Morten Hjorth-Jensen

Quantum mechanics for many-particle systems

From standard methods to quantum computing and machine learning

December 28, 2023

Springer

Use the template dedic.tex *together with the Springer document class SVMono for monograph-type books or SVMult for contributed volumes to style a quotation or a dedication at the very beginning of your book in the Springer layout*

Preface

So, ultimately, in order to understand nature it may be necessary to have a deeper understanding of mathematical relationships. But the real reason is that the subject is enjoyable, and although we humans cut nature up in different ways, and we have different courses in different departments, such compartmentalization is really artificial, and we should take our intellectual pleasures where we find them. *Richard Feynman, The Laws of Thermodynamics.*

Why a preface you may ask? Isn't that just a mere exposition of a raison d'être of an author's choice of material, preferences, biases, teaching philosophy etc.? To a large extent I can answer in the affirmative to that. A preface ought to be personal. Indeed, what you will see in the various chapters of these notes represents how I perceive computational physics should be taught.

This set of lecture notes serves the scope of presenting to you and train you in an algorithmic approach to problems in the sciences, represented here by the unity of three disciplines, physics, mathematics and informatics. This trinity outlines the emerging field of computational physics.

Our insight in a physical system, combined with numerical mathematics gives us the rules for setting up an algorithm, viz. a set of rules for solving a particular problem. Our understanding of the physical system under study is obviously gauged by the natural laws at play, the initial conditions, boundary conditions and other external constraints which influence the given system. Having spelled out the physics, for example in the form of a set of coupled partial differential equations, we need efficient numerical methods in order to set up the final algorithm. This algorithm is in turn coded into a computer program and executed on available computing facilities. To develop such an algorithmic approach, you will be exposed to several physics cases, spanning from the classical pendulum to quantum mechanical systems. We will also present some of the most popular algorithms from numerical mathematics used to solve a plethora of problems in the sciences. Finally we will codify these algorithms using some of the most widely used programming languages, presently C, C++ and Fortran and its most recent standard Fortran 2008[1]. However, a high-level and fully object-oriented language like Python is now emerging as a good alternative although C++ and Fortran still outperform Python when it comes to computational speed. In this text we offer an approach where one can write all programs in C/C++ or Fortran. We will also show you how to develop large programs in Python interfacing C++ and/or Fortran functions for those parts of the program which are CPU intensive. Such an approach allows you to structure the flow of data in a high-level language like Python while tasks of a mere repetitive and CPU intensive nature are left to low-level languages like C++ or Fortran. Python allows you also to smoothly interface your program with other software, such as plotting programs or operating system instructions.

[1] Throughout this text we refer to Fortran 2008 as Fortran, implying the latest standard.

A typical Python program you may end up writing contains everything from compiling and running your codes to preparing the body of a file for writing up your report.

Computer simulations are nowadays an integral part of contemporary basic and applied research in the sciences. Computation is becoming as important as theory and experiment. In physics, computational physics, theoretical physics and experimental physics are all equally important in our daily research and studies of physical systems. Physics is the unity of theory, experiment and computation[2]. Moreover, the ability "to compute" forms part of the essential repertoire of research scientists. Several new fields within computational science have emerged and strengthened their positions in the last years, such as computational materials science, bioinformatics, computational mathematics and mechanics, computational chemistry and physics and so forth, just to mention a few. These fields underscore the importance of simulations as a means to gain novel insights into physical systems, especially for those cases where no analytical solutions can be found or an experiment is too complicated or expensive to carry out. To be able to simulate large quantal systems with many degrees of freedom such as strongly interacting electrons in a quantum dot will be of great importance for future directions in novel fields like nano-techonology. This ability often combines knowledge from many different subjects, in our case essentially from the physical sciences, numerical mathematics, computing languages, topics from high-performace computing and some knowledge of computers.

In 1999, when I started this course at the department of physics in Oslo, computational physics and computational science in general were still perceived by the majority of physicists and scientists as topics dealing with just mere tools and number crunching, and not as subjects of their own. The computational background of most students enlisting for the course on computational physics could span from dedicated hackers and computer freaks to people who basically had never used a PC. The majority of undergraduate and graduate students had a very rudimentary knowledge of computational techniques and methods. Questions like 'do you know of better methods for numerical integration than the trapezoidal rule' were not uncommon. I do happen to know of colleagues who applied for time at a supercomputing centre because they needed to invert matrices of the size of $10^4 \times 10^4$ since they were using the trapezoidal rule to compute integrals. With Gaussian quadrature this dimensionality was easily reduced to matrix problems of the size of $10^2 \times 10^2$, with much better precision.

More than a decade later most students have now been exposed to a fairly uniform introduction to computers, basic programming skills and use of numerical exercises. Practically every undergraduate student in physics has now made a Matlab or Maple simulation of for example the pendulum, with or without chaotic motion. Nowadays most of you are familiar, through various undergraduate courses in physics and mathematics, with interpreted languages such as Maple, Matlab and/or Mathematica. In addition, the interest in scripting languages such as Python or Perl has increased considerably in recent years. The modern programmer would typically combine several tools, computing environments and programming languages. A typical example is the following. Suppose you are working on a project which demands extensive visualizations of the results. To obtain these results, that is to solve a physics problems like obtaining the density profile of a Bose-Einstein condensate, you need however a program which is fairly fast when computational speed matters. In this case you would most

[2] We mentioned previously the trinity of physics, mathematics and informatics. Viewing physics as the trinity of theory, experiment and simulations is yet another example. It is obviously tempting to go beyond the sciences. History shows that triunes, trinities and for example triple deities permeate the Indo-European cultures (and probably all human cultures), from the ancient Celts and Hindus to modern days. The ancient Celts revered many such trinues, their world was divided into earth, sea and air, nature was divided in animal, vegetable and mineral and the cardinal colours were red, yellow and blue, just to mention a few. As a curious digression, it was a Gaulish Celt, Hilary, philosopher and bishop of Poitiers (AD 315-367) in his work *De Trinitate* who formulated the Holy Trinity concept of Christianity, perhaps in order to accomodate millenia of human divination practice.

likely write a high-performance computing program using Monte Carlo methods in languages which are tailored for that. These are represented by programming languages like Fortran and C++. However, to visualize the results you would find interpreted languages like Matlab or scripting languages like Python extremely suitable for your tasks. You will therefore end up writing for example a script in Matlab which calls a Fortran or C++ program where the number crunching is done and then visualize the results of say a wave equation solver via Matlab's large library of visualization tools. Alternatively, you could organize everything into a Python or Perl script which does everything for you, calls the Fortran and/or C++ programs and performs the visualization in Matlab or Python. Used correctly, these tools, spanning from scripting languages to high-performance computing languages and vizualization programs, speed up your capability to solve complicated problems. Being multilingual is thus an advantage which not only applies to our globalized modern society but to computing environments as well. This text shows you how to use C++ and Fortran as programming languages.

There is however more to the picture than meets the eye. Although interpreted languages like Matlab, Mathematica and Maple allow you nowadays to solve very complicated problems, and high-level languages like Python can be used to solve computational problems, computational speed and the capability to write an efficient code are topics which still do matter. To this end, the majority of scientists still use languages like C++ and Fortran to solve scientific problems. When you embark on a master or PhD thesis, you will most likely meet these high-performance computing languages. This course emphasizes thus the use of programming languages like Fortran, Python and C++ instead of interpreted ones like Matlab or Maple. You should however note that there are still large differences in computer time between for example numerical Python and a corresponding C++ program for many numerical applications in the physical sciences, with a code in C++ or Fortran being the fastest.

Computational speed is not the only reason for this choice of programming languages. Another important reason is that we feel that at a certain stage one needs to have some insights into the algorithm used, its stability conditions, possible pitfalls like loss of precision, ranges of applicability, the possibility to improve the algorithm and taylor it to special purposes etc etc. One of our major aims here is to present to you what we would dub 'the algorithmic approach', a set of rules for doing mathematics or a precise description of how to solve a problem. To device an algorithm and thereafter write a code for solving physics problems is a marvelous way of gaining insight into complicated physical systems. The algorithm you end up writing reflects in essentially all cases your own understanding of the physics and the mathematics (the way you express yourself) of the problem. We do therefore devote quite some space to the algorithms behind various functions presented in the text. Especially, insight into how errors propagate and how to avoid them is a topic we would like you to pay special attention to. Only then can you avoid problems like underflow, overflow and loss of precision. Such a control is not always achievable with interpreted languages and canned functions where the underlying algorithm and/or code is not easily accesible. Although we will at various stages recommend the use of library routines for say linear algebra[3], our belief is that one should understand what the given function does, at least to have a mere idea. With such a starting point, we strongly believe that it can be easier to develope more complicated programs on your own using Fortran, C++ or Python.

We have several other aims as well, namely:

- We would like to give you an opportunity to gain a deeper understanding of the physics you have learned in other courses. In most courses one is normally confronted with simple systems which provide exact solutions and mimic to a certain extent the realistic cases. Many are however the comments like 'why can't we do something else than the particle in

[3] Such library functions are often taylored to a given machine's architecture and should accordingly run faster than user provided ones.

a box potential?’. In several of the projects we hope to present some more ’realistic’ cases to solve by various numerical methods. This also means that we wish to give examples of how physics can be applied in a much broader context than it is discussed in the traditional physics undergraduate curriculum.
- To encourage you to "discover" physics in a way similar to how researchers learn in the context of research.
- Hopefully also to introduce numerical methods and new areas of physics that can be studied with the methods discussed.
- To teach structured programming in the context of doing science.
- The projects we propose are meant to mimic to a certain extent the situation encountered during a thesis or project work. You will tipically have at your disposal 2-3 weeks to solve numerically a given project. In so doing you may need to do a literature study as well. Finally, we would like you to write a report for every project.

Our overall goal is to encourage you to learn about science through experience and by asking questions. Our objective is always understanding and the purpose of computing is further insight, not mere numbers! Simulations can often be considered as experiments. Rerunning a simulation need not be as costly as rerunning an experiment.

Needless to say, these lecture notes are upgraded continuously, from typos to new input. And we do always benefit from your comments, suggestions and ideas for making these notes better. It’s through the scientific discourse and critics we advance. Moreover, I have benefitted immensely from many discussions with fellow colleagues and students. In particular I must mention Hans Petter Langtangen, Anders Malthe-Sørenssen, Knut Mørken and Øyvind Ryan, whose input during the last fifteen years has considerably improved these lecture notes. Furthermore, the time we have spent and keep spending together on the Computing in Science Education project at the University, is just marvelous. Thanks so much. Concerning the Computing in Science Education initiative, you can read more at `http://www.mn.uio.no/english/about/collaboration/cse/`.

Finally, I would like to add a petit note on referencing. These notes have evolved over many years and the idea is that they should end up in the format of a web-based learning environment for doing computational science. It will be fully free and hopefully represent a much more efficient way of conveying teaching material than traditional textbooks. I have not yet settled on a specific format, so any input is welcome. At present however, it is very easy for me to upgrade and improve the material on say a yearly basis, from simple typos to adding new material. When accessing the web page of the course, you will have noticed that you can obtain all source files for the programs discussed in the text. Many people have thus written to me about how they should properly reference this material and whether they can freely use it. My answer is rather simple. You are encouraged to use these codes, modify them, include them in publications, thesis work, your lectures etc. As long as your use is part of the dialectics of science you can use this material freely. However, since many weekends have elapsed in writing several of these programs, testing them, sweating over bugs, swearing in front of a f*@?%g code which didn’t compile properly ten minutes before monday morning’s eight o’clock lecture etc etc, I would dearly appreciate in case you find these codes of any use, to reference them properly. That can be done in a simple way, refer to M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2013). The weblink to the course should also be included. Hope it is not too much to ask for. Enjoy!

Acknowledgements

Use the template *acknow.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) if you prefer to set your acknowledgement section as a separate chapter instead of including it as last part of your preface.

Contents

Acronyms

Use the template *acronym.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your list(s) of abbreviations or symbols in the Springer layout.

Lists of abbreviations, symbols and the like are easily formatted with the help of the Springer-enhanced `description` environment.

ABC	Spelled-out abbreviation and definition
BABI	Spelled-out abbreviation and definition
CABR	Spelled-out abbreviation and definition

Part I
Linear algebra and second quantization

Chapter 1
Many-body Hamiltonians, basic linear algebra and Second Quantization

Definitions and notations

Before we proceed we need some definitions. We will assume that the interacting part of the Hamiltonian can be approximated by a two-body interaction. This means that our Hamiltonian is written as the sum of some onebody part and a twobody part

$$\hat{H} = \hat{H}_0 + \hat{H}_I = \sum_{i=1}^{A} \hat{h}_0(x_i) + \sum_{i<j}^{A} \hat{v}(r_{ij}), \tag{1.1}$$

with

$$H_0 = \sum_{i=1}^{A} \hat{h}_0(x_i). \tag{1.2}$$

The onebody part $u_{\text{ext}}(x_i)$ is normally approximated by a harmonic oscillator potential or the Coulomb interaction an electron feels from the nucleus. However, other potentials are fully possible, such as one derived from the self-consistent solution of the Hartree-Fock equations to be discussed here.

Our Hamiltonian is invariant under the permutation (interchange) of two particles. Since we deal with fermions however, the total wave function is antisymmetric. Let $\hat{P}$ be an operator which interchanges two particles. Due to the symmetries we have ascribed to our Hamiltonian, this operator commutes with the total Hamiltonian,

$$[\hat{H}, \hat{P}] = 0,$$

meaning that $\Psi_\lambda(x_1, x_2, \dots, x_A)$ is an eigenfunction of $\hat{P}$ as well, that is

$$\hat{P}_{ij}\Psi_\lambda(x_1, x_2, \dots, x_i, \dots, x_j, \dots, x_A) = \beta\Psi_\lambda(x_1, x_2, \dots, x_i, \dots, x_j, \dots, x_A),$$

where β is the eigenvalue of $\hat{P}$. We have introduced the suffix ij in order to indicate that we permute particles i and j. The Pauli principle tells us that the total wave function for a system of fermions has to be antisymmetric, resulting in the eigenvalue $\beta = -1$.

In our case we assume that we can approximate the exact eigenfunction with a Slater determinant

$$\Phi(x_1, x_2, \dots, x_A, \alpha, \beta, \dots, \sigma) = \frac{1}{\sqrt{A!}} \left| \begin{array}{ccccc} \psi_\alpha(x_1) & \psi_\alpha(x_2) & \dots & \dots & \psi_\alpha(x_A) \\ \psi_\beta(x_1) & \psi_\beta(x_2) & \dots & \dots & \psi_\beta(x_A) \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \psi_\sigma(x_1) & \psi_\sigma(x_2) & \dots & \dots & \psi_\sigma(x_A) \end{array} \right|, \tag{1.3}$$

where x_i stand for the coordinates and spin values of a particle i and $\alpha,\beta,\dots,\gamma$ are quantum numbers needed to describe remaining quantum numbers.

Brief reminder on some linear algebra properties.

Before we proceed with a more compact representation of a Slater determinant, we would like to repeat some linear algebra properties which will be useful for our derivations of the energy as function of a Slater determinant, Hartree-Fock theory and later the nuclear shell model.

The inverse of a matrix is defined by

$$\mathbf{A}^{-1}\cdot\mathbf{A} = I$$

A unitary matrix $\mathbf{A}$ is one whose inverse is its adjoint

$$\mathbf{A}^{-1} = \mathbf{A}^{\dagger}$$

A real unitary matrix is called orthogonal and its inverse is equal to its transpose. A hermitian matrix is its own self-adjoint, that is

$$\mathbf{A} = \mathbf{A}^{\dagger}.$$

Relations	Name	matrix elements
$A = A^T$	symmetric	$a_{ij} = a_{ji}$
$A = \left(A^T\right)^{-1}$	real orthogonal	$\sum_k a_{ik}a_{jk} = \sum_k a_{ki}a_{kj} = \delta_{ij}$
$A = A^*$	real matrix	$a_{ij} = a_{ij}^*$
$A = A^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$A = \left(A^\dagger\right)^{-1}$	unitary	$\sum_k a_{ik}a_{jk}^* = \sum_k a_{ki}^*a_{kj} = \delta_{ij}$

Since we will deal with Fermions (identical and indistinguishable particles) we will form an ansatz for a given state in terms of so-called Slater determinants determined by a chosen basis of single-particle functions.

For a given $n\times n$ matrix $\mathbf{A}$ we can write its determinant

$$det(\mathbf{A}) = |\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{vmatrix},$$

in a more compact form as

$$|\mathbf{A}| = \sum_{i=1}^{n!}(-1)^{p_i}\hat{P}_i a_{11}a_{22}\dots a_{nn},$$

where $\hat{P}_i$ is a permutation operator which permutes the column indices $1,2,3,\dots,n$ and the sum runs over all $n!$ permutations. The quantity p_i represents the number of transpositions of column indices that are needed in order to bring a given permutation back to its initial ordering, in our case given by $a_{11}a_{22}\dots a_{nn}$ here.

A simple 2×2 determinant illustrates this. We have

$$det(\mathbf{A}) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = (-1)^0 a_{11}a_{22} + (-1)^1 a_{12}a_{21},$$

where in the last term we have interchanged the column indices 1 and 2. The natural ordering we have chosen is $a_{11}a_{22}$.

Back to the derivation of the energy.

The single-particle function $\psi_\alpha(x_i)$ are eigenfunctions of the onebody Hamiltonian h_i, that is

$$\hat{h}_0(x_i) = \hat{t}(x_i) + \hat{u}_{\mathrm{ext}}(x_i),$$

with eigenvalues

$$\hat{h}_0(x_i)\psi_\alpha(x_i) = (\hat{t}(x_i) + \hat{u}_{\mathrm{ext}}(x_i))\,\psi_\alpha(x_i) = \varepsilon_\alpha \psi_\alpha(x_i).$$

The energies ε_α are the so-called non-interacting single-particle energies, or unperturbed energies. The total energy is in this case the sum over all single-particle energies, if no two-body or more complicated many-body interactions are present.

Let us denote the ground state energy by E_0. According to the variational principle we have

$$E_0 \le E[\Phi] = \int \Phi^* \hat{H} \Phi d\tau$$

where Φ is a trial function which we assume to be normalized

$$\int \Phi^* \Phi d\tau = 1,$$

where we have used the shorthand $d\tau = dx_1 dr_2 \ldots dr_A$.

In the Hartree-Fock method the trial function is the Slater determinant of Eq. (1.3) which can be rewritten as

$$\Phi(x_1, x_2, \ldots, x_A, \alpha, \beta, \ldots, \nu) = \frac{1}{\sqrt{A!}} \sum_P (-)^P \hat{P} \psi_\alpha(x_1) \psi_\beta(x_2) \ldots \psi_\nu(x_A) = \sqrt{A!}\hat{A}\Phi_H,$$

where we have introduced the antisymmetrization operator $\hat{A}$ defined by the summation over all possible permutations of two particles.

It is defined as

$$\hat{A} = \frac{1}{A!} \sum_p (-)^p \hat{P}, \tag{1.4}$$

with p standing for the number of permutations. We have introduced for later use the so-called Hartree-function, defined by the simple product of all possible single-particle functions

$$\Phi_H(x_1, x_2, \ldots, x_A, \alpha, \beta, \ldots, \nu) = \psi_\alpha(x_1) \psi_\beta(x_2) \ldots \psi_\nu(x_A).$$

Both $\hat{H}_0$ and $\hat{H}_I$ are invariant under all possible permutations of any two particles and hence commute with $\hat{A}$

$$[H_0, \hat{A}] = [H_I, \hat{A}] = 0. \tag{1.5}$$

Furthermore, $\hat{A}$ satisfies

$$\hat{A}^2 = \hat{A}, \tag{1.6}$$

since every permutation of the Slater determinant reproduces it.

The expectation value of $\hat{H}_0$

$$\int \Phi^* \hat{H}_0 \Phi d\tau = A! \int \Phi_H^* \hat{A} \hat{H}_0 \hat{A} \Phi_H d\tau$$

is readily reduced to

$$\int \Phi^* \hat{H}_0 \Phi d\tau = A! \int \Phi_H^* \hat{H}_0 \hat{A} \Phi_H d\tau,$$

where we have used Eqs. (1.5) and (1.6). The next step is to replace the antisymmetrization operator by its definition and to replace $\hat{H}_0$ with the sum of one-body operators

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{i=1}^{A} \sum_{p} (-)^p \int \Phi_H^* \hat{h}_0 \hat{P} \Phi_H d\tau.$$

The integral vanishes if two or more particles are permuted in only one of the Hartree-functions Φ_H because the individual single-particle wave functions are orthogonal. We obtain then

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{i=1}^{A} \int \Phi_H^* \hat{h}_0 \Phi_H d\tau.$$

Orthogonality of the single-particle functions allows us to further simplify the integral, and we arrive at the following expression for the expectation values of the sum of one-body Hamiltonians

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{\mu=1}^{A} \int \psi_\mu^*(x) \hat{h}_0 \psi_\mu(x) dx d\mathbf{r}. \tag{1.7}$$

We introduce the following shorthand for the above integral

$$\langle \mu | \hat{h}_0 | \mu \rangle = \int \psi_\mu^*(x) \hat{h}_0 \psi_\mu(x) dx,$$

and rewrite Eq. (1.7) as

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{\mu=1}^{A} \langle \mu | \hat{h}_0 | \mu \rangle. \tag{1.8}$$

The expectation value of the two-body part of the Hamiltonian is obtained in a similar manner. We have

$$\int \Phi^* \hat{H}_I \Phi d\tau = A! \int \Phi_H^* \hat{A} \hat{H}_I \hat{A} \Phi_H d\tau,$$

which reduces to

$$\int \Phi^* \hat{H}_I \Phi d\tau = \sum_{i \le j=1}^{A} \sum_{p} (-)^p \int \Phi_H^* \hat{v}(r_{ij}) \hat{P} \Phi_H d\tau,$$

by following the same arguments as for the one-body Hamiltonian.

Because of the dependence on the inter-particle distance r_{ij}, permutations of any two particles no longer vanish, and we get

$$\int \Phi^* \hat{H}_I \Phi d\tau = \sum_{i < j=1}^{A} \int \Phi_H^* \hat{v}(r_{ij}) (1 - P_{ij}) \Phi_H d\tau.$$

where P_{ij} is the permutation operator that interchanges particle i and particle j. Again we use the assumption that the single-particle wave functions are orthogonal.

We obtain

$$\int \Phi^* \hat{H}_I \Phi d\tau = \frac{1}{2} \sum_{\mu=1}^{A} \sum_{\nu=1}^{A} \left[\int \psi_\mu^*(x_i) \psi_\nu^*(x_j) \hat{v}(r_{ij}) \psi_\mu(x_i) \psi_\nu(x_j) dx_i dx_j \right. \tag{1.9}$$

$$\left. - \int \psi_\mu^*(x_i) \psi_\nu^*(x_j) \hat{v}(r_{ij}) \psi_\nu(x_i) \psi_\mu(x_j) dx_i dx_j \right]. \tag{1.10}$$

The first term is the so-called direct term. It is frequently also called the Hartree term, while the second is due to the Pauli principle and is called the exchange term or just the Fock term. The factor $1/2$ is introduced because we now run over all pairs twice.

The last equation allows us to introduce some further definitions. The single-particle wave functions $\psi_\mu(x)$, defined by the quantum numbers μ and x are defined as the overlap

$$\psi_\alpha(x) = \langle x | \alpha \rangle.$$

We introduce the following shorthands for the above two integrals

$$\langle \mu\nu|\hat{v}|\mu\nu\rangle = \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)\hat{v}(r_{ij})\psi_\mu(x_i)\psi_\nu(x_j)dx_idx_j,$$

and

$$\langle \mu\nu|\hat{v}|\nu\mu\rangle = \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)\hat{v}(r_{ij})\psi_\nu(x_i)\psi_\mu(x_j)dx_idx_j.$$

Preparing for later studies: varying the coefficients of a wave function expansion and orthogonal transformations

It is common to expand the single-particle functions in a known basis and vary the coefficients, that is, the new single-particle wave function is written as a linear expansion in terms of a fixed chosen orthogonal basis (for example the well-known harmonic oscillator functions or the hydrogen-like functions etc). We define our new single-particle basis (this is a normal approach for Hartree-Fock theory) by performing a unitary transformation on our previous basis (labelled with greek indices) as

$$\psi_p^{new} = \sum_\lambda C_{p\lambda}\phi_\lambda. \tag{1.11}$$

In this case we vary the coefficients $C_{p\lambda}$. If the basis has infinitely many solutions, we need to truncate the above sum. We assume that the basis ϕ_λ is orthogonal.

It is normal to choose a single-particle basis defined as the eigenfunctions of parts of the full Hamiltonian. The typical situation consists of the solutions of the one-body part of the Hamiltonian, that is we have

$$\hat{h}_0\phi_\lambda = \varepsilon_\lambda\phi_\lambda.$$

The single-particle wave functions $\phi_\lambda(\mathbf{r})$, defined by the quantum numbers λ and $\mathbf{r}$ are defined as the overlap

$$\phi_\lambda(\mathbf{r}) = \langle \mathbf{r}|\lambda\rangle.$$

In deriving the Hartree-Fock equations, we will expand the single-particle functions in a known basis and vary the coefficients, that is, the new single-particle wave function is written as a linear expansion in terms of a fixed chosen orthogonal basis (for example the well-known harmonic oscillator functions or the hydrogen-like functions etc).

We stated that a unitary transformation keeps the orthogonality. To see this consider first a basis of vectors $\mathbf{v}_i$,

$$\mathbf{v}_i = \begin{bmatrix} v_{i1} \\ \dots \\ \dots \\ v_{in} \end{bmatrix}$$

We assume that the basis is orthogonal, that is

$$\mathbf{v}_j^T\mathbf{v}_i = \delta_{ij}.$$

An orthogonal or unitary transformation

$$\mathbf{w}_i = \mathbf{U}\mathbf{v}_i,$$

preserves the dot product and orthogonality since

$$\mathbf{w}_j^T\mathbf{w}_i = (\mathbf{U}\mathbf{v}_j)^T\mathbf{U}\mathbf{v}_i = \mathbf{v}_j^T\mathbf{U}^T\mathbf{U}\mathbf{v}_i = \mathbf{v}_j^T\mathbf{v}_i = \delta_{ij}.$$

This means that if the coefficients $C_{p\lambda}$ belong to a unitary or orthogonal trasformation (using the Dirac bra-ket notation)

$$|p\rangle = \sum_\lambda C_{p\lambda}|\lambda\rangle,$$

orthogonality is preserved, that is $\langle\alpha|\beta\rangle = \delta_{\alpha\beta}$ and $\langle p|q\rangle = \delta_{pq}$.

This propertry is extremely useful when we build up a basis of many-body Stater determinant based states.

Note also that although a basis $|\alpha\rangle$ contains an infinity of states, for practical calculations we have always to make some truncations.

Before we develop for example the Hartree-Fock equations, there is another very useful property of determinants that we will use both in connection with Hartree-Fock calculations and later shell-model calculations.

Consider the following determinant

$$\begin{vmatrix} \alpha_1 b_{11} + \alpha_2 s b_{12} & a_{12} \\ \alpha_1 b_{21} + \alpha_2 b_{22} & a_{22} \end{vmatrix} = \alpha_1 \begin{vmatrix} b_{11} & a_{12} \\ b_{21} & a_{22} \end{vmatrix} + \alpha_2 \begin{vmatrix} b_{12} & a_{12} \\ b_{22} & a_{22} \end{vmatrix}$$

We can generalize this to an $n\times n$ matrix and have

$$\begin{vmatrix} a_{11} & a_{12} & \dots & \sum_{k=1}^n c_k b_{1k} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \sum_{k=1}^n c_k b_{2k} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \sum_{k=1}^n c_k b_{nk} & \dots & a_{nn} \end{vmatrix} = \sum_{k=1}^n c_k \begin{vmatrix} a_{11} & a_{12} & \dots & b_{1k} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & b_{2k} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & b_{nk} & \dots & a_{nn} \end{vmatrix}.$$

This is a property we will use in our Hartree-Fock discussions.

We can generalize the previous results, now with all elements a_{ij} being given as functions of linear combinations of various coefficients c and elements b_{ij},

$$\begin{vmatrix} \sum_{k=1}^n b_{1k}c_{k1} & \sum_{k=1}^n b_{1k}c_{k2} & \dots & \sum_{k=1}^n b_{1k}c_{kj} & \dots & \sum_{k=1}^n b_{1k}c_{kn} \\ \sum_{k=1}^n b_{2k}c_{k1} & \sum_{k=1}^n b_{2k}c_{k2} & \dots & \sum_{k=1}^n b_{2k}c_{kj} & \dots & \sum_{k=1}^n b_{2k}c_{kn} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \sum_{k=1}^n b_{nk}c_{k1} & \sum_{k=1}^n b_{nk}c_{k2} & \dots & \sum_{k=1}^n b_{nk}c_{kj} & \dots & \sum_{k=1}^n b_{nk}c_{kn} \end{vmatrix} = det(\mathbf{C})det(\mathbf{B}),$$

where $det(\mathbf{C})$ and $det(\mathbf{B})$ are the determinants of $n\times n$ matrices with elements c_{ij} and b_{ij} respectively. This is a property we will use in our Hartree-Fock discussions. Convince yourself about the correctness of the above expression by setting $n=2$.

