.3 Nuclear gradients for RHF energy

When evaluating the HF nuclear gradients in Eq. (16.9), we can simplify the computation by leveraging the permutation symmetry within the integrals. For instance, the term $K^{(X_A)}$ can be simplified to:

$$\begin{aligned} K^{(X_A)} &= \sum_{\mu\nu\kappa\lambda} \frac{1}{2} D_{\lambda\mu} D_{\nu\kappa} \left((\frac{\partial \mu}{\partial X_A} \nu | \kappa \lambda) + (\mu \frac{\partial \nu}{\partial X_A} | \kappa \lambda) + (\mu \nu | \frac{\partial \kappa}{\partial X_A} \lambda) + (\mu \nu | \kappa \frac{\partial \lambda}{\partial X_A}) \right) \\ &= -4 \sum_{\mu\nu\kappa\lambda} \frac{1}{2} D_{\lambda\mu} D_{\nu\kappa} ((\nabla_x \mu_A) \nu | \kappa \lambda). \end{aligned} \quad (16.22)$$

Similar transformations can be applied to the $J^{(X_A)}$ term and the overlap term. By combining these simplifications with the integral functions we introduced previously, we can derive the program for RHF nuclear gradients.

```
def grad_hf_energy(mol, mo_energy, mo_coeff, nocc):
    mo_o = mo_coeff[:, :nocc]
    dm = einsum('pi,qi->pq', mo_o, mo_o) * 2
    dme = einsum('pi,i,qi->pq', mo_o, mo_energy[:nocc], mo_o) * 2
    eril1 = mol.intor('int2e_ip1')
    j = einsum('xpqrs,sr->xpq', eril1, dm)
    k = einsum('xpqrs,qr->xps', eril1, dm)
    vhf = j - k * .5
    s1 = mol.intor('intle_ipovlp')

    aoslices = mol-aoslice_by_atom()
    de = np.zeros((mol.natm, 3))
```

```

for k in range(mol.natm):
    p0, p1 = aoslices[k,2:]
    h1 = grad_hcore(mol, k)
    de[k] += np.einsum('xij,ji->x', h1, dm)
    de[k] += np.einsum('xij,ji->x', -vhf[:,p0:p1], dm[:,p0:p1]) * 2
    de[k] -= np.einsum('xij,ji->x', -s1[:,p0:p1], dme[:,p0:p1]) * 2

coords = mol.atom_coords()
Z = mol.atom_charges()
de_nuc = jax.grad(nuclear_repulsion_energy)(coords, Z)      # (1)
return de + de_nuc

def nuclear_repulsion_energy(coords, z):
    rr = coords[:,None,:] - coords
    zz = z[:,None] * z
    tril = np.tril_indices(len(z), -1)
    d = jnp.linalg.norm(rr[tril], axis=1)
    return (zz[tril] / d).sum()

```

In line (1), we utilize the automatic differentiation (autodiff) feature of the JAX library to compute the gradients of nuclear repulsion energy. This function has been discussed in Section 5.3 of Chapter 5.

Given the zeroth-order HF results, we can utilize the `grad_hf_energy` function to calculate the nuclear gradients of the RHF energy.

```

import pyscf
mol = pyscf.M(atom='N 0. 0 0; N 1.5 0 0', basis='ccpvdz')
mf = mol.RHF().run()
nooc = mol.nelectron // 2
de = grad_hf_energy(mol, mf.mo_energy, mf.mo_coeff, nooc)

```

The results can be verified using the finite difference method.

```

def finite_diff(mol, i_j, delta=1e-2):
    mol = mol.copy()
    coords = mol.atom_coords()
    coords[i_j] += .5 * delta
    mol.set_geom_(coords, unit='Bohr')
    mf = mol.RHF(verbose=0).run()
    ehf0 = mf.e_tot

    coords[i_j] -= delta
    mol.set_geom_(coords, unit='Bohr')
    mf = mol.RHF(verbose=0).run()
    ehf1 = mf.e_tot

```

```
    return (ehf0 - ehf1) / delta

for i in range(mol.natm):
    for j in range(3):
        diff1 = de[i,j] - finite_diff(mol, (i, j))
        diff2 = de[i,j] - finite_diff(mol, (i, j), 0.5e-2)
        print(i, j, diff1, diff2)
```

16.2.4 Nuclear gradients with finite difference

In quantum chemistry, programs designed to compute analytical nuclear gradients are often complex. It is crucial to validate the correctness of these programs by comparing them against results obtained from the finite difference method. The finite difference gradients should be computed using the two-point stencil, as this formula introduces only a quadratic error:

$$f' = \frac{f(x+h) - f(x-h)}{2h} + O(h^2). \quad (16.23)$$

A suitable step size h should be chosen for the two-point stencil. While a smaller step size can help minimize the impact of higher order terms, an excessively small step size may amplify the errors due to the subtraction in Eq. (16.23), which can lead to increased numerical errors.

The results of finite difference gradients and analytical gradients are close, yet exhibit slight discrepancies. In such circumstances, how can we ensure the correctness of the analytical gradients?

Given the quadratic error term in the two-point stencil formula, the difference between the finite difference and analytical results should change quadratically with respect to the step size. This quadratic change is a reliable indicator of the correctness of analytical gradients. If the difference does not follow the quadratic trend, it usually suggests several possible issues:

- The analytical gradients program might be incorrect.
- The step size might be too small, leading to a scenario where the precision in the single-point calculation becomes the dominant factor. This often occurs when the convergence or other thresholds for the single-point calculation are not set tightly enough.
- Certain quantities that implicitly depend on the molecular geometry might be missing from the analytical gradients program.
- Certain quantities might not change in a continuous and smooth manner with respect to the molecular geometry.

If we compute integrals using numerical integration formulas, we might encounter the latter two issues. For example, the exchange-correlation functional in density functional theory is numerically evaluated on grids. Both the coordinates and the

weights of these grids are influenced by the nuclear coordinates. Additionally, grids with small weights are often discarded to reduce computational costs, which can lead to discontinuities with respect to the molecular geometry. These errors can become significant near the equilibrium geometry, especially when compared to the near-zero nuclear gradients in these regions. To ensure high precision in analytical nuclear gradients, it is crucial to explicitly consider the derivatives of all variables and to avoid discarding grids.

16.2.5 First order molecular orbitals

Here, we refer to the derivatives of molecular orbitals as first-order orbitals, adopting the terminology from perturbation theory. Although first-order orbitals are not required for calculating nuclear gradients, they are often involved in various second-order properties, such as the nuclear Hessian and the nuclear magnetic resonance (NMR). If you are interested in the calculations of second-order properties, more details on the methodologies can be found in quantum chemistry textbooks [2]. In the following, we will focus on the numerical program for first-order orbitals.

The coefficients of first-order orbitals can be derived from the first-order expansion of the Hartree-Fock-Roothaan equation:

$$\mathbf{F}^1 \mathbf{C}^0 + \mathbf{F}^0 \mathbf{C}^1 = \mathbf{S}^1 \mathbf{C}^0 e^0 + \mathbf{S}^0 \mathbf{C}^1 e^0 + \mathbf{S}^0 \mathbf{C}^0 e^1. \quad (16.24)$$

As a standard method to solve this equation, we can take the zero-order molecular orbitals (MO) as the basis functions to transform the equation. This leads to the equation for the virtual-occupied block of the first-order orbitals:

$$F_{ai}^1 + e_a^0 C_{ai}^1 = S_{ai}^1 e_i^0 + C_{ai}^1 e_i^0. \quad (16.25)$$

The occupied-occupied block and the virtual-virtual block can be determined by the first order equations derived from the orthogonality constraints:

$$(\mathbf{C}^1)^\dagger + \mathbf{S}^1 + \mathbf{C}^1 = 0 \rightarrow \begin{cases} C_{ij}^1 = -\frac{1}{2} S_{ij}^1 & i, j \in \text{occupied} \\ C_{ab}^1 = -\frac{1}{2} S_{ab}^1 & a, b \in \text{virtual} \end{cases}. \quad (16.26)$$

By combining Eq. (16.25) and (16.26), we derive the coupled perturbed Hartree-Fock (CPHF) equation for the closed-shell RHF model:

$$(e_a - e_i) C_{ai}^1 + \sum_{bj} (4(ai|jb) C_{bj}^1 - (ab|ji) C_{bj}^1 - (aj|bi) C_{bj}^1) = -f_{ai}, \quad (16.27)$$

where

$$\begin{aligned} f_{ai} = & \frac{\partial h_{ai}^{\text{core}}}{\partial X_A} + \sum_j \frac{2\delta(ai|jj)}{\partial X_A} - \frac{\partial(aj|ji)}{\partial X_A} - \frac{\partial S_{ai}}{\partial X_A} e_i^0 \\ & - \sum_{jk} (2(ai|jk) - (ak|ji)) S_{kj}^1. \end{aligned} \quad (16.28)$$

Although the four-index integrals in Eq. (16.28) are represented in the MO basis, it is not necessary to explicitly construct these MO integrals. A more efficient approach can be adopted:

1. First, compute the integrals in the atomic orbital (AO) basis;
2. Next, perform the tensor contractions between the integral tensors and the density matrix in the AO basis;
3. Finally, transform the integrals from the AO basis to the MO basis.

During this procedure, we can utilize the permutation symmetry in the integrals and density matrix to further reduce computational costs. For example, the term $\frac{\partial(aj|ji)}{\partial X_A}$ in Eq. (16.28) can be reformulated as:

$$\begin{aligned} \sum_j \frac{\partial(aj|ji)}{\partial X_A} &= \sum_{\mu\lambda} C_{\mu a} C_{\lambda i} \left(\frac{1}{2} \sum_{\nu\kappa} D_{\nu\kappa} \frac{\partial(\mu\nu|\kappa\lambda)}{\partial X_A} \right) \\ &= \frac{1}{2} \sum_{\mu\lambda} C_{\mu a} C_{\lambda i} (1 + P(\mu, \lambda)) \\ &\quad \cdot \sum_{\nu\kappa} \left(D_{\nu\kappa} ((\nabla_x \mu_A) \nu | \kappa\lambda) + D_{\nu\kappa} ((\nabla_x \nu_A) \mu | \kappa\lambda) \right). \end{aligned} \quad (16.29)$$

After reformulating the other terms of Eq. (16.28) in a similar manner, we can proceed to implement the computational program for Eq. (16.28).

```
def _fock_partial_deriv(mol, mo_coeff, nocc):
    nao, nmo = mo_coeff.shape
    mo_o = mo_coeff[:, :nocc]

    eri1 = mol.intor('int2e_ip1')
    eri = mol.intor('int2e')
    dm = einsum('pi,qi->pq', mo_o, mo_o) * 2
    j = einsum('xpqrs,sr->xpq', eri1, dm)
    k = einsum('xpqrs,qr->xps', eri1, dm)
    vhfl = j - k * .5
    s1 = mol.intor('intle_ipovlp')

    aoslices = mol.aoslice_by_atom()
    s1_buf = []
    f1_buf = []
    for ia in range(mol.natm):
        shl0, shl1, p0, p1 = aoslices[ia]
        slao = np.zeros((3, nao, nao))
        slao[:, p0:p1] -= s1[:, p0:p1]
        slao[:, :, p0:p1] -= s1[:, p0:p1].transpose(0, 2, 1)
        s1_buf.append(slao)
```

```

j = einsum('xpqrs,qp->xrs', eri1[:,p0:p1], dm[:,p0:p1])
k = einsum('xpqrs,sp->xrq', eri1[:,p0:p1], dm[:,p0:p1])
v1 = -(j - k * .5)
v1[:,p0:p1] -= vhf1[:,p0:p1]
v1 = v1 + v1.transpose(0,2,1)
s1_oo = einsum('pi,xpq,qj->xij', mo_o, slao, mo_o)
s1_oo = einsum('pi,xij,qj->xpq', mo_o, s1_oo, mo_o) * 2
fock1 = grad_hcore(mol, ia) + v1 - get_vhf(eri, s1_oo)
f1_buf.append(fock1)
return np.vstack(f1_buf), np.vstack(s1_buf)

def get_vhf(eri, dm):
    j = einsum('pqrs,xqp->xrs', eri, dm)
    k = einsum('pqrs,xqr->xps', eri, dm)
    return j - k * .5

```

The CPHF equation (16.27) is essentially a linear equation

$$(\mathbf{I} + \mathbf{A})x = \mathbf{b}. \quad (16.30)$$

This equation can be solved using the Krylov linear equation solver, which will be discussed in Section 16.3. To solve the CPHF equation, we implement a custom matrix-vector function that incorporates the factor $(e_a - e_i)$ in Eq. (16.27). The program for computing first-order molecular orbitals is demonstrated below.

```

from .krylov import solve_krylov

def solve_mol(mol, mo_energy, mo_coeff, nocc):
    nao, nmo = mo_coeff.shape
    nvir = nmo - nocc
    mo_o = mo_coeff[:, :nocc]
    mo_v = mo_coeff[:, nocc:]
    e_o = mo_energy[:nocc]
    e_v = mo_energy[nocc:]

    eri = mol.intor('int2e')
    def matvec(x):
        x = x.reshape(-1, nvir, nocc)
        dm = einsum('pi,xij,qj->xpq', mo_v, x, mo_o) * 2
        dm = dm + dm.transpose(0, 2, 1)
        vhf = einsum('pi,xpq,qj->xij', mo_v, get_vhf(eri, dm), mo_o)
        vhf /= e_v[:, None] - e_o
    return vhf.ravel()

```

```

f1, s1 = _fock_partial_deriv(mol, mo_coeff, nocc)
s1 = einsum('pi,xpq,qj->xij', mo_coeff, s1, mo_coeff)
f1 = einsum('pi,xpq,qj->xij', mo_v, f1, mo_o)
b = (f1 - s1[:,nocc:,:nocc] * e_o) / (e_o - e_v[:,None])
mol_vo = solve_krylov(matvec, b.ravel()).reshape(-1,nvir,nocc)

mol = -.5 * s1 # The occupied-occupied and virtual-virtual blocks
mol[:,nocc,:,:nocc] = mol_vo
# mol.T + s1 + mol = 0
mol[:, :, nocc, nocc:] = -s1[:, :, nocc, nocc:] - mol_vo.transpose(0, 2, 1)
mol = einsum('pq,xqi->xpi', mo_coeff, mol)
return mol.reshape(mol.natm, 3, nao, nmo)

```

For ease of discussion in subsequent sections, we convert the resultant first-order orbitals back to the AO basis representation in this program.

How can we verify whether the first-order orbitals are correctly calculated? In this scenario, it is not suitable to use the finite difference method to validate the first-order orbitals. This is because the occupied-occupied block of the first-order orbitals, as determined by the condition in Eq. (16.26), does not align with the orthonormality of canonical HF orbitals. An alternative approach is to use the finite difference method to examine the first-order density matrix instead. The first-order density matrix can be obtained using the following code:

```

mol = solve_mol(mol, mf.mo_energy, mf.mo_coeff, nocc)
nao, nmo = mf.mo_coeff.shape
mol = mol.reshape(mol.natm, 3, nao, nmo)
dm = einsum('pi,qi->pq', mol[0,0], mf.mo_coeff) * 2
dm = dm + dm.T

```

16.3 Krylov subspace linear equation solver

Many quantum chemistry problems can be formulated as linear equations involving a square matrix A and a vector b ,

$$x + Ax = b. \quad (16.31)$$

Often, matrix A is either too costly to compute, or too large to be fully stored in memory. Under such circumstances, it is impractical to directly construct matrix A and then solve the linear equation using conventional linear algebra techniques for dense matrices. Instead, matrix A is represented by a function that computes the product of A and x . To find the solution of the linear equation (16.31), we must rely on iterative methods.