With our definition of the new basis in terms of an orthogonal basis we have

$$\psi_p(x) = \sum_\lambda C_{p\lambda}\phi_\lambda(x).$$

If the coefficients $C_{p\lambda}$ belong to an orthogonal or unitary matrix, the new basis is also orthogonal. Our Slater determinant in the new basis $\psi_p(x)$ is written as

$$\frac{1}{\sqrt{A!}}\begin{vmatrix} \psi_p(x_1) & \psi_p(x_2) & \dots & \dots & \psi_p(x_A) \\ \psi_q(x_1) & \psi_q(x_2) & \dots & \dots & \psi_q(x_A) \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \psi_t(x_1) & \psi_t(x_2) & \dots & \dots & \psi_t(x_A) \end{vmatrix} = \frac{1}{\sqrt{A!}}\begin{vmatrix} \sum_\lambda C_{p\lambda}\phi_\lambda(x_1) & \sum_\lambda C_{p\lambda}\phi_\lambda(x_2) & \dots & \dots & \sum_\lambda C_{p\lambda}\phi_\lambda(x_A) \\ \sum_\lambda C_{q\lambda}\phi_\lambda(x_1) & \sum_\lambda C_{q\lambda}\phi_\lambda(x_2) & \dots & \dots & \sum_\lambda C_{q\lambda}\phi_\lambda(x_A) \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \sum_\lambda C_{t\lambda}\phi_\lambda(x_1) & \sum_\lambda C_{t\lambda}\phi_\lambda(x_2) & \dots & \dots & \sum_\lambda C_{t\lambda}\phi_\lambda(x_A) \end{vmatrix},$$

which is nothing but $det(\mathbf{C})det(\Phi)$, with $det(\Phi)$ being the determinant given by the basis functions $\phi_\lambda(x)$.

In our discussions hereafter we will use our definitions of single-particle states above and below the Fermi (F) level given by the labels $ijkl\dots \le F$ for so-called single-hole states and $abcd\dots > F$ for so-called particle states. For general single-particle states we employ the labels $pqrs\dots$.

The energy functional is

$$E[\Phi] = \sum_{\mu=1}^{A} \langle \mu|h|\mu\rangle + \frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A} \langle \mu\nu|\hat{v}|\mu\nu\rangle_{AS},$$

we found the expression for the energy functional in terms of the basis function $\phi_\lambda(\mathbf{r})$. We then varied the above energy functional with respect to the basis functions $|\mu\rangle$. Now we are interested in defining a new basis defined in terms of a chosen basis as defined in Eq. (4.4). We can then rewrite the energy functional as

$$E[\Phi^{New}] = \sum_{i=1}^{A} \langle i|h|i\rangle + \frac{1}{2}\sum_{ij=1}^{A} \langle ij|\hat{v}|ij\rangle_{AS}, \tag{1.12}$$

where Φ^{New} is the new Slater determinant defined by the new basis of Eq. (4.4).

Using Eq. (4.4) we can rewrite Eq. (4.5) as

$$E[\Psi] = \sum_{i=1}^{A}\sum_{\alpha\beta} C^*_{i\alpha}C_{i\beta}\langle\alpha|h|\beta\rangle + \frac{1}{2}\sum_{ij=1}^{A}\sum_{\alpha\beta\gamma\delta} C^*_{i\alpha}C^*_{j\beta}C_{i\gamma}C_{j\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{AS}. \tag{1.13}$$

Chapter 2
Second quantization

We introduce the time-independent operators $a^\dagger_\alpha$ and a_α which create and annihilate, respectively, a particle in the single-particle state φ_α. We define the fermion creation operator $a^\dagger_\alpha$

$$a^\dagger_\alpha|0\rangle \equiv |\alpha\rangle, \tag{2.1}$$

and

$$a^\dagger_\alpha|\alpha_1\dots\alpha_n\rangle_{\mathrm{AS}} \equiv |\alpha\alpha_1\dots\alpha_n\rangle_{\mathrm{AS}} \tag{2.2}$$

In Eq. (2.1) the operator $a^\dagger_\alpha$ acts on the vacuum state $|0\rangle$, which does not contain any particles. Alternatively, we could define a closed-shell nucleus or atom as our new vacuum, but then we need to introduce the particle-hole formalism, see the discussion to come.

In Eq. (2.2) $a^\dagger_\alpha$ acts on an antisymmetric n-particle state and creates an antisymmetric $(n+1)$-particle state, where the one-body state φ_α is occupied, under the condition that $\alpha \neq \alpha_1,\alpha_2,\dots,\alpha_n$. It follows that we can express an antisymmetric state as the product of the creation operators acting on the vacuum state.

$$|\alpha_1\dots\alpha_n\rangle_{\mathrm{AS}} = a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}\dots a^\dagger_{\alpha_n}|0\rangle \tag{2.3}$$

It is easy to derive the commutation and anticommutation rules for the fermionic creation operators $a^\dagger_\alpha$. Using the antisymmetry of the states (2.3)

$$|\alpha_1\dots\alpha_i\dots\alpha_k\dots\alpha_n\rangle_{\mathrm{AS}} = -|\alpha_1\dots\alpha_k\dots\alpha_i\dots\alpha_n\rangle_{\mathrm{AS}} \tag{2.4}$$

we obtain

$$a^\dagger_{\alpha_i}a^\dagger_{\alpha_k} = -a^\dagger_{\alpha_k}a^\dagger_{\alpha_i} \tag{2.5}$$

Using the Pauli principle

$$|\alpha_1\dots\alpha_i\dots\alpha_i\dots\alpha_n\rangle_{\mathrm{AS}} = 0 \tag{2.6}$$

it follows that

$$a^\dagger_{\alpha_i}a^\dagger_{\alpha_i} = 0. \tag{2.7}$$

If we combine Eqs. (2.5) and (2.7), we obtain the well-known anti-commutation rule

$$a^\dagger_\alpha a^\dagger_\beta + a^\dagger_\beta a^\dagger_\alpha \equiv \{a^\dagger_\alpha, a^\dagger_\beta\} = 0 \tag{2.8}$$

The hermitian conjugate of $a^\dagger_\alpha$ is

$$a_\alpha = (a^\dagger_\alpha)^\dagger \tag{2.9}$$

If we take the hermitian conjugate of Eq. (2.8), we arrive at

$$\{a_\alpha, a_\beta\} = 0 \tag{2.10}$$

What is the physical interpretation of the operator a_α and what is the effect of a_α on a given state $|\alpha_1\alpha_2\dots\alpha_n\rangle_{\mathrm{AS}}$? Consider the following matrix element

$$\langle\alpha_1\alpha_2\dots\alpha_n|a_\alpha|\alpha'_1\alpha'_2\dots\alpha'_m\rangle \tag{2.11}$$

where both sides are antisymmetric. We distinguish between two cases. The first (1) is when $\alpha\in\{\alpha_i\}$. Using the Pauli principle of Eq. (2.6) it follows

$$\langle\alpha_1\alpha_2\dots\alpha_n|a_\alpha = 0 \tag{2.12}$$

The second (2) case is when $\alpha\notin\{\alpha_i\}$. It follows that an hermitian conjugation

$$\langle\alpha_1\alpha_2\dots\alpha_n|a_\alpha = \langle\alpha\alpha_1\alpha_2\dots\alpha_n| \tag{2.13}$$

Eq. (2.13) holds for case (1) since the lefthand side is zero due to the Pauli principle. We write Eq. (2.11) as

$$\langle\alpha_1\alpha_2\dots\alpha_n|a_\alpha|\alpha'_1\alpha'_2\dots\alpha'_m\rangle = \langle\alpha_1\alpha_2\dots\alpha_n|\alpha\alpha'_1\alpha'_2\dots\alpha'_m\rangle \tag{2.14}$$

Here we must have $m=n+1$ if Eq. (2.14) has to be trivially different from zero.

For the last case, the minus and plus signs apply when the sequence $\alpha,\alpha_1,\alpha_2,\dots,\alpha_n$ and $\alpha'_1,\alpha'_2,\dots,\alpha'_{n+1}$ are related to each other via even and odd permutations. If we assume that $\alpha\notin\{\alpha_i\}$ we obtain

$$\langle\alpha_1\alpha_2\dots\alpha_n|a_\alpha|\alpha'_1\alpha'_2\dots\alpha'_{n+1}\rangle = 0 \tag{2.15}$$

when $\alpha\in\{\alpha'_i\}$. If $\alpha\notin\{\alpha'_i\}$, we obtain

$$a_\alpha\underbrace{|\alpha'_1\alpha'_2\dots\alpha'_{n+1}\rangle}_{\neq\alpha} = 0 \tag{2.16}$$

and in particular

$$a_\alpha|0\rangle = 0 \tag{2.17}$$

If $\{\alpha\alpha_i\}=\{\alpha'_i\}$, performing the right permutations, the sequence $\alpha,\alpha_1,\alpha_2,\dots,\alpha_n$ is identical with the sequence $\alpha'_1,\alpha'_2,\dots,\alpha'_{n+1}$. This results in

$$\langle\alpha_1\alpha_2\dots\alpha_n|a_\alpha|\alpha\alpha_1\alpha_2\dots\alpha_n\rangle = 1 \tag{2.18}$$

and thus

$$a_\alpha|\alpha\alpha_1\alpha_2\dots\alpha_n\rangle = |\alpha_1\alpha_2\dots\alpha_n\rangle \tag{2.19}$$

The action of the operator a_α from the left on a state vector is to to remove one particle in the state α. If the state vector does not contain the single-particle state α, the outcome of the operation is zero. The operator a_α is normally called for a destruction or annihilation operator.

The next step is to establish the commutator algebra of $a^\dagger_\alpha$ and a_β.

The action of the anti-commutator $\{a^\dagger_\alpha,a_\alpha\}$ on a given n-particle state is

$$\begin{aligned} a^\dagger_\alpha a_\alpha\underbrace{|\alpha_1\alpha_2\dots\alpha_n\rangle}_{\neq\alpha} &= 0 \\ a_\alpha a^\dagger_\alpha\underbrace{|\alpha_1\alpha_2\dots\alpha_n\rangle}_{\neq\alpha} &= a_\alpha\underbrace{|\alpha\alpha_1\alpha_2\dots\alpha_n\rangle}_{\neq\alpha} = \underbrace{|\alpha_1\alpha_2\dots\alpha_n\rangle}_{\neq\alpha} \end{aligned} \tag{2.20}$$

if the single-particle state α is not contained in the state.

If it is present we arrive at

$$\begin{aligned}
a^\dagger_\alpha a_\alpha|\alpha_1\alpha_2\dots\alpha_k\alpha\alpha_{k+1}\dots\alpha_{n-1}\rangle &= a^\dagger_\alpha a_\alpha(-1)^k|\alpha\alpha_1\alpha_2\dots\alpha_{n-1}\rangle \\
&= (-1)^k|\alpha\alpha_1\alpha_2\dots\alpha_{n-1}\rangle = |\alpha_1\alpha_2\dots\alpha_k\alpha\alpha_{k+1}\dots\alpha_{n-1}\rangle \\
a_\alpha a^\dagger_\alpha|\alpha_1\alpha_2\dots\alpha_k\alpha\alpha_{k+1}\dots\alpha_{n-1}\rangle &= 0
\end{aligned} \tag{2.21}$$

From Eqs. (2.20) and (2.21) we arrive at

$$\{a^\dagger_\alpha, a_\alpha\} = a^\dagger_\alpha a_\alpha + a_\alpha a^\dagger_\alpha = 1 \tag{2.22}$$

The action of $\left\{a^\dagger_\alpha, a_\beta\right\}$, with $\alpha \neq \beta$ on a given state yields three possibilities. The first case is a state vector which contains both α and β, then either α or β and finally none of them.

The first case results in

$$\begin{aligned}
a^\dagger_\alpha a_\beta|\alpha\beta\alpha_1\alpha_2\dots\alpha_{n-2}\rangle &= 0 \\
a_\beta a^\dagger_\alpha|\alpha\beta\alpha_1\alpha_2\dots\alpha_{n-2}\rangle &= 0
\end{aligned} \tag{2.23}$$

while the second case gives

$$\begin{aligned}
a^\dagger_\alpha a_\beta|\beta\underbrace{\alpha_1\alpha_2\dots\alpha_{n-1}}_{\neq\alpha}\rangle &= |\alpha\underbrace{\alpha_1\alpha_2\dots\alpha_{n-1}}_{\neq\alpha}\rangle \\
a_\beta a^\dagger_\alpha|\beta\underbrace{\alpha_1\alpha_2\dots\alpha_{n-1}}_{\neq\alpha}\rangle &= a_\beta|\alpha\beta\underbrace{\beta\alpha_1\alpha_2\dots\alpha_{n-1}}_{\neq\alpha}\rangle \\
&= -|\alpha\underbrace{\alpha_1\alpha_2\dots\alpha_{n-1}}_{\neq\alpha}\rangle
\end{aligned} \tag{2.24}$$

Finally if the state vector does not contain α and β

$$\begin{aligned}
a^\dagger_\alpha a_\beta|\underbrace{\alpha_1\alpha_2\dots\alpha_n}_{\neq\alpha,\beta}\rangle &= \qquad 0 \\
a_\beta a^\dagger_\alpha|\underbrace{\alpha_1\alpha_2\dots\alpha_n}_{\neq\alpha,\beta}\rangle &= a_\beta|\alpha\underbrace{\alpha_1\alpha_2\dots\alpha_n}_{\neq\alpha,\beta}\rangle = 0
\end{aligned} \tag{2.25}$$

For all three cases we have

$$\{a^\dagger_\alpha, a_\beta\} = a^\dagger_\alpha a_\beta + a_\beta a^\dagger_\alpha = 0, \quad \alpha \neq \beta \tag{2.26}$$

We can summarize our findings in Eqs. (2.22) and (2.26) as

$$\{a^\dagger_\alpha, a_\beta\} = \delta_{\alpha\beta} \tag{2.27}$$

with $\delta_{\alpha\beta}$ is the Kroenecker δ-symbol.

The properties of the creation and annihilation operators can be summarized as (for fermions)

$$a^\dagger_\alpha|0\rangle \equiv |\alpha\rangle,$$

and

$$a^\dagger_\alpha|\alpha_1\dots\alpha_n\rangle_{\mathrm{AS}} \equiv |\alpha\alpha_1\dots\alpha_n\rangle_{\mathrm{AS}}.$$

from which follows

$$|\alpha_1\dots\alpha_n\rangle_{\mathrm{AS}} = a^\dagger_{\alpha_1}a^\dagger_{\alpha_2}\dots a^\dagger_{\alpha_n}|0\rangle.$$

The hermitian conjugate has the folowing properties

$$a_\alpha = (a_\alpha^\dagger)^\dagger.$$

Finally we found

$$a_\alpha \underbrace{|\alpha'_1\alpha'_2\dots\alpha'_{n+1}\rangle}_{\neq\alpha} = 0, \quad \text{in particular } a_\alpha|0\rangle = 0,$$

and

$$a_\alpha|\alpha\alpha_1\alpha_2\dots\alpha_n\rangle = |\alpha_1\alpha_2\dots\alpha_n\rangle,$$

and the corresponding commutator algebra

$$\{a_\alpha^\dagger, a_\beta^\dagger\} = \{a_\alpha, a_\beta\} = 0 \qquad \{a_\alpha^\dagger, a_\beta\} = \delta_{\alpha\beta}.$$

One-body operators in second quantization

A very useful operator is the so-called number-operator. Most physics cases we will study in this text conserve the total number of particles. The number operator is therefore a useful quantity which allows us to test that our many-body formalism conserves the number of particles. In for example (d,p) or (p,d) reactions it is important to be able to describe quantum mechanical states where particles get added or removed. A creation operator $a_\alpha^\dagger$ adds one particle to the single-particle state α of a give many-body state vector, while an annihilation operator a_α removes a particle from a single-particle state α.

Let us consider an operator proportional with $a_\alpha^\dagger a_\beta$ and $\alpha = \beta$. It acts on an n-particle state resulting in

$$a_\alpha^\dagger a_\alpha|\alpha_1\alpha_2\dots\alpha_n\rangle = \begin{cases} 0 & \alpha \notin \{\alpha_i\} \\ \\ |\alpha_1\alpha_2\dots\alpha_n\rangle & \alpha \in \{\alpha_i\} \end{cases} \tag{2.28}$$

Summing over all possible one-particle states we arrive at

$$\left(\sum_\alpha a_\alpha^\dagger a_\alpha\right)|\alpha_1\alpha_2\dots\alpha_n\rangle = n|\alpha_1\alpha_2\dots\alpha_n\rangle \tag{2.29}$$

The operator

$$\hat{N} = \sum_\alpha a_\alpha^\dagger a_\alpha \tag{2.30}$$

is called the number operator since it counts the number of particles in a give state vector when it acts on the different single-particle states. It acts on one single-particle state at the time and falls therefore under category one-body operators. Next we look at another important one-body operator, namely $\hat{H}_0$ and study its operator form in the occupation number representation.

We want to obtain an expression for a one-body operator which conserves the number of particles. Here we study the one-body operator for the kinetic energy plus an eventual external one-body potential. The action of this operator on a particular n-body state with its pertinent expectation value has already been studied in coordinate space. In coordinate space the operator reads

$$\hat{H}_0 = \sum_i \hat{h}_0(x_i) \tag{2.31}$$

and the anti-symmetric n-particle Slater determinant is defined as

$$\Phi(x_1, x_2, \dots, x_n, \alpha_1, \alpha_2, \dots, \alpha_n) = \frac{1}{\sqrt{n!}} \sum_p (-1)^p \hat{P} \psi_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi_{\alpha_n}(x_n).$$

Defining

$$\hat{h}_0(x_i)\psi_{\alpha_i}(x_i) = \sum_{\alpha'_k} \psi_{\alpha'_k}(x_i)\langle\alpha'_k|\hat{h}_0|\alpha_k\rangle \tag{2.32}$$

we can easily evaluate the action of $\hat{H}_0$ on each product of one-particle functions in Slater determinant. From Eq. (2.32) we obtain the following result without permuting any particle pair

$$\begin{aligned} & \left(\sum_i \hat{h}_0(x_i)\right)\psi_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi_{\alpha_n}(x_n) \\ = & \sum_{\alpha'_1}\langle\alpha'_1|\hat{h}_0|\alpha_1\rangle\psi_{\alpha'_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi_{\alpha_n}(x_n) \\ + & \sum_{\alpha'_2}\langle\alpha'_2|\hat{h}_0|\alpha_2\rangle\psi_{\alpha_1}(x_1)\psi_{\alpha'_2}(x_2)\dots\psi_{\alpha_n}(x_n) \\ + & \dots \\ + & \sum_{\alpha'_n}\langle\alpha'_n|\hat{h}_0|\alpha_n\rangle\psi_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi_{\alpha'_n}(x_n) \end{aligned} \tag{2.33}$$

If we interchange particles 1 and 2 we obtain

$$\begin{aligned} & \left(\sum_i \hat{h}_0(x_i)\right)\psi_{\alpha_1}(x_2)\psi_{\alpha_1}(x_2)\dots\psi_{\alpha_n}(x_n) \\ = & \sum_{\alpha'_2}\langle\alpha'_2|\hat{h}_0|\alpha_2\rangle\psi_{\alpha_1}(x_2)\psi_{\alpha'_2}(x_1)\dots\psi_{\alpha_n}(x_n) \\ + & \sum_{\alpha'_1}\langle\alpha'_1|\hat{h}_0|\alpha_1\rangle\psi_{\alpha'_1}(x_2)\psi_{\alpha_2}(x_1)\dots\psi_{\alpha_n}(x_n) \\ + & \dots \\ + & \sum_{\alpha'_n}\langle\alpha'_n|\hat{h}_0|\alpha_n\rangle\psi_{\alpha_1}(x_2)\psi_{\alpha_1}(x_2)\dots\psi_{\alpha'_n}(x_n) \end{aligned} \tag{2.34}$$

We can continue by computing all possible permutations. We rewrite also our Slater determinant in its second quantized form and skip the dependence on the quantum numbers x_i. Summing up all contributions and taking care of all phases $(-1)^p$ we arrive at

$$\begin{aligned} \hat{H}_0|\alpha_1,\alpha_2,\dots,\alpha_n\rangle = & \sum_{\alpha'_1}\langle\alpha'_1|\hat{h}_0|\alpha_1\rangle|\alpha'_1\alpha_2\dots\alpha_n\rangle \\ + & \sum_{\alpha'_2}\langle\alpha'_2|\hat{h}_0|\alpha_2\rangle|\alpha_1\alpha'_2\dots\alpha_n\rangle \\ + & \dots \\ + & \sum_{\alpha'_n}\langle\alpha'_n|\hat{h}_0|\alpha_n\rangle|\alpha_1\alpha_2\dots\alpha'_n\rangle \end{aligned} \tag{2.35}$$

In Eq. (2.35) we have expressed the action of the one-body operator of Eq. (2.31) on the n-body state in its second quantized form. This equation can be further manipulated if we use the properties of the creation and annihilation operator on each primed quantum number, that is

$$|\alpha_1\alpha_2\dots\alpha'_k\dots\alpha_n\rangle = a^{\dagger}_{\alpha'_k}a_{\alpha_k}|\alpha_1\alpha_2\dots\alpha_k\dots\alpha_n\rangle \tag{2.36}$$

Inserting this in the right-hand side of Eq. (2.35) results in

$$\begin{aligned}
\hat{H}_0|\alpha_1\alpha_2\dots\alpha_n\rangle = & \sum_{\alpha_1'}\langle\alpha_1'|\hat{h}_0|\alpha_1\rangle a^{\dagger}_{\alpha_1'}a_{\alpha_1}|\alpha_1\alpha_2\dots\alpha_n\rangle \\
+ & \sum_{\alpha_2'}\langle\alpha_2'|\hat{h}_0|\alpha_2\rangle a^{\dagger}_{\alpha_2'}a_{\alpha_2}|\alpha_1\alpha_2\dots\alpha_n\rangle \\
+ & \dots \\
+ & \sum_{\alpha_n'}\langle\alpha_n'|\hat{h}_0|\alpha_n\rangle a^{\dagger}_{\alpha_n'}a_{\alpha_n}|\alpha_1\alpha_2\dots\alpha_n\rangle \\
= & \sum_{\alpha,\beta}\langle\alpha|\hat{h}_0|\beta\rangle a^{\dagger}_{\alpha}a_{\beta}|\alpha_1\alpha_2\dots\alpha_n\rangle
\end{aligned} \tag{2.37}$$

In the number occupation representation or second quantization we get the following expression for a one-body operator which conserves the number of particles

$$\hat{H}_0 = \sum_{\alpha\beta}\langle\alpha|\hat{h}_0|\beta\rangle a^{\dagger}_{\alpha}a_{\beta} \tag{2.38}$$

Obviously, $\hat{H}_0$ can be replaced by any other one-body operator which preserved the number of particles. The stucture of the operator is therefore not limited to say the kinetic or single-particle energy only.

The opearator $\hat{H}_0$ takes a particle from the single-particle state β to the single-particle state α with a probability for the transition given by the expectation value $\langle\alpha|\hat{h}_0|\beta\rangle$.

It is instructive to verify Eq. (2.38) by computing the expectation value of $\hat{H}_0$ between two single-particle states

$$\langle\alpha_1|\hat{h}_0|\alpha_2\rangle = \sum_{\alpha\beta}\langle\alpha|\hat{h}_0|\beta\rangle\langle 0|a_{\alpha_1}a^{\dagger}_{\alpha}a_{\beta}a^{\dagger}_{\alpha_2}|0\rangle \tag{2.39}$$

Using the commutation relations for the creation and annihilation operators we have

$$a_{\alpha_1}a^{\dagger}_{\alpha}a_{\beta}a^{\dagger}_{\alpha_2} = (\delta_{\alpha\alpha_1} - a^{\dagger}_{\alpha}a_{\alpha_1})(\delta_{\beta\alpha_2} - a^{\dagger}_{\alpha_2}a_{\beta}), \tag{2.40}$$

which results in

$$\langle 0|a_{\alpha_1}a^{\dagger}_{\alpha}a_{\beta}a^{\dagger}_{\alpha_2}|0\rangle = \delta_{\alpha\alpha_1}\delta_{\beta\alpha_2} \tag{2.41}$$

and

$$\langle\alpha_1|\hat{h}_0|\alpha_2\rangle = \sum_{\alpha\beta}\langle\alpha|\hat{h}_0|\beta\rangle\delta_{\alpha\alpha_1}\delta_{\beta\alpha_2} = \langle\alpha_1|\hat{h}_0|\alpha_2\rangle \tag{2.42}$$

Two-body operators in second quantization

Let us now derive the expression for our two-body interaction part, which also conserves the number of particles. We can proceed in exactly the same way as for the one-body operator. In the coordinate representation our two-body interaction part takes the following expression

$$\hat{H}_I = \sum_{i<j} V(x_i, x_j) \tag{2.43}$$

where the summation runs over distinct pairs. The term V can be an interaction model for the nucleon-nucleon interaction or the interaction between two electrons. It can also include additional two-body interaction terms.

The action of this operator on a product of two single-particle functions is defined as

$$V(x_i, x_j)\psi_{\alpha_k}(x_i)\psi_{\alpha_l}(x_j) = \sum_{\alpha_k'\alpha_l'}\psi'_{\alpha_k}(x_i)\psi'_{\alpha_l}(x_j)\langle\alpha_k'\alpha_l'|\hat{v}|\alpha_k\alpha_l\rangle \tag{2.44}$$

We can now let $\hat{H}_I$ act on all terms in the linear combination for $|\alpha_1\alpha_2\dots\alpha_n\rangle$. Without any permutations we have

$$\begin{aligned}
&\left(\sum_{i<j} V(x_i,x_j)\right)\psi_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi_{\alpha_n}(x_n) \\
= & \sum_{\alpha_1'\alpha_2'}\langle\alpha_1'\alpha_2'|\hat{v}|\alpha_1\alpha_2\rangle\psi'_{\alpha_1}(x_1)\psi'_{\alpha_2}(x_2)\dots\psi_{\alpha_n}(x_n) \\
+ & \dots \\
+ & \sum_{\alpha_1'\alpha_n'}\langle\alpha_1'\alpha_n'|\hat{v}|\alpha_1\alpha_n\rangle\psi'_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi'_{\alpha_n}(x_n) \\
+ & \dots \\
+ & \sum_{\alpha_2'\alpha_n'}\langle\alpha_2'\alpha_n'|\hat{v}|\alpha_2\alpha_n\rangle\psi_{\alpha_1}(x_1)\psi'_{\alpha_2}(x_2)\dots\psi'_{\alpha_n}(x_n) \\
+ & \dots
\end{aligned} \tag{2.45}$$

where on the rhs we have a term for each distinct pairs.

For the other terms on the rhs we obtain similar expressions and summing over all terms we obtain

$$\begin{aligned}
H_I|\alpha_1\alpha_2\dots\alpha_n\rangle = & \sum_{\alpha_1',\alpha_2'}\langle\alpha_1'\alpha_2'|\hat{v}|\alpha_1\alpha_2\rangle|\alpha_1'\alpha_2'\dots\alpha_n\rangle \\
+ & \dots \\
+ & \sum_{\alpha_1',\alpha_n'}\langle\alpha_1'\alpha_n'|\hat{v}|\alpha_1\alpha_n\rangle|\alpha_1'\alpha_2\dots\alpha_n'\rangle \\
+ & \dots \\
+ & \sum_{\alpha_2',\alpha_n'}\langle\alpha_2'\alpha_n'|\hat{v}|\alpha_2\alpha_n\rangle|\alpha_1\alpha_2'\dots\alpha_n'\rangle \\
+ & \dots
\end{aligned} \tag{2.46}$$

We introduce second quantization via the relation

$$\begin{aligned}
& a^\dagger_{\alpha_k'}a^\dagger_{\alpha_l'}a_{\alpha_l}a_{\alpha_k}|\alpha_1\alpha_2\dots\alpha_k\dots\alpha_l\dots\alpha_n\rangle \\
= & (-1)^{k-1}(-1)^{l-2}a^\dagger_{\alpha_k'}a^\dagger_{\alpha_l'}a_{\alpha_l}a_{\alpha_k}|\alpha_k\alpha_l\underbrace{\alpha_1\alpha_2\dots\alpha_n}_{\neq\alpha_k,\alpha_l}\rangle \\
= & (-1)^{k-1}(-1)^{l-2}|\alpha_k'\alpha_l'\underbrace{\alpha_1\alpha_2\dots\alpha_n}_{\neq\alpha_k',\alpha_l'}\rangle \\
= & |\alpha_1\alpha_2\dots\alpha_k'\dots\alpha_l'\dots\alpha_n\rangle
\end{aligned} \tag{2.47}$$

Inserting this in (2.46) gives

$$\begin{aligned}
H_I|\alpha_1\alpha_2\dots\alpha_n\rangle = & \sum_{\alpha_1',\alpha_2'} \langle\alpha_1'\alpha_2'|\hat{v}|\alpha_1\alpha_2\rangle a^{\dagger}_{\alpha_1'} a^{\dagger}_{\alpha_2'} a_{\alpha_2} a_{\alpha_1} |\alpha_1\alpha_2\dots\alpha_n\rangle \\
+ & \dots \\
= & \sum_{\alpha_1',\alpha_n'} \langle\alpha_1'\alpha_n'|\hat{v}|\alpha_1\alpha_n\rangle a^{\dagger}_{\alpha_1'} a^{\dagger}_{\alpha_n'} a_{\alpha_n} a_{\alpha_1} |\alpha_1\alpha_2\dots\alpha_n\rangle \\
+ & \dots \\
= & \sum_{\alpha_2',\alpha_n'} \langle\alpha_2'\alpha_n'|\hat{v}|\alpha_2\alpha_n\rangle a^{\dagger}_{\alpha_2'} a^{\dagger}_{\alpha_n'} a_{\alpha_n} a_{\alpha_2} |\alpha_1\alpha_2\dots\alpha_n\rangle \\
+ & \dots \\
= & \sum_{\alpha,\beta,\gamma,\delta}{}' \langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^{\dagger}_{\alpha} a^{\dagger}_{\beta} a_{\delta} a_{\gamma} |\alpha_1\alpha_2\dots\alpha_n\rangle
\end{aligned} \tag{2.48}$$

Here we let $\sum'$ indicate that the sums running over α and β run over all single-particle states, while the summations γ and δ run over all pairs of single-particle states. We wish to remove this restriction and since

$$\langle\alpha\beta|\hat{v}|\gamma\delta\rangle = \langle\beta\alpha|\hat{v}|\delta\gamma\rangle \tag{2.49}$$

we get

$$\sum_{\alpha\beta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^{\dagger}_{\alpha}a^{\dagger}_{\beta}a_{\delta}a_{\gamma} = \sum_{\alpha\beta}\langle\beta\alpha|\hat{v}|\delta\gamma\rangle a^{\dagger}_{\alpha}a^{\dagger}_{\beta}a_{\delta}a_{\gamma} \tag{2.50}$$

$$= \sum_{\alpha\beta}\langle\beta\alpha|\hat{v}|\delta\gamma\rangle a^{\dagger}_{\beta}a^{\dagger}_{\alpha}a_{\gamma}a_{\delta} \tag{2.51}$$

where we have used the anti-commutation rules.