One approach is to reformulate the problem as a fixed-point iteration problem:

$$x = b - Ax. \quad (16.32)$$

The fixed-point iteration can be accelerated using the DIIS method. Alternatively, the Krylov subspace method can be employed to iteratively solve the equation [3,4].

The Krylov space $\{v_0, v_1, \dots, v_{n-1}\}$ for matrix A and vector b is generated through the recursive matrix-vector multiplication

$$v_0 = b, \quad (16.33)$$

$$v_{i+1} = Av_i. \quad (16.34)$$

The solution x can be represented using the basis vectors

$$x = \sum_i c_i v_i. \quad (16.35)$$

By using the basis vectors v to represent A and b , we can derive a linear equation for the coefficients c :

$$\sum_i (v_j \cdot v_i + v_j \cdot A \cdot v_i) c_i = v_j \cdot b. \quad (16.36)$$

This linear equation can be efficiently solved to determine the solution x .

Mathematically, this method is quite straightforward. However, several technical issues must be considered in real applications:

- The process of repeatedly applying matrix-vector multiplications can result in huge numbers within the basis vectors. Additionally, the generated basis vectors might be close to linearly dependent, which can cause numerical issues in subsequent linear algebra computation. To address this issue, it is necessary to check for linear dependency among the existing basis vectors each time a new vector is generated. If linear dependency is detected, the generation of the Krylov space should be halted.
- It is not necessary to generate all linearly independent basis vectors v before solving the subspace linear equation (16.36). Often, a relatively small Krylov subspace is sufficient to achieve the desired accuracy. To efficiently truncate the Krylov subspace, we can solve the subspace equation (16.36) and calculate the residual

$$\sigma = Ax_i - b \quad (16.37)$$

each time a new basis vector is generated. If the magnitude of the residual falls below the convergence threshold, the iteration of basis generation can be stopped.

The function `solve_krylov` can be implemented as follows:

```
def solve_krylov(A, b, tol=1e-5, maxiter=50):
    vs = [b]
    hv = []
    h = np.empty((maxiter, maxiter))
    g = np.empty(maxiter)
    c = []

    for cycle in range(maxiter):
        # Generate new vector in Krylov subspace and orthogonalize it
        v1 = A(vs[-1])
        hv.append(v1 + vs[cycle]) # (I+A)*v
        for v in vs:
            v1 -= v * np.dot(v, v1)
        # Add to subspace if it linearly independent to others
        v_norm = np.linalg.norm(v1)
        if v_norm < 1e-10:
            break
        v1 /= v_norm
        vs.append(v1)

        for i in range(cycle+1):
            h[i,cycle] = np.dot(vs[i], hv[cycle])
            h[cycle,i] = np.dot(vs[cycle], hv[i])
        g[cycle] = np.dot(vs[cycle], b)
        c = np.linalg.solve(h[:cycle+1,:cycle+1], g[:cycle+1])

        residual = -b
        for i, ci in enumerate(c):
            # Reuse hv vectors to reduce the matvec A(x)
            residual += hv[i] * ci
        norm = np.linalg.norm(residual)
        print(f'krylov {cycle=} residual={norm:.5g}')
        if norm < tol:
            break
    else:
        raise RuntimeError('solve_krylov not converged')

    x = np.zeros_like(b)
    for i, ci in enumerate(c):
        x += vs[i] * ci
    return x
```

Let's consider how to incorporate an initial guess into the linear equation solver. Given an initial guess x_0 , the solution to the original linear equation can be written as $x = x_0 + x'$. We can then rewrite the linear equation (16.31) as

$$x' + Ax' = b - Ax_0 - x_0. \quad (16.38)$$

By adjusting vector b , we can utilize the same Krylov subspace solver for x' , leading to the revised version of the function:

```
def solve_krylov(A, b, x0=None, tol=1e-5, maxiter=50):
    if x0 is not None:
        x = solve_krylov(A, b-A(x0)-x0, tol=tol, maxiter=maxiter)
    return x + x0
...
```

The Krylov subspace iterative method may encounter convergence issues. Within a limited number of iterations, the generated Krylov space might not sufficiently represent the solution of the linear equation. To address the convergence issue, one approach is to use the unconverged results as an initial guess and then reapply the solver. Alternatively, we can use the unconverged results as the initial guess for the fixed-point iteration Eq. (16.32) and employ DIIS to find the solution.

The underlying assumption in Eq. (16.31) is that the matrix A is relatively “small” compared to the identity matrix, essentially acting as a perturber to the identity matrix. However, this assumption might not always hold true in practice, and the Krylov solver may not always lead to a converged solution. To adjust the strength of the perturber matrix A , we can introduce a diagonal matrix d and rewrite the linear equation as

$$x + (I + d)^{-1}(A - d)x = (I + d)^{-1}b. \quad (16.39)$$

By selecting an appropriate diagonal matrix d , the convergence of the Krylov subspace iteration can be accelerated.

For challenging linear equations, a combination of the previously mentioned techniques, along with other solvers for sparse matrices as discussed in Chapter 4, may be required to achieve convergence.

16.4 Nuclear gradients with automatic differentiation

An emerging method for computing nuclear gradients is automatic differentiation (AD). Generally, there are two approaches to applying AD in a Python program:

- Developing a package entirely based on AD libraries such as JAX or PyTorch.
- Integrating AD features into existing programs with analytical gradients.

The first approach enables the application of AD to nearly all parameters. However, it requires writing a substantial amount of code to establish the fundamental

framework. In practice, new research often builds upon existing codebases, many of which were not developed with AD libraries. To effectively utilize AD, it is often necessary to integrate the hard-coded differentiation program into AD toolkits. The hybrid AD approach is particularly useful in the context of quantum chemistry applications [5–8].

Common AD libraries such as PyTorch, JAX, and TensorFlow all provide capabilities for custom AD. Which tool should we choose for the hybrid AD code?

Unlike PyTorch, which follows the object-oriented programming style of Python, JAX programming adopts a functional programming (FP) approach. If you are unfamiliar with the FP coding style, you might find it challenging to work with JAX initially. In such cases, the FP textbook *SICP in Python* [9] can be a helpful resource for learning FP in Python.

There are two modes for computing derivatives with AD: *forward-mode* and *reverse-mode*. Given a vector as input, forward-mode AD computes the *Jacobian-vector product* (JVP), while reverse-mode AD computes the *vector-Jacobian product* (VJP). The two AD modes employ different strategies for customizing function derivatives. As we will see, forward-mode AD tends to be more straightforward for customizing AD rules. Compared to reverse-mode AD, functions implemented with forward-mode AD often maintain a structure that is closer to the original program.

PyTorch supports only reverse-mode for AD customization. JAX, on the other hand, supports both forward-mode and reverse-mode AD. It offers the capability to automatically transform the derivative rules from forward-mode to reverse-mode. This feature provides a more flexible way to develop JAX AD programs. In the subsequent examples, we will demonstrate how to customize derivative rules using the JAX library.

An essential task in JAX AD customization is to rewrite functions that are not JAX-transformable into JAX-transformable ones. The terminology *JAX-transformable* refers to functions or operations that JAX can manipulate to generate reverse-mode differentiation code. Generally, the process of implementing custom derivative rules in JAX can be carried out through the following steps:

1. Clean up and refactor the program to ensure it follows the JAX FP style. This includes replacing all NumPy and SciPy functions with their JAX equivalents.
2. Identify the variables and functions that require customization.
3. Utilize the `custom_jvp` and `custom_vjp` APIs to implement these custom derivative rules.
4. Optimize the back-propagation process.

Let us consider the CCD program developed in Chapter 15 as an example. The nuclear gradients of CCD energy depend on the derivatives of integrals and the gradients of the Hartree-Fock energy. Given the complexity and computational costs associated with quantum chemistry integrals, it is practical to utilize an existing integral library for computing integrals (and their derivatives), than developing an integral program from scratch using JAX library. Although the nuclear gradients of Hartree-Fock part could be generated using AD, to demonstrate the functionality of AD customization,

we utilize the Hartree-Fock analytical gradients program developed in Section 16.2.1 for the gradients of the Hartree-Fock part.

Step 1: refactoring the program according to JAX FP style

This step involves the following checks and modifications:

- Creating an explicit list of input arguments and clearly defined outputs for each function. JAX traces both the inputs and outputs to establish the derivative rules for a function. For instance, the main function of the CCD program in Section 15.2 of Chapter 15, `CCD_solve`, is initially defined as:

```
def CCD_solve(mf, conv_tol=1e-5, max_cycle=100):
    ...
    return e_ccd, t2
```

To clearly specify the function that computes the CCD total energy with respect to the atomic coordinates, the `coords` variable should be included in the argument list. Therefore, the function signature of `CCD_solve` is updated to:

```
def CCD_solve(coords, mf, conv_tol=1e-5, max_cycle=100):
    ...
    return e_ccd
```

The second argument, `mf`, and the remaining keyword arguments can be retained in the function's argument list. These parameters are excluded when performing the JAX Jacobian computation. Similarly, the function signature of `mo_integrals` needs to be rewritten to reflect the dependency on `coords`:

```
def mo_integrals(coords, mf: pyscf.scf.hf.RHF):
    ...
    return H
```

- Eliminating any side-effects within each function. This means that the function should not modify the input arguments or global variables that the derivative computation depends on. Ideally, to incorporate the FP coding style, it is also preferable to remove any side-effects, including modifications to the non-differentiable input arguments. In the present CCD program, the functions are already *side-effects free*. No further action is required.
- Avoiding any in-place modifications. Although in-place modifications to local variables are not considered side-effects of the function, they can interfere with JAX's ability to trace the status of intermediate variables. In the `update_CCD_amplitudes` function, several statements involve in-place operations. These should be rewritten to ensure compatibility with JAX's tracing capabilities. For example, the following statements involve in-place modifications to the local variables:

```
Fvv[np.diag_indices(nvir)] -= e_v
t2out += .5 * einsum('ijkl,abkl->abij', oooo, t2)
```

These should be changed to:

```
Fvv = Fvv - jnp.diag(e_v)
t2out = t2out + .5 * einsum('ijkl,abkl->abij', oooo, t2)
```

- Replacing NumPy and SciPy functions with JAX equivalents. It is necessary to replace all instances of NumPy and SciPy functions with their JAX counterparts to ensure compatibility with JAX's AD. For example, all occurrences of `np.einsum` should be replaced with `jax.numpy.einsum`.

After incorporating these modifications, we obtained a refactored CCD program that adapts to the JAX FP style.

```
einsum = jnp.einsum

def update_CCD_amplitudes(H, t2, level_shift=0):
    nvir, nocc = t2.shape[1:3]
    fock = H['fock']
    foo = fock[:nocc,:,:nocc]
    fvv = fock[nocc:,:,nocc:]
    e_o = foo.diagonal()
    e_v = fvv.diagonal() + level_shift

    Fvv = fvv - .5 * einsum('klcd,bdkl->bc', H['oovv'], t2)
    Foo = foo + .5 * einsum('klcd,cdjl->kj', H['oovv'], t2)
    Fvv = Fvv - jnp.diag(e_v)
    Foo = Foo - jnp.diag(e_o)

    t2out = .25 * H['vvoo']
    t2out = t2out - einsum('bkcj,acik->abij', H['vovo'], t2)
    t2out = t2out + .5 * einsum('bc,acij->abij', Fvv, t2)
    t2out = t2out - .5 * einsum('kj,abik->abij', Foo, t2)
    t2out = t2out + .5 * einsum('klcd,acik,bdj->abij', H['oovv'], t2, t2)
    t2out = t2out - t2out.transpose(0,1,3,2)
    t2out = t2out - t2out.transpose(1,0,2,3)
    oooo = .5 * einsum('klcd,cdij->ijkl', H['oovv'], t2)
    oooo = oooo + H['oooo']
    t2out = t2out + .5 * einsum('ijkl,abkl->abij', oooo, t2)
    t2out = t2out + .5 * einsum('abcd,cdij->abij', H['vvvv'], t2)
    eijab = e_o + e_o[:,None] - e_v[:,None,None] - e_v[:,None,None,None]
    t2out = t2out / eijab
    return t2out
```

```

def get_CCD_corr_energy(H, t2):
    return .25 * einsum('ijab,abij->', H['oovv'], t2)

def mp2(H):
    nocc = H['oooo'].shape[0]
    e_o = H['fock'].diagonal()[:nocc]
    e_v = H['fock'].diagonal()[nocc:]
    eijab = e_o + e_o[:,None] - e_v[:,None,None] - e_v[:,None,None,None]
    t2 = H['vvoo'] / eijab
    e = get_CCD_corr_energy(H, t2)
    return H['e_hf'] + e, t2

def CCD_solve(coords: np.ndarray, mf: pyscf.scf.hf.RHF):
    H = mo_integrals(coords, mf)
    e_ccd, t2 = mp2(H) # initial guess
    print(f'E(MP2)={e_ccd}')
    e_hf = H['e_hf']
    for cycle in range(max_cycle):
        t2, t2_prev = update_CCD_amplitudes(H, t2), t2
        e_ccd, e_prev = get_CCD_corr_energy(H, t2) + e_hf, e_ccd
        print(f'{cycle=}, E(CCD)={e_ccd}, dE={e_ccd-e_prev}')
        if jnp.abs(t2 - t2_prev).max() < conv_tol:
            break
    return e_ccd

def mo_integrals(coords, mf: pyscf.scf.hf.RHF): # (1)
    '''MO integrals in physists notation <pq||rs>'''
    mol = mf.mol
    assert jnp.allclose(mol.atom_coords(), coords)
    orb = mf.mo_coeff
    nmo = orb.shape[1]
    eri = mol.intor('int2e', aosym='s1')
    eri = einsum('pqrs,pi,qj,rk,sl->ijkl', eri, orb, orb, orb, orb)
    i2 = jnp.eye(2) # (2)
    eri = einsum('pqrs,ab,cd->parcqbsd', eri, i2, i2).reshape([nmo*2]*4)
    eri = eri - eri.transpose(1,0,2,3)

    no = mol.nelectron
    H = {}
    H['vvoo'] = vvoo = eri[no:,no,:,:no,:no]
    H['oovv'] = vvoo.transpose(2,3,0,1)
    H['vovo'] = eri[no,:,:no,no,:,:no]
    H['oooo'] = eri[:no,:no,:no,:no]

```

```

H['vvvv'] = eri[no:,no:,no:,no:]

hcore = mol.intor('int1e_kin') + mol.intor('int1e_nuc')
hcore = einsum('pq,pi,qj->ij', hcore, orb, orb)
hcore = einsum('pq,ab->paqb', hcore, i2).reshape([nmo*2]*2)
H['fock'] = hcore + einsum('ipiq->pq', eri[:no,:,:no,:])
H['e_hf'] = mf.e_tot
return H

```

The argument `coords` in line (1) is not explicitly used by the `mo_integrals` function. You might wonder: can we remove this argument as we would with a typical Python function? The answer is no. If we were to remove the `coords` argument, JAX would not be able to detect the dependency of `mo_integrals` on `coords`, and incorrectly exclude the differentiation of `mo_integrals` during the AD process. The `mo_integrals` function has an implicit dependency on the nuclear coordinates. Since we need the `mf` object to provide integral computation, `coords` must be consistent with the molecular structure stored in `mf`.

In the `mo_integrals` presented in Chapter 15, we used the slice notation `::2` and `1::2` to assign MO integrals for alpha and beta orbitals. These assignments are replaced by the `einsum` operation with the 2×2 diagonal matrix `i2` in line (2). If you wish to implement the assignment code using fancy indexing, the `.at` and `.set` methods of JAX can be utilized, as shown in the following code.

```
output = output.at[:, :, :, ::2].set(eri)
```

Please note that this code creates a new copy of the output array rather than updating it in-place. The `.at` method should be used with caution. It is recommended to check the JAX documentation of the `.at` method [10] for more details.

For simplicity, we have removed the DIIS acceleration from the CCD iteration function. Mathematically, this only slows down the performance of the CCD program but does not affect the results.

The DIIS module is a typical example of the OOP (object-oriented programming) paradigm. It tracks various intermediate states and modifies them implicitly. In theory, this OOP function could also be rewritten in a FP style. It can be achieved by exposing the DIIS intermediate states within the `CCD_solve` function, and passing them along with other arguments to the DIIS extrapolation function. By doing so, the forward and backward propagation of the DIIS method could be traced and managed by JAX's AD. However, as we will see in Section 16.4.2, there is no need to refactor the DIIS implementation. The existing DIIS module can be used to accelerate the JAX-transformable CCD program without any conflicts.

Step 2: identifying the variables and functions that require customization

If all quantities within a function are produced by operations that are JAX transformable, including simple arithmetic operations and JAX built-in functions, JAX can automatically handle the derivatives for the function. This allows us to easily

identify which functions are compatible with JAX transformations. In the context of the CCD program, both `update_CCD_amplitudes` and `get_CCD_corr_energy` are examples of such functions.

If a function relies on functions not provided by the JAX package, we need to further analyze its dependencies. A function is JAX-transformable if it recursively depends on other JAX-transformable functions. For example, the function `mp2` is a JAX-transformable function, as its only dependency, `get_CCD_corr_energy`, is also JAX-transformable. In contrast, the primal function `mo_integrals` cannot be transformed by JAX because it utilizes several integral evaluation functions from the PySCF package which are not JAX-transformable. Here, the term *primal function* refers to the function in its original, untransformed form. To make `mo_integrals` JAX-transformable, this function needs to be augmented with custom derivative rules.

If a function, such as `CCD_solve`, depends on the function that has been augmented with custom rules, like `mo_integrals`, there is no need to augment `CCD_solve` with additional custom rules, assuming the remaining statements in the `CCD_solve` function are already JAX transformable. By augmenting `mo_integrals`, `CCD_solve` automatically becomes JAX-transformable.

Step 3: implementing the custom rules for forward-mode (or reverse-mode) AD

A normal computation (via the primal function) is executed in a *forward* direction. JVP derivative rules [11] can be implemented with small changes to the code of primal function. When calculating gradients, JAX can automatically convert JVP rules into VJP rules, but not vice versa. Therefore, it is generally preferable to customize the JVP functions if the performance of AD is not a major concern.

To customize the JVP derivative rules, the `jax.custom_jvp` function is used. For the `mo_integrals` function in the CCD program, the JVP rules can be implemented as follows:

```
@partial(jax.custom_jvp, nondiff_argnums=(1,))
def mo_integrals(coords, mf: pyscf.scf.hf.RHF):
    '''MO integrals in physists notation <pq||rs>'''
    mol = mf.mol
    assert jnp.allclose(mol.atom_coords(), coords)
    orb = mf.mo_coeff
    eri = mol.intor('int2e', aosym='s1')
    eri = einsum('pqrs,pi,qj,rk,s1->ijkl', eri, orb, orb, orb, orb)
    hccore = mol.intor('int1e_kin') + mol.intor('int1e_nuc')
    hccore = einsum('pq,pi,qj->ij', hccore, orb, orb)
    return _mo_integrals_common(mol, hccore, eri, mf.e_tot)

@mo_integrals.defjvp
def mo_integrals_jvp(mf, primals, tangents): # (1)
    coords, = primals
    primals_out = mo_integrals(coords, mf)
```

```
mol = mf.mol
disp, = tangents
assert disp.shape == (mol.natm, 3)
nocc = mol.nelectron // 2
mol = solve_mol(mol, mf.mo_energy, mf.mo_coeff, nocc)
mol = einsum('ixpq,ix->pq', mol, disp)
eri = _eri_deriv(mol, mf.mo_coeff, mol, disp)
hcore = _hcore_deriv(mol, mf.mo_coeff, mol, disp)
e_hf1 = grad_hf_energy(mol, mf.mo_energy, mf.mo_coeff, nocc)
e_hf = einsum('ix,ix->', e_hf1, disp)
return primals_out, _mo_integrals_common(mol, hcore, eri, e_hf)

def _mo_integrals_common(mol, hcore, eri, e_hf):
    nmo = hcore.shape[0]
    no = mol.nelectron
    i2 = jnp.eye(2)
    eri = einsum('pqrs,ab,cd->parcqbsd', eri, i2, i2).reshape([nmo*2]*4)
    eri = eri - eri.transpose(1,0,2,3)
    H = {}
    H['vvoo'] = vvoo = eri[:,no,:,:no,:no]
    H['oovv'] = vvoo.transpose(2,3,0,1)
    H['vovo'] = eri[:,no,no,:,:no]
    H['oooo'] = eri[:no,no,:,:no]
    H['vvvv'] = eri[:,no,no,no,:]
    hcore = einsum('pq,ab->paqb', hcore, i2).reshape([nmo*2]*2)
    H['fock'] = hcore + einsum('ipiq->pq', eri[:no,:,:no,:])
    H['e_hf'] = e_hf
    return H

def _hcore_deriv(mol, mo0, mol, disp):
    hcore = mol.intor('intle_kin') + mol.intor('intle_nuc')
    hcore = einsum('pq,pi,qj->ij', hcore, mol, mo0)
    hcore = hcore + hcore.T
    for i in range(mol.natm):
        h1 = grad_hcore(mol, i)
        hcore = hcore + einsum('x,xpq,pi,qj->ij', disp[i], h1, mo0, mo0)
    return hcore

def _eri_deriv(mol, mo0, mol, disp):
    eri1 = mol.intor('int2e_ip1', aosym='s1')
    eri1_partial = einsum('xpqrs,qj,rk,sl->xpjkl', eri1, mo0, mo0, mo0)
    eri = mol.intor('int2e', aosym='s1')
    eri = einsum('pqrs,pi,qj,rk,sl->ijkl', eri, mol, mo0, mo0, mo0)
```

```

aoslices = mol.aoslice_by_atom()[:,2:4]
for i in range(mol.natm):
    p0, p1 = aoslices[i]
    eri = eri + einsum('x,xpjk\l,pi->ijk\l',
                        disp[i], -eril_partial[:,p0:p1], mo0[p0:p1])
    # symmetrize eri
    eri = eri + eri.transpose(1,0,2,3)
    eri = eri + eri.transpose(2,3,0,1)
return eri

```

The `custom_jvp` function is used to decorate the primal function `mo_integrals`. Its `.defjvp` method, as shown in line (1), adds a function that represents the custom JVP rules. In the subsequent discussion, we will refer to this decorated function as the *JVP augmentation function*. The augmentation function must follow a specific input and output format [12].

```

@custom_jvp
def f(x):
    ...

@f.defjvp
def f_jvp(primals, tangents):
    ...
    return primals_out, jvp_product

```

By default, all arguments in the primal function are treated as variables eligible for differentiation. The differentiable variables are passed to the JVP augmentation function under the input argument `primals`. The other argument, `tangents`, represents the tangent vectors associated with the `primals`. The return values of the JVP augmentation function are composed of two elements. The first element is the output of the primal function, denoted as `primals_out` in the code example. It is computed using the primal function with the argument `primals`. The second element, `jvp_product`, represents the result of the Jacobian-vector product.

Some arguments in the primal function may not be differentiable. They are identified by the `nondiff_argnums` parameter in `custom_jvp`. When dealing with non-differentiable arguments, such as the `mf` object in the `mo_integrals` function, no matter how many and where these arguments are defined in the primal function, these arguments are placed at the beginning of the signature of the augmentation function [13]. The code signature of the `custom_jvp` functions would appear as follows:

```

@partial(custom_jvp, nondiff_argnums=(0,2))
def f(a, x, b):
    ...
    @f.defjvp

```

```
def f_jvp(a, b, primals, tangents):
    ...
    return primals_out, jvp_product
```

The JVP augmentation function `mo_integrals_jvp` computes the product of the Jacobian and the residual associated with the primal input variables. It is not necessary to fully construct the Jacobian before performing the product. Instead, we can consume the residual vector early to simplify the JVP rules. For instance, by defining the `_hcore_deriv` and `eri_deriv` functions, we calculate the product of the first-order integrals and the displacements of nuclear coordinates. The resulting `hcore` and `eri` tensors maintain the same shape as those in the primal function. As a result, some portions of the primal function code can be reused in the JVP action, which are consolidated in the `_mo_integrals_common` function.

In the JVP augmentation functions, the operation applied to the tangent space must be a *linear transformation*. Non-linear operations on the tangent argument, such as `jnp.sqrt(tangent)` or `jnp.sin(tangent)`, even though they can be transformed by JAX, are not permissible within the JVP augmentation functions. This implies that the finite difference method cannot be used to approximate the Jacobian-vector product, even though applying finite difference method might be simple in certain scenarios. The reason for the requirement of linear transformations will be explored later.

After defining the JVP rules for `mo_integrals`, we can utilize the `jax.grad` function to compute the CCD nuclear gradients, as demonstrated below:

```
In [1]: mol = pyscf.M(atom='N 0. 0 0; N 3. 0 0', unit='Bohr',
                     basis='cc-pvdz')
        mf = mol.RHF().run()
        grad = jax.grad(CCD_solve, argnums=0)
        print(grad(mol.atom_coords(), mf))

Out[1]:
E(MP2)=Traced<ConcreteArray(-109.13850360728081, dtype=float64)>with<
    JVPTrace(level=2/0)> with
    primal = Array(-109.13850361, dtype=float64)
    tangent = Traced<ShapedArray(float64[])>with<JaxprTrace(level=1/0)> with
        pval = (ShapedArray(float64[]), None)
        recipe = JaxprEqnRecipe(eqn_id=<object object at 0x7f50e45c4e30>,
                               in_tracers=(Traced<ShapedArray(...

...
cycle=19, E(CCD)=Traced<ConcreteArray(-109.03387828514111, dtype=float64)>
    with<JVPTrace(level=2/0)> with
    primal = Array(-109.03387829, dtype=float64)
    tangent = Traced<ShapedArray(float64[])>with<JaxprTrace(level=1/0)> with
        pval = (ShapedArray(float64[]), None)
        recipe = JaxprEqnRecipe(eqn_id=<object object at 0x7f50e407bc40>,
                               in_tracers=(Traced<ShapedArray(...
    primal = Array(-2.76017473e-05, dtype=float64)
```

```
tangent = Traced<ShapedArray(float64[])>with<JaxprTrace(level=1/0)> with
    pval = (ShapedArray(float64[]), None)
    recipe = JaxprEqnRecipe(eqn_id=<object object at 0x7f50e407bda0>,
                           in_tracers=(Traced<ShapedArray(...),
[[-2.78494891e-01 1.80279569e-16 7.12077149e-17]
 [ 2.78494891e-01 -1.80279569e-16 -7.12077149e-17]]
```

Step 4, optimizing the back-propagation

As illustrated by the output above, JAX creates back-propagation *tracers* for each iteration, which hold additional state information. To ensure enough information for back-propagation, JAX tracks the state generated during the normal execution of functions. To optimize back-propagation, we can utilize implicit differentiation, a technique that will be explored in more detail later in Section 16.4.2.

So far, we have demonstrated how to integrate non-JAX functions with JAX AD capabilities. In practice, to enhance the interaction between JAX AD code and the remaining Python code, it may be necessary to customize Python classes when differentiation involves their attributes.

16.4.1 Differentiable Python class

In our previous CCD-AD implementation, we directly passed the differentiable variable into the function that required differentiation. This approach implies that if additional parameters need to be included in the differentiation process, we would have to continually update the signature of the `CCD_solve` function. New parameters must be added to the argument list of `CCD_solve` as well as to the subsequent functions it calls. However, in many OOP Python projects, parameters are typically managed through classes. Introducing explicit arguments for differentiation might not align with the existing APIs of the underlying Python programs.

Fortunately, JAX provides customizable *PyTree nodes* to manage class attributes for JAX AD functions. This feature allows us to use classes as the primal to carry various differentiable attributes between JAX functions, which greatly simplifies the function signatures.

What is a PyTree? In JAX, a PyTrees represents a collection of data organized in a tree-like structure. When a differentiable argument consists of nested Python built-in data containers, such as lists, tuples, and dictionaries, JAX recognizes it as a PyTree. JAX can *flatten* the leaves of a PyTree into a list for internal processing. For the output of derivatives, a reverse operation *unflatten*, is applied to reassemble the leaves into the original tree structure. The leaves and the PyTree definitions are accessible through the JAX function

```
jax.tree_util.tree_flatten()
```

The PyTree can be reconstructed from the leaves using the function

```
jax.tree_util.tree_unflatten()
```

Leaves, tree definitions, and the applications of the two functions are illustrated in the following example, which is taken from the JAX documentation:

```
In [2]: from jax.tree_util import tree_flatten, tree_unflatten
      # The leaves in value_flat correspond to the '*' markers in
      value_tree
      value_flat, value_tree = tree_flatten([1., (2., 3.)])
      print(f"value_flat={}\n{value_tree=}")

      # Transform the flat value list using an element-wise numeric
      transformer
      transformed_flat = list(map(lambda v: v * 2., value_flat))
      print(f"transformed_flat={}")

      # Reconstruct the structured output, using the original
      transformed_structured = tree_unflatten(value_tree,
      transformed_flat)
      print(f"transformed_structured={}")

Out[2]: value_flat=[1.0, 2.0, 3.0]
        value_tree=PyTreeDef([*, (*, *)])
        transformed_flat=[2.0, 4.0, 6.0]
        transformed_structured=[2.0, (4.0, 6.0)]
```

JAX allows us to register custom classes as PyTree nodes [14] by adhering to the following guidelines:

- Flatten and unflatten functions must be provided to define how to convert the attributes of an instance into a `(children, metadata)` pair and how to perform the reverse conversion into an instance. The `children` variable represents the objects that are subject to differentiation, while `metadata` is used to store the non-differentiable attributes or other necessary information.
- To add a custom PyTree node to JAX’s internal types, the class must be registered using the `jax.tree_util.register_pytree_node` function, along with the provided flatten and unflatten functions.
- To compute the derivatives of an attribute, such as `a`, one can access it via `instance.a`. This represents the output of `jax.grad` or `jax.jacobian`. When processing higher-order derivatives, such as `jax.hessian`, the attribute can be accessed twice, like `instance.a.a`.