Changing the summation indices α and β in (2.51) we obtain

$$\sum_{\alpha\beta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^{\dagger}_{\alpha}a^{\dagger}_{\beta}a_{\delta}a_{\gamma} = \sum_{\alpha\beta}\langle\alpha\beta|\hat{v}|\delta\gamma\rangle a^{\dagger}_{\alpha}a^{\dagger}_{\beta}a_{\gamma}a_{\delta} \tag{2.52}$$

From this it follows that the restriction on the summation over γ and δ can be removed if we multiply with a factor $\frac{1}{2}$, resulting in

$$\hat{H}_I = \frac{1}{2}\sum_{\alpha\beta\gamma\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^{\dagger}_{\alpha}a^{\dagger}_{\beta}a_{\delta}a_{\gamma} \tag{2.53}$$

where we sum freely over all single-particle states α, β, γ og δ.

With this expression we can now verify that the second quantization form of $\hat{H}_I$ in Eq. (2.53) results in the same matrix between two anti-symmetrized two-particle states as its corresponding coordinate space representation. We have

$$\langle\alpha_1\alpha_2|\hat{H}_I|\beta_1\beta_2\rangle = \frac{1}{2}\sum_{\alpha\beta\gamma\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle\langle 0|a_{\alpha_2}a_{\alpha_1}a^{\dagger}_{\alpha}a^{\dagger}_{\beta}a_{\delta}a_{\gamma}a^{\dagger}_{\beta_1}a^{\dagger}_{\beta_2}|0\rangle. \tag{2.54}$$

Using the commutation relations we get

$$\begin{aligned}
& a_{\alpha_2}a_{\alpha_1}a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma a^\dagger_{\beta_1}a^\dagger_{\beta_2} \\
= \quad & a_{\alpha_2}a_{\alpha_1}a^\dagger_\alpha a^\dagger_\beta (a_\delta \delta_{\gamma\beta_1} a^\dagger_{\beta_2} - a_\delta a^\dagger_{\beta_1} a_\gamma a^\dagger_{\beta_2}) \\
= \quad & a_{\alpha_2}a_{\alpha_1}a^\dagger_\alpha a^\dagger_\beta (\delta_{\gamma\beta_1}\delta_{\delta\beta_2} - \delta_{\gamma\beta_1}a^\dagger_{\beta_2}a_\delta - a_\delta a^\dagger_{\beta_1}\delta_{\gamma\beta_2} + a_\delta a^\dagger_{\beta_1}a^\dagger_{\beta_2}a_\gamma) \\
= \quad & a_{\alpha_2}a_{\alpha_1}a^\dagger_\alpha a^\dagger_\beta (\delta_{\gamma\beta_1}\delta_{\delta\beta_2} - \delta_{\gamma\beta_1}a^\dagger_{\beta_2}a_\delta \\
& -\delta_{\delta\beta_1}\delta_{\gamma\beta_2} + \delta_{\gamma\beta_2}a^\dagger_{\beta_1}a_\delta + a_\delta a^\dagger_{\beta_1}a^\dagger_{\beta_2}a_\gamma) \qquad (2.55)
\end{aligned}$$

The vacuum expectation value of this product of operators becomes

$$\begin{aligned}
& \langle 0|a_{\alpha_2}a_{\alpha_1}a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma a^\dagger_{\beta_1}a^\dagger_{\beta_2}|0\rangle \\
= \quad & (\delta_{\gamma\beta_1}\delta_{\delta\beta_2} - \delta_{\delta\beta_1}\delta_{\gamma\beta_2})\langle 0|a_{\alpha_2}a_{\alpha_1}a^\dagger_\alpha a^\dagger_\beta|0\rangle \\
= \quad & (\delta_{\gamma\beta_1}\delta_{\delta\beta_2} - \delta_{\delta\beta_1}\delta_{\gamma\beta_2})(\delta_{\alpha\alpha_1}\delta_{\beta\alpha_2} - \delta_{\beta\alpha_1}\delta_{\alpha\alpha_2}) \qquad (2.56)
\end{aligned}$$

Insertion of Eq. (2.56) in Eq. (2.54) results in

$$\begin{aligned}
\langle \alpha_1\alpha_2|\hat{H}_I|\beta_1\beta_2\rangle = \quad & \frac{1}{2}\big[\langle \alpha_1\alpha_2|\hat{v}|\beta_1\beta_2\rangle - \langle \alpha_1\alpha_2|\hat{v}|\beta_2\beta_1\rangle \\
& -\langle \alpha_2\alpha_1|\hat{v}|\beta_1\beta_2\rangle + \langle \alpha_2\alpha_1|\hat{v}|\beta_2\beta_1\rangle\big] \\
= \quad & \langle \alpha_1\alpha_2|\hat{v}|\beta_1\beta_2\rangle - \langle \alpha_1\alpha_2|\hat{v}|\beta_2\beta_1\rangle \\
= \quad & \langle \alpha_1\alpha_2|\hat{v}|\beta_1\beta_2\rangle_{\mathrm{AS}}. \qquad (2.57)
\end{aligned}$$

The two-body operator can also be expressed in terms of the anti-symmetrized matrix elements we discussed previously as

$$\begin{aligned}
\hat{H}_I = \quad & \frac{1}{2}\sum_{\alpha\beta\gamma\delta}\langle \alpha\beta|\hat{v}|\gamma\delta\rangle a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma \\
= \quad & \frac{1}{4}\sum_{\alpha\beta\gamma\delta}[\langle \alpha\beta|\hat{v}|\gamma\delta\rangle - \langle \alpha\beta|\hat{v}|\delta\gamma\rangle] a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma \\
= \quad & \frac{1}{4}\sum_{\alpha\beta\gamma\delta}\langle \alpha\beta|\hat{v}|\gamma\delta\rangle_{\mathrm{AS}} a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma \qquad (2.58)
\end{aligned}$$

The factors in front of the operator, either $\frac{1}{4}$ or $\frac{1}{2}$ tells whether we use antisymmetrized matrix elements or not.

We can now express the Hamiltonian operator for a many-fermion system in the occupation basis representation as

$$H = \sum_{\alpha,\beta}\langle \alpha|\hat{t} + \hat{u}_{\mathrm{ext}}|\beta\rangle a^\dagger_\alpha a_\beta + \frac{1}{4}\sum_{\alpha\beta\gamma\delta}\langle \alpha\beta|\hat{v}|\gamma\delta\rangle a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma. \qquad (2.59)$$

This is the form we will use in the rest of these lectures, assuming that we work with anti-symmetrized two-body matrix elements.

Particle-hole formalism

Second quantization is a useful and elegant formalism for constructing many-body states and quantum mechanical operators. One can express and translate many physical processes into simple pictures such as Feynman diagrams. Expecation values of many-body states are also easily calculated. However, although the equations are seemingly easy to set up, from a practical point of view, that is the solution of Schroedinger's equation, there is no particular gain. The many-body equation is equally hard to solve, irrespective of representation. The cliche that there is no free lunch brings us down to earth again. Note however that a transformation to a particular basis, for cases where the interaction obeys specific symmetries, can ease the solution of Schroedinger's equation.

But there is at least one important case where second quantization comes to our rescue. It is namely easy to introduce another reference state than the pure vacuum $|0\rangle$, where all single-particle states are active. With many particles present it is often useful to introduce another reference state than the vacuum state$|0\rangle$. We will label this state $|c\rangle$ (c for core) and as we will see it can reduce considerably the complexity and thereby the dimensionality of the many-body problem. It allows us to sum up to infinite order specific many-body correlations. The particle-hole representation is one of these handy representations.

In the original particle representation these states are products of the creation operators $a^{\dagger}_{\alpha_i}$ acting on the true vacuum $|0\rangle$. Following Eq. (2.3) we have

$$|\alpha_1\alpha_2\dots\alpha_{n-1}\alpha_n\rangle = a^{\dagger}_{\alpha_1}a^{\dagger}_{\alpha_2}\dots a^{\dagger}_{\alpha_{n-1}}a^{\dagger}_{\alpha_n}|0\rangle \tag{2.60}$$

$$|\alpha_1\alpha_2\dots\alpha_{n-1}\alpha_n\alpha_{n+1}\rangle = a^{\dagger}_{\alpha_1}a^{\dagger}_{\alpha_2}\dots a^{\dagger}_{\alpha_{n-1}}a^{\dagger}_{\alpha_n}a^{\dagger}_{\alpha_{n+1}}|0\rangle \tag{2.61}$$

$$|\alpha_1\alpha_2\dots\alpha_{n-1}\rangle = a^{\dagger}_{\alpha_1}a^{\dagger}_{\alpha_2}\dots a^{\dagger}_{\alpha_{n-1}}|0\rangle \tag{2.62}$$

If we use Eq. (2.60) as our new reference state, we can simplify considerably the representation of this state

$$|c\rangle \equiv |\alpha_1\alpha_2\dots\alpha_{n-1}\alpha_n\rangle = a^{\dagger}_{\alpha_1}a^{\dagger}_{\alpha_2}\dots a^{\dagger}_{\alpha_{n-1}}a^{\dagger}_{\alpha_n}|0\rangle \tag{2.63}$$

The new reference states for the $n+1$ and $n-1$ states can then be written as

$$|\alpha_1\alpha_2\dots\alpha_{n-1}\alpha_n\alpha_{n+1}\rangle = (-1)^n a^{\dagger}_{\alpha_{n+1}}|c\rangle \equiv (-1)^n|\alpha_{n+1}\rangle_c \tag{2.64}$$

$$|\alpha_1\alpha_2\dots\alpha_{n-1}\rangle = (-1)^{n-1}a_{\alpha_n}|c\rangle \equiv (-1)^{n-1}|\alpha_{n-1}\rangle_c \tag{2.65}$$

The first state has one additional particle with respect to the new vacuum state $|c\rangle$ and is normally referred to as a one-particle state or one particle added to the many-body reference state. The second state has one particle less than the reference vacuum state $|c\rangle$ and is referred to as a one-hole state. When dealing with a new reference state it is often convenient to introduce new creation and annihilation operators since we have from Eq. (2.65)

$$a_{\alpha}|c\rangle \neq 0 \tag{2.66}$$

since α is contained in $|c\rangle$, while for the true vacuum we have $a_{\alpha}|0\rangle = 0$ for all α.

The new reference state leads to the definition of new creation and annihilation operators which satisfy the following relations

$$b_{\alpha}|c\rangle = 0 \tag{2.67}$$

$$\{b^{\dagger}_{\alpha}, b^{\dagger}_{\beta}\} = \{b_{\alpha}, b_{\beta}\} = 0$$

$$\{b^{\dagger}_{\alpha}, b_{\beta}\} = \delta_{\alpha\beta} \tag{2.68}$$

We assume also that the new reference state is properly normalized

$$\langle c|c\rangle = 1 \tag{2.69}$$

The physical interpretation of these new operators is that of so-called quasiparticle states. This means that a state defined by the addition of one extra particle to a reference state $|c\rangle$ may not necesseraly be interpreted as one particle coupled to a core. We define now new creation operators that act on a state α creating a new quasiparticle state

$$b_\alpha^\dagger|c\rangle = \begin{cases} a_\alpha^\dagger|c\rangle = |\alpha\rangle, & \alpha > F \\ \\ a_\alpha|c\rangle = |\alpha^{-1}\rangle, & \alpha \leq F \end{cases} \tag{2.70}$$

where F is the Fermi level representing the last occupied single-particle orbit of the new reference state $|c\rangle$.

The annihilation is the hermitian conjugate of the creation operator

$$b_\alpha = (b_\alpha^\dagger)^\dagger,$$

resulting in

$$b_\alpha^\dagger = \begin{cases} a_\alpha^\dagger \ \alpha > F \\ \\ a_\alpha \ \alpha \leq F \end{cases} \qquad b_\alpha = \begin{cases} a_\alpha \ \alpha > F \\ \\ a_\alpha^\dagger \ \alpha \leq F \end{cases} \tag{2.71}$$

With the new creation and annihilation operator we can now construct many-body quasi-particle states, with one-particle-one-hole states, two-particle-two-hole states etc in the same fashion as we previously constructed many-particle states. We can write a general particle-hole state as

$$|\beta_1\beta_2\dots\beta_{n_p}\gamma_1^{-1}\gamma_2^{-1}\dots\gamma_{n_h}^{-1}\rangle \equiv \underbrace{b_{\beta_1}^\dagger b_{\beta_2}^\dagger\dots b_{\beta_{n_p}}^\dagger}_{>F}\underbrace{b_{\gamma_1}^\dagger b_{\gamma_2}^\dagger\dots b_{\gamma_{n_h}}^\dagger}_{\leq F}|c\rangle \tag{2.72}$$

We can now rewrite our one-body and two-body operators in terms of the new creation and annihilation operators. The number operator becomes

$$\hat{N} = \sum_\alpha a_\alpha^\dagger a_\alpha = \sum_{\alpha > F} b_\alpha^\dagger b_\alpha + n_c - \sum_{\alpha \leq F} b_\alpha^\dagger b_\alpha \tag{2.73}$$

where n_c is the number of particle in the new vacuum state $|c\rangle$. The action of $\hat{N}$ on a many-body state results in

$$N|\beta_1\beta_2\dots\beta_{n_p}\gamma_1^{-1}\gamma_2^{-1}\dots\gamma_{n_h}^{-1}\rangle = (n_p + n_c - n_h)|\beta_1\beta_2\dots\beta_{n_p}\gamma_1^{-1}\gamma_2^{-1}\dots\gamma_{n_h}^{-1}\rangle \tag{2.74}$$

Here $n = n_p + n_c - n_h$ is the total number of particles in the quasi-particle state of Eq. (2.72). Note that $\hat{N}$ counts the total number of particles present

$$N_{qp} = \sum_\alpha b_\alpha^\dagger b_\alpha, \tag{2.75}$$

gives us the number of quasi-particles as can be seen by computing

$$N_{qp} = |\beta_1\beta_2\dots\beta_{n_p}\gamma_1^{-1}\gamma_2^{-1}\dots\gamma_{n_h}^{-1}\rangle = (n_p + n_h)|\beta_1\beta_2\dots\beta_{n_p}\gamma_1^{-1}\gamma_2^{-1}\dots\gamma_{n_h}^{-1}\rangle \tag{2.76}$$

where $n_{qp} = n_p + n_h$ is the total number of quasi-particles.

We express the one-body operator $\hat{H}_0$ in terms of the quasi-particle creation and annihilation operators, resulting in

$$\begin{aligned} \hat{H}_0 = & \sum_{\alpha\beta > F} \langle\alpha|\hat{h}_0|\beta\rangle b_\alpha^\dagger b_\beta + \sum_{\alpha > F, \beta \leq F} \left[\langle\alpha|\hat{h}_0|\beta\rangle b_\alpha^\dagger b_\beta^\dagger + \langle\beta|\hat{h}_0|\alpha\rangle b_\beta b_\alpha\right] \\ + & \sum_{\alpha \leq F} \langle\alpha|\hat{h}_0|\alpha\rangle - \sum_{\alpha\beta \leq F} \langle\beta|\hat{h}_0|\alpha\rangle b_\alpha^\dagger b_\beta \end{aligned} \tag{2.77}$$

The first term gives contribution only for particle states, while the last one contributes only for holestates. The second term can create or destroy a set of quasi-particles and the third term is the contribution from the vacuum state $|c\rangle$.

Before we continue with the expressions for the two-body operator, we introduce a nomenclature we will use for the rest of this text. It is inspired by the notation used in quantum chemistry. We reserve the labels $i,j,k,\dots$ for hole states and $a,b,c,\dots$ for states above F, viz. particle states. This means also that we will skip the constraint $\leq F$ or $> F$ in the summation symbols. Our operator $\hat{H}_0$ reads now

$$\begin{aligned}\hat{H}_0 = & \sum_{ab}\langle a|\hat{h}|b\rangle b_a^\dagger b_b + \sum_{ai}\left[\langle a|\hat{h}|i\rangle b_a^\dagger b_i^\dagger + \langle i|\hat{h}|a\rangle b_i b_a\right] \\ + & \sum_{i}\langle i|\hat{h}|i\rangle - \sum_{ij}\langle j|\hat{h}|i\rangle b_i^\dagger b_j \end{aligned} \tag{2.78}$$

The two-particle operator in the particle-hole formalism is more complicated since we have to translate four indices $\alpha\beta\gamma\delta$ to the possible combinations of particle and hole states. When performing the commutator algebra we can regroup the operator in five different terms

$$\hat{H}_I = \hat{H}_I^{(a)} + \hat{H}_I^{(b)} + \hat{H}_I^{(c)} + \hat{H}_I^{(d)} + \hat{H}_I^{(e)} \tag{2.79}$$

Using anti-symmetrized matrix elements, bthe term $\hat{H}_I^{(a)}$ is

$$\hat{H}_I^{(a)} = \frac{1}{4}\sum_{abcd}\langle ab|\hat{V}|cd\rangle b_a^\dagger b_b^\dagger b_d b_c \tag{2.80}$$

The next term $\hat{H}_I^{(b)}$ reads

$$\hat{H}_I^{(b)} = \frac{1}{4}\sum_{abci}\left(\langle ab|\hat{V}|ci\rangle b_a^\dagger b_b^\dagger b_i^\dagger b_c + \langle ai|\hat{V}|cb\rangle b_a^\dagger b_i b_b b_c\right) \tag{2.81}$$

This term conserves the number of quasiparticles but creates or removes a three-particle-one-hole state. For $\hat{H}_I^{(c)}$ we have

$$\begin{aligned}\hat{H}_I^{(c)} = & \frac{1}{4}\sum_{abij}\left(\langle ab|\hat{V}|ij\rangle b_a^\dagger b_b^\dagger b_j^\dagger b_i^\dagger + \langle ij|\hat{V}|ab\rangle b_a b_b b_j b_i\right) + \\ & \frac{1}{2}\sum_{abij}\langle ai|\hat{V}|bj\rangle b_a^\dagger b_j^\dagger b_b b_i + \frac{1}{2}\sum_{abi}\langle ai|\hat{V}|bi\rangle b_a^\dagger b_b. \end{aligned} \tag{2.82}$$

The first line stands for the creation of a two-particle-two-hole state, while the second line represents the creation to two one-particle-one-hole pairs while the last term represents a contribution to the particle single-particle energy from the hole states, that is an interaction between the particle states and the hole states within the new vacuum state. The fourth term reads

$$\begin{aligned}\hat{H}_I^{(d)} = & \frac{1}{4}\sum_{aijk}\left(\langle ai|\hat{V}|jk\rangle b_a^\dagger b_k^\dagger b_j^\dagger b_i + \langle ji|\hat{V}|ak\rangle b_k^\dagger b_j b_i b_a\right) + \\ & \frac{1}{4}\sum_{aij}\left(\langle ai|\hat{V}|ji\rangle b_a^\dagger b_j^\dagger + \langle ji|\hat{V}|ai\rangle - \langle ji|\hat{V}|ia\rangle b_j b_a\right). \end{aligned} \tag{2.83}$$

The terms in the first line stand for the creation of a particle-hole state interacting with hole states, we will label this as a two-hole-one-particle contribution. The remaining terms are a particle-hole state interacting with the holes in the vacuum state. Finally we have

$$\hat{H}_I^{(e)} = \frac{1}{4}\sum_{ijkl}\langle kl|\hat{V}|ij\rangle b_i^\dagger b_j^\dagger b_l b_k + \frac{1}{2}\sum_{ijk}\langle ij|\hat{V}|kj\rangle b_k^\dagger b_i + \frac{1}{2}\sum_{ij}\langle ij|\hat{V}|ij\rangle \tag{2.84}$$

The first terms represents the interaction between two holes while the second stands for the interaction between a hole and the remaining holes in the vacuum state. It represents a contribution to single-hole energy to first order. The last term collects all contributions to the energy of the ground state of a closed-shell system arising from hole-hole correlations.

Summarizing and defining a normal-ordered Hamiltonian

$$\Phi_{AS}(\alpha_1,\dots,\alpha_A;x_1,\dots x_A)=\frac{1}{\sqrt{A}}\sum_{\hat{P}}(-1)^P\hat{P}\prod_{i=1}^{A}\psi_{\alpha_i}(x_i),$$

which is equivalent with $|\alpha_1\dots\alpha_A\rangle=a^\dagger_{\alpha_1}\dots a^\dagger_{\alpha_A}|0\rangle$. We have also

$$a^\dagger_p|0\rangle=|p\rangle,\quad a_p|q\rangle=\delta_{pq}|0\rangle$$

$$\delta_{pq}=\left\{a_p,a^\dagger_q\right\},$$

and

$$0=\left\{a^\dagger_p,a_q\right\}=\left\{a_p,a_q\right\}=\left\{a^\dagger_p,a^\dagger_q\right\}$$

$$|\Phi_0\rangle=|\alpha_1\dots\alpha_A\rangle,\quad \alpha_1,\dots,\alpha_A\leq\alpha_F$$

$$\left\{a^\dagger_p,a_q\right\}=\delta_{pq},p,q\leq\alpha_F$$

$$\left\{a_p,a^\dagger_q\right\}=\delta_{pq},p,q>\alpha_F$$

with $i,j,\ldots\leq\alpha_F,\quad a,b,\ldots>\alpha_F,\quad p,q,\ldots-$ any

$$a_i|\Phi_0\rangle=|\Phi_i\rangle,\qquad a^\dagger_a|\Phi_0\rangle=|\Phi^a\rangle$$

and

$$a^\dagger_i|\Phi_0\rangle=0\qquad a_a|\Phi_0\rangle=0$$

The one-body operator is defined as

$$\hat{F}=\sum_{pq}\langle p|\hat{f}|q\rangle a^\dagger_p a_q$$

while the two-body opreator is defined as

$$\hat{V}=\frac{1}{4}\sum_{pqrs}\langle pq|\hat{v}|rs\rangle_{AS}a^\dagger_p a^\dagger_q a_s a_r$$

where we have defined the antisymmetric matrix elements

$$\langle pq|\hat{v}|rs\rangle_{AS}=\langle pq|\hat{v}|rs\rangle-\langle pq|\hat{v}|sr\rangle.$$

We can also define a three-body operator

$$\hat{V}_3=\frac{1}{36}\sum_{pqrstu}\langle pqr|\hat{v}_3|stu\rangle_{AS}a^\dagger_p a^\dagger_q a^\dagger_r a_u a_t a_s$$

with the antisymmetrized matrix element

$$\langle pqr|\hat{v}_3|stu\rangle_{AS} = \langle pqr|\hat{v}_3|stu\rangle + \langle pqr|\hat{v}_3|tus\rangle + \langle pqr|\hat{v}_3|ust\rangle - \langle pqr|\hat{v}_3|sut\rangle - \langle pqr|\hat{v}_3|tsu\rangle - \langle pqr|\hat{v}_3|uts\rangle. \tag{2.85}$$

Operators in second quantization

In the build-up of a shell-model or FCI code that is meant to tackle large dimensionalities is the action of the Hamiltonian $\hat{H}$ on a Slater determinant represented in second quantization as

$$|\alpha_1 \dots \alpha_n\rangle = a^\dagger_{\alpha_1} a^\dagger_{\alpha_2} \dots a^\dagger_{\alpha_n} |0\rangle.$$

The time consuming part stems from the action of the Hamiltonian on the above determinant,

$$\left(\sum_{\alpha\beta} \langle\alpha|t+u|\beta\rangle a^\dagger_\alpha a_\beta + \frac{1}{4} \sum_{\alpha\beta\gamma\delta} \langle\alpha\beta|\hat{v}|\gamma\delta\rangle a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma \right) a^\dagger_{\alpha_1} a^\dagger_{\alpha_2} \dots a^\dagger_{\alpha_n} |0\rangle.$$

A practically useful way to implement this action is to encode a Slater determinant as a bit pattern.

Assume that we have at our disposal n different single-particle orbits $\alpha_0, \alpha_2, \dots, \alpha_{n-1}$ and that we can distribute among these orbits $N \le n$ particles.

A Slater determinant can then be coded as an integer of n bits. As an example, if we have $n = 16$ single-particle states $\alpha_0, \alpha_1, \dots, \alpha_{15}$ and $N = 4$ fermions occupying the states α_3, α_6, α_{10} and α_{13} we could write this Slater determinant as

$$\Phi_\Lambda = a^\dagger_{\alpha_3} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle.$$

The unoccupied single-particle states have bit value 0 while the occupied ones are represented by bit state 1. In the binary notation we would write this 16 bits long integer as

$$\begin{array}{cccccccccccccccc} \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \alpha_5 & \alpha_6 & \alpha_7 & \alpha_8 & \alpha_9 & \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} & \alpha_{15} \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$$

which translates into the decimal number

$$2^3 + 2^6 + 2^{10} + 2^{13} = 9288.$$

We can thus encode a Slater determinant as a bit pattern.

With N particles that can be distributed over n single-particle states, the total number of Slater determinats (and defining thereby the dimensionality of the system) is

$$\dim(\mathscr{H}) = \binom{n}{N}.$$

The total number of bit patterns is 2^n.

We assume again that we have at our disposal n different single-particle orbits $\alpha_0, \alpha_2, \dots, \alpha_{n-1}$ and that we can distribute among these orbits $N \le n$ particles. The ordering among these states is important as it defines the order of the creation operators. We will write the determinant

$$\Phi_\Lambda = a^\dagger_{\alpha_3} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle,$$

in a more compact way as

$$\Phi_{3,6,10,13} = |0001001000100100\rangle.$$

The action of a creation operator is thus

$$a^{\dagger}_{\alpha_4}\Phi_{3,6,10,13} = a^{\dagger}_{\alpha_4}|0001001000100100\rangle = a^{\dagger}_{\alpha_4}a^{\dagger}_{\alpha_3}a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle,$$

which becomes

$$-a^{\dagger}_{\alpha_3}a^{\dagger}_{\alpha_4}a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle = -|0001101000100100\rangle.$$

Similarly

$$a^{\dagger}_{\alpha_6}\Phi_{3,6,10,13} = a^{\dagger}_{\alpha_6}|0001001000100100\rangle = a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_3}a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle,$$

which becomes

$$-a^{\dagger}_{\alpha_4}(a^{\dagger}_{\alpha_6})^2a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle = 0!$$

This gives a simple recipe:

- If one of the bits b_j is 1 and we act with a creation operator on this bit, we return a null vector
- If $b_j = 0$, we set it to 1 and return a sign factor $(-1)^l$, where l is the number of bits set before bit j.

Consider the action of $a^{\dagger}_{\alpha_2}$ on various slater determinants:

$$\begin{aligned}
a^{\dagger}_{\alpha_2}\Phi_{00111} &= a^{\dagger}_{\alpha_2}|00111\rangle &&= 0\times|00111\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{01011} &= a^{\dagger}_{\alpha_2}|01011\rangle &&= (-1)\times|01111\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{01101} &= a^{\dagger}_{\alpha_2}|01101\rangle &&= 0\times|01101\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{01110} &= a^{\dagger}_{\alpha_2}|01110\rangle &&= 0\times|01110\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{10011} &= a^{\dagger}_{\alpha_2}|10011\rangle &&= (-1)\times|10111\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{10101} &= a^{\dagger}_{\alpha_2}|10101\rangle &&= 0\times|10101\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{10110} &= a^{\dagger}_{\alpha_2}|10110\rangle &&= 0\times|10110\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{11001} &= a^{\dagger}_{\alpha_2}|11001\rangle &&= (+1)\times|11101\rangle\\
a^{\dagger}_{\alpha_2}\Phi_{11010} &= a^{\dagger}_{\alpha_2}|11010\rangle &&= (+1)\times|11110\rangle
\end{aligned}$$

What is the simplest way to obtain the phase when we act with one annihilation(creation) operator on the given Slater determinant representation?

We have an SD representation

$$\Phi_{\Lambda} = a^{\dagger}_{\alpha_0}a^{\dagger}_{\alpha_3}a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle,$$

in a more compact way as

$$\Phi_{0,3,6,10,13} = |1001001000100100\rangle.$$

The action of

$$a^{\dagger}_{\alpha_4}a_{\alpha_0}\Phi_{0,3,6,10,13} = a^{\dagger}_{\alpha_4}|0001001000100100\rangle = a^{\dagger}_{\alpha_4}a^{\dagger}_{\alpha_3}a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle,$$

which becomes

$$-a^{\dagger}_{\alpha_3}a^{\dagger}_{\alpha_4}a^{\dagger}_{\alpha_6}a^{\dagger}_{\alpha_{10}}a^{\dagger}_{\alpha_{13}}|0\rangle = -|0001101000100100\rangle.$$

The action

$$a_{\alpha_0}\Phi_{0,3,6,10,13} = |0001001000100100\rangle,$$

can be obtained by subtracting the logical sum (AND operation) of $\Phi_{0,3,6,10,13}$ and a word which represents only α_0, that is

$$|1000000000000000\rangle,$$

from $\Phi_{0,3,6,10,13} = |1001001000100100\rangle$.