Using the CCD program as an example, the nuclear coordinates that we wish to differentiate are provided by the `mol` object, which is an instance of the `pyscf.gto.mole.Mole` class. Additionally, the `mol` object is an attribute of the `mf` object, an instance of the `pyscf.scf.hf.RHF` class. Consequently, it is necessary to register both the `RHF` and `Mole` classes as JAX PyTree types.

In the following, we introduce the `_RHF_flatten` function and the `_tree_unflatten` function to register the `RHF` class.

```
def _RHF_flatten(mf):
    children = (mf.mol,)
    metadata = (mf, ['mol']) # (1)
    return children, metadata

def _tree_unflatten(metadata, children):
    obj, keys = metadata
    # Create a copy of the original object
    out = object.__new__(obj.__class__)
    out.__dict__.update(obj.__dict__)
    for k, a in zip(keys, children):
        setattr(out, k, a)
    return out

jax.tree_util.register_pytree_node(
    pyscf.scf.hf.RHF, _RHF_flatten, _tree_unflatten)
```

In the `_RHF_flatten` function, the `mol` object is included in the `children` tuple because it contains the differentiable attribute `coords`. To rebuild the `RHF` instance in the `_unflatten` function, it is necessary to specify the names of the differentiable attributes and the remaining non-differentiable attributes of the `mf` object. For simplicity, we place the `mf` object in the `metadata` variable, as shown in line (1), to convey all the necessary information. Please note that the PyTree nodes created by this method may encounter type compatibility issues when dealing with higher-order derivatives. We will not explore this issue in depth here. Readers are encouraged to consult the JAX PyTree documentation for further details.

By applying a similar approach, we can also register the `Mole` class. Since the `coords` variable is not an attribute of the PySCF `Mole` instance, we check its availability and switch to the `Mole.atom_coords` method to generate the default values the first time this attribute is referenced. When the derivative functions return, the `coords` attribute, which represents the derivatives of the nuclear coordinates, becomes accessible. This attribute is created within the `_tree_unflatten` function.

```
def _Mole_flatten(mol):
    if hasattr(mol, 'coords'):
        coords = mol.coords
    else:
        coords = mol.atom_coords()
    children = (coords,)
    metadata = (mol, ['coords'])
    return children, metadata
```

```
jax.tree_util.register_pytree_node(
    pyscf.gto.mole.Mole, _Mole_flatten, _tree_unflatten)
```

As the two classes have been registered as JAX PyTree nodes, it is no longer necessary to explicitly pass the `coords` variable to CCD programs. We can pass only the `mf` object to the CCD program because it carries the differentiable attribute `coords` via the `mol` object. The function signatures of `CCD_solve` and `mo_integrals` are updated to reflect this change.

```
def CCD_solve(mf: pyscf.scf.hf.RHF, max_cycle=20, conv_tol=1e-5):
    H = mo_integrals(mf)
    ...

@jax.custom_jvp
def mo_integrals(mf: pyscf.scf.hf.RHF):
    ...

@mo_integrals.defjvp
def mo_integrals_jvp(primals, tangents):
    mf, = tangents
    disp = mf.mol.coords
    ...

...
```

After registering these Python classes and applying the necessary modifications, these classes can be seamlessly used in the JAX AD code and the original Python program. For example, the computation of the CCD nuclear gradients can be simplified to:

```
mol = pyscf.M(atom='N 0. 0 0; N 3. 0 0', unit='Bohr', basis='cc-pvdz')
mf = mol.RHF().run()
grad = jax.grad(CCD_solve, argnums=0)
dmf = grad(mf)
print(dmf.mol.coords)
```

16.4.2 Implicit differentiation

The `CCD_solve` function we developed earlier uses a fixed-point iteration to solve for the `t2` tensor. During this iteration process, JAX AD generates tracers for each iteration. Each tracer is responsible for storing the intermediate states of `t2` for that specific iteration, leading to a significantly large memory footprint. This AD code can be optimized using the implicit differentiation technique [15,16].

Consider the equation for a differentiable variable a , such as the nuclear coordinates, and certain free variables \mathbf{x} , like the `t2` tensor in the case of the CCD program

$$\mathbf{x} = f(a, \mathbf{x}(a)). \quad (16.40)$$

The solution \mathbf{x}^* to this equation can be obtained using fixed-point iteration, as implemented in the `CCD_solve` function. The resulting \mathbf{x}^* is then utilized to compute another quantity, such as the CCD energy. A Lagrangian for the CCD energy can be formulated with a Lagrange multiplier Λ to include the constraint (16.40):

$$\mathcal{L}(a, \mathbf{x}, \Lambda) = E(a, \mathbf{x}) - \Lambda \cdot (\mathbf{x} - f(a, \mathbf{x})). \quad (16.41)$$

We can then proceed to search for the maximum or minimum values of the Lagrangian. The stationary conditions of the Lagrangian lead to two equations:

$$\frac{\partial \mathcal{L}}{\partial \Lambda} = \mathbf{x} - f(a, \mathbf{x}) = 0, \quad (16.42)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = 0. \quad (16.43)$$

The first equation corresponds to the constraint (16.40). The second equation defines Λ . After simplification, it can be written in the following form:

$$(\mathbf{I} - \frac{\partial f(a, \mathbf{x})}{\partial \mathbf{x}}) \cdot \Lambda = \frac{\partial E(a, \mathbf{x})}{\partial \mathbf{x}}. \quad (16.44)$$

In terms of the Lagrangian defined in Eq. (16.41), the energy derivatives are ready to evaluate:

$$\frac{dE(a, \mathbf{x})}{da} = \frac{d\mathcal{L}}{da} = \frac{\partial \mathcal{L}}{\partial a} = \frac{\partial E}{\partial a} + \Lambda \cdot \frac{\partial f(a, \mathbf{x})}{\partial a}. \quad (16.45)$$

The linear equation (16.44) can be efficiently solved using a Krylov subspace solver. In the context of CCD fixed-point iteration, the function for this linear equation can be implemented as follows:

```
from .krylov import solve_krylov
def solve_lambda(H, t2_star, v):
    _, vjp_Ht = jax.vjp(lambda t2: update_CCD_amplitudes(H, t2), t2_star)
    def matvec(x):
        return -np.asarray(vjp_Ht(x.reshape(v.shape))[0]).ravel()
    l = solve_krylov(matvec, np.asarray(v).ravel())
    return l.reshape(v.shape)
```

The `jax.vjp` function produces the operation `vjp_Ht`, which facilitates the computation of the product between the vector Λ and the Jacobian $\frac{\partial f(a, \mathbf{x})}{\partial \mathbf{x}}$. The term $\frac{\partial E(a, \mathbf{x})}{\partial \mathbf{x}}$ on the right-hand side of Eq. (16.44), which acts as the “force” in the CPHF equation, is derived using JAX AD

```
jax.jacrev(lambda t2: get_CCD_corr_energy(H, t2))(t2_star)
```

The output of the `solve_lambda` function corresponds to the de-excitation operator Λ in the coupled cluster energy functional theory [17]. In the framework of coupled cluster energy functional, the equations for Λ are derived using the complex diagram

method. In contrast, in the JAX AD approach, there is no need for any complex algebraic derivations. The Λ operator is obtained with just a few JAX AD statements!

Let's explore further the solvers for the linear equation (16.44) here. Besides the Krylov subspace solver, Eq. (16.44) can be solved using other methods, such as DIIS acceleration or the GMRES (Generalized Minimal Residual Method) solver. For scenarios that require JAX to perform higher-order derivatives, the linear equation solver must also be JAX-transformable. In such cases, solvers provided by the `JAX.scipy.linalg.sparse` module are suitable choices.

```
def solve_lambda(H, t2_star, v):
    _, vjp_Ht = jax.vjp(lambda t2: update_CCD_amplitudes(H, t2), t2_star)
    def matvec(x):
        return x - vjp_Ht(x)[0]
    return jax.scipy.sparse.linalg.gmres(matvec, v, tol=1e-5)[0]
```

Integrating the Λ solver into the CCD program requires some modifications to the `CCD_solve` function.

```
def CCD_solve(mf):
    H = mo_integrals(mf)
    t2 = fixed_point(H)
    e = H['e_hf'] + get_CCD_corr_energy(H, t2)
    return e

# The fixed point iteration
@partial(jax.custom_vjp, nondiff_argnums=(1, 2))
def fixed_point(H, conv_tol=1e-5, max_cycle=100):
    '''A fixed-point iteration solver for spin-orbital CCD'''
    e_ccd, t2 = mp2(H) # initial guess
    print(f'E(MP2)={e_ccd}')
    e_hf = H['e_hf']
    with tempfile.TemporaryDirectory() as tmpdir:
        diis = DIIS(f'{tmpdir}/diis')
        for cycle in range(max_cycle):
            t2, t2_prev = update_CCD_amplitudes(H, t2), t2
            e_ccd, e_prev = get_CCD_corr_energy(H, t2) + e_hf, e_ccd
            print(f'(cycle={cycle}), E(CCD)={e_ccd}, dE={e_ccd-e_prev}')
            if jnp.abs(t2 - t2_prev).max() < conv_tol:
                break
            t2 = diis.update(t2 - t2_prev, t2)
    return t2

def fp_fwd(H, conv_tol=1e-4, max_cycle=100): # (1)
    t2 = fixed_point(H, conv_tol, max_cycle)
    return t2, (H, t2)
```

```

def fp_bwd(conv_tol, max_cycle, res, cotangent): # (2)
    H, t2_star = res
    L = solve_lambda(H, t2_star, cotangent)
    _, vjp_R = jax.vjp(lambda H: update_CCD_amplitudes(H, t2_star), H)
    # vjp_R computes the product of L and the Jacobian of update_t2
    return vjp_R(L)

fixed_point.defvjp(fp_fwd, fp_bwd) # (3)

```

The iterations for t_2 are collected in a dedicated `fixed_point` function. Subsequently, the CCD energy calculation is performed separately, taking the output of the `fixed_point` function. The `custom_vjp` is utilized to define the reverse-mode rules for the `fixed_point` function.

In line (3), the `custom_vjp.defvjp` method is invoked with two augmentation functions: `fp_fwd` in line (1) for forward propagation, and `fp_bwd` in line (2) for backward propagation. The signature of the forward propagation function, `fp_fwd`, is identical to that of the primal function. In its output, `fp_fwd` is expected to provide the primal results along with a residual, which will be passed to the `fp_bwd` function. This design convention enables the forward and backward propagation processes to share certain intermediates, thereby minimizing computational overhead to some extent. The backward function, `fp_bwd`, requires at least two input arguments: the residual data from `fp_fwd` and the cotangents associated with the output of the primal function. If the primal function includes non-differentiable arguments, as demonstrated in our example, these arguments should be passed as the leading arguments in `fp_bwd`, regardless of their position in the primal function's signature.

In the primal function `fixed_point`, DIIS acceleration has been applied. Unlike our initial CCD-AD program, this version does not require the DIIS function to be JAX-transformable. The DIIS iteration is executed by the JAX forward propagation only. During the back-propagation, JAX invokes the custom `fp_bwd` function, thereby skipping the DIIS procedure.

Please note the anonymous `lambda` functions in `fp_bwd` and `solve_lambda`. They are defined against differentiable variables for the same `update_CCD_amplitudes` function. In `solve_lambda`, the differentiable variable is t_2 , which corresponds to the VJP operation:

```
jax.vjp(lambda t2: update_CCD_amplitudes(H, t2), t2_star)
```

In `fp_bwd`, the differentiable variable is H . The VJP operation is achieved with the following code:

```
jax.vjp(lambda H: update_CCD_amplitudes(H, t2_star), H)
```

Compared to the original CCD-AD program, the implicit differentiation method introduced in this section provides lower computational costs and a reduced memory footprint. These advantages can be attributed to the DIIS acceleration in `CCD_solve`,

and the fewer tracers generated by back-propagation. In the implicit differentiation approach, only one partially evaluated `update_CCD_amplitudes` tracer is created, in contrast to the multiple tracers generated for each iteration in the earlier versions of CCD-AD program.

Fixed-point iterations are extensively used in quantum chemistry programs. Besides the coupled cluster programs, we can also find them in self-consistent field iterations, CPHF equations, and other areas. The implicit differentiation method is an ideal tool for computing derivatives in all these methods. This approach has been adopted as the default solver in the PySCFAD library [8]. If you are interested in exploring more quantum chemistry applications of JAX AD, within the PySCFAD library, you can find additional practical examples of the techniques discussed in this section.

Fixed-point iteration is a fundamental technique in scientific computing applications. As such, the `jaxopt` library provides a specialized tool for implementing implicit differentiation. This tool is the `custom_fixed_point` function, accessible through the module `jaxopt.implicit_diff` [16]. Consequently, it is unnecessary to implement `custom_vjp` rules for the fixed-point function. We only need to specify the updating function and the corresponding fixed-point solver for the `custom_fixed_point` decorator as follows:

```
@custom_fixed_point
    lambda t2, H, *args, **kwargs: update_CCD_amplitudes(H, t2),
    solve=partial(jaxopt.linear_solve.solve_gmres, tol=1e-5))
def fixed_point(t2, H, conv_tol=1e-5, max_cycle=100):
    """A fixed-point iteration solver for spin-orbital CCD"""
    e_ccd, t2 = mp2(H) # initial guess
    print(f'E(MP2)={e_ccd}')
    e_hf = H['e_hf']
    with tempfile.TemporaryDirectory() as tmpdir:
        diis = DIIS(f'{tmpdir}/diis')
        for cycle in range(max_cycle):
            t2, t2_prev = update_CCD_amplitudes(H, t2), t2
            e_ccd, e_prev = get_CCD_corr_energy(H, t2) + e_hf, e_ccd
            print(f'{cycle}, E(CCD)={e_ccd}, dE={e_ccd-e_prev}')
            if jnp.abs(t2 - t2_prev).max() < conv_tol:
                break
            t2 = diis.update(t2 - t2_prev, t2)
    return t2
```

16.4.3 Higher-order derivatives

JAX supports higher-order derivatives for normal JAX-transformable functions. Does this capability extend to functions defined with custom derivative rules?

If the JVP or VJP augmentation functions and all subsequent functions are defined with JAX-transformable functions, JAX AD can handle higher-order derivatives smoothly. Let's examine the case of `custom_jvp` in our CCD program. A similar approach is applicable to the `custom_vjp` scenario.

In our initial CCD-AD program, the `custom_jvp` for `mo_integrals` is implemented as follows:

```
@partial(jax.custom_jvp, nondiff_argnums=(1,))
def mo_integrals(coords, mf: pyscf.scf.hf.RHF):
    ...
    @mo_integrals.defjvp
    def mo_integrals_jvp(mf, primals, tangents):
        coords, = primals
        ...
        primals_out = mo_integrals(coords, mf)
    return primals_out, jvp_product
```

Inside the `mo_integrals_jvp` function, the `primals_out` is generated by the JAX-transformable function `mo_integrals`. When invoking the nuclear Hessian in this CCD program, JAX AD can execute without raising any errors. However, the results must be incorrect because we have not implemented any integrals for higher-order derivatives.

To prevent JAX from erroneously attempting to compute higher-order derivatives, one effective method is to use a non-JAX-transformable function to generate the `primals_out`. For example, by isolating the functionality of `mo_integrals` into a separate function, `_mo_integrals_no_ad`, which is not JAX-transformable, we restrict the application of the CCD-AD code to only first-order derivatives.

```
def _mo_integrals_no_ad(coords, mf: pyscf.scf.hf.RHF):
    ...
    @partial(jax.custom_jvp, nondiff_argnums=(1,))
    def mo_integrals(coords, mf: pyscf.scf.hf.RHF):
        return _mo_integrals_no_ad(coords, mf)

    @mo_integrals.defjvp
    def mo_integrals_jvp(mf, primals, tangents):
        coords, = primals
        ...
        primals_out = mo_integrals_no_ad(coords, mf)
    return primals_out, _mo_integrals_common(mol, hcore, eri, e_hf)
```

When computing higher-order derivatives, the `primals` argument will contain nested tracers instead of regular inputs, corresponding the level of derivatives to be

computed. Passing these arguments to the customized JVP primal function can initiate a recursive call to the JVP augmentation function, with a reduced level of nested tracers in `primals`. This recursive calling pattern must be carefully considered when designing the JVP augmentation function for higher-order derivatives.

To customize higher-order derivatives, such as integrating higher-order derivatives for GTO integrals within the augmentation function, we may need to track the current derivative level within the JVP augmentation function. Testing the level of the tracers in the `primals` argument is one method. Alternatively, we can explicitly introduce a counter to represent the level of the derivatives. For example, in the JVP augmentation code provided below, different JVP rules are applied depending on the level of the derivatives. The derivative counter is managed through the non-differentiable arguments.

```
@partial(jax.custom_jvp, nondiff_argnums=(1,))
def f(x, level): # level=0 indicates the second order
    return x

@f.defjvp
def f_jvp(level, primals, tangents):
    x, = primals
    primals_out = f(x, level-1)
    tangent_out = custom_jvp_rules(level, primals, tangents)
    return primals_out, tangent_out

jax.hessian(f)(3.14, 2)
```

16.4.4 Conversion between JVP and VJP

Earlier, we mentioned that JAX can automatically convert the standard forward-mode AD (the JVP operation) into the reverse-mode AD (the VJP operation). You might have the question: how does JAX achieve this magical transformation from JVP to VJP? In short, JAX employs a *linearize-transpose* algorithm to accomplish the JVP-VJP conversion [18].

The JVP augmentation function defines the Jacobian-vector product, acting as a linear transformation for the tangent input. JAX employs specific tracers to track the tangent inputs of the JVP augmentation function. By isolating and partially evaluating code that is irrelevant to the tangent input, the JVP rules are transformed into a linearized function. The linearized function is usually simpler and faster. It primarily involves tensor contractions with the tangents or accumulation operations for some intermediates.

JAX then carries out *transposition*, which is a process of binding the linearized operations to the cotangent input associated with the VJP action. This process results in an operation that is equivalent to the VJP product between the transposed Jacobian and the cotangent input.

Due to the requirements of linearization, the JVP rules cannot be approximated using the finite difference method. JAX is unable to transform the finite difference method into a linearized operation. In fact, *JAX cannot process any non-linear operations on the tangent input.* It will identify and reject any attempts to perform non-linear operations on the tangent input, regardless of whether the operations are JAX-transformable.

The VJP custom rules are directly applicable in back-propagation, whereas the JVP customization involves the linearize-transpose transformation. This raises an interesting question: Is `custom_vjp` more efficient than `custom_jvp` for back-propagation tasks?

Although certain terminologies are shared between reverse-mode AD and back-propagation, it does not imply that `jax.custom_vjp` is inherently more efficient than `jax.custom_jvp`. JAX provides these two APIs, allowing us to flexibly define derivative rules using either method. When the JAX JIT compiler processes the differentiation, it can optimize the entire computation graph. As a result, the two derivative customization approaches may eventually yield functions with comparable computational complexity.

Regarding the use of `custom_jvp` and `custom_vjp`, it is beneficial to understand two additional points:

- JAX does not support defining both `custom_jvp` and `custom_vjp` for the same primal function.
- If `custom_vjp` is implemented, JAX cannot automatically convert the VJP rules into JVP rules. Consequently, the forward-mode AD functions, such as `jax.jacfwd`, will not work with `custom_vjp`.

Therefore, except for special optimization methods such as implicit differentiation, it is advisable to focus on JVP rules when customizing derivatives in JAX.

Summary

In this chapter, we explored various methods for calculating molecular properties, focusing particularly on the use of JAX automatic differentiation to compute nuclear gradients. A unique feature of JAX automatic differentiation is its hybrid mode, which combines analytical gradient functionalities from existing programs with automatic differentiation capabilities. We demonstrated the process of developing hybrid code for calculating the nuclear gradients of the coupled cluster with double excitation (CCD) method. In this nuclear gradients program, certain first-order quantities, such as integrals and Hartree-Fock orbitals, are computed analytically, while the remainder is derived through JAX automatic differentiation.

The challenge of developing a JAX automatic differentiation program is to ensure that all relevant functions are transformable by JAX. To achieve this in the hybrid code, we made several adjustments to the existing CCD program. These adjustments include adopting a functional programming style, rewriting certain code blocks to

eliminate side effects, and substituting NumPy and SciPy functions with their JAX equivalents. For functions that are not directly transformable by JAX, we defined `custom_jvp` interfaces to make them compatible with JAX transformations. Additionally, for user-defined classes within the program, we employed JAX PyTree nodes to extend the data types that JAX can handle. Given that solving the CCD problem involves a process of fixed-point iteration, we utilized the technique of implicit differentiation to enhance the automatic differentiation of this fixed-point iteration. The `custom_vjp` interface was employed in the implicit differentiation method. However, aside from this special scenario, it is generally preferable to customize the JVP interface as it is more straightforward to develop and maintain.

References

- [1] The PySCF Developers, Quantum chemistry with Python, <https://pyscf.org/>, 2024.
- [2] Y. Yamaguchi, H.F. Schaefer III, Analytic Derivative Methods in Molecular Electronic Structure Theory: A New Dimension to Quantum Chemistry and Its Applications to Spectroscopy, John Wiley & Sons, Ltd, 2011, <https://doi.org/10.1002/9780470749593.hrs006>, <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470749593.hrs006>, <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470749593.hrs006>.
- [3] J.A. Pople, R. Krishnan, H.B. Schlegel, J.S. Binkley, Derivative studies in Hartree–Fock and Møller–Plesset theories, in: Proceedings of the International Symposium on Atomic, Molecular, and Solid-State Theory, Collision Phenomena, and Quantum Statistics, and Computational Methods, International Journal of Quantum Chemistry 16 (S13) (1979) 225–241, <https://doi.org/10.1002/qua.560160825>.
- [4] G.H. Golub, C.F. Van Loan, Matrix Computations, 4th edition, Johns Hopkins University Press, Philadelphia, PA, 2013, <https://doi.org/10.1137/1.9781421407944>, <https://pubs.siam.org/doi/pdf/10.1137/1.9781421407944>, <https://pubs.siam.org/doi/abs/10.1137/1.9781421407944>.
- [5] M.F. Kasim, S. Lehtola, S.M. Vinko, DQC: a Python program package for differentiable quantum chemistry, Journal of Chemical Physics 156 (8) (2022) 084801, <https://doi.org/10.1063/5.0076202>, https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0076202/18277993/084801_1_5.0076202.pdf.
- [6] A.S. Abbott, B.Z. Abbott, J.M. Turney, H.F.I. Schaefer, Arbitrary-order derivatives of quantum chemical methods via automatic differentiation, Journal of Physical Chemistry Letters 12 (12) (2021) 3232–3239, pMID: 33764068, <https://doi.org/10.1021/acs.jpclett.1c00607>.
- [7] G. Zhou, B. Nebgen, N. Lubbers, W. Malone, A.M.N. Niklasson, S. Tretiak, Graphics processing unit-accelerated semiempirical Born Oppenheimer Molecular Dynamics using PyTorch, Journal of Chemical Theory and Computation 16 (8) (2020) 4951–4962, pMID: 32609513, <https://doi.org/10.1021/acs.jctc.0c00243>.
- [8] X. Zhang, G.K.-L. Chan, Differentiable quantum chemistry with PySCF for molecules and materials at the mean-field level and beyond, Journal of Chemical Physics 157 (20) (2022) 204801, <https://doi.org/10.1063/5.0118200>, https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0118200/18281172/204801_1_5.0118200.pdf.
- [9] J. Denero, SICP in Python, <https://wizardforcel.gitbooks.io/sicp-in-python/content/>, 2024.

- [10] The JAX authors, JAX documentation - jax.numpy.ndarray.at, https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html, 2024.
- [11] The JAX authors, Custom derivative rules for JAX-transformable Python functions, https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html, 2024.
- [12] The JAX authors, Use jax.custom_jvp to define forward-mode (and, indirectly, reverse-mode) rules, https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html#use-jax-custom-jvp-to-define-forward-mode-and-indirectly-reverse-mode-rules, 2024.
- [13] The JAX authors, jax.custom_jvp with nondiff_argnums, https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html#jax-custom-jvp-with-nondiff-argnums, 2024.
- [14] The JAX authors, Extending PyTrees, <https://jax.readthedocs.io/en/latest/pytrees.html#extending-pytrees>, 2024.
- [15] The JAX authors, Implicit function differentiation of iterative implementations, https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html#implicit-function-differentiation-of-iterative-implementations, 2024.
- [16] The JAXOpt authors, JAXOpt documentation - implicit differentiation, https://jaxopt.github.io/stable/implicit_diff.html, 2024.
- [17] P.G. Szalay, M. Nooijen, R.J. Bartlett, Alternative ansätze in single reference coupled-cluster theory. III. A critical analysis of different methods, Journal of Chemical Physics 103 (1) (1995) 281–298, <https://doi.org/10.1063/1.469641>, https://pubs.aip.org/aip/jcp/article-pdf/103/1/281/19147216/281_1_online.pdf.
- [18] A. Radul, A. Paszke, R. Frostig, M.J. Johnson, D. Maclaurin, You only linearize once: tangents transpose to gradients, Proc. ACM Program. Lang., 7 (POPL), <https://doi.org/10.1145/3571236>.

Index

A

Abstract Syntax Tree (AST), 176, 177, 193, 364, 652
code, 181
documentation, 179
output, 181
tree, 177
Adaptive planner, 161
Addition (ADD) units, 324
Ahead-of-Time (AOT)
compilation, 347, 352, 364
compilers, 362
AI programming assistant, 51
Analytical
GTO integrals, 484
integral evaluation, 478
nuclear gradients, 677
Angular momentum coupling, 185
Apache arrow, 218
API
cloud, 231
code, 265, 283, 303
CUDA, 466, 468
CuPy, 453
functionality, 30
MPI, 437
NumPy, 453, 455, 456
OpenMP, 435
Python, 436, 456
PyTorch, 454
Application Binary Interface (ABI)
conventions, 296, 299
level interface, 289
Approximating quadratures, 544
Architecture-aware optimization, 456
Architecture-independent optimization, 452
Array
addressing efficiency, 517
views, 75
Assembly code, 353
Asynchronous
code, 374, 399, 417
data transfers, 458, 459, 470
execution, 315, 454
I/O operations, 554
modules, 395
programming, 416, 417, 446

Atomic

instructions, 307
Python operations, 307
results superposition, 607
Atomic orbital (AO), 552
basis, 685
integrals, 675
Attribute-based indexing, 101
Augmented Roothaan-Hall DIIS (ADIIS), 605
Auto-scaling group (ASG), 230
Automated
build, 38
testing, 38
Automatic
code deduction, 173
threading parallelization, 633–635, 639
Automatic differentiation (AD), 173, 186, 385, 388, 682, 690, 711
capabilities, 710
technique, 193, 675
Auxiliary basis function (ABF), 598

AWS

access credentials, 252
APIs, 257

B

B-spline interpolation, 545
Back-propagation tracers, 700
Baker-Campbell-Hausdorff (BCH) expansion, 662
Base
class, 3, 48, 593, 596
SCF class, 565
Basis Set Exchange (BSE)
database, 138, 480
service, 212
Batch services, 227
Benchmark tests, 327
Binary
code, 347, 535
serialization, 196
BLAS library, 155, 280, 290, 294, 296
Block Sparse Row (BSR) format, 148, 371
Blocked threads, 404
Boys function, 489, 492, 525, 526, 532, 539, 541
Breadth-first search, 411
Broadcasting, 62, 64, 65, 96
code, 349, 361, 515, 519
mechanism, 96

- NumPy, 156, 187
- operations, 349, 355, 360, 515
- Pandas, 98
- rules, 64, 69, 81, 99
- scalar functions, 86
- Buffer
 - contiguous memory, 438
 - data, 71, 72, 75, 77, 78, 207, 290, 294, 312, 421, 428, 431
 - memory, 73, 290, 292–294, 312, 316, 323, 479, 552–555
 - mode, 438
 - NumPy array, 294, 430
 - object, 294, 437, 438
 - protocol, 220
- Bypassing intermediate arrays, 533
- Bytecode
 - analysis, 340
 - execution, 398
 - information, 340
 - instructions, 338, 402
 - parallel processing, 398
 - Python, 312, 347, 392, 632
- C**
 - C-contiguous storage, 73, 74
 - C/C++ interfaces, 267
 - Cache
 - accesses, 314, 345
 - coherence, 318, 319, 323, 324, 392, 398, 635
 - CPU, 314, 315, 323, 571, 630, 635
 - data, 319
 - dictionary, 649, 651
 - effectiveness, 349
 - efficiency, 318, 320, 322, 327
 - GPU, 318
 - hit, 571
 - hit rates, 62, 534, 630
 - line, 316, 319, 321, 323, 324
 - lookups, 377, 510
 - management strategies, 375
 - managing, 375
 - memory, 648
 - miss rates, 315
 - size, 317, 319, 322, 630, 648, 649
 - synchronization, 323, 324
 - utilization, 315, 319, 346, 392
 - Cached
 - data, 603
 - function, 376
 - indirect indices, 347
 - keys, 503
 - Callback, 288
 - Callback Python object, 274
 - Canonical HF orbitals, 678
 - Cartesian
 - components, 487, 490, 491
 - directions, 486, 487, 489
 - GTOs, 486, 487
 - Celery, 247
 - library, 248
 - tasks, 253
 - Center-of-charge coordinates, 486
 - CFFI package, 289
 - Chebyshev
 - approximation, 546, 549
 - polynomial, 546, 547
 - polynomial methods, 548
 - Child
 - class, 43, 46, 47
 - threads, 397, 399, 421, 425
 - Cholesky decomposition (CD), 598
 - Cholesky-decomposed ERI (CDERI)
 - class, 604
 - tensor, 598, 600–603
 - Chunk storage, 601
 - Chunked storage, 366
 - Class
 - attributes, 596, 700
 - DIIS, 587, 605
 - hierarchy, 48, 590
 - inheritance, 43, 590, 610
 - management, 589
 - serialization challenge, 597
 - Clebsch-Gordan coefficients, 185
 - Client
 - attributes, 592
 - class, 592
 - Closure, 586
 - Cloud
 - APIs, 231
 - computing APIs, 224
 - management toolkits, 226
 - resource management, 247
 - services, 234
 - storage APIs, 225
 - Cluster amplitudes, 642
 - Coalesced memory
 - access, 471–473
 - transaction, 471