This operation gives $|0001001000100100\rangle$.

Similarly, we can form $a^{\dagger}_{\alpha_4}a_{\alpha_0}\Phi_{0,3,6,10,13}$, say, by adding $|0000100000000000\rangle$ to $a_{\alpha_0}\Phi_{0,3,6,10,13}$, first checking that their logical sum is zero in order to make sure that orbital α_4 is not already occupied.

It is trickier however to get the phase $(-1)^l$. One possibility is as follows

- Let S_1 be a word that represents the $1-$bit to be removed and all others set to zero.

In the previous example $S_1 = |1000000000000000\rangle$

- Define S_2 as the similar word that represents the bit to be added, that is in our case

$S_2 = |0000100000000000\rangle$.

- Compute then $S = S_1 - S_2$, which here becomes

$$S = |0111000000000000\rangle$$

- Perform then the logical AND operation of S with the word containing

$$\Phi_{0,3,6,10,13} = |1001001000100100\rangle,$$

which results in $|0001000000000000\rangle$. Counting the number of $1-$bits gives the phase. Here you need however an algorithm for bitcounting. Several efficient ones available.

Chapter 3
Full configuration interaction theory

Slater determinants as basis states, Repetition

The simplest possible choice for many-body wavefunctions are **product** wavefunctions. That is

$$\Psi(x_1,x_2,x_3,\ldots,x_A) \approx \phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\ldots$$

because we are really only good at thinking about one particle at a time. Such product wavefunctions, without correlations, are easy to work with; for example, if the single-particle states $\phi_i(x)$ are orthonormal, then the product wavefunctions are easy to orthonormalize.

Similarly, computing matrix elements of operators are relatively easy, because the integrals factorize.

The price we pay is the lack of correlations, which we must build up by using many, many product wavefunctions. (Thus we have a trade-off: compact representation of correlations but difficult integrals versus easy integrals but many states required.)

Slater determinants as basis states, repetition

Because we have fermions, we are required to have antisymmetric wavefunctions, e.g.

$$\Psi(x_1,x_2,x_3,\ldots,x_A) = -\Psi(x_2,x_1,x_3,\ldots,x_A)$$

etc. This is accomplished formally by using the determinantal formalism

$$\Psi(x_1,x_2,\ldots,x_A) = \frac{1}{\sqrt{A!}}| \left| \begin{array}{cccc} \phi_1(x_1) & \phi_1(x_2) & \ldots & \phi_1(x_A) \\ \phi_2(x_1) & \phi_2(x_2) & \ldots & \phi_2(x_A) \\ \vdots & & & \\ \phi_A(x_1) & \phi_A(x_2) & \ldots & \phi_A(x_A) \end{array} \right|$$

Product wavefunction + antisymmetry = Slater determinant.

Slater determinants as basis states

$$\Psi(x_1,x_2,\dots,x_A) = \frac{1}{\sqrt{A!}}| \left| \begin{matrix} \phi_1(x_1) & \phi_1(x_2) & \dots & \phi_1(x_A) \\ \phi_2(x_1) & \phi_2(x_2) & \dots & \phi_2(x_A) \\ \vdots & & & \\ \phi_A(x_1) & \phi_A(x_2) & \dots & \phi_A(x_A) \end{matrix} \right|$$

Properties of the determinant (interchange of any two rows or any two columns yields a change in sign; thus no two rows and no two columns can be the same) lead to the Pauli principle:

- No two particles can be at the same place (two columns the same); and
- No two particles can be in the same state (two rows the same).

Slater determinants as basis states

As a practical matter, however, Slater determinants beyond $N=4$ quickly become unwieldy. Thus we turn to the **occupation representation** or **second quantization** to simplify calculations.

The occupation representation or number representation, using fermion **creation** and **annihilation** operators, is compact and efficient. It is also abstract and, at first encounter, not easy to internalize. It is inspired by other operator formalism, such as the ladder operators for the harmonic oscillator or for angular momentum, but unlike those cases, the operators **do not have coordinate space representations**.

Instead, one can think of fermion creation/annihilation operators as a game of symbols that compactly reproduces what one would do, albeit clumsily, with full coordinate-space Slater determinants.

Quick repetition of the occupation representation

We start with a set of orthonormal single-particle states $\{\phi_i(x)\}$. (Note: this requirement, and others, can be relaxed, but leads to a more involved formalism.) **Any** orthonormal set will do.

To each single-particle state $\phi_i(x)$ we associate a creation operator $\hat{a}^\dagger_i$ and an annihilation operator $\hat{a}_i$.

When acting on the vacuum state $|0\rangle$, the creation operator $\hat{a}^\dagger_i$ causes a particle to occupy the single-particle state $\phi_i(x)$:

$$\phi_i(x) \rightarrow \hat{a}^\dagger_i|0\rangle$$

Quick repetition of the occupation representation

But with multiple creation operators we can occupy multiple states:

$$\phi_i(x)\phi_j(x')\phi_k(x'') \to \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_k^\dagger |0\rangle.$$

Now we impose antisymmetry, by having the fermion operators satisfy **anticommutation relations**:

$$\hat{a}_i^\dagger \hat{a}_j^\dagger + \hat{a}_j^\dagger \hat{a}_i^\dagger = [\hat{a}_i^\dagger, \hat{a}_j^\dagger]_+ = \{\hat{a}_i^\dagger, \hat{a}_j^\dagger\} = 0$$

so that

$$\hat{a}_i^\dagger \hat{a}_j^\dagger = -\hat{a}_j^\dagger \hat{a}_i^\dagger$$

Quick repetition of the occupation representation

Because of this property, automatically $\hat{a}_i^\dagger \hat{a}_i^\dagger = 0$, enforcing the Pauli exclusion principle. Thus when writing a Slater determinant using creation operators,

$$\hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_k^\dagger \dots |0\rangle$$

each index $i, j, k, \dots$ must be unique.

For some relevant exercises with solutions see chapter 8 of Lecture Notes in Physics, volume 936.

Full Configuration Interaction Theory

We have defined the ansatz for the ground state as

$$|\Phi_0\rangle = \left(\prod_{i\le F} \hat{a}_i^\dagger\right)|0\rangle,$$

where the index i defines different single-particle states up to the Fermi level. We have assumed that we have N fermions. A given one-particle-one-hole ($1p1h$) state can be written as

$$|\Phi_i^a\rangle = \hat{a}_a^\dagger \hat{a}_i |\Phi_0\rangle,$$

while a $2p2h$ state can be written as

$$|\Phi_{ij}^{ab}\rangle = \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i |\Phi_0\rangle,$$

and a general $NpNh$ state as

$$|\Phi_{ijk\dots}^{abc\dots}\rangle = \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \dots \hat{a}_k \hat{a}_j \hat{a}_i |\Phi_0\rangle.$$

Full Configuration Interaction Theory

We can then expand our exact state function for the ground state as

$$|\Psi_0\rangle = C_0|\Phi_0\rangle + \sum_{ai} C_i^a|\Phi_i^a\rangle + \sum_{abij} C_{ij}^{ab}|\Phi_{ij}^{ab}\rangle + \cdots = (C_0 + \hat{C})|\Phi_0\rangle,$$

where we have introduced the so-called correlation operator

$$\hat{C} = \sum_{ai} C_i^a \hat{a}_a^\dagger \hat{a}_i + \sum_{abij} C_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i + \dots$$

Since the normalization of Ψ_0 is at our disposal and since C_0 is by hypothesis non-zero, we may arbitrarily set $C_0 = 1$ with corresponding proportional changes in all other coefficients. Using this so-called intermediate normalization we have

$$\langle\Psi_0|\Phi_0\rangle = \langle\Phi_0|\Phi_0\rangle = 1,$$

resulting in

$$|\Psi_0\rangle = (1 + \hat{C})|\Phi_0\rangle.$$

Full Configuration Interaction Theory

We rewrite

$$|\Psi_0\rangle = C_0|\Phi_0\rangle + \sum_{ai} C_i^a|\Phi_i^a\rangle + \sum_{abij} C_{ij}^{ab}|\Phi_{ij}^{ab}\rangle + \dots,$$

in a more compact form as

$$|\Psi_0\rangle = \sum_{PH} C_H^P \Phi_H^P = \left(\sum_{PH} C_H^P \hat{A}_H^P\right)|\Phi_0\rangle,$$

where H stands for $0,1,\dots,n$ hole states and P for $0,1,\dots,n$ particle states. Our requirement of unit normalization gives

$$\langle\Psi_0|\Phi_0\rangle = \sum_{PH} |C_H^P|^2 = 1,$$

and the energy can be written as

$$E = \langle\Psi_0|\hat{H}|\Phi_0\rangle = \sum_{PP'HH'} C_H^{*P} \langle\Phi_H^P|\hat{H}|\Phi_{H'}^{P'}\rangle C_{H'}^{P'}.$$

Full Configuration Interaction Theory

Normally

$$E = \langle \Psi_0 | \hat{H} | \Phi_0 \rangle = \sum_{PP'HH'} C_H^{*P} \langle \Phi_H^P | \hat{H} | \Phi_{H'}^{P'} \rangle C_{H'}^{P'},$$

is solved by diagonalization setting up the Hamiltonian matrix defined by the basis of all possible Slater determinants. A diagonalization is equivalent to finding the variational minimum of

$$\langle \Psi_0 | \hat{H} | \Phi_0 \rangle - \lambda \langle \Psi_0 | \Phi_0 \rangle,$$

where λ is a variational multiplier to be identified with the energy of the system. The minimization process results in

$$\delta\left[\langle \Psi_0 | \hat{H} | \Phi_0 \rangle - \lambda \langle \Psi_0 | \Phi_0 \rangle \right] =$$

$$\sum_{P'H'} \left\{ \delta[C_H^{*P}] \langle \Phi_H^P | \hat{H} | \Phi_{H'}^{P'} \rangle C_{H'}^{P'} + C_H^{*P} \langle \Phi_H^P | \hat{H} | \Phi_{H'}^{P'} \rangle \delta[C_{H'}^{P'}] - \lambda (\delta[C_H^{*P}] C_{H'}^{P'} + C_H^{*P} \delta[C_{H'}^{P'}] \right\} = 0.$$

Since the coefficients $\delta[C_H^{*P}]$ and $\delta[C_{H'}^{P'}]$ are complex conjugates it is necessary and sufficient to require the quantities that multiply with $\delta[C_H^{*P}]$ to vanish.

Full Configuration Interaction Theory

This leads to

$$\sum_{P'H'} \langle \Phi_H^P | \hat{H} | \Phi_{H'}^{P'} \rangle C_{H'}^{P'} - \lambda C_H^P = 0,$$

for all sets of P and H.

If we then multiply by the corresponding C_H^{*P} and sum over PH we obtain

$$\sum_{PP'HH'} C_H^{*P} \langle \Phi_H^P | \hat{H} | \Phi_{H'}^{P'} \rangle C_{H'}^{P'} - \lambda \sum_{PH} |C_H^P|^2 = 0,$$

leading to the identification $\lambda = E$. This means that we have for all PH sets

$$\sum_{P'H'} \langle \Phi_H^P | \hat{H} - E | \Phi_{H'}^{P'} \rangle = 0. \qquad (3.1)$$

Full Configuration Interaction Theory

An alternative way to derive the last equation is to start from

$$(\hat{H} - E) | \Psi_0 \rangle = (\hat{H} - E) \sum_{P'H'} C_{H'}^{P'} | \Phi_{H'}^{P'} \rangle = 0,$$

and if this equation is successively projected against all Φ_H^P in the expansion of Ψ, then the last equation on the previous slide results. As stated previously, one solves this equation normally by diagonalization. If we are able to solve this equation exactly (that is numerically exactly) in a large Hilbert space (it will be truncated in terms of the number of single-particle states included in the definition of Slater determinants), it can then serve as a benchmark for other many-body methods which approximate the correlation operator $\hat{C}$.

Example of a Hamiltonian matrix

Suppose, as an example, that we have six fermions below the Fermi level. This means that we can make at most $6p-6h$ excitations. If we have an infinity of single particle states above the Fermi level, we will obviously have an infinity of say $2p-2h$ excitations. Each such way to configure the particles is called a **configuration**. We will always have to truncate in the basis of single-particle states. This gives us a finite number of possible Slater determinants. Our Hamiltonian matrix would then look like (where each block can have a large dimensionalities):

	$0p-0h$	$1p-1h$	$2p-2h$	$3p-3h$	$4p-4h$	$5p-5h$	$6p-6h$
$0p-0h$	x	x	x	0	0	0	0
$1p-1h$	x	x	x	x	0	0	0
$2p-2h$	x	x	x	x	x	0	0
$3p-3h$	0	x	x	x	x	x	0
$4p-4h$	0	0	x	x	x	x	x
$5p-5h$	0	0	0	x	x	x	x
$6p-6h$	0	0	0	0	x	x	x

with a two-body force. Why are there non-zero blocks of elements?

Example of a Hamiltonian matrix with a Hartree-Fock basis

If we use a Hartree-Fock basis, this corresponds to a particular unitary transformation where matrix elements of the type $\langle 0p-0h|\hat{H}|1p-1h\rangle = \langle \Phi_0|\hat{H}|\Phi_i^a\rangle = 0$ and our Hamiltonian matrix becomes

	$0p-0h$	$1p-1h$	$2p-2h$	$3p-3h$	$4p-4h$	$5p-5h$	$6p-6h$
$0p-0h$	$\tilde{x}$	0	$\tilde{x}$	0	0	0	0
$1p-1h$	0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	0	0	0
$2p-2h$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	0	0
$3p-3h$	0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	0
$4p-4h$	0	0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
$5p-5h$	0	0	0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$
$6p-6h$	0	0	0	0	$\tilde{x}$	$\tilde{x}$	$\tilde{x}$

Shell-model jargon

If we do not make any truncations in the possible sets of Slater determinants (many-body states) we can make by distributing A nucleons among n single-particle states, we call such a calculation for **Full configuration interaction theory**

If we make truncations, we have different possibilities

- The standard nuclear shell-model. Here we define an effective Hilbert space with respect to a given core. The calculations are normally then performed for all many-body states that can be constructed from the effective Hilbert spaces. This approach requires a properly defined effective Hamiltonian

- We can truncate in the number of excitations. For example, we can limit the possible Slater determinants to only $1p-1h$ and $2p-2h$ excitations. This is called a configuration interaction calculation at the level of singles and doubles excitations, or just CISD.
- We can limit the number of excitations in terms of the excitation energies. If we do not define a core, this defines normally what is called the no-core shell-model approach.

What happens if we have a three-body interaction and a Hartree-Fock basis?

FCI and the exponential growth

Full configuration interaction theory calculations provide in principle, if we can diagonalize numerically, all states of interest. The dimensionality of the problem explodes however quickly.

The total number of Slater determinants which can be built with say N neutrons distributed among n single particle states is

$$\binom{n}{N} = \frac{n!}{(n-N)!N!}.$$

For a model space which comprises the first for major shells only $0s$, $0p$, $1s0d$ and $1p0f$ we have 40 single particle states for neutrons and protons. For the eight neutrons of oxygen-16 we would then have

$$\binom{40}{8} = \frac{40!}{(32)!8!} \sim 10^9,$$

and multiplying this with the number of proton Slater determinants we end up with approximately with a dimensionality d of $d \sim 10^{18}$.

Exponential wall

This number can be reduced if we look at specific symmetries only. However, the dimensionality explodes quickly!

- For Hamiltonian matrices of dimensionalities which are smaller than $d \sim 10^5$, we would use so-called direct methods for diagonalizing the Hamiltonian matrix
- For larger dimensionalities iterative eigenvalue solvers like Lanczos' method are used. The most efficient codes at present can handle matrices of $d \sim 10^{10}$.

A non-practical way of solving the eigenvalue problem

To see this, we look at the contributions arising from

$$\langle \Phi_H^P| = \langle \Phi_0|$$

in Eq. (3.1), that is we multiply with $\langle \Phi_0|$ from the left in

$$(\hat{H}-E)\sum_{P'H'} C_{H'}^{P'}|\Phi_{H'}^{P'}\rangle = 0.$$

If we assume that we have a two-body operator at most, Slater's rule gives then an equation for the correlation energy in terms of C_i^a and C_{ij}^{ab} only. We get then

$$\langle \Phi_0|\hat{H}-E|\Phi_0\rangle + \sum_{ai}\langle \Phi_0|\hat{H}-E|\Phi_i^a\rangle C_i^a + \sum_{abij}\langle \Phi_0|\hat{H}-E|\Phi_{ij}^{ab}\rangle C_{ij}^{ab} = 0,$$

or

$$E-E_0 = \Delta E = \sum_{ai}\langle \Phi_0|\hat{H}|\Phi_i^a\rangle C_i^a + \sum_{abij}\langle \Phi_0|\hat{H}|\Phi_{ij}^{ab}\rangle C_{ij}^{ab},$$

where the energy E_0 is the reference energy and ΔE defines the so-called correlation energy. The single-particle basis functions could be the results of a Hartree-Fock calculation or just the eigenstates of the non-interacting part of the Hamiltonian.

A non-practical way of solving the eigenvalue problem

To see this, we look at the contributions arising from

$$\langle \Phi_H^P| = \langle \Phi_0|$$

in Eq. (3.1), that is we multiply with $\langle \Phi_0|$ from the left in

$$(\hat{H}-E)\sum_{P'H'} C_{H'}^{P'}|\Phi_{H'}^{P'}\rangle = 0.$$

A non-practical way of solving the eigenvalue problem

If we assume that we have a two-body operator at most, Slater's rule gives then an equation for the correlation energy in terms of C_i^a and C_{ij}^{ab} only. We get then

$$\langle \Phi_0|\hat{H}-E|\Phi_0\rangle + \sum_{ai}\langle \Phi_0|\hat{H}-E|\Phi_i^a\rangle C_i^a + \sum_{abij}\langle \Phi_0|\hat{H}-E|\Phi_{ij}^{ab}\rangle C_{ij}^{ab} = 0,$$

or

$$E-E_0 = \Delta E = \sum_{ai}\langle \Phi_0|\hat{H}|\Phi_i^a\rangle C_i^a + \sum_{abij}\langle \Phi_0|\hat{H}|\Phi_{ij}^{ab}\rangle C_{ij}^{ab},$$

where the energy E_0 is the reference energy and ΔE defines the so-called correlation energy. The single-particle basis functions could be the results of a Hartree-Fock calculation or just the eigenstates of the non-interacting part of the Hamiltonian.

Rewriting the FCI equation

In our notes on Hartree-Fock calculations, we have already computed the matrix $\langle \Phi_0 | \hat{H} | \Phi_i^a \rangle$ and $\langle \Phi_0 | \hat{H} | \Phi_{ij}^{ab} \rangle$. If we are using a Hartree-Fock basis, then the matrix elements $\langle \Phi_0 | \hat{H} | \Phi_i^a \rangle = 0$ and we are left with a *correlation energy* given by

$$E - E_0 = \Delta E^{HF} = \sum_{abij} \langle \Phi_0 | \hat{H} | \Phi_{ij}^{ab} \rangle C_{ij}^{ab}.$$

Rewriting the FCI equation

Inserting the various matrix elements we can rewrite the previous equation as

$$\Delta E = \sum_{ai} \langle i | \hat{f} | a \rangle C_i^a + \sum_{abij} \langle ij | \hat{v} | ab \rangle C_{ij}^{ab}.$$

This equation determines the correlation energy but not the coefficients C.

Rewriting the FCI equation, does not stop here

We need more equations. Our next step is to set up

$$\langle \Phi_i^a | \hat{H} - E | \Phi_0 \rangle + \sum_{bj} \langle \Phi_i^a | \hat{H} - E | \Phi_j^b \rangle C_j^b + \sum_{bcjk} \langle \Phi_i^a | \hat{H} - E | \Phi_{jk}^{bc} \rangle C_{jk}^{bc} + \sum_{bcdjkl} \langle \Phi_i^a | \hat{H} - E | \Phi_{jkl}^{bcd} \rangle C_{jkl}^{bcd} = 0,$$

as this equation will allow us to find an expression for the coefficents C_i^a since we can rewrite this equation as

$$\langle i | \hat{f} | a \rangle + \langle \Phi_i^a | \hat{H} | \Phi_i^a \rangle C_i^a + \sum_{bj \neq ai} \langle \Phi_i^a | \hat{H} | \Phi_j^b \rangle C_j^b + \sum_{bcjk} \langle \Phi_i^a | \hat{H} | \Phi_{jk}^{bc} \rangle C_{jk}^{bc} + \sum_{bcdjkl} \langle \Phi_i^a | \hat{H} | \Phi_{jkl}^{bcd} \rangle C_{jkl}^{bcd} = E C_i^a.$$

Rewriting the FCI equation, please stop here

We see that on the right-hand side we have the energy E. This leads to a non-linear equation in the unknown coefficients. These equations are normally solved iteratively (that is we can start with a guess for the coefficients C_i^a). A common choice is to use perturbation theory for the first guess, setting thereby

$$C_i^a = \frac{\langle i | \hat{f} | a \rangle}{\varepsilon_i - \varepsilon_a}.$$

Rewriting the FCI equation, more to add

The observant reader will however see that we need an equation for C_{jk}^{bc} and C_{jkl}^{bcd} as well. To find equations for these coefficients we need then to continue our multiplications from the left with the various Φ_H^P terms.

For C_{jk}^{bc} we need then

$$\langle\Phi_{ij}^{ab}|\hat{H}-E|\Phi_0\rangle+\sum_{kc}\langle\Phi_{ij}^{ab}|\hat{H}-E|\Phi_k^c\rangle C_k^c+$$

$$\sum_{cdkl}\langle\Phi_{ij}^{ab}|\hat{H}-E|\Phi_{kl}^{cd}\rangle C_{kl}^{cd}+\sum_{cdeklm}\langle\Phi_{ij}^{ab}|\hat{H}-E|\Phi_{klm}^{cde}\rangle C_{klm}^{cde}+\sum_{cdefklmn}\langle\Phi_{ij}^{ab}|\hat{H}-E|\Phi_{klmn}^{cdef}\rangle C_{klmn}^{cdef}=0,$$

and we can isolate the coefficients C_{kl}^{cd} in a similar way as we did for the coefficients C_i^a.

Rewriting the FCI equation, more to add

A standard choice for the first iteration is to set

$$C_{ij}^{ab}=\frac{\langle ij|\hat{v}|ab\rangle}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b}.$$

At the end we can rewrite our solution of the Schroedinger equation in terms of n coupled equations for the coefficients C_H^P. This is a very cumbersome way of solving the equation. However, by using this iterative scheme we can illustrate how we can compute the various terms in the wave operator or correlation operator $\hat{C}$. We will later identify the calculation of the various terms C_H^P as parts of different many-body approximations to full CI. In particular, we can relate this non-linear scheme with Coupled Cluster theory and many-body perturbation theory.

Summarizing FCI and bringing in approximative methods

If we can diagonalize large matrices, FCI is the method of choice since:

- It gives all eigenvalues, ground state and excited states
- The eigenvectors are obtained directly from the coefficients C_H^P which result from the diagonalization
- We can compute easily expectation values of other operators, as well as transition probabilities
- Correlations are easy to understand in terms of contributions to a given operator beyond the Hartree-Fock contribution. This is the standard approach in many-body theory.

Definition of the correlation energy

The correlation energy is defined as, with a two-body Hamiltonian,

$$\Delta E = \sum_{ai} \langle i|\hat{f}|a\rangle C_i^a + \sum_{abij} \langle ij|\hat{v}|ab\rangle C_{ij}^{ab}.$$

The coefficients C result from the solution of the eigenvalue problem. The energy of say the ground state is then

$$E = E_{ref} + \Delta E,$$

where the so-called reference energy is the energy we obtain from a Hartree-Fock calculation, that is

$$E_{ref} = \langle \Phi_0|\hat{H}|\Phi_0\rangle.$$

FCI equation and the coefficients

However, as we have seen, even for a small case like the four first major shells and a nucleus like oxygen-16, the dimensionality becomes quickly intractable. If we wish to include single-particle states that reflect weakly bound systems, we need a much larger single-particle basis. We need thus approximative methods that sum specific correlations to infinite order.

Popular methods are

- Many-body perturbation theory (in essence a Taylor expansion)
- Coupled cluster theory (coupled non-linear equations)
- Green's function approaches (matrix inversion)
- Similarity group transformation methods (coupled ordinary differential equations)

All these methods start normally with a Hartree-Fock basis as the calculational basis.

Important ingredients to have in codes

- Be able to validate and verify the algorithms.
- Include concepts like unit testing. Gives the possibility to test and validate several or all parts of the code.
- Validation and verification are then included *naturally* and one can develop a better attitude to what is meant with an ethically sound scientific approach.

A structured approach to solving problems

In the steps that lead to the development of clean code you should think of

1. How to structure a code in terms of functions (use IDEs or advanced text editors like sublime or atom)
2. How to make a module
3. How to read input data flexibly from the command line or files
4. How to create graphical/web user interfaces
5. How to write unit tests
6. How to refactor code in terms of classes (instead of functions only)
7. How to conduct and automate large-scale numerical experiments
8. How to write scientific reports in various formats ($\LaTeX$, HTML, doconce)

Additional benefits

Many of the above aspetcs will save you a lot of time when you incrementally extend software over time from simpler to more complicated problems. In particular, you will benefit from many good habits:

1. New code is added in a modular fashion to a library (modules)
2. Programs are run through convenient user interfaces
3. It takes one quick command to let all your code undergo heavy testing
4. Tedious manual work with running programs is automated,
5. Your scientific investigations are reproducible, scientific reports with top quality typesetting are produced both for paper and electronic devices. Use version control software like git and repositories like github

Unit Testing

Unit Testing is the practice of testing the smallest testable parts, called units, of an application individually and independently to determine if they behave exactly as expected.

Unit tests (short code fragments) are usually written such that they can be preformed at any time during the development to continually verify the behavior of the code.

In this way, possible bugs will be identified early in the development cycle, making the debugging at later stages much easier.

Unit Testing, benefits

There are many benefits associated with Unit Testing, such as

- It increases confidence in changing and maintaining code. Big changes can be made to the code quickly, since the tests will ensure that everything still is working properly.
- Since the code needs to be modular to make Unit Testing possible, the code will be easier to reuse. This improves the code design.
- Debugging is easier, since when a test fails, only the latest changes need to be debugged.
  - Different parts of a project can be tested without the need to wait for the other parts to be available.
- A unit test can serve as a documentation on the functionality of a unit of the code.

Simple example of unit test

Look up the guide on how to install unit tests for c++ at course webpage. This is the version with classes.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <unittest++/UnitTest++.h>

class MyMultiplyClass public: double multiply(double x, double y) return x * y; ;

TEST(MyMath) MyMultiplyClass my; CHECK$_EQUAL(56, my.multiply(7,8))$;

int main() return UnitTest::RunAllTests();

Simple example of unit test

And without classes

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <unittest++/UnitTest++.h>

double multiply(double x, double y) return x * y;

TEST(MyMath) CHECK$_EQUAL(56, multiply(7,8))$;

int main() return UnitTest::RunAllTests();

For Fortran users, the link at `http://sourceforge.net/projects/fortranxunit/` contains a similar software for unit testing. For Python go to `https://docs.python.org/2/library/unittest.html`.

Unit tests

There are many types of **unit test** libraries. One which is very popular with C++ programmers is Catch

Catch is header only. All you need to do is drop the file(s) somewhere reachable from your project - either in some central location you can set your header search path to find, or directly into your project tree itself!

This is a particularly good option for other Open-Source projects that want to use Catch for their test suite.

Examples

Computing factorials

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ inline unsigned int Factorial(unsigned int number) return number > 1 ? Factorial(number-1)*number : 1;

Factorial Example

Simple test where we put everything in a single file

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ define CATCH$_{CONFIG_MAIN}//ThistellsCatchtoprovideamain()include"catch.hpp"inlineunsignedintFactorial(unsignedintnumber)returnn$

TEST$_{CASE}("Factorialsarecomputed","[factorial]")REQUIRE(Factorial(0)==1);REQUIRE(Factorial(1)==1);REQUIRE(Factoria$

This will compile to a complete executable which responds to command line arguments. If you just run it with no arguments it will execute all test cases (in this case there is just one), report any failures, report a summary of how many tests passed and failed and return the number of failed tests.

What did we do (1)?

All we did was

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ define

one identifier and

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include

one header and we got everything - even an implementation of main() that will respond to command line arguments. Once you have more than one file with unit tests in you'll just need to

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include "catch.hpp"

and go. Usually it's a good idea to have a dedicated implementation file that just has [fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ define CATCH$_{C}ONFIG_{M}AIN include$"$catch.hpp$".

You can also provide your own implementation of main and drive Catch yourself.

What did we do (2)?

We introduce test cases with the

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ TEST$_{C}ASE$

macro.

The test name must be unique. You can run sets of tests by specifying a wildcarded test name or a tag expression. All we did was **define** one identifier and **include** one header and we got everything.

We write our individual test assertions using the

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ REQUIRE

macro.

Unit test summary and testing approach

Three levels of tests

1. Microscopic level: testing small parts of code, use often unit test libraries
2. Mesoscopic level: testing the integration of various parts of your code
3. Macroscopic level: testing that the final result is ok

Coding Recommendations

Writing clean and clear code is an art and reflects your understanding of

1. derivation, verification, and implementation of algorithms
2. what can go wrong with algorithms
3. overview of important, known algorithms
4. how algorithms are used to solve mathematical problems
5. reproducible science and ethics
6. algorithmic thinking for gaining deeper insights about scientific problems

Computing is understanding and your understanding is reflected in your abilities to write clear and clean code.

Summary and recommendations

Some simple hints and tips in order to write clean and clear code

1. Spell out the algorithm and have a top-down approach to the flow of data
2. Start with coding as close as possible to eventual mathematical expressions
3. Use meaningful names for variables
4. Split tasks in simple functions and modules/classes
5. Functions should return as few as possible variables
6. Use unit tests and make sure your codes are producing the correct results
7. Where possible use symbolic coding to autogenerate code and check results
8. Make a proper timing of your algorithms
9. Use version control and make your science reproducible
10. Use IDEs or smart editors with debugging and analysis tools.
11. Automatize your computations interfacing high-level and compiled languages like C++ and Fortran.
12. .....

Building a many-body basis

Here we will discuss how we can set up a single-particle basis which we can use in the various parts of our projects, from the simple pairing model to infinite nuclear matter. We will use here the simple pairing model to illustrate in particular how to set up a single-particle basis. We will also use this do discuss standard FCI approaches like:

1. Standard shell-model basis in one or two major shells
2. Full CI in a given basis and no truncations
3. CISD and CISDT approximations
4. No-core shell model and truncation in excitation energy

Building a many-body basis

An important step in an FCI code is to construct the many-body basis.

While the formalism is independent of the choice of basis, the **effectiveness** of a calculation will certainly be basis dependent.

Furthermore there are common conventions useful to know.

First, the single-particle basis has angular momentum as a good quantum number. You can imagine the single-particle wavefunctions being generated by a one-body Hamiltonian, for example a harmonic oscillator. Modifications include harmonic oscillator plus spin-orbit splitting, or self-consistent mean-field potentials, or the Woods-Saxon potential which mocks up the self-consistent mean-field. For nuclei, the harmonic oscillator, modified by spin-orbit splitting, provides a useful language for describing single-particle states.

Building a many-body basis

Each single-particle state is labeled by the following quantum numbers:

- Orbital angular momentum l
- Intrinsic spin s = 1/2 for protons and neutrons
- Angular momentum $j = l \pm 1/2$
- z-component j_z (or m)
- Some labeling of the radial wavefunction, typically n the number of nodes in the radial wavefunction, but in the case of harmonic oscillator one can also use the principal quantum number N, where the harmonic oscillator energy is $(N+3/2)\omega$.

In this format one labels states by $n(l)_j$, with (l) replaced by a letter: s for $l=0$, p for $l=1$, d for $l=2$, f for $l=3$, and thenceforth alphabetical.

Building a many-body basis

In practice the single-particle space has to be severely truncated. This truncation is typically based upon the single-particle energies, which is the effective energy from a mean-field potential.

Sometimes we freeze the core and only consider a valence space. For example, one may assume a frozen ^{4}He core, with two protons and two neutrons in the $0s_{1/2}$ shell, and then only allow active particles in the $0p_{1/2}$ and $0p_{3/2}$ orbits.

Another example is a frozen ^{16}O core, with eight protons and eight neutrons filling the $0s_{1/2}$, $0p_{1/2}$ and $0p_{3/2}$ orbits, with valence particles in the $0d_{5/2}, 1s_{1/2}$ and $0d_{3/2}$ orbits.

Sometimes we refer to nuclei by the valence space where their last nucleons go. So, for example, we call ^{12}C a p-shell nucleus, while ^{26}Al is an sd-shell nucleus and ^{56}Fe is a pf-shell nucleus.

Building a many-body basis

There are different kinds of truncations.

- For example, one can start with 'filled' orbits (almost always the lowest), and then allow one, two, three... particles excited out of those filled orbits. These are called 1p-1h, 2p-2h, 3p-3h excitations.
- Alternately, one can state a maximal orbit and allow all possible configurations with particles occupying states up to that maximum. This is called *full configuration*.
- Finally, for particular use in nuclear physics, there is the *energy* truncation, also called the $N\Omega$ or N_{max} truncation.

Building a many-body basis

Here one works in a harmonic oscillator basis, with each major oscillator shell assigned a principal quantum number $N = 0, 1, 2, 3, \ldots$. The $N\Omega$ or N_{max} truncation: Any configuration is given an noninteracting energy, which is the sum of the single-particle harmonic oscillator energies. (Thus this ignores spin-orbit splitting.)

Excited state are labeled relative to the lowest configuration by the number of harmonic oscillator quanta.

This truncation is useful because if one includes *all* configuration up to some N_{max}, and has a translationally invariant interaction, then the intrinsic motion and the center-of-mass motion factor. In other words, we can know exactly the center-of-mass wavefunction.

In almost all cases, the many-body Hamiltonian is rotationally invariant. This means it commutes with the operators $\hat{J}^2, \hat{J}_z$ and so eigenstates will have good J, M. Furthermore, the eigenenergies do not depend upon the orientation M.

Therefore we can choose to construct a many-body basis which has fixed M; this is called an M-scheme basis.

Alternately, one can construct a many-body basis which has fixed J, or a J-scheme basis.

Building a many-body basis

The Hamiltonian matrix will have smaller dimensions (a factor of 10 or more) in the J-scheme than in the M-scheme. On the other hand, as we'll show in the next slide, the M-scheme is very easy to construct with Slater determinants, while the J-scheme basis states, and thus the matrix elements, are more complicated, almost always being linear combinations of M-scheme states. J-scheme bases are important and useful, but we'll focus on the simpler M-scheme.

The quantum number m is additive (because the underlying group is Abelian): if a Slater determinant $\hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_k^\dagger \ldots |0\rangle$ is built from single-particle states all with good m, then the total

$$M = m_i + m_j + m_k + \ldots$$

This is *not* true of J, because the angular momentum group SU(2) is not Abelian.

Building a many-body basis

The upshot is that

- It is easy to construct a Slater determinant with good total M;
- It is trivial to calculate M for each Slater determinant;
- So it is easy to construct an M-scheme basis with fixed total M.

Note that the individual M-scheme basis states will *not*, in general, have good total J. Because the Hamiltonian is rotationally invariant, however, the eigenstates will have good J. (The situation is muddied when one has states of different J that are nonetheless degenerate.)

Building a many-body basis

Example: two $j = 1/2$ orbits

Index	n	l	j	m_j
1	0	0	1/2	-1/2
2	0	0	1/2	1/2
3	1	0	1/2	-1/2
4	1	0	1/2	1/2

Note that the order is arbitrary.

Building a many-body basis

There are $\left(\begin{array}{c} 4 \\ 2 \end{array} \right) = 6$ two-particle states, which we list with the total M:

Occupied	M
1,2	0
1,3	-1
1,4	0
2,3	0
2,4	1
3,4	0

There are 4 states with $M = 0$, and 1 each with $M = \pm 1$.

Building a many-body basis

As another example, consider using only single particle states from the $0d_{5/2}$ space. They have the following quantum numbers

Index	n	l	j	m_j
1	0	2	5/2	-5/2
2	0	2	5/2	-3/2
3	0	2	5/2	-1/2
4	0	2	5/2	1/2
5	0	2	5/2	3/2
6	0	2	5/2	5/2

Building a many-body basis

There are $\binom{6}{2} = 15$ two-particle states, which we list with the total M:

Occupied	M	Occupied	M	Occupied	M
1,2	-4	2,3	-2	3,5	1
1,3	-3	2,4	-1	3,6	2
1,4	-2	2,5	0	4,5	2
1,5	-1	2,6	1	4,6	3
1,6	0	3,4	0	5,6	4

There are 3 states with $M = 0$, 2 with $M = 1$, and so on.

Shell-model project

The first step is to construct the M-scheme basis of Slater determinants. Here M-scheme means the total J_z of the many-body states is fixed.

The steps could be:

- Read in a user-supplied file of single-particle states (examples can be given) or just code these internally;
- Ask for the total M of the system and the number of particles N;
- Construct all the N-particle states with given M. You will validate the code by comparing both the number of states and specific states.

Shell-model project

The format of a possible input file could be

Index	n	l	$2j$	$2m_j$
1	1	0	1	-1
2	1	0	1	1
3	0	2	3	-3
4	0	2	3	-1
5	0	2	3	1
6	0	2	3	3
7	0	2	5	-5
8	0	2	5	-3
9	0	2	5	-1
10	0	2	5	1
11	0	2	5	3
12	0	2	5	5

This represents the $1s_{1/2}0d_{3/2}0d_{5/2}$ valence space, or just the sd-space. There are twelve single-particle states, labeled by an overall index, and which have associated quantum numbers the number of radial nodes, the orbital angular momentum l, and the angular momentum j and third component j_z. To keep everything as integers, we could store $2 \times j$ and $2 \times j_z$.

Shell-model project

To read in the single-particle states you need to:

- Open the file
  - Read the number of single-particle states (in the above example, 12); allocate memory; all you need is a single array storing $2 \times j_z$ for each state, labeled by the index.
- Read in the quantum numbers and store $2 \times j_z$ (and anything else you happen to want).

Shell-model project

The next step is to read in the number of particles N and the fixed total M (or, actually, $2 \times M$). For this project we assume only a single species of particles, say neutrons, although this can be relaxed. **Note**: Although it is often a good idea to try to write a more general code, given the short time alloted we would suggest you keep your ambition in check, at least in the initial phases of the project.

You should probably write an error trap to make sure N and M are congruent; if N is even, then $2 \times M$ should be even, and if N is odd then $2 \times M$ should be odd.

Shell-model project

The final step is to generate the set of N-particle Slater determinants with fixed M. The Slater determinants will be stored in occupation representation. Although in many codes this representation is done compactly in bit notation with ones and zeros, but for greater transparency and simplicity we will list the occupied single particle states.

Hence we can store the Slater determinant basis states as $sd(i,j)$, that is an array of dimension N_{SD}, the number of Slater determinants, by N, the number of occupied state. So if for the 7th Slater determinant the 2nd, 3rd, and 9th single-particle states are occupied, then $sd(7,1) = 2$, $sd(7,2) = 3$, and $sd(7,3) = 9$.

Shell-model project

We can construct an occupation representation of Slater determinants by the *odometer* method. Consider $N_{sp} = 12$ and $N = 4$. Start with the first 4 states occupied, that is:

- $sd(1,:) = 1,2,3,4$ (also written as $|1,2,3,4\rangle$)

Now increase the last occupancy recursively:

- $sd(2,:) = 1,2,3,5$
- $sd(3,:) = 1,2,3,6$
- $sd(4,:) = 1,2,3,7$
- $\dots$
- $sd(9,:) = 1,2,3,12$

Then start over with

- $sd(10,:) = 1,2,4,5$

and again increase the rightmost digit

- $sd(11,:) = 1,2,4,6$
- $sd(12,:) = 1,2,4,7$
- $\dots$
- $sd(17,:) = 1,2,4,12$

Shell-model project

When we restrict ourselves to an M-scheme basis, we could choose two paths. The first is simplest (and simplest is often best, at least in the first draft of a code): generate all possible Slater determinants, and then extract from this initial list a list of those Slater determinants with a given M. (You will need to write a short function or routine that computes M for any given occupation.)

Alternately, and not too difficult, is to run the odometer routine twice: each time, as as a Slater determinant is calculated, compute M, but do not store the Slater determinants except the current one. You can then count up the number of Slater determinants with a chosen M. Then allocated storage for the Slater determinants, and run the odometer algorithm again, this time storing Slater determinants with the desired M (this can be done with a simple logical flag).

Shell-model project

Some example solutions: Let's begin with a simple case, the $0d_{5/2}$ space containing six single-particle states

Index	n	l	j	m_j
1	0	2	5/2	-5/2
2	0	2	5/2	-3/2
3	0	2	5/2	-1/2
4	0	2	5/2	1/2
5	0	2	5/2	3/2
6	0	2	5/2	5/2

For two particles, there are a total of 15 states, which we list here with the total M:

- $|1,2\rangle$, $M=-4$, $|1,3\rangle$, $M=-3$
- $|1,4\rangle$, $M=-2$, $|1,5\rangle$, $M=-1$
- $|1,5\rangle$, $M=0$, $vert2,3\rangle$, $M=-2$
- $|2,4\rangle$, $M=-1$, $|2,5\rangle$, $M=0$
- $|2,6\rangle$, $M=1$, $|3,4\rangle$, $M=0$
- $|3,5\rangle$, $M=1$, $|3,6\rangle$, $M=2$
- $|4,5\rangle$, $M=2$, $|4,6\rangle$, $M=3$
- $|5,6\rangle$, $M=4$

Of these, there are only 3 states with $M=0$.

Shell-model project

You should try by hand to show that in this same single-particle space, that for $N=3$ there are 3 states with $M=1/2$ and for $N=4$ there are also only 3 states with $M=0$.

To test your code, confirm the above.

Also, for the sd-space given above, for $N=2$ there are 14 states with $M=0$, for $N=3$ there are 37 states with $M=1/2$, for $N=4$ there are 81 states with $M=0$.

Shell-model project

For our project, we will only consider the pairing model. A simple space is the $(1/2)^2$ space with four single-particle states

Index	n	l	s	m_s
1	0	0	1/2	-1/2
2	0	0	1/2	1/2
3	1	0	1/2	-1/2
4	1	0	1/2	1/2

For $N=2$ there are 4 states with $M=0$; show this by hand and confirm your code reproduces it.

Shell-model project

Another, slightly more challenging space is the $(1/2)^4$ space, that is, with eight single-particle states we have

Index	n	l	s	m_s
1	0	0	1/2	-1/2
2	0	0	1/2	1/2
3	1	0	1/2	-1/2
4	1	0	1/2	1/2
5	2	0	1/2	-1/2
6	2	0	1/2	1/2
7	3	0	1/2	-1/2
8	3	0	1/2	1/2

For $N = 2$ there are 16 states with $M = 0$; for $N = 3$ there are 24 states with $M = 1/2$, and for $N = 4$ there are 36 states with $M = 0$.

Shell-model project

In the shell-model context we can interpret this as 4 $s_{1/2}$ levels, with $m = \pm 1/2$, we can also think of these are simple four pairs, $\pm k, k = 1, 2, 3, 4$. Later on we will assign single-particle energies, depending on the radial quantum number n, that is, $\varepsilon_k = |k|\delta$ so that they are equally spaced.

Shell-model project

For application in the pairing model we can go further and consider only states with no "broken pairs," that is, if $+k$ is filled (or $m = +1/2$, so is $-k$ ($m = -1/2$). If you want, you can write your code to accept only these, and obtain the following six states:

- $|1,2,3,4\rangle$,
- $|1,2,5,6\rangle$,
- $|1,2,7,8\rangle$,
- $|3,4,5,6\rangle$,
- $|3,4,7,8\rangle$,
- $|5,6,7,8\rangle$

Shell-model project

Hints for coding.

- Write small modules (routines/functions) ; avoid big functions that do everything. (But not too small.)
- Use Unit tests! Write lots of error traps, even for things that are ‘obvious.’
- Document as you go along. The Unit tests serve as documentation. For each function write a header that includes:

  1. Main purpose of function and/or unit test
  2. names and brief explanation of input variables, if any
  3. names and brief explanation of output variables, if any
  4. functions called by this function
  5. called by which functions

Shell-model project

Hints for coding

- Unit tests will save time. Use also IDEs for debugging. If you insist on brute force debugging, print out intermediate values. It’s almost impossible to debug a code by looking at it–the code will almost always win a ‘staring contest.’
- Validate code with SIMPLE CASES. Validate early and often. Unit tests!!

The number one mistake is using a too complex a system to test. For example , if you are computing particles in a potential in a box, try removing the potential–you should get particles in a box. And start with one particle, then two, then three... Don’t start with eight particles.

Shell-model project

Our recommended occupation representation, e.g. $|1,2,4,8\rangle$, is easy to code, but numerically inefficient when one has hundreds of millions of Slater determinants.

In state-of-the-art shell-model codes, one generally uses bit representation, i.e. $|1101000100...\rangle$ where one stores the Slater determinant as a single (or a small number of) integer.

This is much more compact, but more intricate to code with considerable more overhead. There exist bit-manipulation functions. We will discuss these in more detail at the beginning of the third week.

Example case: pairing Hamiltonian

We consider a space with 2Ω single-particle states, with each state labeled by $k = 1,2,3,\Omega$ and $m = \pm 1/2$. The convention is that the state with $k > 0$ has $m = +1/2$ while $-k$ has $m = -1/2$.

The Hamiltonian we consider is

$$\hat{H} = -G\hat{P}_+\hat{P}_-,$$

where

$$\hat{P}_+ = \sum_{k>0} \hat{a}^\dagger_k \hat{a}^\dagger_{-k}.$$

and $\hat{P}_- = (\hat{P}_+)^\dagger$.

This problem can be solved using what is called the quasi-spin formalism to obtain the exact results. Thereafter we will try again using the explicit Slater determinant formalism.

Example case: pairing Hamiltonian

One can show (and this is part of the project) that

$$\left[\hat{P}_+, \hat{P}_-\right] = \sum_{k>0} \left(\hat{a}^\dagger_k \hat{a}_k + \hat{a}^\dagger_{-k}\hat{a}_{-k} - 1\right) = \hat{N} - \Omega.$$

Now define

$$\hat{P}_z = \frac{1}{2}(\hat{N} - \Omega).$$

Finally you can show

$$\left[\hat{P}_z, \hat{P}_\pm\right] = \pm\hat{P}_\pm.$$

This means the operators $\hat{P}_\pm, \hat{P}_z$ form a so-called $SU(2)$ algebra, and we can use all our insights about angular momentum, even though there is no actual angular momentum involved.

So we rewrite the Hamiltonian to make this explicit:

$$\hat{H} = -G\hat{P}_+\hat{P}_- = -G\left(\hat{P}^2 - \hat{P}_z^2 + \hat{P}_z\right)$$

Example case: pairing Hamiltonian

Because of the SU(2) algebra, we know that the eigenvalues of $\hat{P}^2$ must be of the form $p(p+1)$, with p either integer or half-integer, and the eigenvalues of $\hat{P}_z$ are m_p with $p \geq |m_p|$, with m_p also integer or half-integer.

But because $\hat{P}_z = (1/2)(\hat{N} - \Omega)$, we know that for N particles the value $m_p = (N - \Omega)/2$. Furthermore, the values of m_p range from $-\Omega/2$ (for $N = 0$) to $+\Omega/2$ (for $N = 2\Omega$, with all states filled).

We deduce the maximal $p = \Omega/2$ and for a given n the values range of p range from $|N - \Omega|/2$ to $\Omega/2$ in steps of 1 (for an even number of particles)

Following Racah we introduce the notation $p = (\Omega - v)/2$ where $v = 0, 2, 4, ..., \Omega - |N - \Omega|$ With this it is easy to deduce that the eigenvalues of the pairing Hamiltonian are

$$-G(N - v)(2\Omega + 2 - N - v)/4$$

This also works for N odd, with $v = 1, 3, 5, \dots$.

Example case: pairing Hamiltonian

Let's take a specific example: $\Omega = 3$ so there are 6 single-particle states, and $N = 3$, with $v = 1, 3$. Therefore there are two distinct eigenvalues,

$$E = -2G, 0$$

Now let's work this out explicitly. The single particle degrees of freedom are defined as

Index	k	m
1	1	-1/2
2	-1	1/2
3	2	-1/2
4	-2	1/2
5	3	-1/2
6	-3	1/2

There are $\binom{6}{3} = 20$ three-particle states, but there are 9 states with $M = +1/2$, namely $|1,2,3\rangle, |1,2,5\rangle, |1,4,6\rangle, |2,3,4\rangle, |2,3,6\rangle, |2,4,5\rangle, |2,5,6\rangle, |3,4,6\rangle, |4,5,6\rangle$.

Example case: pairing Hamiltonian

In this basis, the operator

$$\hat{P}_+ = \hat{a}_1^\dagger \hat{a}_2^\dagger + \hat{a}_3^\dagger \hat{a}_4^\dagger + \hat{a}_5^\dagger \hat{a}_6^\dagger$$

From this we can determine that

$$\hat{P}_- |1,4,6\rangle = \hat{P}_- |2,3,6\rangle = \hat{P}_- |2,4,5\rangle = 0$$

so those states all have eigenvalue 0.

Example case: pairing Hamiltonian

Now for further example,

$$\hat{P}_- |1,2,3\rangle = |3\rangle$$

so

$$\hat{P}_+\hat{P}_-|1,2,3\rangle = |1,2,3\rangle + |3,4,3\rangle + |5,6,3\rangle$$

The second term vanishes because state 3 is occupied twice, and reordering the last term we get

$$\hat{P}_+\hat{P}_-|1,2,3\rangle = |1,2,3\rangle + |3,5,6\rangle$$

without picking up a phase.

Example case: pairing Hamiltonian

Continuing in this fashion, with the previous ordering of the many-body states ($|1,2,3\rangle, |1,2,5\rangle, |1,4,6\rangle, |2,3,4\rangle, |2,3,6\rangle,$
the Hamiltonian matrix of this system is

$$H = -G\begin{pmatrix} 1&0&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&0&1 \end{pmatrix}$$

This is useful for our project. One can by hand confirm that there are 3 eigenvalues $-2G$ and 6 with value zero.

Example case: pairing Hamiltonian

Another example Using the $(1/2)^4$ single-particle space, resulting in eight single-particle states

Index	n	l	s	m_s
1	0	0	1/2	-1/2
2	0	0	1/2	1/2
3	1	0	1/2	-1/2
4	1	0	1/2	1/2
5	2	0	1/2	-1/2
6	2	0	1/2	1/2
7	3	0	1/2	-1/2
8	3	0	1/2	1/2

and then taking only 4-particle, $M = 0$ states that have no ‘broken pairs’, there are six basis Slater determinants:

- $|1,2,3,4\rangle,$
- $|1,2,5,6\rangle,$

- $|1,2,7,8\rangle$,
- $|3,4,5,6\rangle$,
- $|3,4,7,8\rangle$,
- $|5,6,7,8\rangle$

Example case: pairing Hamiltonian

Now we take the following Hamiltonian

$$\hat{H} = \sum_n n\delta \hat{N}_n - G\hat{P}^\dagger \hat{P}$$

where

$$\hat{N}_n = \hat{a}^\dagger_{n,m=+1/2}\hat{a}_{n,m=+1/2} + \hat{a}^\dagger_{n,m=-1/2}\hat{a}_{n,m=-1/2}$$

and

$$\hat{P}^\dagger = \sum_n \hat{a}^\dagger_{n,m=+1/2}\hat{a}^\dagger_{n,m=-1/2}$$

We can write down the 6×6 Hamiltonian in the basis from the prior slide:

$$H = \begin{pmatrix} 2\delta - 2G & -G & -G & -G & -G & 0 \\ -G & 4\delta - 2G & -G & -G & -0 & -G \\ -G & -G & 6\delta - 2G & 0 & -G & -G \\ -G & -G & 0 & 6\delta - 2G & -G & -G \\ -G & 0 & -G & -G & 8\delta - 2G & -G \\ 0 & -G & -G & -G & -G & 10\delta - 2G \end{pmatrix}$$

(You should check by hand that this is correct.)

For $\delta = 0$ we have the closed form solution of the g.s. energy given by $-6G$.

Building a Hamiltonian matrix

The goal is to compute the matrix elements of the Hamiltonian, specifically matrix elements between many-body states (Slater determinants) of two-body operators

$$\sum_{p<q,r<s} V_{pqr}\hat{a}^\dagger_p\hat{a}^\dagger_q\hat{a}_s\hat{a}_r$$

In particular we will need to compute

$$\langle\beta|\hat{a}^\dagger_p\hat{a}^\dagger_q\hat{a}_s\hat{a}_r|\alpha\rangle$$

where α,β are indices labeling Slater determinants and p,q,r,s label single-particle states.

Building a Hamiltonian matrix

Note: there are other, more efficient ways to do this than the method we describe, but you will be able to produce a working code quickly.

As we coded in the first step, a Slater determinant $|\alpha\rangle$ with index α is a list of N occupied single-particle states $i_1 < i_2 < i_3 \dots i_N$.

Furthermore, for the two-body matrix elements V_{pqrs} we normally assume $p < q$ and $r < s$. For our specific project, the interaction is much simpler and you can use this to simplify considerably the setup of a shell-model code for project 2.

What follows here is a more general, but still brute force, approach.

Building a Hamiltonian matrix

Write a function that:

1. Has as input the single-particle indices p, q, r, s for the two-body operator and the index α for the ket Slater determinant;
2. Returns the index β of the unique (if any) Slater determinant such that

$$|\beta\rangle = \pm \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r |\alpha\rangle$$

as well as the phase

This is equivalent to computing

$$\langle\beta| \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r |\alpha\rangle$$

Building a Hamiltonian matrix, first step

The first step can take as input an initial Slater determinant (whose position in the list of basis Slater determinants is α) written as an ordered listed of occupied single-particle states, e.g. $1, 2, 5, 8$, and the indices p, q, r, s from the two-body operator.

It will return another final Slater determinant if the single-particle states r and s are occupied, else it will return an empty Slater determinant (all zeroes).

If r and s are in the list of occupied single particle states, then replace the initial single-particle states ij as $i \rightarrow r$ and $j \rightarrow r$.

Building a Hamiltonian matrix, second step

The second step will take the final Slater determinant from the first step (if not empty), and then order by pairwise permutations (i.e., if the Slater determinant is $i_1, i_2, i_3, \ldots$, then if $i_n > i_{n+1}$, interchange $i_n \leftrightarrow i_{n+1}$.

Building a Hamiltonian matrix

It will also output a phase. If any two single-particle occupancies are repeated, the phase is 0. Otherwise it is +1 for an even permutation and -1 for an odd permutation to bring the final Slater determinant into ascending order, $j_1 < j_2 < j_3 \ldots$.

Building a Hamiltonian matrix

Example: Suppose in the sd single-particle space that the initial Slater determinant is $1,3,9,12$. If $p,q,r,s = 2,8,1,12$, then after the first step the final Slater determinant is $2,3,9,8$. The second step will return $2,3,8,9$ and a phase of -1, because an odd number of interchanges is required.

Building a Hamiltonian matrix

Example: Suppose in the sd single-particle space that the initial Slater determinant is $1,3,9,12$. If $p,q,r,s = 3,8,1,12$, then after the first step the final Slater determinant is $3,3,9,8$, but after the second step the phase is 0 because the single-particle state 3 is occupied twice.

Lastly, the final step takes the ordered final Slater determinant and we search through the basis list to determine its index in the many-body basis, that is, β.

Building a Hamiltonian matrix

The Hamiltonian is then stored as an $N_{SD} \times N_{SD}$ array of real numbers, which can be allocated once you have created the many-body basis and know N_{SD}.

Building a Hamiltonian matrix

1. Initialize $H(\alpha,\beta) = 0.0$
2. Set up an outer loop over β
3. Loop over $\alpha = 1, NSD$
4. For each α, loop over $a = 1, ntbme$ and fetch $V(a)$ and the single-particle indices p,q,r,s
5. If $V(a) = 0$ skip. Otherwise, apply $\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r$ to the Slater determinant labeled by α.
6. Find, if any, the label β of the resulting Slater determinant and the phase (which is 0, +1, -1).
7. If phase $\neq 0$, then update $H(\alpha,\beta)$ as $H(\alpha,\beta) + phase * V(a)$. The sum is important because multiple operators might contribute to the same matrix element.
8. Continue loop over a
9. Continue loop over α.
10. End the outer loop over β.

You should force the resulting matrix H to be symmetric. To do this, when updating $H(\alpha,\beta)$, if $\alpha \neq \beta$, also update $H(\beta,\alpha)$.

Building a Hamiltonian matrix

You will also need to include the single-particle energies. This is easy: they only contribute to diagonal matrix elements, that is, $H(\alpha,\alpha)$. Simply find the occupied single-particle states i and add the corresponding $\varepsilon(i)$.

Hamiltonian matrix without the bit representation

Consider the many-body state Ψ_λ expressed as linear combinations of Slater determinants (SD) of orthonormal single-particle states $\phi(\mathbf{r})$:

$$\Psi_\lambda = \sum_i C_{\lambda i} SD_i \tag{3.2}$$

Using the Slater-Condon rules the matrix elements of any one-body ($\mathscr{O}_\infty$) or two-body ($\mathscr{O}_\in$) operator expressed in the determinant space have simple expressions involving one- and two-fermion integrals in our given single-particle basis. The diagonal elements are given by:

$$\langle SD|\mathscr{O}_\infty|\mathscr{S}\mathscr{D}\rangle = \sum_{i\in SD} \langle \phi_i|\mathscr{O}_\infty|\phi_\rangle\rangle \tag{3.3}$$

$$\langle SD|\mathscr{O}_\in|\mathscr{S}\mathscr{D}\rangle = \frac{1}{2} \sum_{(i,j)\in SD} \langle \phi_i\phi_j|\mathscr{O}_\in|\phi_\rangle\phi_|\rangle - \langle \phi_i\phi_j|\mathscr{O}_\in|\phi_|\phi_\rangle\rangle$$

Hamiltonian matrix without the bit representation, one and two-body operators

For two determinants which differ only by the substitution of single-particle states i with a single-particle state j:

$$
\begin{aligned}
\langle SD|\mathscr{O}_\infty|\mathscr{S}\mathscr{D}^{|}_{\rangle}\rangle &= \langle \phi_i|\mathscr{O}_\infty|\phi_|\rangle \\
\langle SD|\mathscr{O}_\in|\mathscr{S}\mathscr{D}^{|}_{\rangle}\rangle &= \sum_{k\in SD} \langle \phi_i\phi_k|\mathscr{O}_\in|\phi_|\phi_\|\rangle - \langle \phi_\rangle\phi_\||\mathscr{O}_\in|\phi_\|\phi_|\rangle
\end{aligned} \tag{3.4}
$$

For two determinants which differ by two single-particle states

$$
\begin{aligned}
\langle SD|\mathscr{O}_\infty|\mathscr{S}\mathscr{D}^{|\updownarrow}_{\rangle\|}\rangle &= 0 \\
\langle SD|\mathscr{O}_\in|\mathscr{S}\mathscr{D}^{|\updownarrow}_{\rangle\|}\rangle &= \langle \phi_i\phi_k|\mathscr{O}_\in|\phi_|\phi_\updownarrow\rangle - \langle \phi_\rangle\phi_\||\mathscr{O}_\in|\phi_\updownarrow\phi_|\rangle
\end{aligned} \tag{3.5}
$$

All other matrix elements involving determinants with more than two substitutions are zero.