Code
 annotation, 361
 asynchronous, 374, 399, 417
 binary, 347, 535
 broadcasting, 349, 361, 515, 519
 compiled, 58, 361, 570
 coverage, 30
 CUDA, 460, 461, 469
 Cython, 283, 357, 359, 361
 data serialization, 429
 duplication, 590
 editors, 31, 593
 ERI programs, 504
 example, 23, 206, 215, 500, 526, 543, 592, 624, 666
 execution, 49
 files, 11, 348
 generation, 173, 175, 180, 185, 300, 477, 520, 641, 656, 672
 generator, 520, 521, 523
 JAX AD, 700, 703
 memory management, 268
 NumPy, 58, 64, 70, 74, 75, 77, 86, 190, 453, 472
 NumPy arrays, 290
 NumPy ufuncs, 360
 optimization, 70, 477, 614
 parallel programs, 395
 quality analysis, 38
 readability, 31, 38, 155
 release, 38
 SciPy, 452
 symbolic, 672
 SymPy, 184
 tensor contraction, 155, 491, 604
 unrolling, 315, 520, 528
 Coding style, 31
 Collective communication methods, 437
 Command-line interface (CLI), 12
 functionality, 13
 tools, 12
 Compiled
 PTX assembly code, 468
 shared library, 469
 Compiler
 Cython, 51, 275, 276
 directives, 356, 363, 535
 flag, 279
 intrinsics, 363, 535
 manual, 297
 Numba, 531, 547
 Numba JIT, 515, 526, 531
 options, 34, 363, 364
 settings, 281
 Compiling Python code, 305, 347, 355, 515
 Compound data type, 91
 Comprehension code, 174
 Compressed Sparse Column (CSC) format, 148
 Compressed Sparse Row (CSR) format, 148, 371
 Computation
 bound, 316
 overhead, 322, 520
 thread, 403–405
 Compute capability, 451
 Computer Algebra System (CAS), 656
 Computing resources, 227
 Concatenating strings, 208, 343
 Concise Python integral program, 531
 Concurrent
 mode, 313
 program, 405
 threads, 395
 Conda, 17
 environment, 18
 operations, 17
 package, 32, 37, 38
 package management system, 17, 18
 recipe file, 37
 Condition variable, 403
 Configuration Interaction (CI), 613
 coefficients, 619
 energy, 614
 method, 614
 strings, 616
 Consecutive memory
 addresses, 471
 locations, 74
 Containerized solution, 227
 Context
 manager, 405, 459, 467
 switch, 312
 Contiguous memory
 access, 322
 address, 635
 buffer, 438
 Continuous Deployment (CD), 38
 Continuous Integration (CI), 30, 38
 Contracted strings, 667, 668
 Contraction
 coefficients, 533
 paths, 533
 Control statements, 136
 Converting recursive code, 502
 Coordinate (COO) format, 148, 371
 Copy-on-write (COW)
 mechanism, 421, 423
 technique, 419, 431

- Coroutine, 586
 - Cost estimation, 306
 - Coulomb
 - energy, 560
 - matrix, 567, 572
 - matrix construction code, 559
 - Coupled cluster (CC)
 - diagram methods, 671
 - methods, 641
 - programs, 641, 643
 - theory, 641, 642, 656
 - Coupled cluster with doubles (CCD)
 - code, 641
 - equations, 665
 - method, 642, 675, 710
 - program, 641, 643, 645, 647, 648, 652, 653, 692, 695, 696, 701, 703, 705
 - tensors, 666
 - Coupled perturbed Hartree-Fock (CPHF) equation, 684, 686, 704, 707
 - CPU
 - cache, 314, 315, 323, 571, 630, 635
 - cache utilization, 639
 - thread, 371
 - CPython
 - executables, 4
 - interpreter, 19
 - Cross-platform compatibility, 198
 - Cube
 - file, 136, 141
 - format, 140
 - CUDA
 - APIs, 466, 468
 - library, 452
 - programming, 449, 460, 466
 - programming model, 460
 - runtime, 450, 451
 - environments, 451
 - toolkit, 451
 - stream, 458
 - threads, 461–463
 - Cumulative overhead, 516
 - CuPy, 453
 - APIs, 453
 - array, 457, 458, 469–471
 - custom kernel, 460, 466
 - documentation, 453, 467
 - functions, 454, 458–460, 467
 - library, 454, 455, 469
 - RawKernel, 460
 - Custom
 - data types, 288, 543
 - diagonalization program, 619
 - Docker runtime, 42
 - encoder, 200
 - GPU kernels, 460
 - memory allocator, 537
 - Customized JVP primal function, 709
 - Cython
 - annotations, 359, 519
 - compilation, 335, 338, 361, 362, 519, 531, 557, 639
 - compiler, 51, 275, 276
 - compiler directives, 356, 365, 519
 - header file, 276
 - memoryview, 355
 - transpiler, 355, 531
 - version, 519, 531
 - Cythonized code, 359, 361
- D**
- Dangling Pointer, 292
 - Dask, 253
 - client, 254
 - cluster, 253, 254
 - cluster scheduler, 254
 - distributed executors, 253
 - distributed systems, 256
 - scheduler, 254
 - workers, 254, 255
 - Data
 - accessibility, 224
 - buffer, 71, 72, 75, 77, 78, 207, 290, 294, 312, 421, 428, 431
 - cache, 319
 - communication, 419, 422
 - compression, 602
 - layout, 601
 - movement overhead, 168, 473
 - prefetching thread, 403
 - processing threads, 403
 - serialization, 196, 221
 - code, 429
 - process, 207
 - schemes, 196
 - sharing, 397
 - structure, 70, 478
 - transfer, 224, 229, 254, 312, 316, 318, 419–421, 427–429, 434, 437, 452, 455, 458, 459, 472, 613, 648
 - efficiency, 434, 445, 470
 - method, 434
 - operations, 445
 - overhead, 159, 613, 637
 - process, 316, 419, 457
 - rates, 314

- transferring services, 224
- type
 - conversions, 310, 311
 - declarations, 275
 - incompatibilities, 303
- DataFrame, 93
- Davidson diagonalization, 617
- Deadlocks, 406
- Declaration code, 276
- Default
 - data types, 455, 456
 - Docker command, 451
 - implementation, 155
 - pickle protocol, 198
 - storage mode, 73
 - stream, 458, 459, 467
- Density fitting (DF) approximation, 566, 598
- Density Functional Theory (DFT), 43, 225
 - class, 44
 - task executor, 235
- Dependency dictionary, 179
- Deploying RPC server, 239
- Depth-first search (DFS) pattern, 411
- Derivative
 - integrals, 679
 - tensors, 579, 584
- Deserialization
 - operations, 220
 - pickle, 202
 - processes, 587
- Device memory, 457
- Dictionary
 - cache, 649, 651
 - comprehension, 174, 342, 392
 - lookup, 308, 309, 376, 382
 - nested, 94, 199
 - unpacking operation, 342
- Dictionary of Keys (DOK) format, 148
- Dirac-Hartree-Fock (DHF), 566
- Dirac-Kohn-Sham (DKS), 566
- Direct CI algorithm, 623
- Direct Inversion in Iterative Subspace (DIIS), 584
 - class, 587, 605
 - code, 587
 - extrapolation method, 584
 - functionalities, 586
 - object, 587, 589, 608
 - program, 585, 586
 - serialization, 605
- Direct memory access (DMA), 316, 470
- Disassembled
 - bytecode, 340
 - Python source code, 338
- Discrete Fourier Transform (DFT), 157, 158
 - matrix, 146
 - operations, 86
- Disk performance, 311
- Distributed job executors, 246
- Distribution file, 32
- Docker
 - command, 30
 - configuration file, 451
 - container, 7, 18–21, 23, 30, 451
 - image, 20, 21, 40, 224–226, 228, 230, 250
 - official documentation, 19
 - rootless mode, 19
 - runtime, 42
- Docstring, 49
- Documentation files, 33
- Duplicated
 - function names, 294
 - objects, 654
- Dynamic
 - mean-field classes, 594
 - patches, 592
- Dynamic programming (DP), 380
 - approach, 485
 - code, 381
 - implementation, 486
 - methodology, 380
 - state, 382–384, 487, 503, 505–507, 510, 513
 - state array, 510

E

- Early contraction, 533
- Efficiency
 - cache, 318, 320, 322, 327
 - computational, 170, 477, 524, 630
 - data transfer, 434, 445, 470
 - FFT, 169
 - improvements, 392
 - memory, 76, 163, 268, 615, 628, 632, 639
 - memory access, 74, 322, 398, 477
 - memory management, 410, 614, 639
 - multithreading, 446
 - parallel, 168, 170, 413, 633, 638
 - parallel computation, 613
 - serialization, 198
- Elastic Container Registry (ECR), 227
- Elastic Container Service (ECS), 227
- Electron repulsion integral (ERI), 484, 488, 491, 496, 498, 501, 566
 - code, 520
 - computation, 528
 - performance factors, 531

- program, 529
- tensor, 396, 399, 457, 464, 471, 472, 491, 495, 528, 552, 567, 570, 572, 598, 600, 699
- Electron Spin Resonance (ESR), 675
- Electron-transfer RR (TRR), 492
- Electronic coordinates, 679
- Eliminating
 - Python classes, 518
 - temporary objects, 343
- Encoder
 - custom, 200
 - JSON, 200
- Energy DIIS (EDIIS), 605
- Energy efficiency, 315
- Error
 - handling code, 440
 - management, 397
- Eval function, 657
- Excessive
 - memory usage, 400, 413
 - threads, 329
- Exchange matrices, 567, 572
- Exchange-correlation (XC)
 - derivative tensors, 582
 - energy, 560, 565
 - functional, 561, 565, 576
 - functional libraries, 577
 - libraries, 580
 - matrix, 609
 - matrix elements, 577
 - potential, 576
- Executable and Linkable Format (ELF), 281
- Executing
 - bytecode, 397
 - legacy Python code, 23
- Explicit data types, 518
- Expression template technique, 359
- External libraries, 279

- F**
- F-contiguous storage, 73, 79
- False
 - cache sharing contention, 635
 - sharing, 398
- Fancy indexing, 64, 67–69, 76, 695
 - approach, 69
 - code, 64, 65
 - expression, 65
 - method, 69, 70
 - NumPy, 57, 99
 - operation, 65
 - syntax, 66
- Fast Fourier Transform (FFT)
 - algorithm, 158
 - dimension, 169
 - efficiency, 169
 - length, 169
 - libraries, 159, 160, 162, 163, 165, 167–169
 - performance, 162, 169, 170
 - performance benchmark, 162
 - planner, 161
 - SciPy, 158, 165
- Fast math operations, 349
- FFTPACK library, 159
- File
 - descriptors, 376
 - formats, 135, 137
 - I/O, 202
 - locations, 436
 - object, 197, 202, 207, 208
 - operations, 208
 - path, 436
 - system, 6, 7, 431, 432, 434, 603
- First order molecular orbitals, 684
- First-In/First-Out (FIFO), 375, 648
 - cache, 648, 656
 - queue, 399
- Flexible array structure, 71
- Floating-point operation (FLOP)
 - counts, 499
 - efficiency, 325
- Foreign Function Interface (FFI), 283, 296, 303
- Fortran
 - code, 301
 - function, 297
 - interfaces, 296
 - MPI programs, 437
 - programming language, 73
 - programs, 435
- Fourier transform, 549
- Free unused memory blocks, 454
- Full Configuration Interaction (FCI), 153, 613, 614
 - code, 635
 - method, 380, 614, 617
 - program, 317, 318, 416, 615, 639
 - wavefunction, 613, 620, 639
- Function
 - attributes, 363
 - cached, 376
 - inline, 315, 348
 - multithreaded, 397
 - pointer, 274
 - prototype, 284
 - runtime, 295
 - signature, 518

Function-as-a-Service (FaaS), 241
 Functional programming (FP), 375–378, 380
 approach, 691
 paradigm, 190
 style, 190, 376, 710
 Fundamental symbolic elements, 657
 Fused Multiply-Add (FMA)
 instruction, 349, 355
 operations, 324
 units, 324

G

Gaussian quadrature, 538
 Gaussian type orbital (GTO), 477, 478
 basis, 478
 shells, 479, 481, 483, 532, 568–571
 GCC compiler, 279, 355, 363, 364
 Generalized Gradient Approximation (GGA)
 functional, 577
 Generalized HF (GHF), 565
 Generalized Minimal Residual Method (GMRES)
 solver, 705
 Git
 commands, 23
 submodule, 29
 GitHub repository, 24
 Global Interpreter Lock (GIL), 312, 397, 446, 637
 Graphical user interface (GUI), 127
 Graphics Processing Unit (GPU), 449
 acceleration, 449
 cache, 318
 memory, 454, 455, 470, 472
 bandwidth, 318
 pointers, 469
 programming, 449
 concepts, 461
 in Python, 472
 toolkit, 460
 runtime environment, 449, 450
 threads, 470

H

Hard disk drive (HDD), 314
 Hartree-Fock (HF), 559, 560
 derivatives basic equations, 677
 exchange elements, 75
 method, 675
 Hashability, 82
 HDF5, 370
 format, 366
 storage, 205
 Head-Gordon-Pople (HGP) algorithm, 533

Header files, 276, 283
 NumPy, 283
 High-Performance Computing (HPC), 436
 environments, 209, 246
 Higher-order
 derivatives, 707
 function, 586
 Horizontal RR (HRR), 492
 tensor, 506, 508, 509, 513, 514, 517, 519, 534
 HTML file, 359

I

I/O
 bound, 316
 buffer, 208
 efficiency, 368, 552
 operations, 311, 552
 readahead, 652
 IAM (Identity and Access Management) roles, 227
 Identity methods resolution, 598
 Illegal memory operations, 275
 Immutable data types, 376
 Imperative programming, 377
 Implicit differentiation, 703
 Importable Python
 extension, 359
 module, 352
 In-memory I/O, 208
 Incompatible data types, 62
 Independent Mixin classes, 596
 Indexing, 99
 code, 69, 582
 efficiency, 355
 efficiency NumPy array, 519
 elements, 102
 Indirect dependencies, 8
 Inefficient array accessing patterns, 510
 Initialization
 commands, 259
 overhead, 163, 424
 Inline buffer allocation, 294
 Inplace operations, 61, 62, 178, 179
 Inspecting dependencies, 5
 Instruction cache (I-cache), 535
 misses, 337
 performance, 353
 Instruction level parallelism (ILP), 324, 535
 Instructions per cycle (IPC), 317
 Integral
 code, 525
 contraction, 498
 derivatives, 679
 evaluation code, 559

- program, 477, 531, 534
- screening, 532
- tensor, 315, 598, 646, 672
- transformation, 552
- Integrated circuit (IC), 315
- Integrated development environment (IDE), 31
- Integration tests, 29
- Intel
 - compilers, 279
 - FFT library, 160
 - Fortran compilers, 297
- Inter-Process Communication (IPC), 265, 426
 - interfaces, 266
 - method, 419
- Intermediate Representation (IR), 348
- IPython, 49, 51, 52
 - cell, 50
 - extensions, 127
 - magics, 50, 127
 - shell, 51, 127
- Iterable objects, 94, 95, 653
- Iteration code, 349, 382, 502
- Iterator object, 197
- J**
- J-engine, 573
- Jacobian-vector product (JVP), 691
 - augmentation function, 698
 - function, 192
 - rules, 696
- JavaScript Object Notation (JSON), 196, 199
 - encoder, 200
 - module, 200
 - serialization, 196, 589
- JAX, 190, 456
 - AD, 703, 704, 710
 - approach, 705
 - capabilities, 700
 - code, 700, 703
 - customization, 691
 - functionality, 190
 - technique, 675
 - derivative customization, 192
 - differentiable
 - attributes, 700, 702
 - Python class, 700
 - forward direction, 696
 - forward-mode, 691
 - differentiation, 186
 - hybrid code, 710
 - JIT compilation, 456
 - library, 456
 - primal function, 696
- primals, 192
- PyTree nodes, 700
- reverse-mode, 691
 - differentiation, 186
- tangents, 192
- transformations, 190
- Jinja
 - code, 138
 - template, 136
 - templating techniques, 175
- Job
 - command, 258
 - management, 215
 - scheduling, 224
- Jupyter, 49
 - notebook, 49, 50, 52, 53, 124, 127, 327, 329, 330
 - notebook functionality, 50
- Just-in-Time (JIT)
 - compilation, 335, 338, 347–349, 352, 353, 362, 450, 464, 515, 518, 528, 656
 - JAX, 456
 - Python, 306
 - technique, 652
- compiler, 19, 322, 641
- technology, 672
- K**
- Keyword argument, 60, 78, 92, 116, 126, 131, 136, 147, 455, 457
- Kohn-Sham Density Functional Theory (KS-DFT), 559
- Krylov subspace linear equation solver, 687
- L**
- Label-based indexing, 100
- Last-In/First-Out (LIFO), 375
- Lazy evaluation, 186, 306, 378, 384–386, 388, 390, 409, 411, 416, 417, 666
- Least Frequently Used (LFU), 375
- Least Recently Used (LRU), 375
 - algorithm, 603
 - cache, 375, 376, 503, 557, 639
- Legacy code, 23
- Libraries
 - CuPy, 469
 - FFT, 159, 160, 162, 163, 165, 167–169
 - MPI, 436, 437
 - OpenMP, 436
 - optimized, 395
 - PyFFTW, 163
 - Python, 3, 12, 13, 32, 37, 226, 277
 - QtCore, 127

- SciPy, 88, 452, 545
 - searching order, 294
 - tensor, 398
 - Lifecycle management system, 292
 - Line
 - profiler, 330, 332
 - styles, 123
 - Linear
 - algebra, 145, 152
 - operator, 153
 - transformation, 699
 - Linearize-transpose algorithm, 709
 - List
 - comprehension, 174, 342, 409
 - container, 400
 - Python, 58, 59, 79, 309, 311
 - Python standard, 421
 - List of Lists (LIL) format, 148
 - Local Density Approximation (LDA) functional, 577
 - Local repository, 24
 - Location-based indexing, 99
 - Log file, 208
 - Loop
 - orders, 532
 - tiling, 319
 - unrolling, 516
 - Low-latency cache, 315
 - Low-Level Virtual Machine (LLVM) compiler, 348, 349, 353, 355
- M**
- Machine code, 347, 348, 355
 - optimized, 364
 - Magic command, 52, 327–330
 - Magics, 50
 - Management
 - capabilities, 256
 - class, 589
 - data, 215
 - design, 206
 - memory, 114, 363, 390, 434, 454, 457, 630, 631
 - system, 246
 - threading, 634
 - Managing
 - cache, 375
 - Dask workers, 256
 - Python projects, 3
 - Python virtual environments, 16
 - Mask array, 67–70, 91, 99
 - indexer, 103
 - NumPy, 103
 - Math Kernel Library (MKL), 160, 279
 - Matplotlib library, 119, 120, 127, 134
 - Mayavi library, 119, 127, 135
 - McMurchie-Davidson (MD)
 - algorithm, 488, 505
 - contraction code, 514
 - integral program, 575
 - Mean-field
 - calculations, 597
 - classes, 590
 - methods, 589
 - objects, 592
 - Memoization, 374, 533
 - Memory
 - access, 74, 307, 315, 317–320, 462, 464, 472, 473, 510, 514, 536
 - efficiency, 74, 322, 398, 477
 - optimized version, 631
 - overhead, 567
 - pattern, 75, 161, 319, 321, 514, 635
 - requests, 322
 - address, 70, 71, 273, 285, 296, 363, 378, 397, 398, 470, 472, 634
 - alignment, 160–162, 363
 - allocation, 208, 311, 329, 458, 630
 - allocation overhead, 61, 454, 455, 630
 - allocator, 311, 363, 633
 - allocator library, 311
 - architecture, 318
 - bandwidth, 74, 76, 312, 314, 317, 318
 - block, 275
 - bound, 317
 - buffer, 73, 290, 292–294, 312, 316, 323, 479, 552–555
 - cache, 648
 - coherence issues, 635
 - consumption, 61, 71, 76, 153, 341, 401, 639
 - contiguity requirements, 599
 - efficiency, 76, 163, 268, 615, 628, 632, 639
 - footprint, 147, 312, 438, 576, 587, 599, 628, 639
 - GPU, 454, 455, 470, 472
 - layout, 353
 - leaks, 292–294, 301, 593
 - management, 114, 363, 390, 434, 454, 457, 630, 631
 - efficiency, 410, 614, 639
 - overhead, 62, 528, 639
 - strategy, 146, 613
 - techniques, 363
 - operations, 370, 398
 - overhead, 204, 400, 599
 - overlaps, 349, 350
 - page, 74, 420, 470
 - pool, 454

- size, 70, 450
 - space, 61, 62, 70, 196, 265, 267, 311, 316, 396, 419, 425, 613, 637
 - storage, 70
 - temporary, 69
 - usage, 58, 147, 150, 159, 376, 400, 434, 609, 628
 - utilization, 413
 - Memory mapping (memmap), 207, 316, 368, 370
 - array, 204, 207, 370, 432, 433
 - files, 370, 434, 435
 - mode, 204
 - object, 369, 434
 - Memoryview, 220, 361, 369
 - Cython, 355
 - objects, 220
 - Merging dictionaries, 342
 - Message Passing Interface (MPI), 225, 281, 395, 427, 436
 - APIs, 437
 - compilers, 281
 - libraries, 436, 437
 - parallel computation, 440
 - parallel programming, 437
 - processes, 439
 - program, 437, 439–441, 445
 - Python program, 440
 - Meta-programming techniques, 520
 - Method Resolution Order (MRO), 43, 45, 595
 - Minimal overhead, 267, 553
 - Mixin, 47, 595, 597
 - Modular design, 42
 - Module
 - multiprocessing, 395, 419, 421, 423, 425, 427, 428, 445
 - NumPy, 58, 190, 516
 - profiler, 454
 - Python standard, 177
 - threading, 312, 395–398, 419
 - Molden
 - file, 137, 139
 - format, 137, 138
 - Molecular Design Limited (MOL), 135
 - Molecular Electrostatic Potential (MEP), 675
 - Molecular mechanics (MM) particles, 566
 - Molecular orbital (MO), 552, 684
 - Molecular properties, 675
 - Molecule structure file, 135
 - Multi-Configuration Self-Consistent Field (MCSCF) wavefunction, 671
 - Multiple
 - code versions, 364
 - eigenstates, 621
 - inheritance, 43, 46
 - Multiple Program Multiple Data (MPMD), 440
 - design pattern, 439
 - model, 440
 - paradigm, 446
 - Multiplication (MUL) units, 324
 - Multiprocessing, 419
 - module, 395, 419, 421, 423, 425, 427, 428, 445
 - parallel computation efficiency, 419
 - parallel program, 328
 - programming, 419
 - Multithreaded
 - function, 397
 - programs, 398
 - Multithreading, 396–398, 420, 421
 - efficiency, 446
 - parallel performance, 397
 - parallelism, 397, 420
 - Python, 554
 - tools, 398
 - Mutable objects, 376, 377
 - Mutex, 401
- N**
- Namespace packages, 11
 - Nested
 - dictionary, 94, 199
 - function calls, 378
 - multiprocessing issues, 638
 - threads, 633
 - Network
 - I/O, 209
 - requests, 210
 - Non-Hermitian matrix, 622
 - Non-python mode, 516, 518
 - Non-uniform memory access (NUMA), 324
 - Notification, 39
 - Nuclear gradients, 681, 683, 690
 - Nuclear Magnetic Resonance (NMR), 675, 684
 - Numba, 348
 - alternatives, 531
 - commands, 348
 - compilation, 519
 - compiled program, 530
 - compiler, 531, 547
 - CUDA APIs, 466
 - CUDA JIT, 464
 - documentation, 348
 - JIT, 322, 348, 353, 362, 464, 467, 570
 - compilation, 322, 349, 472, 477, 515, 557
 - compiler, 515, 526, 531
 - decorator, 348
 - performance code, 349

- optimization, 356
- package, 349
- threads, 633, 634
- vectorization issues, 353
- Numerical integration, 537, 549
- NumPy, 58
 - APIs, 453, 455, 456
 - array, 286
 - buffer, 294, 430, 457
 - indexing efficiency, 519
 - object, 79, 82, 349, 428, 438, 587
 - serialization, 200
 - basics, 58
 - boolean array, 103
 - broadcasting, 98, 156, 187, 349
 - core library, 71
 - counterparts, 455
 - documentation, 89
 - fancy indexing, 57, 99
 - FFT code, 165
 - functionalities, 86, 347, 361
 - functions, 78, 82, 190, 200, 312, 332, 349, 452, 454–456, 531, 691, 693
 - header files, 283
 - implementations in Pythran, 362
 - indexing methods, 57
 - learning documentation, 58
 - library, 70, 159, 190, 268, 290, 294, 299, 329, 345, 380, 452, 641
 - mask array, 103
 - module, 58, 190, 516
 - ndarray, 70
 - operations, 77, 80, 361
 - reduction functions, 81, 89, 453
 - scalar variables, 82
 - structured arrays, 91, 273, 274
 - ufunc, 60, 86, 93, 332, 349, 355, 361, 456, 515, 519
 - operations, 96
 - statements, 331
 - vector operations, 344
- O**
- Obara-Saika (OS) algorithm, 491, 506
- Object
 - buffer, 294, 437, 438
 - cached, 376
 - construction, 94
 - DIIS, 587, 589, 608
 - file, 197, 202, 207, 208
 - iteration, 340
 - memmap, 369, 434
 - NumPy array, 79, 82, 349, 428, 438, 587
 - picklable, 197
 - serialization, 441
- Object-Oriented Programming (OOP), 42, 43, 590
 - classes, 43, 592
 - paradigm, 695
 - programs, 46, 47
 - Python projects, 700
- Official
 - NumPy tutorial, 58
 - Pandas document, 108
 - Pandas documentation, 92, 105
- OpenMP, 435
 - APIs, 435
 - functionalities, 435
 - libraries, 436
 - shared libraries, 435
 - threads, 356, 435, 436, 633
- Operating systems (OS), 311
- Operation code (opcode), 332, 339
- Optimal performance, 164, 361, 392, 450, 631
- Optimization
 - hints, 361
 - procedure, 306
 - strategies, 306
- Optimized
 - iteration, 530
 - machine code, 364
- Optimizing
 - array strides, 356
 - cache coherence, 323
 - code implementation, 306
 - Cython code, 355, 519
 - data locality, 319
 - data transferring, 647
 - memory management, 632
 - tensor contractions, 625
 - tensor indexing efficiency, 346
- Ordinary differential equation (ODE), 184
- Outcore method, 552
- Overhead
 - calling NumPy, 361
 - data transfer, 159, 613, 637
 - in
 - data transfer, 458
 - function calls, 310
 - Python interpretation, 631
 - memory, 204, 400, 599
 - access, 567
 - management, 62, 528, 639
 - serialization, 196, 220
 - system calls, 402
 - Overlap integrals, 486

P

- Package
 - dependencies, 8
 - file, 32
 - management systems, 3, 19, 334
 - managers, 127
 - Numba, 349
 - PyFFTW, 162
 - PySCF, 530, 583, 584, 597, 646, 647, 679, 696
 - SymPy, 186, 193
- Page faults, 62, 312
- Pageable host memory, 457
- Pandas, 91
 - applications, 112
 - broadcasting, 98
 - data
 - objects, 92, 96, 99, 100, 111, 113, 119, 124
 - structures, 99
 - visualization, 126
 - documentation, 108
 - indexing, 94, 99
 - library, 91, 92
 - objects, 57, 84, 96, 98–100, 113, 220
 - plots, 126
 - split-apply-combine operations, 112
 - visualization, 124
- Parallel
 - computation, 169, 197, 208, 247, 316, 328, 398, 418, 440, 613, 633, 635
 - efficiency, 613
 - environments, 170
 - in Python, 395, 419
 - scenario, 403
 - technique, 247, 396, 446
 - efficiency, 168, 170, 395, 413, 633, 638
 - execution, 348
 - performance, 315, 635
 - program, 43, 311, 395, 406, 436
 - programming framework, 435
 - threads, 328
- Parallel Thread Execution (PTX) code, 450
- Parallelism, 170
- Parent class, 43, 44, 46, 47, 176, 590
- Partial differential equation (PDE), 184
- Path file, 8
- Peak
 - FLOP performance, 325
 - performance, 325
- Performance
 - analysis, 390, 614
 - benchmark, 161, 162, 165, 322, 529
 - bottlenecks, 327
 - CPU, 307, 314
 - CuPy, 472
 - data transfer, 457
 - FFT, 162, 169, 170
 - NumPy functions, 312
 - optimization, 305, 306, 310
 - overhead, 216
 - parallel, 315, 635
 - penalty, 60, 307, 337, 460
 - profiling, 632
 - quantum chemistry, 145
 - statistics, 329, 330
- Performance-friendly Python code, 340
- Periodic boundary condition (PBC), 567
- Permutation symmetry, 555, 559, 567, 572, 575, 628, 646, 666–669, 681, 685
- Pickleable object, 197, 198
- Pickle, 196–198, 426, 438
 - compatibility, 198
 - deserialization, 202
 - modules, 197, 199, 201
 - protocols, 198, 199
 - Python, 199
 - serialization, 197–199, 423, 428, 434, 435
- Pinned
 - buffer, 458
 - memory, 316, 449, 456–458
- Pipeline executor, 407–409, 416, 446, 614, 635, 638, 639
- Planner cache, 162
- Plotting functionalities, 120
- Point-to-point communication methods, 437
- Polynomial approximation, 532
- Post-Hartree-Fock (post-HF) methods, 613
- Potential energy surface (PES), 225
- Precomputation, 374
- Precomputing, 533
- Preconditioner, 621
- Prefetch, 603
- Prefetching data, 315
- Private
 - memory space, 420
 - PyPI server, 21
- Process management methods, 437
- Producer-consumer model, 406
- Profiler
 - line, 330, 332
 - module, 454
- Profiling, 326, 361
- Program
 - asynchronous, 416–418
 - bottlenecks, 392
 - CCD, 641, 643, 645, 647, 648, 652, 653, 692, 695, 696, 701, 703, 705