Strategies for setting up an algorithm

An efficient implementation of these rules requires

- to find the number of single-particle state substitutions between two determinants
- to find which single-particle states are involved in the substitution
- to compute the phase factor if a reordering of the single-particle states has occured

We can solve this problem using our odometric approach or alternatively using a bit representation as discussed below and in more detail in

- Scemama and Gimer's article (Fortran codes)
- Simen Kvaal's article on how to build an FCI code (C++ code)

We recommend in particular the article by Simen Kvaal. It contains nice general classes for creation and annihilation operators as well as the calculation of the phase (see below).

Computing expectation values and transitions in the shell-model

When we diagonalize the Hamiltonian matrix, the eigenvectors are the coefficients $C_{\lambda i}$ used to express the many-body state Ψ_λ in terms of a linear combinations of Slater determinants (SD) of orthonormal single-particle states $\phi(\mathbf{r})$.

With these eigenvectors we can compute say the transition likelyhood of a one-body operator as

$$\langle \Psi_\lambda|\mathscr{O}_\infty|\ominus_\sigma\rangle = \sum_{\rangle|} \mathscr{C}^*_{\lambda\rangle}\mathscr{C}_{\sigma|}\langle \mathscr{S}\mathscr{D}_\rangle|\mathscr{O}_\infty|\mathscr{S}\mathscr{D}_|\rangle.$$

Writing the one-body operator in second quantization as

$$\mathscr{O}_\infty = \sum_{\surd\blacksquare} \langle_{\surd} |\wr_\infty| \blacksquare\rangle \dashv^\dagger_\surd \dashv_\blacksquare,$$

we have

$$\langle \Psi_\lambda | \mathscr{O}_\infty | \ominus_\sigma \rangle = \sum_{\surd\blacksquare} \langle_{\surd} |\wr_\infty| \blacksquare\rangle \sum_{\rangle|} \mathscr{C}^*_{\lambda\rangle} \mathscr{C}_{\sigma|} \langle \mathscr{S}\mathscr{D}_\rangle | \dashv^\dagger_\surd \dashv_\blacksquare | \mathscr{S}\mathscr{D}_| \rangle.$$

Computing expectation values and transitions in the shell-model and spectroscopic factors

The terms we need to evalute then are just the elements

$$\langle SD_i | a^\dagger_p a_q | SD_j \rangle,$$

which can be rewritten in terms of spectroscopic factors by inserting a complete set of Slater determinats as

$$\langle SD_i | a^\dagger_p a_q | SD_j \rangle = \sum_l \langle SD_i | a^\dagger_p | SD_l \rangle \langle SD_l | a_q | SD_j \rangle,$$

where $\langle SD_l | a_q (a^\dagger_p) | SD_j \rangle$ are the spectroscopic factors. These can be easily evaluated in m-scheme. Using the Wigner-Eckart theorem we can transform these to a J-coupled scheme through so-called reduced matrix elements.

Operators in second quantization

In the build-up of a shell-model or FCI code that is meant to tackle large dimensionalities we need to deal with the action of the Hamiltonian $\hat{H}$ on a Slater determinant represented in second quantization as

$$|\alpha_1 \dots \alpha_n\rangle = a^\dagger_{\alpha_1} a^\dagger_{\alpha_2} \dots a^\dagger_{\alpha_n} |0\rangle.$$

The time consuming part stems from the action of the Hamiltonian on the above determinant,

$$\left(\sum_{\alpha\beta} \langle \alpha | t + u | \beta \rangle a^\dagger_\alpha a_\beta + \frac{1}{4} \sum_{\alpha\beta\gamma\delta} \langle \alpha\beta | \hat{v} | \gamma\delta \rangle a^\dagger_\alpha a^\dagger_\beta a_\delta a_\gamma \right) a^\dagger_{\alpha_1} a^\dagger_{\alpha_2} \dots a^\dagger_{\alpha_n} |0\rangle.$$

A practically useful way to implement this action is to encode a Slater determinant as a bit pattern.

Operators in second quantization

Assume that we have at our disposal n different single-particle states $\alpha_0, \alpha_2, \dots, \alpha_{n-1}$ and that we can distribute among these states $N \le n$ particles.

A Slater determinant can then be coded as an integer of n bits. As an example, if we have $n = 16$ single-particle states $\alpha_0, \alpha_1, \dots, \alpha_{15}$ and $N = 4$ fermions occupying the states α_3, α_6, α_{10} and α_{13} we could write this Slater determinant as

$$\Phi_\Lambda = a^\dagger_{\alpha_3} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle.$$

The unoccupied single-particle states have bit value 0 while the occupied ones are represented by bit state 1. In the binary notation we would write this 16 bits long integer as

$$\begin{array}{cccccccccccccccc} \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \alpha_5 & \alpha_6 & \alpha_7 & \alpha_8 & \alpha_9 & \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} & \alpha_{15} \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$$

which translates into the decimal number

$$2^3 + 2^6 + 2^{10} + 2^{13} = 9288.$$

We can thus encode a Slater determinant as a bit pattern.

Operators in second quantization

With N particles that can be distributed over n single-particle states, the total number of Slater determinats (and defining thereby the dimensionality of the system) is

$$\dim(\mathcal{H}) = \binom{n}{N}.$$

The total number of bit patterns is 2^n.

Operators in second quantization

We assume again that we have at our disposal n different single-particle orbits $\alpha_0, \alpha_2, \dots, \alpha_{n-1}$ and that we can distribute among these orbits $N \le n$ particles. The ordering among these states is important as it defines the order of the creation operators. We will write the determinant

$$\Phi_\Lambda = a^\dagger_{\alpha_3} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle,$$

in a more compact way as

$$\Phi_{3,6,10,13} = |0001001000100100\rangle.$$

The action of a creation operator is thus

$$a^\dagger_{\alpha_4}\Phi_{3,6,10,13} = a^\dagger_{\alpha_4}|0001001000100100\rangle = a^\dagger_{\alpha_4}a^\dagger_{\alpha_3}a^\dagger_{\alpha_6}a^\dagger_{\alpha_{10}}a^\dagger_{\alpha_{13}}|0\rangle,$$

which becomes

$$-a^\dagger_{\alpha_3}a^\dagger_{\alpha_4}a^\dagger_{\alpha_6}a^\dagger_{\alpha_{10}}a^\dagger_{\alpha_{13}}|0\rangle = -|0001101000100100\rangle.$$

Operators in second quantization

Similarly

$$a^\dagger_{\alpha_6}\Phi_{3,6,10,13} = a^\dagger_{\alpha_6}|0001001000100100\rangle = a^\dagger_{\alpha_6}a^\dagger_{\alpha_3}a^\dagger_{\alpha_6}a^\dagger_{\alpha_{10}}a^\dagger_{\alpha_{13}}|0\rangle,$$

which becomes

$$-a^\dagger_{\alpha_4}(a^\dagger_{\alpha_6})^2a^\dagger_{\alpha_{10}}a^\dagger_{\alpha_{13}}|0\rangle = 0!$$

This gives a simple recipe:

- If one of the bits b_j is 1 and we act with a creation operator on this bit, we return a null vector
- If $b_j = 0$, we set it to 1 and return a sign factor $(-1)^l$, where l is the number of bits set before bit j.

Operators in second quantization

Consider the action of $a^\dagger_{\alpha_2}$ on various slater determinants:

$$\begin{aligned}
a^\dagger_{\alpha_2}\Phi_{00111} &= a^\dagger_{\alpha_2}|00111\rangle \quad = 0\times|00111\rangle\\
a^\dagger_{\alpha_2}\Phi_{01011} &= a^\dagger_{\alpha_2}|01011\rangle = (-1)\times|01111\rangle\\
a^\dagger_{\alpha_2}\Phi_{01101} &= a^\dagger_{\alpha_2}|01101\rangle \quad = 0\times|01101\rangle\\
a^\dagger_{\alpha_2}\Phi_{01110} &= a^\dagger_{\alpha_2}|01110\rangle \quad = 0\times|01110\rangle\\
a^\dagger_{\alpha_2}\Phi_{10011} &= a^\dagger_{\alpha_2}|10011\rangle = (-1)\times|10111\rangle\\
a^\dagger_{\alpha_2}\Phi_{10101} &= a^\dagger_{\alpha_2}|10101\rangle \quad = 0\times|10101\rangle\\
a^\dagger_{\alpha_2}\Phi_{10110} &= a^\dagger_{\alpha_2}|10110\rangle \quad = 0\times|10110\rangle\\
a^\dagger_{\alpha_2}\Phi_{11001} &= a^\dagger_{\alpha_2}|11001\rangle = (+1)\times|11101\rangle\\
a^\dagger_{\alpha_2}\Phi_{11010} &= a^\dagger_{\alpha_2}|11010\rangle = (+1)\times|11110\rangle
\end{aligned}$$

What is the simplest way to obtain the phase when we act with one annihilation(creation) operator on the given Slater determinant representation?

Operators in second quantization

We have an SD representation

$$\Phi_\Lambda = a^\dagger_{\alpha_0} a^\dagger_{\alpha_3} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle,$$

in a more compact way as

$$\Phi_{0,3,6,10,13} = |1001001000100100\rangle.$$

The action of

$$a^\dagger_{\alpha_4} a_{\alpha_0} \Phi_{0,3,6,10,13} = a^\dagger_{\alpha_4} |0001001000100100\rangle = a^\dagger_{\alpha_4} a^\dagger_{\alpha_3} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle,$$

which becomes

$$-a^\dagger_{\alpha_3} a^\dagger_{\alpha_4} a^\dagger_{\alpha_6} a^\dagger_{\alpha_{10}} a^\dagger_{\alpha_{13}} |0\rangle = -|0001101000100100\rangle.$$

Operators in second quantization

The action

$$a_{\alpha_0} \Phi_{0,3,6,10,13} = |0001001000100100\rangle,$$

can be obtained by subtracting the logical sum (AND operation) of $\Phi_{0,3,6,10,13}$ and a word which represents only α_0, that is

$$|1000000000000000\rangle,$$

from $\Phi_{0,3,6,10,13} = |1001001000100100\rangle$.

This operation gives $|0001001000100100\rangle$.

Similarly, we can form $a^\dagger_{\alpha_4} a_{\alpha_0} \Phi_{0,3,6,10,13}$, say, by adding $|0000100000000000\rangle$ to $a_{\alpha_0} \Phi_{0,3,6,10,13}$, first checking that their logical sum is zero in order to make sure that the state α_4 is not already occupied.

Operators in second quantization

It is trickier however to get the phase $(-1)^l$. One possibility is as follows

- Let S_1 be a word that represents the 1-bit to be removed and all others set to zero.

In the previous example $S_1 = |1000000000000000\rangle$

- Define S_2 as the similar word that represents the bit to be added, that is in our case

$S_2 = |0000100000000000\rangle$.

- Compute then $S = S_1 - S_2$, which here becomes

$$S = |0111000000000000\rangle$$

- Perform then the logical AND operation of S with the word containing

$$\Phi_{0,3,6,10,13} = |1001001000100100\rangle,$$

which results in $|0001000000000000\rangle$. Counting the number of 1-bits gives the phase. Here you need however an algorithm for bitcounting.

Bit counting

We include here a python program which may aid in this direction. It uses bit manipulation functions from `http://wiki.python.org/moin/BitManipulation`.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python import math

""" A simple Python class for Slater determinant manipulation Bit-manipulation stolen from:

http://wiki.python.org/moin/BitManipulation """

bitCount() counts the number of bits set (not an optimal function)

def bitCount(int$_{t}ype):""""Countbitssetininteger""""count=0while(int_{t}ype):int_{t}ype=int_{t}ype-1count+=1return(count)$

testBit() returns a nonzero result, 2**offset, if the bit at 'offset' is one.

def testBit(int$_{t}ype,offset):mask=1<<offsetreturn(int_{t}ypemask)>>offset$

setBit() returns an integer with the bit at 'offset' set to 1.

def setBit(int$_{t}ype,offset):mask=1<<offsetreturn(int_{t}ype|mask)$

clearBit() returns an integer with the bit at 'offset' cleared.

def clearBit(int$_{t}ype,offset):mask=\ (1<<offset)return(int_{t}ypemask)$

toggleBit() returns an integer with the bit at 'offset' inverted, 0 -> 1 and 1 -> 0.

def toggleBit(int$_{t}ype,offset):mask=1<<offsetreturn(int_{t}ype^{m}ask)$

binary string made from number

def bin0(s): return str(s) if s<=1 else bin0(s»1) + str(s1)

def bin(s, L = 0): ss = bin0(s) if L > 0: return '0'*(L-len(ss)) + ss else: return ss

class Slater: """ Class for Slater determinants """ def $_{i}nit_{(self):self.word=int(0)}$

def create(self, j): print "c$^{+}_{"}+str(j)+"|"+bin(self.word)+">=",Assumebitjisset,thenwereturnzero.s=0Checkifbitjisset.isset=testBit(self.word,j)ifisset==0:bits=bitCount(self.word((1<<j)-1))s=pow(-1,bits)self.word=setBit(self.word,j)$

print str(s) + " x |" + bin(self.word) + ">" return s

def annihilate(self, j): print "c$_{"}+str(j)+"|"+bin(self.word)+">=",Assumebitjisnotset,thenwereturnzero.s=0Checkifbitjisset.isset=testBit(self.word,j)ifisset==1:bits=bitCount(self.word((1<<j)-1))s=pow(-1,bits)self.word=clearBit(self.word,j)$

print str(s) + " x |" + bin(self.word) + ">" return s

Do some testing:

phi = Slater() phi.create(0) phi.create(1) phi.create(2) phi.create(3)

print

s = phi.annihilate(2) s = phi.create(7) s = phi.annihilate(0) s = phi.create(200)

Eigenvalue problems, basic definitions

Let us consider the matrix $\mathbf{A}$ of dimension n. The eigenvalues of $\mathbf{A}$ are defined through the matrix equation

$$\mathbf{A}\mathbf{x}^{(\nu)} = \lambda^{(\nu)}\mathbf{x}^{(\nu)},$$

where $\lambda^{(\nu)}$ are the eigenvalues and $\mathbf{x}^{(\nu)}$ the corresponding eigenvectors. Unless otherwise stated, when we use the wording eigenvector we mean the right eigenvector. The left eigenvalue problem is defined as

$$\mathbf{x}_L^{(\nu)}\mathbf{A} = \lambda^{(\nu)}\mathbf{x}_L^{(\nu)}$$

The above right eigenvector problem is equivalent to a set of n equations with n unknowns x_i.

Eigenvalue problems, basic definitions

The eigenvalue problem can be rewritten as

$$\left(\mathbf{A} - \lambda^{(\nu)}\mathbf{I}\right)\mathbf{x}^{(\nu)} = 0,$$

with $\mathbf{I}$ being the unity matrix. This equation provides a solution to the problem if and only if the determinant is zero, namely

$$\left|\mathbf{A} - \lambda^{(\nu)}\mathbf{I}\right| = 0,$$

which in turn means that the determinant is a polynomial of degree n in λ and in general we will have n distinct zeros.

Eigenvalue problems, basic definitions

The eigenvalues of a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ are thus the n roots of its characteristic polynomial

$$P(\lambda) = det(\lambda\mathbf{I} - \mathbf{A}),$$

or

$$P(\lambda) = \prod_{i=1}^{n}(\lambda_i - \lambda).$$

The set of these roots is called the spectrum and is denoted as $\lambda(\mathbf{A})$. If $\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ then we have

$$det(\mathbf{A}) = \lambda_1\lambda_2\dots\lambda_n,$$

and if we define the trace of $\mathbf{A}$ as

$$Tr(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

then

$$Tr(\mathbf{A}) = \lambda_1 + \lambda_2 + \cdots + \lambda_n.$$

Abel-Ruffini Impossibility Theorem

The *Abel-Ruffini* theorem (also known as Abel's impossibility theorem) states that there is no general solution in radicals to polynomial equations of degree five or higher.

The content of this theorem is frequently misunderstood. It does not assert that higher-degree polynomial equations are unsolvable. In fact, if the polynomial has real or complex coefficients, and we allow complex solutions, then every polynomial equation has solutions; this is the fundamental theorem of algebra. Although these solutions cannot always be computed exactly with radicals, they can be computed to any desired degree of accuracy using numerical methods such as the Newton-Raphson method or Laguerre method, and in this way they are no different from solutions to polynomial equations of the second, third, or fourth degrees.

The theorem only concerns the form that such a solution must take. The content of the theorem is that the solution of a higher-degree equation cannot in all cases be expressed in terms of the polynomial coefficients with a finite number of operations of addition, subtraction, multiplication, division and root extraction. Some polynomials of arbitrary degree, of which the simplest nontrivial example is the monomial equation $ax^n = b$, are always solvable with a radical.

Abel-Ruffini Impossibility Theorem

The *Abel-Ruffini* theorem says that there are some fifth-degree equations whose solution cannot be so expressed. The equation $x^5 - x + 1 = 0$ is an example. Some other fifth degree equations can be solved by radicals, for example $x^5 - x^4 - x + 1 = 0$. The precise criterion that distinguishes between those equations that can be solved by radicals and those that cannot was given by Galois and is now part of Galois theory: a polynomial equation can be solved by radicals if and only if its Galois group is a solvable group.

Today, in the modern algebraic context, we say that second, third and fourth degree polynomial equations can always be solved by radicals because the symmetric groups S_2, S_3 and S_4 are solvable groups, whereas S_n is not solvable for $n \geq 5$.

Eigenvalue problems, basic definitions

In the present discussion we assume that our matrix is real and symmetric, that is $\mathbf{A} \in \mathbb{R}^{n\times n}$. The matrix $\mathbf{A}$ has n eigenvalues $\lambda_1 \dots \lambda_n$ (distinct or not). Let $\mathbf{D}$ be the diagonal matrix with the eigenvalues on the diagonal

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n \end{pmatrix}.$$

If $\mathbf{A}$ is real and symmetric then there exists a real orthogonal matrix $\mathbf{S}$ such that

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n),$$

and for $j = 1 : n$ we have $\mathbf{AS}(:, j) = \lambda_j \mathbf{S}(:, j)$.

Eigenvalue problems, basic definitions

To obtain the eigenvalues of $\mathbf{A} \in \mathbb{R}^{n\times n}$, the strategy is to perform a series of similarity transformations on the original matrix $\mathbf{A}$, in order to reduce it either into a diagonal form as above or into a tridiagonal form.

We say that a matrix $\mathbf{B}$ is a similarity transform of $\mathbf{A}$ if

$$\mathbf{B} = \mathbf{S}^T\mathbf{AS}, \qquad \text{where} \qquad \mathbf{S}^T\mathbf{S} = \mathbf{S}^{-1}\mathbf{S} = \mathbf{I}.$$

The importance of a similarity transformation lies in the fact that the resulting matrix has the same eigenvalues, but the eigenvectors are in general different.

Eigenvalue problems, basic definitions

To prove this we start with the eigenvalue problem and a similarity transformed matrix $\mathbf{B}$.

$$\mathbf{Ax} = \lambda\mathbf{x} \qquad \text{and} \qquad \mathbf{B} = \mathbf{S}^T\mathbf{AS}.$$

We multiply the first equation on the left by $\mathbf{S}^T$ and insert $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between $\mathbf{A}$ and $\mathbf{x}$. Then we get

$$(\mathbf{S}^T\mathbf{AS})(\mathbf{S}^T\mathbf{x}) = \lambda\mathbf{S}^T\mathbf{x}, \tag{3.6}$$

which is the same as

$$\mathbf{B}\left(\mathbf{S}^T\mathbf{x}\right) = \lambda\left(\mathbf{S}^T\mathbf{x}\right).$$

The variable λ is an eigenvalue of $\mathbf{B}$ as well, but with eigenvector $\mathbf{S}^T\mathbf{x}$.

Eigenvalue problems, basic definitions

The basic philosophy is to

- Either apply subsequent similarity transformations (direct method) so that

$$\mathbf{S}_N^T \dots \mathbf{S}_1^T\mathbf{AS}_1 \dots \mathbf{S}_N = \mathbf{D}, \tag{3.7}$$

- Or apply subsequent similarity transformations so that $\mathbf{A}$ becomes tridiagonal (Householder) or upper/lower triangular (the *QR* method to be discussed later).
- Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.
- Or use so-called power methods
- Or use iterative methods (Krylov, Lanczos, Arnoldi). These methods are popular for huge matrix problems.

Discussion of methods for eigenvalues

The general overview.

One speaks normally of two main approaches to solving the eigenvalue problem.

- The first is the formal method, involving determinants and the characteristic polynomial. This proves how many eigenvalues there are, and is the way most of you learned about how to solve the eigenvalue problem, but for matrices of dimensions greater than 2 or 3, it is rather impractical.
- The other general approach is to use similarity or unitary tranformations to reduce a matrix to diagonal form. This is normally done in two steps: first reduce to for example a *tridiagonal* form, and then to diagonal form. The main algorithms we will discuss in detail, Jacobi's and Householder's (so-called direct method) and Lanczos algorithms (an iterative method), follow this methodology.

Eigenvalues methods

Direct or non-iterative methods require for matrices of dimensionality $n \times n$ typically $O(n^3)$ operations. These methods are normally called standard methods and are used for dimensionalities $n \sim 10^5$ or smaller. A brief historical overview

Year	n	
1950	$n = 20$	(Wilkinson)
1965	$n = 200$	(Forsythe et al.)
1980	$n = 2000$	Linpack
1995	$n = 20000$	Lapack
This decade	$n \sim 10^5$	Lapack

shows that in the course of 60 years the dimension that direct diagonalization methods can handle has increased by almost a factor of 10^4 (note this is for serial versions). However, it pales beside the progress achieved by computer hardware, from flops to petaflops, a factor of almost 10^{15}. We see clearly played out in history the $O(n^3)$ bottleneck of direct matrix algorithms.

Sloppily speaking, when $n \sim 10^4$ is cubed we have $O(10^{12})$ operations, which is smaller than the 10^{15} increase in flops.

Discussion of methods for eigenvalues

If the matrix to diagonalize is large and sparse, direct methods simply become impractical, also because many of the direct methods tend to destroy sparsity. As a result large dense matrices may arise during the diagonalization procedure. The idea behind iterative methods is to project the $n-$dimensional problem in smaller spaces, so-called Krylov subspaces. Given a matrix $\mathbf{A}$ and a vector $\mathbf{v}$, the associated Krylov sequences of vectors (and thereby subspaces) $\mathbf{v}$, $\mathbf{Av}$, $\mathbf{A}^2\mathbf{v}$, $\mathbf{A}^3\mathbf{v}, \dots$, represent successively larger Krylov subspaces.

Matrix	$\mathbf{Ax}=\mathbf{b}$	$\mathbf{Ax}=\lambda\mathbf{x}$
$\mathbf{A}=\mathbf{A}^*$	Conjugate gradient	Lanczos
$\mathbf{A}\neq\mathbf{A}^*$	GMRES etc	Arnoldi

Eigenvalues and Lanczos' method

Basic features with a real symmetric matrix (and normally huge $n>10^6$ and sparse) $\hat{A}$ of dimension $n\times n$:

- Lanczos' algorithm generates a sequence of real tridiagonal matrices T_k of dimension $k\times k$ with $k\leq n$, with the property that the extremal eigenvalues of T_k are progressively better estimates of $\hat{A}$' extremal eigenvalues.* The method converges to the extremal eigenvalues.
- The similarity transformation is

$$\hat{T}=\hat{Q}^T\hat{A}\hat{Q},$$

with the first vector $\hat{Q}\hat{e}_1=\hat{q}_1$.

We are going to solve iteratively

$$\hat{T}=\hat{Q}^T\hat{A}\hat{Q},$$

with the first vector $\hat{Q}\hat{e}_1=\hat{q}_1$. We can write out the matrix $\hat{Q}$ in terms of its column vectors

$$\hat{Q}=[\hat{q}_1\hat{q}_2\dots\hat{q}_n].$$

Eigenvalues and Lanczos' method, tridiagonal matrix

The matrix

$$\hat{T}=\hat{Q}^T\hat{A}\hat{Q},$$

can be written as

$$\hat{T}=\begin{pmatrix} \alpha_1 & \beta_1 & 0 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 \\ \dots & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & \dots & \dots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

Eigenvalues and Lanczos' method, tridiagonal and orthogonal matrices

Using the fact that

$$\hat{Q}\hat{Q}^T=\hat{I},$$

we can rewrite

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

as

$$\hat{Q}\hat{T} = \hat{A}\hat{Q}.$$

Eigenvalues and Lanczos' method

If we equate columns

$$\hat{T} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 \\ \dots & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & \dots & \dots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

we obtain

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k \hat{q}_k + \beta_k \hat{q}_{k+1}.$$

Eigenvalues and Lanczos' method, defining the Lanczos' vectors

We have thus

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k \hat{q}_k + \beta_k \hat{q}_{k+1},$$

with $\beta_0\hat{q}_0 = 0$ for $k = 1 : n-1$. Remember that the vectors $\hat{q}_k$ are orthornormal and this implies

$$\alpha_k = \hat{q}_k^T \hat{A} \hat{q}_k,$$

and these vectors are called Lanczos vectors.

Eigenvalues and Lanczos' method, basic steps

We have thus

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k \hat{q}_k + \beta_k \hat{q}_{k+1},$$

with $\beta_0\hat{q}_0 = 0$ for $k = 1 : n-1$ and

$$\alpha_k = \hat{q}_k^T \hat{A} \hat{q}_k.$$

If

$$\hat{r}_k = (\hat{A} - \alpha_k \hat{I})\hat{q}_k - \beta_{k-1}\hat{q}_{k-1},$$

is non-zero, then

$$\hat{q}_{k+1} = \hat{r}_k / \beta_k,$$

with $\beta_k = \pm||\hat{r}_k||_2$.

Part II
Mean-field theories and post Hartree-Fock methods

Chapter 4
Hartree-Fock methods

Why Hartree-Fock? Derivation of Hartree-Fock equations in coordinate space

Hartree-Fock (HF) theory is an algorithm for finding an approximative expression for the ground state of a given Hamiltonian. The basic ingredients are

- Define a single-particle basis $\{\psi_\alpha\}$ so that

$$\hat{h}^{\mathrm{HF}}\psi_\alpha = \varepsilon_\alpha \psi_\alpha$$

with the Hartree-Fock Hamiltonian defined as

$$\hat{h}^{\mathrm{HF}} = \hat{t} + \hat{u}_{\mathrm{ext}} + \hat{u}^{\mathrm{HF}}$$

- The term $\hat{u}^{\mathrm{HF}}$ is a single-particle potential to be determined by the HF algorithm.
- The HF algorithm means to choose $\hat{u}^{\mathrm{HF}}$ in order to have

$$\langle \hat{H} \rangle = E^{\mathrm{HF}} = \langle \Phi_0 | \hat{H} | \Phi_0 \rangle$$

that is to find a local minimum with a Slater determinant Φ_0 being the ansatz for the ground state.

- The variational principle ensures that $E^{\mathrm{HF}} \geq E_0$, with E_0 the exact ground state energy.

We will show that the Hartree-Fock Hamiltonian $\hat{h}^{\mathrm{HF}}$ equals our definition of the operator $\hat{f}$ discussed in connection with the new definition of the normal-ordered Hamiltonian (see later lectures), that is we have, for a specific matrix element

$$\langle p | \hat{h}^{\mathrm{HF}} | q \rangle = \langle p | \hat{f} | q \rangle = \langle p | \hat{t} + \hat{u}_{\mathrm{ext}} | q \rangle + \sum_{i \leq F} \langle pi | \hat{V} | qi \rangle_{AS},$$

meaning that

$$\langle p | \hat{u}^{\mathrm{HF}} | q \rangle = \sum_{i \leq F} \langle pi | \hat{V} | qi \rangle_{AS}.$$

The so-called Hartree-Fock potential $\hat{u}^{\mathrm{HF}}$ brings an explicit medium dependence due to the summation over all single-particle states below the Fermi level F. It brings also in an explicit dependence on the two-body interaction (in nuclear physics we can also have complicated three- or higher-body forces). The two-body interaction, with its contribution from the other bystanding fermions, creates an effective mean field in which a given fermion moves, in addition to the external potential $\hat{u}_{\mathrm{ext}}$ which confines the motion of the fermion. For systems like nuclei, there is no external confining potential. Nuclei are examples of self-bound systems, where the binding arises due to the intrinsic nature of the strong force. For nuclear systems thus, there would be no external one-body potential in the Hartree-Fock Hamiltonian.

Variational Calculus and Lagrangian Multipliers

The calculus of variations involves problems where the quantity to be minimized or maximized is an integral.