- Cython, 276
- DIIS, 585, 586
- ERI, 529
 - execution, 306, 330, 591, 628
 - failures, 62
 - flow, 416
 - implementation, 622
 - integral, 477, 531, 534
 - MPI, 437, 440, 441, 445
 - multiprocessing, 425
 - NumPy, 190
 - optimization, 306, 326, 378, 392
 - optimization steps, 557
 - parallel, 43, 311, 395, 406, 436
 - profiling, 326
 - serial, 395, 399, 440, 444–446
 - symbolic, 186, 656, 657, 666, 671, 672
 - tensor contraction, 610
- Project manager, 26
- Protein Data Bank (PDB), 135
- Pseudocode, 213
- Pull request (PR), 26, 39
- PyArrow buffers, 220
- PyFFTW, 160
 - functions, 164
 - interface, 163
 - libraries, 163
 - package, 161, 162
 - wrapper, 168
- PyPI release workflow, 42
- PySCF
 - integral program, 531
 - package, 530, 583, 584, 597, 646, 647, 679, 696
 - source code, 584
- Python
 - APIs, 436, 456
 - applications, 18, 54, 218, 449, 451
 - arithmetic computation code, 390
 - AST, 176
 - AST documentation, 176
 - bindings, 277, 301
 - buffer object, 470
 - built-in profiling tools, 327
 - bytecode, 312, 347
 - instructions, 402
 - interpretation, 392, 632
 - callback, 289
 - callback function, 274
 - classes, 176, 270, 310, 518, 589, 594, 703
 - classes design, 589
 - community, 31
 - compilation, 362, 364, 392, 557
 - techniques, 338, 344, 363, 390, 392, 525, 557
 - tools, 363
 - data types, 268, 419
 - debuggers, 593
 - decorators, 13
 - dictionary, 309, 375
 - document, 12
 - documentation, 402
 - ecosystem, 3, 213
 - environment, 8, 13, 17, 18, 23, 36
 - executable, 4, 15, 17
 - expressions, 71, 175
 - extensions, 265, 276–278, 280
 - features, 32, 289, 361, 384
 - function
 - call overhead, 557
 - callback, 274
 - GIL, 443, 637
 - GPU applications, 450, 452
 - import mechanism, 8
 - indexing, 101
 - instructions, 340
 - integral program, 477, 531, 537, 557
 - interface, 277, 298, 299, 460, 466, 468
 - interpreter, 9, 14, 15, 19, 23, 305, 309, 310, 332, 333, 395, 397, 398, 402, 436, 454
 - JIT, 306, 464, 466
 - list, 58, 59, 79, 309, 311
 - modules, 6, 34, 278, 300, 360, 361, 436, 444
 - MPI programs, 440
 - multithreading, 163, 554
 - object
 - overhead, 531
 - serialization, 201
 - types, 79
 - official documentation, 419
 - operations, 307, 390
 - package, 3, 5–8, 11, 15, 17, 18, 21, 32, 127, 265, 277, 282, 294, 296
 - configuration, 33
 - management systems, 53
 - system, 3, 4
 - pickle, 199
 - profilers, 361
 - profiling tools, 454
 - program
 - in Docker, 18
 - optimization, 390
 - testing, 29
 - project, 3, 8, 23, 33, 265, 267, 277, 278, 280, 468, 675
 - file structure, 32

- management, 3
- structure, 32
- QT libraries, 127
- quantum chemistry
 - program, 305
 - programs, 268
- raw code, 174
- runtime, 3, 4, 53, 175
 - environment, 53
 - paths, 4
- scalars, 80
- scientific applications, 452
- shell, 49
- source code, 36
- standard
 - library, 13, 201, 283, 405, 419
 - list, 421
 - module, 177
 - operators, 61
- strings, 269, 270
- syntax sugar, 355, 359
- versions, 4, 36, 198, 327, 436, 530, 537
- virtual machine, 340
- wrapper, 127
- Python Enhancement Proposal (PEP), 11
- Python Package Index (PyPI) server, 7
- Pythonic
 - fashion, 104
 - iterators, 516, 519
 - statements, 519
 - style, 219
- Pythran, 359
 - compilation, 364
 - directives, 359
 - extension, 360
- PyTorch, 187, 454
 - AD, 187
 - APIs, 454
 - counterparts, 454
 - documentation, 455
 - equivalents, 455
 - extensions, 277
 - functions, 455
 - tensor objects, 454, 469
 - tutorial, 277
 - version, 455
- Q**
- QT libraries, 127
- Quadrature roots, 541
- Quantum chemistry, 75, 145, 153, 176, 229, 396, 486, 552, 613, 617, 641, 683
 - analytical, 29, 175
 - basis sets, 480
 - data generation workloads, 247
 - FCI, 318
 - integral, 691
 - integral computation program, 76
 - literature, 486, 499
 - methods, 43, 366, 559
 - molecular systems, 478
 - programs, 29, 43, 145, 147, 195, 209, 215, 266, 279, 296, 305, 312, 313, 317, 421, 477, 478, 537, 707
 - simulations, 223
 - software, 479
 - software package, 641
 - textbooks, 500, 559, 684
 - visualization, 135
 - visualization software, 135
- Quantum mechanical (QM) calculations, 566
- Quaternion algebra, 186
- Queue object, 407
- R**
- Race conditions, 395, 397, 401, 402, 458, 459, 462, 635
- Randomly distributed mask array, 69
- Range-then-index approach, 356
- Raw
 - CUDA code, 467, 472
 - Python string, 124
- Ray, 256
 - libraries, 260
 - source code, 259
- Read-after-write synchronization, 403
- Read-before-write issue, 397
- Real-valued solid harmonics, 478
- Recurrence relation (RR), 486
 - iterations, 535
- Recursion program, 346
- Recursive
 - code, 382, 505
 - program, 557
- Regular quantum systems, 185
- Remote
 - execution, 53
 - repository, 24
- Remote Memory Access (RMA) model, 445
- Remote Procedure Call (RPC), 213, 427, 446
 - framework, 235
 - server, 213, 215, 216, 218, 219, 235–241
- Remote process call (RPC) service, 225
- Repository manager, 26
- Representational State Transfer (REST), 212
- Reproducible runtime environment, 18

- Requested memory location, 320
 - Resolution of identity (RI) method, 598
 - Response status code, 210
 - RESTful APIs, 241
 - Restore
 - method, 608
 - symmetry, 582
 - Restricted HF (RHF)
 - class, 565, 590
 - energy, 677, 681
 - Restricted Kohn-Sham (RKS)
 - class, 590
 - method, 565
 - Restricted open-shell HF (ROHF), 565
 - Reused memory space, 62
 - Rootless Docker installation, 19
 - Running shell commands, 311
 - Runtime, 6, 173, 175, 294, 449, 450, 593, 594, 652
 - configurations, 449, 460
 - CUDA, 450, 451
 - dependencies, 38, 40, 265, 303
 - Docker, 42
 - environment, 3, 16, 17, 37, 449, 452
 - paths, 4, 5, 281, 303
 - Rust
 - code, 302
 - interfaces, 301
 - libraries, 301, 302
 - Rys-quadrature algorithm, 496, 509
- S**
- Sampling profilers, 332, 333
 - SciPy
 - code, 452
 - documentation, 146
 - FFT, 158
 - functions, 165
 - modules, 161
 - functions, 691, 693, 711
 - libraries, 88, 452, 545
 - linear algebra module, 146
 - official documentation, 152
 - Second quantization, 185
 - Secure Sockets Layer (SSL) validation errors, 8
 - Security problem, 199
 - Segmentation Fault crash, 292
 - Self-Consistent Field (SCF)
 - base class, 43
 - calculations, 584
 - class, 561, 590
 - iteration, 561
 - program, 561, 562, 589, 607–609
 - Self-hosted runners, 40
 - Semaphore, 402
 - Sequential program, 444
 - Serial
 - code, 633
 - program, 395, 399, 440, 444–446
 - Serialization, 196
 - binary, 196
 - compatibility, 199
 - data, 196, 221
 - DIIS, 605
 - efficiency, 198
 - failure, 196
 - formats, 221
 - functions, 199
 - JSON, 196, 589
 - limitations, 196
 - method, 199
 - NumPy arrays, 200
 - object, 441
 - overhead, 196, 220
 - pickle, 197–199, 423, 428, 434, 435
 - process, 198, 434
 - schemes, 196
 - Shared memory (shm), 431, 445, 463
 - Shebang, 15
 - Shell commands, 311
 - Side-effects free, 692
 - Signature file, 299, 300
 - Simple Storage Service (S3), 227
 - Single Instruction Multiple Data (SIMD), 324
 - instructions, 324
 - pattern, 310
 - vectorization, 163, 349, 350, 353, 355, 360, 362, 364, 392, 395, 396, 517
 - Single Program Multiple Data (SPMD)
 - design pattern, 439
 - framework, 446
 - MPI programs, 439
 - Single-particle operators, 675
 - Singular value decomposition (SVD), 152
 - Singularity, 621
 - Software
 - stack, 224
 - version dependency, 199
 - Solid-state drive (SSD), 314
 - Sparse
 - arrays, 371
 - matrix, 147, 152
 - tensor, 639
 - Sparsity, 533
 - Standard
 - library directories, 5
 - list object, 421

- Python
 library, 209, 419
 list, 79
 objects, 287
 operators, 61
 Startup configurations, 19, 49
 Static
 checks, 31
 code analysis, 6
 code analyzers, 6, 31
 Storage
 formats, 148, 601
 layout, 366
 Strides, 319, 320, 346, 355, 364
 String
 code, 174
 list, 93
 Python, 269, 270
 representation, 615
 Structure Data File (SDF), 135
 Structured array, 90
 Subclass, 43, 188–190, 595, 596, 657
 Subclassing syntax, 594
 Submodules, 29
 Suboptimal performance, 464
 Superclass, 46
 Superior performance, 70, 165
 Superposition of atomic potentials (SAP) model,
 607
 Swapped strings, 663
 Symbolic
 computation, 181
 program, 186, 656, 657, 666, 671, 672
 programming, 656, 672
 programming approach, 641
 SymPy, 181
 documentation, 183
 online documentation, 185
 package, 186, 193
 programs, 182
 Synchronization primitives, 425
 Synchronizing threads, 402
 Synchronous program, 416, 418
- T**
- Task definition, 230
 Template
 files, 136
 programming, 173
 Temporary
 files, 208, 266, 267
 memory, 69
- Tensor
 coefficients, 663
 contraction, 145, 154, 155, 488, 512, 572, 573,
 578, 623, 624, 645, 647, 648, 676, 685
 algorithm, 619
 code, 155, 491, 604
 computational costs, 639
 operations, 154, 155
 scheme, 580
 tools, 641
 ERI, 396, 399, 457, 464, 471, 472, 491, 495,
 528, 552, 567, 570, 572, 598, 600, 699
 indexing, 347
 indices, 514, 662, 665, 666
 integral, 315, 598, 646, 672
 libraries, 398
 operations, 344, 392, 453, 609, 639
 transpose, 553
 TensorFlow, 691
 Testing
 automation, 30
 tools, 29
 Text
 formatting, 124
 strings, 267
 Thread
 creation, 400
 creation overhead, 399
 CUDA, 461–463
 executor, 414
 GPU, 470
 nested, 633
 Numba, 633, 634
 OpenMP, 356, 435, 436, 633
 parallelization efficiency, 637
 resources, 399
 switching, 402
 synchronization, 401, 425, 635
 worker, 637
 Threading
 controllers, 634
 efficiency, 397
 level parallelism, 396
 library, 376
 locks, 425
 management, 634
 module, 312, 395–398, 419, 446
 parallelization, 395, 419, 425, 634, 639
 resources, 398
 ThreadPoolExecutor, 398
 Three-center two-electron (3c2e) repulsion
 integrals, 598
 Throughput-oriented approach, 464

Translation Lookaside Buffer (TLB) cache, 314
 TRR tensor, 507, 513, 534
 Two-center two-electron (2c2e) integral tensor, 598
 Type
 casting, 79
 conversion, 268
 declaration, 355
 upcasting, 79
 Typed memoryview, 355

U

Uncommitted files, 24
 Unicode, 269
 Uniform Resource Identifier (URI), 210, 212
 Unit tests, 29, 501
 Universal function (ufunc), 60, 93
 Unmapped virtual memory, 630
 Unpicklable
 instances, 198
 objects, 198
 Unpickle operations, 438
 Unreachable memory, 292
 Unrestricted HF (UHF), 565
 Unrolled code, 350, 351, 520
 Unrolling, 534
 Unused memory blocks, 456
 User interface (UI), 227

V

Vector-Jacobian product (VJP), 691
 augmentation functions, 708
 operation, 706

rules, 696
 Vectorized
 code, 309, 310
 data processing, 58
 operations, 309
 Vertical RR (VRR), 492
 tensor, 506
 Virtual
 environments, 16
 memory, 434
 method, 310
 Virtual machine (VM), 224, 227
 Virtual Private Cloud (VPC), 227
 Virtualization technologies, 19
 Visitor pattern, 590
 Visualization Toolkit (VTK), 127
 Vscode, 593

W

Waiting threads, 402, 403, 405
 Wavefunction object, 564, 565
 Wheel, 35, 36
 Whitespace control, 137
 Wigner coefficients, 185
 Wrapped Python function, 526

Y

YAML, 201
 decoder, 202
 document, 201, 202
 format, 22, 201
 library, 201

Python for Quantum Chemistry

A Full Stack Programming Guide

Provides a better understanding of the power of Python, advanced Python programming skills, modern technologies related to Python program, and Python high-performance programming technologies for use in quantum chemistry and quantum computing

Key Features

- Covers in depth the crucial topic of Python code optimization methods with high-performance computing technologies and their practice in quantum chemistry
- Provides examples of Python applications with cutting-edge technologies such as automatic code generation, cloud computing, and GPGPU
- Includes discussion of Python runtime mechanism and advanced Python technologies

Python for Quantum Chemistry provides a comprehensive, end-to-end practical resource for researchers and engineers who have basic Python programming experience (chiefly in computational chemistry) but want to take their use of the software forwards to the next level. The book provides an insightful exploration of data analysis tools, input/output utilities, parallel computation, hybrid language programming, and so forth. Readers will learn various tools and protocols for computational chemistry research exhibited and analyzed from a technical perspective. Multiple programming paradigms including object-oriented, functional, meta-programming, dynamic, concurrent, and vector-oriented are illustrated in various technology scenarios. Readers will learn how to leverage these paradigms to enhance their program projects. The book introduces various optimization technologies to speed up Python applications, even to the level as fast as a native C++ implementation. These optimization strategies are then demonstrated using quantum chemistry Python applications.

Python for Quantum Chemistry is written primarily for graduate students, researchers, and software engineers working primarily in the fields of theoretical chemistry, computational chemistry, condensed matter physics, materials modeling, molecular simulations, and quantum computing.

About the Author

Qiming Sun is primarily known for his contributions to the field of quantum chemistry, particularly through his work in developing the Python package PySCF. He has experience across multiple technical fields such as machine learning, quantum computing algorithms, and cloud computing. Around 2014, he began promoting the use of Python in quantum chemistry research and led the development of the widely used Python quantum chemistry package PySCF, which is now utilized by thousands of universities and companies in quantum chemistry, quantum computing, and AI-chemistry research.



ELSEVIER

elsevier.com/books-and-journals

ISBN 978-0-443-23837-6

9 780443 238376