In the general case we have an integral of the type

$$E[\Phi] = \int_a^b f(\Phi(x), \frac{\partial \Phi}{\partial x}, x)dx,$$

where E is the quantity which is sought minimized or maximized. The problem is that although f is a function of the variables Φ, $\partial\Phi/\partial x$ and x, the exact dependence of Φ on x is not known. This means again that even though the integral has fixed limits a and b, the path of integration is not known. In our case the unknown quantities are the single-particle wave functions and we wish to choose an integration path which makes the functional $E[\Phi]$ stationary. This means that we want to find minima, or maxima or saddle points. In physics we search normally for minima. Our task is therefore to find the minimum of $E[\Phi]$ so that its variation δE is zero subject to specific constraints. In our case the constraints appear as the integral which expresses the orthogonality of the single-particle wave functions. The constraints can be treated via the technique of Lagrangian multipliers

Let us specialize to the expectation value of the energy for one particle in three-dimensions. This expectation value reads

$$E = \int dxdydz \psi^*(x,y,z)\hat{H}\psi(x,y,z),$$

with the constraint

$$\int dxdydz \psi^*(x,y,z)\psi(x,y,z) = 1,$$

and a Hamiltonian

$$\hat{H} = -\frac{1}{2}\nabla^2 + V(x,y,z).$$

We will, for the sake of notational convenience, skip the variables x,y,z below, and write for example $V(x,y,z) = V$.

The integral involving the kinetic energy can be written as, with the function ψ vanishing strongly for large values of x,y,z (given here by the limits a and b),

$$\int_a^b dxdydz \psi^* \left(-\frac{1}{2}\nabla^2\right)\psi dxdydz = \psi^*\nabla\psi|_a^b + \int_a^b dxdydz \frac{1}{2}\nabla\psi^*\nabla\psi.$$

We will drop the limits a and b in the remaining discussion. Inserting this expression into the expectation value for the energy and taking the variational minimum we obtain

$$\delta E = \delta\left\{\int dxdydz\left(\frac{1}{2}\nabla\psi^*\nabla\psi + V\psi^*\psi\right)\right\} = 0.$$

The constraint appears in integral form as

$$\int dxdydz \psi^*\psi = \text{constant},$$

and multiplying with a Lagrangian multiplier λ and taking the variational minimum we obtain the final variational equation

$$\delta\left\{\int dxdydz\left(\frac{1}{2}\nabla\psi^*\nabla\psi+V\psi^*\psi-\lambda\psi^*\psi\right)\right\}=0.$$

We introduce the function f

$$f=\frac{1}{2}\nabla\psi^*\nabla\psi+V\psi^*\psi-\lambda\psi^*\psi=\frac{1}{2}(\psi_x^*\psi_x+\psi_y^*\psi_y+\psi_z^*\psi_z)+V\psi^*\psi-\lambda\psi^*\psi,$$

where we have skipped the dependence on x,y,z and introduced the shorthand ψ_x, ψ_y and ψ_z for the various derivatives.

For ψ^* the Euler-Lagrange equations yield

$$\frac{\partial f}{\partial\psi^*}-\frac{\partial}{\partial x}\frac{\partial f}{\partial\psi_x^*}-\frac{\partial}{\partial y}\frac{\partial f}{\partial\psi_y^*}-\frac{\partial}{\partial z}\frac{\partial f}{\partial\psi_z^*}=0,$$

which results in

$$-\frac{1}{2}(\psi_{xx}+\psi_{yy}+\psi_{zz})+V\psi=\lambda\psi.$$

We can then identify the Lagrangian multiplier as the energy of the system. The last equation is nothing but the standard Schroedinger equation and the variational approach discussed here provides a powerful method for obtaining approximate solutions of the wave function.

Derivation of Hartree-Fock equations in coordinate space

Let us denote the ground state energy by E_0. According to the variational principle we have

$$E_0\le E[\Phi]=\int\Phi^*\hat{H}\Phi d\tau$$

where Φ is a trial function which we assume to be normalized

$$\int\Phi^*\Phi d\tau=1,$$

where we have used the shorthand $d\tau=dx_1dx_2\dots dx_A$.

In the Hartree-Fock method the trial function is a Slater determinant which can be rewritten as

$$\Psi(x_1,x_2,\dots,x_A,\alpha,\beta,\dots,\nu)=\frac{1}{\sqrt{A!}}\sum_P(-)^P P\psi_\alpha(x_1)\psi_\beta(x_2)\dots\psi_\nu(x_A)=\sqrt{A!}\hat{A}\Phi_H,$$

where we have introduced the anti-symmetrization operator $\hat{A}$ defined by the summation over all possible permutations p of two fermions. It is defined as

$$\hat{A}=\frac{1}{A!}\sum_p(-)^p\hat{P},$$

with the the Hartree-function given by the simple product of all possible single-particle function

$$\Phi_H(x_1,x_2,\dots,x_A,\alpha,\beta,\dots,\nu)=\psi_\alpha(x_1)\psi_\beta(x_2)\dots\psi_\nu(x_A).$$

Our functional is written as

$$E[\Phi]=\sum_{\mu=1}^{A}\int\psi_\mu^*(x_i)\hat{h}_0(x_i)\psi_\mu(x_i)dx_i+\frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\left[\int\psi_\mu^*(x_i)\psi_\nu^*(x_j)\hat{v}(r_{ij})\psi_\mu(x_i)\psi_\nu(x_j)dx_idx_j-\int\psi_\mu^*(x_i)\psi_\nu^*(x_j)\hat{v}(r_{ij})\psi_\nu(x_i)\psi_\mu(x_j)\right.$$

The more compact version reads

$$E[\Phi]=\sum_{\mu}^{A}\langle\mu|\hat{h}_0|\mu\rangle+\frac{1}{2}\sum_{\mu\nu}^{A}\left[\langle\mu\nu|\hat{v}|\mu\nu\rangle-\langle\nu\mu|\hat{v}|\mu\nu\rangle\right].$$

Since the interaction is invariant under the interchange of two particles it means for example that we have

$$\langle\mu\nu|\hat{v}|\mu\nu\rangle=\langle\nu\mu|\hat{v}|\nu\mu\rangle,$$

or in the more general case

$$\langle\mu\nu|\hat{v}|\sigma\tau\rangle=\langle\nu\mu|\hat{v}|\tau\sigma\rangle.$$

The direct and exchange matrix elements can be brought together if we define the antisymmetrized matrix element

$$\langle\mu\nu|\hat{v}|\mu\nu\rangle_{AS}=\langle\mu\nu|\hat{v}|\mu\nu\rangle-\langle\mu\nu|\hat{v}|\nu\mu\rangle,$$

or for a general matrix element

$$\langle\mu\nu|\hat{v}|\sigma\tau\rangle_{AS}=\langle\mu\nu|\hat{v}|\sigma\tau\rangle-\langle\mu\nu|\hat{v}|\tau\sigma\rangle.$$

It has the symmetry property

$$\langle\mu\nu|\hat{v}|\sigma\tau\rangle_{AS}=-\langle\mu\nu|\hat{v}|\tau\sigma\rangle_{AS}=-\langle\nu\mu|\hat{v}|\sigma\tau\rangle_{AS}.$$

The antisymmetric matrix element is also hermitian, implying

$$\langle\mu\nu|\hat{v}|\sigma\tau\rangle_{AS}=\langle\sigma\tau|\hat{v}|\mu\nu\rangle_{AS}.$$

With these notations we rewrite the Hartree-Fock functional as

$$\int\Phi^*\hat{H}_I\Phi d\tau=\frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\langle\mu\nu|\hat{v}|\mu\nu\rangle_{AS}. \tag{4.1}$$

Adding the contribution from the one-body operator $\hat{H}_0$ to (4.1) we obtain the energy functional

$$E[\Phi]=\sum_{\mu=1}^{A}\langle\mu|h|\mu\rangle+\frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\langle\mu\nu|\hat{v}|\mu\nu\rangle_{AS}. \tag{4.2}$$

In our coordinate space derivations below we will spell out the Hartree-Fock equations in terms of their integrals.

If we generalize the Euler-Lagrange equations to more variables and introduce N^2 Lagrange multipliers which we denote by $\varepsilon_{\mu\nu}$, we can write the variational equation for the functional of E

$$\delta E-\sum_{\mu\nu}^{A}\varepsilon_{\mu\nu}\delta\int\psi_\mu^*\psi_\nu=0.$$

For the orthogonal wave functions ψ_i this reduces to

$$\delta E-\sum_{\mu=1}^{A}\varepsilon_\mu\delta\int\psi_\mu^*\psi_\mu=0.$$

Variation with respect to the single-particle wave functions ψ_μ yields then

$$\sum_{\mu=1}^{A}\int\delta\psi_\mu^*\hat{h_0}(x_i)\psi_\mu dx_i+\frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\left[\int\delta\psi_\mu^*\psi_\nu^*\hat{v}(r_{ij})\psi_\mu\psi_\nu dx_idx_j-\int\delta\psi_\mu^*\psi_\nu^*\hat{v}(r_{ij})\psi_\nu\psi_\mu dx_idx_j\right]+$$

$$\sum_{\mu=1}^{A}\int\psi_\mu^*\hat{h_0}(x_i)\delta\psi_\mu dx_i+\frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\left[\int\psi_\mu^*\psi_\nu^*\hat{v}(r_{ij})\delta\psi_\mu\psi_\nu dx_idx_j-\int\psi_\mu^*\psi_\nu^*\hat{v}(r_{ij})\psi_\nu\delta\psi_\mu dx_idx_j\right]-\sum_{\mu=1}^{A}E_\mu\int\delta\psi_\mu^*\psi_\mu dx_i-\sum_{\mu=1}^{A}E_\mu\int$$

Although the variations $\delta\psi$ and $\delta\psi^*$ are not independent, they may in fact be treated as such, so that the terms dependent on either $\delta\psi$ and $\delta\psi^*$ individually may be set equal to zero. To see this, simply replace the arbitrary variation $\delta\psi$ by $i\delta\psi$, so that $\delta\psi^*$ is replaced by $-i\delta\psi^*$, and combine the two equations. We thus arrive at the Hartree-Fock equations

$$\left[-\frac{1}{2}\nabla_i^2+\sum_{\nu=1}^{A}\int\psi_\nu^*(x_j)\hat{v}(r_{ij})\psi_\nu(x_j)dx_j\right]\psi_\mu(x_i)-\left[\sum_{\nu=1}^{A}\int\psi_\nu^*(x_j)\hat{v}(r_{ij})\psi_\mu(x_j)dx_j\right]\psi_\nu(x_i)=\varepsilon_\mu\psi_\mu(x_i). \tag{4.3}$$

Notice that the integration $\int dx_j$ implies an integration over the spatial coordinates $\mathbf{r_j}$ and a summation over the spin-coordinate of fermion j. We note that the factor of $1/2$ in front of the sum involving the two-body interaction, has been removed. This is due to the fact that we need to vary both $\delta\psi_\mu^*$ and $\delta\psi_\nu^*$. Using the symmetry properties of the two-body interaction and interchanging μ and ν as summation indices, we obtain two identical terms.

The two first terms in the last equation are the one-body kinetic energy and the electron-nucleus potential. The third or *direct* term is the averaged electronic repulsion of the other electrons. As written, the term includes the *self-interaction* of electrons when $\mu=\nu$. The self-interaction is cancelled in the fourth term, or the *exchange* term. The exchange term results from our inclusion of the Pauli principle and the assumed determinantal form of the wave-function. Equation (4.3), in addition to the kinetic energy and the attraction from the atomic nucleus that confines the motion of a single electron, represents now the motion of a single-particle modified by the two-body interaction. The additional contribution to the Schroedinger equation due to the two-body interaction, represents a mean field set up by all the other bystanding electrons, the latter given by the sum over all single-particle states occupied by N electrons.

The Hartree-Fock equation is an example of an integro-differential equation. These equations involve repeated calculations of integrals, in addition to the solution of a set of coupled differential equations. The Hartree-Fock equations can also be rewritten in terms of an eigenvalue problem. The solution of an eigenvalue problem represents often a more practical algorithm and the solution of coupled integro-differential equations. This alternative derivation of the Hartree-Fock equations is given below.

Analysis of Hartree-Fock equations in coordinate space

A theoretically convenient form of the Hartree-Fock equation is to regard the direct and exchange operator defined through

$$V_\mu^d(x_i)=\int\psi_\mu^*(x_j)\hat{v}(r_{ij})\psi_\mu(x_j)dx_j$$

and

$$V_\mu^{ex}(x_i)g(x_i)=\left(\int\psi_\mu^*(x_j)\hat{v}(r_{ij})g(x_j)dx_j\right)\psi_\mu(x_i),$$

respectively.

The function $g(x_i)$ is an arbitrary function, and by the substitution $g(x_i) = \psi_\nu(x_i)$ we get

$$V_\mu^{ex}(x_i)\psi_\nu(x_i) = \left(\int \psi_\mu^*(x_j)\hat{v}(r_{ij})\psi_\nu(x_j)dx_j\right)\psi_\mu(x_i).$$

We may then rewrite the Hartree-Fock equations as

$$\hat{h}^{HF}(x_i)\psi_\nu(x_i) = \varepsilon_\nu\psi_\nu(x_i),$$

with

$$\hat{h}^{HF}(x_i) = \hat{h}_0(x_i) + \sum_{\mu=1}^{A} V_\mu^d(x_i) - \sum_{\mu=1}^{A} V_\mu^{ex}(x_i),$$

and where $\hat{h}_0(i)$ is the one-body part. The latter is normally chosen as a part which yields solutions in closed form. The harmonic oscilltor is a classical problem thereof. We normally rewrite the last equation as

$$\hat{h}^{HF}(x_i) = \hat{h}_0(x_i) + \hat{u}^{HF}(x_i).$$

Hartree-Fock by varying the coefficients of a wave function expansion

Another possibility is to expand the single-particle functions in a known basis and vary the coefficients, that is, the new single-particle wave function is written as a linear expansion in terms of a fixed chosen orthogonal basis (for example the well-known harmonic oscillator functions or the hydrogen-like functions etc). We define our new Hartree-Fock single-particle basis by performing a unitary transformation on our previous basis (labelled with greek indices) as

$$\psi_p^{HF} = \sum_\lambda C_{p\lambda}\phi_\lambda. \tag{4.4}$$

In this case we vary the coefficients $C_{p\lambda}$. If the basis has infinitely many solutions, we need to truncate the above sum. We assume that the basis ϕ_λ is orthogonal.

It is normal to choose a single-particle basis defined as the eigenfunctions of parts of the full Hamiltonian. The typical situation consists of the solutions of the one-body part of the Hamiltonian, that is we have

$$\hat{h}_0\phi_\lambda = \varepsilon_\lambda\phi_\lambda.$$

The single-particle wave functions $\phi_\lambda(\mathbf{r})$, defined by the quantum numbers λ and $\mathbf{r}$ are defined as the overlap

$$\phi_\lambda(\mathbf{r}) = \langle\mathbf{r}|\lambda\rangle.$$

In deriving the Hartree-Fock equations, we will expand the single-particle functions in a known basis and vary the coefficients, that is, the new single-particle wave function is written as a linear expansion in terms of a fixed chosen orthogonal basis (for example the well-known harmonic oscillator functions or the hydrogen-like functions etc).

We stated that a unitary transformation keeps the orthogonality. To see this consider first a basis of vectors $\mathbf{v}_i$,

$$\mathbf{v}_i = \begin{bmatrix} v_{i1} \\ \dots \\ \dots \\ v_{in} \end{bmatrix}$$

We assume that the basis is orthogonal, that is

$$\mathbf{v}_j^T\mathbf{v}_i = \delta_{ij}.$$

An orthogonal or unitary transformation

$$\mathbf{w}_i = \mathbf{U}\mathbf{v}_i,$$

preserves the dot product and orthogonality since

$$\mathbf{w}_j^T\mathbf{w}_i = (\mathbf{U}\mathbf{v}_j)^T\mathbf{U}\mathbf{v}_i = \mathbf{v}_j^T\mathbf{U}^T\mathbf{U}\mathbf{v}_i = \mathbf{v}_j^T\mathbf{v}_i = \delta_{ij}.$$

This means that if the coefficients $C_{p\lambda}$ belong to a unitary or orthogonal trasformation (using the Dirac bra-ket notation)

$$|p\rangle = \sum_\lambda C_{p\lambda}|\lambda\rangle,$$

orthogonality is preserved, that is $\langle\alpha|\beta\rangle = \delta_{\alpha\beta}$ and $\langle p|q\rangle = \delta_{pq}$.

This propertry is extremely useful when we build up a basis of many-body Stater determinant based states.

Note also that although a basis $|\alpha\rangle$ contains an infinity of states, for practical calculations we have always to make some truncations.

Before we develop the Hartree-Fock equations, there is another very useful property of determinants that we will use both in connection with Hartree-Fock calculations and later shell-model calculations.

Consider the following determinant

$$\begin{vmatrix} \alpha_1 b_{11} + \alpha_2 s b_{12} & a_{12} \\ \alpha_1 b_{21} + \alpha_2 b_{22} & a_{22} \end{vmatrix} = \alpha_1 \begin{vmatrix} b_{11} & a_{12} \\ b_{21} & a_{22} \end{vmatrix} + \alpha_2 \begin{vmatrix} b_{12} & a_{12} \\ b_{22} & a_{22} \end{vmatrix}$$

We can generalize this to an $n\times n$ matrix and have

$$\begin{vmatrix} a_{11} & a_{12} & \dots & \sum_{k=1}^n c_k b_{1k} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \sum_{k=1}^n c_k b_{2k} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \sum_{k=1}^n c_k b_{nk} & \dots & a_{nn} \end{vmatrix} = \sum_{k=1}^n c_k \begin{vmatrix} a_{11} & a_{12} & \dots & b_{1k} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & b_{2k} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & b_{nk} & \dots & a_{nn} \end{vmatrix}.$$

This is a property we will use in our Hartree-Fock discussions.

We can generalize the previous results, now with all elements a_{ij} being given as functions of linear combinations of various coefficients c and elements b_{ij},

$$\begin{vmatrix} \sum_{k=1}^n b_{1k}c_{k1} & \sum_{k=1}^n b_{1k}c_{k2} & \dots & \sum_{k=1}^n b_{1k}c_{kj} & \dots & \sum_{k=1}^n b_{1k}c_{kn} \\ \sum_{k=1}^n b_{2k}c_{k1} & \sum_{k=1}^n b_{2k}c_{k2} & \dots & \sum_{k=1}^n b_{2k}c_{kj} & \dots & \sum_{k=1}^n b_{2k}c_{kn} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \sum_{k=1}^n b_{nk}c_{k1} & \sum_{k=1}^n b_{nk}c_{k2} & \dots & \sum_{k=1}^n b_{nk}c_{kj} & \dots & \sum_{k=1}^n b_{nk}c_{kn} \end{vmatrix} = det(\mathbf{C})det(\mathbf{B}),$$

where $det(\mathbf{C})$ and $det(\mathbf{B})$ are the determinants of $n\times n$ matrices with elements c_{ij} and b_{ij} respectively. This is a property we will use in our Hartree-Fock discussions. Convince yourself about the correctness of the above expression by setting $n=2$.

With our definition of the new basis in terms of an orthogonal basis we have

$$\psi_p(x) = \sum_\lambda C_{p\lambda}\phi_\lambda(x).$$

If the coefficients $C_{p\lambda}$ belong to an orthogonal or unitary matrix, the new basis is also orthogonal. Our Slater determinant in the new basis $\psi_p(x)$ is written as

$$\frac{1}{\sqrt{A!}}\left|\begin{array}{ccccc} \psi_p(x_1) & \psi_p(x_2) & \dots & \dots & \psi_p(x_A)\\ \psi_q(x_1) & \psi_q(x_2) & \dots & \dots & \psi_q(x_A)\\ \dots & \dots & \dots & \dots & \dots\\ \dots & \dots & \dots & \dots & \dots\\ \psi_t(x_1) & \psi_t(x_2) & \dots & \dots & \psi_t(x_A)\end{array}\right| = \frac{1}{\sqrt{A!}}\left|\begin{array}{ccccc} \sum_\lambda C_{p\lambda}\phi_\lambda(x_1) & \sum_\lambda C_{p\lambda}\phi_\lambda(x_2) & \dots & \dots & \sum_\lambda C_{p\lambda}\phi_\lambda(x_A)\\ \sum_\lambda C_{q\lambda}\phi_\lambda(x_1) & \sum_\lambda C_{q\lambda}\phi_\lambda(x_2) & \dots & \dots & \sum_\lambda C_{q\lambda}\phi_\lambda(x_A)\\ \dots & \dots & \dots & \dots & \dots\\ \dots & \dots & \dots & \dots & \dots\\ \sum_\lambda C_{t\lambda}\phi_\lambda(x_1) & \sum_\lambda C_{t\lambda}\phi_\lambda(x_2) & \dots & \dots & \sum_\lambda C_{t\lambda}\phi_\lambda(x_A)\end{array}\right|,$$

which is nothing but $det(\mathbf{C})det(\Phi)$, with $det(\Phi)$ being the determinant given by the basis functions $\phi_\lambda(x)$.

In our discussions hereafter we will use our definitions of single-particle states above and below the Fermi (F) level given by the labels $ijkl\cdots \le F$ for so-called single-hole states and $abcd\cdots > F$ for so-called particle states. For general single-particle states we employ the labels $pqrs\dots$.

In Eq. (4.2), restated here

$$E[\Phi] = \sum_{\mu=1}^{A}\langle\mu|h|\mu\rangle + \frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\langle\mu\nu|\hat{v}|\mu\nu\rangle_{AS},$$

we found the expression for the energy functional in terms of the basis function $\phi_\lambda(\mathbf{r})$. We then varied the above energy functional with respect to the basis functions $|\mu\rangle$. Now we are interested in defining a new basis defined in terms of a chosen basis as defined in Eq. (4.4). We can then rewrite the energy functional as

$$E[\Phi^{HF}] = \sum_{i=1}^{A}\langle i|h|i\rangle + \frac{1}{2}\sum_{ij=1}^{A}\langle ij|\hat{v}|ij\rangle_{AS}, \tag{4.5}$$

where Φ^{HF} is the new Slater determinant defined by the new basis of Eq. (4.4).

Using Eq. (4.4) we can rewrite Eq. (4.5) as

$$E[\Psi] = \sum_{i=1}^{A}\sum_{\alpha\beta}C^*_{i\alpha}C_{i\beta}\langle\alpha|h|\beta\rangle + \frac{1}{2}\sum_{ij=1}^{A}\sum_{\alpha\beta\gamma\delta}C^*_{i\alpha}C^*_{j\beta}C_{i\gamma}C_{j\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{AS}. \tag{4.6}$$

We wish now to minimize the above functional. We introduce again a set of Lagrange multipliers, noting that since $\langle i|j\rangle = \delta_{i,j}$ and $\langle\alpha|\beta\rangle = \delta_{\alpha,\beta}$, the coefficients $C_{i\gamma}$ obey the relation

$$\langle i|j\rangle = \delta_{i,j} = \sum_{\alpha\beta}C^*_{i\alpha}C_{i\beta}\langle\alpha|\beta\rangle = \sum_{\alpha}C^*_{i\alpha}C_{i\alpha},$$

which allows us to define a functional to be minimized that reads

$$F[\Phi^{HF}] = E[\Phi^{HF}] - \sum_{i=1}^{A}\varepsilon_i\sum_{\alpha}C^*_{i\alpha}C_{i\alpha}. \tag{4.7}$$

Minimizing with respect to $C^*_{i\alpha}$, remembering that the equations for $C^*_{i\alpha}$ and $C_{i\alpha}$ can be written as two independent equations, we obtain

$$\frac{d}{dC^*_{i\alpha}}\left[E[\Phi^{HF}] - \sum_j\varepsilon_j\sum_\alpha C^*_{j\alpha}C_{j\alpha}\right] = 0,$$

which yields for every single-particle state i and index α (recalling that the coefficients $C_{i\alpha}$ are matrix elements of a unitary (or orthogonal for a real symmetric matrix) matrix) the following

Hartree-Fock equations

$$\sum_{\beta} C_{i\beta}\langle\alpha|h|\beta\rangle + \sum_{j=1}^{A}\sum_{\beta\gamma\delta} C^*_{j\beta}C_{j\delta}C_{i\gamma}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{AS} = \varepsilon_i^{HF}C_{i\alpha}.$$

We can rewrite this equation as (changing dummy variables)

$$\sum_{\beta}\left\{\langle\alpha|h|\beta\rangle + \sum_{j}^{A}\sum_{\gamma\delta} C^*_{j\gamma}C_{j\delta}\langle\alpha\gamma|\hat{v}|\beta\delta\rangle_{AS}\right\}C_{i\beta} = \varepsilon_i^{HF}C_{i\alpha}.$$

Note that the sums over greek indices run over the number of basis set functions (in principle an infinite number).

Defining

$$h_{\alpha\beta}^{HF} = \langle\alpha|h|\beta\rangle + \sum_{j=1}^{A}\sum_{\gamma\delta} C^*_{j\gamma}C_{j\delta}\langle\alpha\gamma|\hat{v}|\beta\delta\rangle_{AS},$$

we can rewrite the new equations as

$$\sum_{\beta} h_{\alpha\beta}^{HF}C_{i\beta} = \varepsilon_i^{HF}C_{i\alpha}. \tag{4.8}$$

The latter is nothing but a standard eigenvalue problem. Compared with Eq. (4.3), we see that we do not need to compute any integrals in an iterative procedure for solving the equations. It suffices to tabulate the matrix elements $\langle\alpha|h|\beta\rangle$ and $\langle\alpha\gamma|\hat{v}|\beta\delta\rangle_{AS}$ once and for all. Successive iterations require thus only a look-up in tables over one-body and two-body matrix elements. These details will be discussed below when we solve the Hartree-Fock equations numerical.

Hartree-Fock algorithm

Our Hartree-Fock matrix is thus

$$\hat{h}_{\alpha\beta}^{HF} = \langle\alpha|\hat{h}_0|\beta\rangle + \sum_{j=1}^{A}\sum_{\gamma\delta} C^*_{j\gamma}C_{j\delta}\langle\alpha\gamma|\hat{v}|\beta\delta\rangle_{AS}.$$

The Hartree-Fock equations are solved in an iterative waym starting with a guess for the coefficients $C_{j\gamma} = \delta_{j,\gamma}$ and solving the equations by diagonalization till the new single-particle energies $\varepsilon_i^{\mathrm{HF}}$ do not change anymore by a prefixed quantity.

Normally we assume that the single-particle basis $|\beta\rangle$ forms an eigenbasis for the operator $\hat{h}_0$, meaning that the Hartree-Fock matrix becomes

$$\hat{h}_{\alpha\beta}^{HF} = \varepsilon_{\alpha}\delta_{\alpha,\beta} + \sum_{j=1}^{A}\sum_{\gamma\delta} C^*_{j\gamma}C_{j\delta}\langle\alpha\gamma|\hat{v}|\beta\delta\rangle_{AS}.$$

The Hartree-Fock eigenvalue problem

$$\sum_{\beta}\hat{h}_{\alpha\beta}^{HF}C_{i\beta} = \varepsilon_i^{\mathrm{HF}}C_{i\alpha},$$

can be written out in a more compact form as

$$\hat{h}^{HF}\hat{C} = \varepsilon^{\mathrm{HF}}\hat{C}.$$

The Hartree-Fock equations are, in their simplest form, solved in an iterative way, starting with a guess for the coefficients $C_{i\alpha}$. We label the coefficients as $C_{i\alpha}^{(n)}$, where the subscript n stands for iteration n. To set up the algorithm we can proceed as follows:

- We start with a guess $C_{i\alpha}^{(0)} = \delta_{i,\alpha}$. Alternatively, we could have used random starting values as long as the vectors are normalized. Another possibility is to give states below the Fermi level a larger weight.
- The Hartree-Fock matrix simplifies then to (assuming that the coefficients $C_{i\alpha}$ are real)

$$\hat{h}_{\alpha\beta}^{HF} = \varepsilon_\alpha \delta_{\alpha,\beta} + \sum_{j=1}^{A}\sum_{\gamma\delta} C_{j\gamma}^{(0)} C_{j\delta}^{(0)} \langle \alpha\gamma|\hat{v}|\beta\delta\rangle_{AS}.$$

Solving the Hartree-Fock eigenvalue problem yields then new eigenvectors $C_{i\alpha}^{(1)}$ and eigenvalues $\varepsilon_i^{HF(1)}$.

- With the new eigenvalues we can set up a new Hartree-Fock potential

$$\sum_{j=1}^{A}\sum_{\gamma\delta} C_{j\gamma}^{(1)} C_{j\delta}^{(1)} \langle \alpha\gamma|\hat{v}|\beta\delta\rangle_{AS}.$$

The diagonalization with the new Hartree-Fock potential yields new eigenvectors and eigenvalues. This process is continued till for example

$$\frac{\sum_p |\varepsilon_i^{(n)} - \varepsilon_i^{(n-1)}|}{m} \le \lambda,$$

where λ is a user prefixed quantity ($\lambda \sim 10^{-8}$ or smaller) and p runs over all calculated single-particle energies and m is the number of single-particle states.

Analysis of Hartree-Fock equations and Koopman's theorem

We can rewrite the ground state energy by adding and subtracting $\hat{u}^{HF}(x_i)$

$$E_0^{HF} = \langle \Phi_0|\hat{H}|\Phi_0\rangle = \sum_{i\le F}^{A} \langle i|\hat{h}_0 + \hat{u}^{HF}|j\rangle + \frac{1}{2}\sum_{i\le F}^{A}\sum_{j\le F}^{A}\left[\langle ij|\hat{v}|ij\rangle - \langle ij|\hat{v}|ji\rangle\right] - \sum_{i\le F}^{A}\langle i|\hat{u}^{HF}|i\rangle,$$

which results in

$$E_0^{HF} = \sum_{i\le F}^{A} \varepsilon_i^{HF} + \frac{1}{2}\sum_{i\le F}^{A}\sum_{j\le F}^{A}\left[\langle ij|\hat{v}|ij\rangle - \langle ij|\hat{v}|ji\rangle\right] - \sum_{i\le F}^{A}\langle i|\hat{u}^{HF}|i\rangle.$$

Our single-particle states $ijk\dots$ are now single-particle states obtained from the solution of the Hartree-Fock equations.

Using our definition of the Hartree-Fock single-particle energies we obtain then the following expression for the total ground-state energy

$$E_0^{HF} = \sum_{i\le F}^{A} \varepsilon_i - \frac{1}{2}\sum_{i\le F}^{A}\sum_{j\le F}^{A}\left[\langle ij|\hat{v}|ij\rangle - \langle ij|\hat{v}|ji\rangle\right].$$

This form will be used in our discussion of Koopman's theorem.

In the atomic physics case we have

$$E[\Phi^{\mathrm{HF}}(N)] = \sum_{i=1}^{H} \langle i|\hat{h}_0|i\rangle + \frac{1}{2}\sum_{ij=1}^{N} \langle ij|\hat{v}|ij\rangle_{AS},$$

where $\Phi^{\mathrm{HF}}(N)$ is the new Slater determinant defined by the new basis of Eq. (4.4) for N electrons (same Z). If we assume that the single-particle wave functions in the new basis do not change when we remove one electron or add one electron, we can then define the corresponding energy for the $N-1$ systems as

$$E[\Phi^{\mathrm{HF}}(N-1)] = \sum_{i=1;i\neq k}^{N} \langle i|\hat{h}_0|i\rangle + \frac{1}{2}\sum_{ij=1;i,j\neq k}^{N} \langle ij|\hat{v}|ij\rangle_{AS},$$

where we have removed a single-particle state $k \le F$, that is a state below the Fermi level.

Calculating the difference

$$E[\Phi^{\mathrm{HF}}(N)] - E[\Phi^{\mathrm{HF}}(N-1)] = \langle k|\hat{h}_0|k\rangle + \frac{1}{2}\sum_{i=1;i\neq k}^{N} \langle ik|\hat{v}|ik\rangle_{AS} + \frac{1}{2}\sum_{j=1;j\neq k}^{N} \langle kj|\hat{v}|kj\rangle_{AS},$$

we obtain

$$E[\Phi^{\mathrm{HF}}(N)] - E[\Phi^{\mathrm{HF}}(N-1)] = \langle k|\hat{h}_0|k\rangle + \sum_{j=1}^{N} \langle kj|\hat{v}|kj\rangle_{AS}$$

which is just our definition of the Hartree-Fock single-particle energy

$$E[\Phi^{\mathrm{HF}}(N)] - E[\Phi^{\mathrm{HF}}(N-1)] = \varepsilon_k^{\mathrm{HF}}$$

Similarly, we can now compute the difference (we label the single-particle states above the Fermi level as $abcd > F$)

$$E[\Phi^{\mathrm{HF}}(N+1)] - E[\Phi^{\mathrm{HF}}(N)] = \varepsilon_a^{\mathrm{HF}}.$$

These two equations can thus be used to the electron affinity or ionization energies, respectively. Koopman's theorem states that for example the ionization energy of a closed-shell system is given by the energy of the highest occupied single-particle state. If we assume that changing the number of electrons from N to $N+1$ does not change the Hartree-Fock single-particle energies and eigenfunctions, then Koopman's theorem simply states that the ionization energy of an atom is given by the single-particle energy of the last bound state. In a similar way, we can also define the electron affinities.

As an example, consider a simple model for atomic sodium, Na. Neutral sodium has eleven electrons, with the weakest bound one being confined the $3s$ single-particle quantum numbers. The energy needed to remove an electron from neutral sodium is rather small, 5.1391 eV, a feature which pertains to all alkali metals. Having performed a Hartree-Fock calculation for neutral sodium would then allows us to compute the ionization energy by using the single-particle energy for the $3s$ states, namely $\varepsilon_{3s}^{\mathrm{HF}}$.

From these considerations, we see that Hartree-Fock theory allows us to make a connection between experimental observables (here ionization and affinity energies) and the underlying interactions between particles. In this sense, we are now linking the dynamics and structure of a many-body system with the laws of motion which govern the system. Our approach is a reductionistic one, meaning that we want to understand the laws of motion in terms of the particles or degrees of freedom which we believe are the fundamental ones. Our Slater determinant, being constructed as the product of various single-particle functions, follows this philosophy.

With similar arguments as in atomic physics, we can now use Hartree-Fock theory to make a link between nuclear forces and separation energies. Changing to nuclear system, we define

$$E[\Phi^{\rm HF}(A)] = \sum_{i=1}^{A}\langle i|\hat{h}_0|i\rangle + \frac{1}{2}\sum_{ij=1}^{A}\langle ij|\hat{v}|ij\rangle_{AS},$$

where $\Phi^{\rm HF}(A)$ is the new Slater determinant defined by the new basis of Eq. (4.4) for A nucleons, where $A = N+Z$, with N now being the number of neutrons and Z th enumber of protons. If we assume again that the single-particle wave functions in the new basis do not change from a nucleus with A nucleons to a nucleus with $A-1$ nucleons, we can then define the corresponding energy for the $A-1$ systems as

$$E[\Phi^{\rm HF}(A-1)] = \sum_{i=1;i\neq k}^{A}\langle i|\hat{h}_0|i\rangle + \frac{1}{2}\sum_{ij=1;i,j\neq k}^{A}\langle ij|\hat{v}|ij\rangle_{AS},$$

where we have removed a single-particle state $k \le F$, that is a state below the Fermi level.

Calculating the difference

$$E[\Phi^{\rm HF}(A)] - E[\Phi^{\rm HF}(A-1)] = \langle k|\hat{h}_0|k\rangle + \frac{1}{2}\sum_{i=1;i\neq k}^{A}\langle ik|\hat{v}|ik\rangle_{AS} + \frac{1}{2}\sum_{j=1;j\neq k}^{A}\langle kj|\hat{v}|kj\rangle_{AS},$$

which becomes

$$E[\Phi^{\rm HF}(A)] - E[\Phi^{\rm HF}(A-1)] = \langle k|\hat{h}_0|k\rangle + \sum_{j=1}^{A}\langle kj|\hat{v}|kj\rangle_{AS}$$

which is just our definition of the Hartree-Fock single-particle energy

$$E[\Phi^{\rm HF}(A)] - E[\Phi^{\rm HF}(A-1)] = \varepsilon_k^{\rm HF}$$

Similarly, we can now compute the difference (recall that the single-particle states $abcd > F$)

$$E[\Phi^{\rm HF}(A+1)] - E[\Phi^{\rm HF}(A)] = \varepsilon_a^{\rm HF}.$$

If we then recall that the binding energy differences

$$BE(A) - BE(A-1) \quad \text{and} \quad BE(A+1) - BE(A),$$

define the separation energies, we see that the Hartree-Fock single-particle energies can be used to define separation energies. We have thus our first link between nuclear forces (included in the potential energy term) and an observable quantity defined by differences in binding energies.

We have thus the following interpretations (if the single-particle fields do not change)

$$BE(A) - BE(A-1) \approx E[\Phi^{\rm HF}(A)] - E[\Phi^{\rm HF}(A-1)] = \varepsilon_k^{\rm HF},$$

and

$$BE(A+1) - BE(A) \approx E[\Phi^{\rm HF}(A+1)] - E[\Phi^{\rm HF}(A)] = \varepsilon_a^{\rm HF}.$$

If we use ^{16}O as our closed-shell nucleus, we could then interpret the separation energy

$$BE(^{16}{\rm O}) - BE(^{15}{\rm O}) \approx \varepsilon^{\rm HF}_{0p^{\nu}_{1/2}},$$

and

$$BE(^{16}{\rm O}) - BE(^{15}{\rm N}) \approx \varepsilon^{\rm HF}_{0p^{\pi}_{1/2}}.$$

Similalry, we could interpret

$$BE(^{17}\mathrm{O}) - BE(^{16}\mathrm{O}) \approx \varepsilon^{\mathrm{HF}}_{0d^{\nu}_{5/2}},$$

and

$$BE(^{17}\mathrm{F}) - BE(^{16}\mathrm{O}) \approx \varepsilon^{\mathrm{HF}}_{0d^{\pi}_{5/2}}.$$

We can continue like this for all $A \pm 1$ nuclei where A is a good closed-shell (or subshell closure) nucleus. Examples are ^{22}O, ^{24}O, ^{40}Ca, ^{48}Ca, ^{52}Ca, ^{54}Ca, ^{56}Ni, ^{68}Ni, ^{78}Ni, ^{90}Zr, ^{88}Sr, ^{100}Sn, ^{132}Sn and ^{208}Pb, to mention some possile cases.

We can thus make our first interpretation of the separation energies in terms of the simplest possible many-body theory. If we also recall that the so-called energy gap for neutrons (or protons) is defined as

$$\Delta S_n = 2BE(N,Z) - BE(N-1,Z) - BE(N+1,Z),$$

for neutrons and the corresponding gap for protons

$$\Delta S_p = 2BE(N,Z) - BE(N,Z-1) - BE(N,Z+1),$$

we can define the neutron and proton energy gaps for ^{16}O as

$$\Delta S_{\nu} = \varepsilon^{\mathrm{HF}}_{0d^{\nu}_{5/2}} - \varepsilon^{\mathrm{HF}}_{0p^{\nu}_{1/2}},$$

and

$$\Delta S_{\pi} = \varepsilon^{\mathrm{HF}}_{0d^{\pi}_{5/2}} - \varepsilon^{\mathrm{HF}}_{0p^{\pi}_{1/2}}.$$

*Exercise : Derivation of Hartree-Fock equations
Consider a Slater determinant built up of single-particle orbitals ψ_{λ}, with $\lambda = 1,2,\dots,N$.
The unitary transformation

$$\psi_a = \sum_{\lambda} C_{a\lambda}\phi_{\lambda},$$

brings us into the new basis. The new basis has quantum numbers $a = 1,2,\dots,N$.

a) Show that the new basis is orthonormal.

b) Show that the new Slater determinant constructed from the new single-particle wave functions can be written as the determinant based on the previous basis and the determinant of the matrix C.

c) Show that the old and the new Slater determinants are equal up to a complex constant with absolute value unity.

Hint.

Use the fact that C is a unitary matrix.
*Exercise : Derivation of Hartree-Fock equations
Consider the Slater determinant

$$\Phi_0 = \frac{1}{\sqrt{n!}}\sum_p (-)^p P \prod_{i=1}^{n} \psi_{\alpha_i}(x_i).$$

A small variation in this function is given by

$$\delta\Phi_0 = \frac{1}{\sqrt{n!}}\sum_p (-)^p P \psi_{\alpha_1}(x_1)\psi_{\alpha_2}(x_2)\dots\psi_{\alpha_{i-1}}(x_{i-1})(\delta\psi_{\alpha_i}(x_i))\psi_{\alpha_{i+1}}(x_{i+1})\dots\psi_{\alpha_n}(x_n).$$

a) Show that

$$\langle\delta\Phi_0|\sum_{i=1}^{n}\{t(x_i)+u(x_i)\}+\frac{1}{2}\sum_{i\neq j=1}^{n}v(x_i,x_j)|\Phi_0\rangle=\sum_{i=1}^{n}\langle\delta\psi_{\alpha_i}|\hat{t}+\hat{u}|\phi_{\alpha_i}\rangle+\sum_{i\neq j=1}^{n}\left\{\langle\delta\psi_{\alpha_i}\psi_{\alpha_j}|\hat{v}|\psi_{\alpha_i}\psi_{\alpha_j}\rangle-\langle\delta\psi_{\alpha_i}\psi_{\alpha_j}|\hat{v}|\psi_{\alpha_j}\psi_{\alpha_i}\rangle\right\}$$

*Exercise : Developing a Hartree-Fock program

Neutron drops are a powerful theoretical laboratory for testing, validating and improving nuclear structure models. Indeed, all approaches to nuclear structure, from ab initio theory to shell model to density functional theory are applicable in such systems. We will, therefore, use neutron drops as a test system for setting up a Hartree-Fock code. This program can later be extended to studies of the binding energy of nuclei like ^{16}O or ^{40}Ca. The single-particle energies obtained by solving the Hartree-Fock equations can then be directly related to experimental separation energies. Since Hartree-Fock theory is the starting point for several many-body techniques (density functional theory, random-phase approximation, shell-model etc), the aim here is to develop a computer program to solve the Hartree-Fock equations in a given single-particle basis, here the harmonic oscillator.

The Hamiltonian for a system of N neutron drops confined in a harmonic potential reads

$$\hat{H}=\sum_{i=1}^{N}\frac{\hat{p}_i^2}{2m}+\sum_{i=1}^{N}\frac{1}{2}m\omega r_i^2+\sum_{i<j}\hat{V}_{ij},$$

with $^2/2m=20.73$ fm^2, $mc^2=938.90590$ MeV, and $\hat{V}_{ij}$ is the two-body interaction potential whose matrix elements are precalculated and to be read in by you.

The Hartree-Fock algorithm can be broken down as follows. We recall that our Hartree-Fock matrix is

$$\hat{h}_{\alpha\beta}^{HF}=\langle\alpha|\hat{h}_0|\beta\rangle+\sum_{j=1}^{N}\sum_{\gamma\delta}C_{j\gamma}^{*}C_{j\delta}\langle\alpha\gamma|V|\beta\delta\rangle_{AS}.$$

Normally we assume that the single-particle basis $|\beta\rangle$ forms an eigenbasis for the operator $\hat{h}_0$ (this is our case), meaning that the Hartree-Fock matrix becomes

$$\hat{h}_{\alpha\beta}^{HF}=\varepsilon_\alpha\delta_{\alpha,\beta}+\sum_{j=1}^{N}\sum_{\gamma\delta}C_{j\gamma}^{*}C_{j\delta}\langle\alpha\gamma|V|\beta\delta\rangle_{AS}.$$

The Hartree-Fock eigenvalue problem

$$\sum_{\beta}\hat{h}_{\alpha\beta}^{HF}C_{i\beta}=\varepsilon_i^{\mathrm{HF}}C_{i\alpha},$$

can be written out in a more compact form as

$$\hat{h}^{HF}\hat{C}=\varepsilon^{\mathrm{HF}}\hat{C}.$$

The equations are often rewritten in terms of a so-called density matrix, which is defined as

$$\rho_{\gamma\delta}=\sum_{i=1}^{N}\langle\gamma|i\rangle\langle i|\delta\rangle=\sum_{i=1}^{N}C_{i\gamma}C_{i\delta}^{*}. \tag{4.9}$$

It means that we can rewrite the Hartree-Fock Hamiltonian as

$$\hat{h}_{\alpha\beta}^{HF}=\varepsilon_\alpha\delta_{\alpha,\beta}+\sum_{\gamma\delta}\rho_{\gamma\delta}\langle\alpha\gamma|V|\beta\delta\rangle_{AS}.$$

It is convenient to use the density matrix since we can precalculate in every iteration the product of two eigenvector components C.

Note that $\langle\alpha|\hat{h}_0|\beta\rangle$ denotes the matrix elements of the one-body part of the starting hamiltonian. For self-bound nuclei $\langle\alpha|\hat{h}_0|\beta\rangle$ is the kinetic energy, whereas for neutron drops, $\langle\alpha|\hat{h}_0|\beta\rangle$ represents the harmonic oscillator hamiltonian since the system is confined in a harmonic trap. If we are working in a harmonic oscillator basis with the same ω as the trapping potential, then $\langle\alpha|\hat{h}_0|\beta\rangle$ is diagonal.

The python program shows how one can, in a brute force way read in matrix elements in m-scheme and compute the Hartree-Fock single-particle energies for four major shells. The interaction which has been used is the so-called N3LO interaction of Machleidt and Entem using the Similarity Renormalization Group approach method to renormalize the interaction, using an oscillator energy $\omega = 10$ MeV.

The nucleon-nucleon two-body matrix elements are in m-scheme and are fully anti-symmetrized. The Hartree-Fock programs uses the density matrix discussed above in order to compute the Hartree-Fock matrix. Here we display the Hartree-Fock part only, assuming that single-particle data and two-body matrix elements have already been read in.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python import numpy as np from decimal import Decimal expectation value for the one body part, Harmonic oscillator in three dimensions def onebody(i, n, l): homega = 10.0 return homega*(2*n[i] + l[i] + 1.5)

if $_{name}{}_{==}{}'_{main}{}'$:

Nparticles = 16 """ Read quantum numbers from file """ index = [] n = [] l = [] j = [] mj = [] tz = [] spOrbitals = 0 with open("nucleispnumbers.dat", "r") as qnumfile: for line in qnumfile: nums = line.split() if len(nums) != 0: index.append(int(nums[0])) n.append(int(nums[1])) l.append(int(nums[2])) j.append(int(nums[3])) mj.append(int(nums[4])) tz.append(int(nums[5])) spOrbitals += 1

""" Read two-nucleon interaction elements (integrals) from file, brute force 4-dim array """ nninteraction = np.zeros([spOrbitals, spOrbitals, spOrbitals, spOrbitals]) with open("nucleitwobody.dat", "r") as infile: for line in infile: number = line.split() a = int(number[0]) - 1 b = int(number[1]) - 1 c = int(number[2]) - 1 d = int(number[3]) - 1 nninteraction[a][b][c][d] = Decimal(number[4]) """ Set up single-particle integral """ singleparticleH = np.zeros(spOrbitals) for i in range(spOrbitals): singleparticleH[i] = Decimal(onebody(i, n, l))

""" Star HF-iterations, preparing variables and density matrix """

""" Coefficients for setting up density matrix, assuming only one along the diagonals """ C = np.eye(spOrbitals) HF coefficients DensityMatrix = np.zeros([spOrbitals,spOrbitals]) for gamma in range(spOrbitals): for delta in range(spOrbitals): sum = 0.0 for i in range(Nparticles): sum += C[gamma][i]*C[delta][i] DensityMatrix[gamma][delta] = Decimal(sum) maxHFiter = 100 epsilon = 1.0e-5 difference = 1.0 hf$_c$*ount* = 0*oldenergies* = *np.zeros*(*spOrbitals*)*newenergies* = *np.zeros*(*spOrbitals*)*whilehf*$_c$*ount* < *maxHFiteranddifference* > *epsilon* : *print*("*IterationHFmatrix* = *np.zeros*([*spOrbitals*, *spOrbitals*])*fora*
forbetainrange(*spOrbitals*) : """"*Iftestsforthree* − *dimensionalsystems*, *includingisospinconservation*""""*ifl*[*alpha*]! = *l*[*beta*]*andj*[*alpha*]! = *j*[*beta*]*andmj*[*alpha*]! = *mj*[*beta*]*andtz*[*alpha*]! = *tz*[*beta*] : *continue*""""*SettinguptheFockmatrixusingthedensitymatrixand*
scheme""""*sumFockTerm* = 0.0*forgammainrange*(*spOrbitals*) : *fordeltainrange*(*spOrbitals*) : *if*(*mj*[*alpha*] + *mj*[*gamma*])! = (*mj*[*beta*] + *mj*[*delta*])*and*(*tz*[*alpha*] + *tz*[*gamma*])! = (*tz*[*beta*] + *tz*[*delta*]) : *continuesumFockTerm*+ = *DensityMatrix*[*gamma*][*delta*] ∗ *nninteraction*[*alpha*][*gamma*][*beta*][*delta*]*HFmatrix*[*alpha*][*beta*] = *Decimal*(*sumFockTerm*)""""*Addingtheone* − *bodyterm*, *hereplainharmonicos*
alpha : *HFmatrix*[*alpha*][*alpha*]+ = *singleparticleH*[*alpha*]*spenergies*, *C* = *np.linalg.eigh*(*HFmatrix*)""""*Settingupnewdensitymatrixinm* − *scheme*""""*DensityMatrix* = *np.zeros*([*spOrbitals*, *spOrbitals*])*forgammainrange*(*spOrbitals*) : *fordeltainrange*(*spOrbitals*) : *sum* = 0.0*foriinrange*(*Nparticles*) : *sum*+ = *C*[*gamma*][*i*] ∗ *C*[*delta*][*i*]*DensityMatrix*[*gamma*][*delta*] = *Decimal*(*sum*)*newenergies* = *spenergies*""""*Bruteforcecomputationofdifferencebetweenpreviousandnewsp HFenergies*""""*sum* = 0.0*foriinrange*(*spOrbitals*) : *sum*+ = (*abs*(*newenergies*[*i*] − *oldenergies*[*i*]))/*spOrbitalsdifference* = *sumoldenergies* = *newenergiesprint*("*Single* − *particleenergies*, *orderingmayhavechanged*")*foriinrange*(*spOrbitals*) : *print*('0 : 4*d*1 : .4*f*'.*format*(*i*, *Decimal*(*oldenergies*[*i*])))*hf*$_c$*ount*+ = 1

Running the program, one finds that the lowest-lying states for a nucleus like ^{16}O, we see that the nucleon-nucleon force brings a natural spin-orbit splitting for the $0p$ states (or other states except the s-states). Since we are using the m-scheme for our calculations, we observe that there are several states with the same eigenvalues. The number of eigenvalues corresponds to the degeneracy $2j+1$ and is well respected in our calculations, as see from the table here.

The values of the lowest-lying states are (π for protons and ν for neutrons)

Quantum numbers	Energy [MeV]
$0s_{1/2}^{\pi}$	-40.4602
$0s_{1/2}^{\pi}$	-40.4602
$0s_{1/2}^{\nu}$	-40.6426
$0s_{1/2}^{\nu}$	-40.6426
$0p_{1/2}^{\pi}$	-6.7133
$0p_{1/2}^{\pi}$	-6.7133
$0p_{1/2}^{\nu}$	-6.8403
$0p_{1/2}^{\nu}$	-6.8403
$0p_{3/2}^{\pi}$	-11.5886
$0p_{3/2}^{\pi}$	-11.5886
$0p_{3/2}^{\pi}$	-11.5886
$0p_{3/2}^{\pi}$	-11.5886
$0p_{3/2}^{\nu}$	-11.7201
$0p_{3/2}^{\nu}$	-11.7201
$0p_{3/2}^{\nu}$	-11.7201
$0p_{3/2}^{\nu}$	-11.7201
$0d_{5/2}^{\pi}$	18.7589
$0d_{5/2}^{\nu}$	18.8082

We can use these results to attempt our first link with experimental data, namely to compute the shell gap or the separation energies. The shell gap for neutrons is given by

$$\Delta S_n = 2BE(N,Z) - BE(N-1,Z) - BE(N+1,Z).$$

For ^{16}O we have an experimental value for the shell gap of 11.51 MeV for neutrons, while our Hartree-Fock calculations result in 25.65 MeV. This means that correlations beyond a simple Hartree-Fock calculation with a two-body force play an important role in nuclear physics. The splitting between the $0p_{3/2}^{\nu}$ and the $0p_{1/2}^{\nu}$ state is 4.88 MeV, while the experimental value for the gap between the ground state $1/2^-$ and the first excited $3/2^-$ states is 6.08 MeV. The two-nucleon spin-orbit force plays a central role here. In our discussion of nuclear forces we will see how the spin-orbit force comes into play here.

Hartree-Fock in second quantization and stability of HF solution

We wish now to derive the Hartree-Fock equations using our second-quantized formalism and study the stability of the equations. Our ansatz for the ground state of the system is approximated as (this is our representation of a Slater determinant in second quantization)

$$|\Phi_0\rangle = |c\rangle = a_i^{\dagger} a_j^{\dagger} \dots a_l^{\dagger}|0\rangle.$$

We wish to determine $\hat{u}^{HF}$ so that $E_0^{HF} = \langle c|\hat{H}|c\rangle$ becomes a local minimum.

In our analysis here we will need Thouless' theorem, which states that an arbitrary Slater determinant $|c'\rangle$ which is not orthogonal to a determinant $|c\rangle = \prod_{i=1}^{n} a_{\alpha_i}^{\dagger}|0\rangle$, can be written as

$$|c'\rangle = exp\left\{\sum_{a>F}\sum_{i\le F} C_{ai}a_a^\dagger a_i\right\}|c\rangle$$

Let us give a simple proof of Thouless' theorem. The theorem states that we can make a linear combination av particle-hole excitations with respect to a given reference state $|c\rangle$. With this linear combination, we can make a new Slater determinant $|c'\rangle$ which is not orthogonal to $|c\rangle$, that is

$$\langle c|c'\rangle \ne 0.$$

To show this we need some intermediate steps. The exponential product of two operators $\exp\hat{A}\times\exp\hat{B}$ is equal to $\exp(\hat{A}+\hat{B})$ only if the two operators commute, that is

$$[\hat{A},\hat{B}] = 0.$$

Thouless' theorem

If the operators do not commute, we need to resort to the Baker-Campbell-Hauersdorf. This relation states that

$$\exp\hat{C} = \exp\hat{A}\exp\hat{B},$$

with

$$\hat{C} = \hat{A}+\hat{B}+\frac{1}{2}[\hat{A},\hat{B}]+\frac{1}{12}[[\hat{A},\hat{B}],\hat{B}]-\frac{1}{12}[[\hat{A},\hat{B}],\hat{A}]+\dots$$

From these relations, we note that in our expression for $|c'\rangle$ we have commutators of the type

$$[a_a^\dagger a_i, a_b^\dagger a_j],$$

and it is easy to convince oneself that these commutators, or higher powers thereof, are all zero. This means that we can write out our new representation of a Slater determinant as

$$|c'\rangle = exp\left\{\sum_{a>F}\sum_{i\le F} C_{ai}a_a^\dagger a_i\right\}|c\rangle = \prod_i\left\{1+\sum_{a>F}C_{ai}a_a^\dagger a_i+\left(\sum_{a>F}C_{ai}a_a^\dagger a_i\right)^2+\dots\right\}|c\rangle$$

We note that

$$\prod_i\sum_{a>F}C_{ai}a_a^\dagger a_i\sum_{b>F}C_{bi}a_b^\dagger a_i|c\rangle = 0,$$

and all higher-order powers of these combinations of creation and annihilation operators disappear due to the fact that $(a_i)^n|c\rangle = 0$ when $n>1$. This allows us to rewrite the expression for $|c'\rangle$ as

$$|c'\rangle = \prod_i\left\{1+\sum_{a>F}C_{ai}a_a^\dagger a_i\right\}|c\rangle,$$

which we can rewrite as

$$|c'\rangle = \prod_i\left\{1+\sum_{a>F}C_{ai}a_a^\dagger a_i\right\}|a_{i_1}^\dagger a_{i_2}^\dagger\dots a_{i_n}^\dagger|0\rangle.$$

The last equation can be written as

$$|c'\rangle = \prod_i \left\{ 1 + \sum_{a>F} C_{ai} a_a^\dagger a_i \right\} |a_{i_1}^\dagger a_{i_2}^\dagger \dots a_{i_n}^\dagger |0\rangle = \left(1 + \sum_{a>F} C_{ai_1} a_a^\dagger a_{i_1} \right) a_{i_1}^\dagger \tag{4.10}$$

$$\times \left(1 + \sum_{a>F} C_{ai_2} a_a^\dagger a_{i_2} \right) a_{i_2}^\dagger \dots |0\rangle = \prod_i \left(a_i^\dagger + \sum_{a>F} C_{ai} a_a^\dagger \right) |0\rangle. \tag{4.11}$$

New operators

If we define a new creation operator

$$b_i^\dagger = a_i^\dagger + \sum_{a>F} C_{ai} a_a^\dagger, \tag{4.12}$$

we have

$$|c'\rangle = \prod_i b_i^\dagger |0\rangle = \prod_i \left(a_i^\dagger + \sum_{a>F} C_{ai} a_a^\dagger \right) |0\rangle,$$

meaning that the new representation of the Slater determinant in second quantization, $|c'\rangle$, looks like our previous ones. However, this representation is not general enough since we have a restriction on the sum over single-particle states in Eq. (4.12). The single-particle states have all to be above the Fermi level. The question then is whether we can construct a general representation of a Slater determinant with a creation operator

$$\tilde{b}_i^\dagger = \sum_p f_{ip} a_p^\dagger,$$

where f_{ip} is a matrix element of a unitary matrix which transforms our creation and annihilation operators $a^\dagger$ and a to $\tilde{b}^\dagger$ and $\tilde{b}$. These new operators define a new representation of a Slater determinant as

$$|\tilde{c}\rangle = \prod_i \tilde{b}_i^\dagger |0\rangle.$$

Showing that $|\tilde{c}\rangle = |c'\rangle$

We need to show that $|\tilde{c}\rangle = |c'\rangle$. We need also to assume that the new state is not orthogonal to $|c\rangle$, that is $\langle c|\tilde{c}\rangle \neq 0$. From this it follows that

$$\langle c|\tilde{c}\rangle = \langle 0|a_{i_n} \dots a_{i_1} \left(\sum_{p=i_1}^{i_n} f_{i_1 p} a_p^\dagger \right) \left(\sum_{q=i_1}^{i_n} f_{i_2 q} a_q^\dagger \right) \dots \left(\sum_{t=i_1}^{i_n} f_{i_n t} a_t^\dagger \right) |0\rangle,$$

which is nothing but the determinant $det(f_{ip})$ which we can, using the intermediate normalization condition, normalize to one, that is

$$det(f_{ip}) = 1,$$

meaning that f has an inverse defined as (since we are dealing with orthogonal, and in our case unitary as well, transformations)

$$\sum_k f_{ik} f_{kj}^{-1} = \delta_{ij},$$

and

$$\sum_j f_{ij}^{-1} f_{jk} = \delta_{ik}.$$

Using these relations we can then define the linear combination of creation (and annihilation as well) operators as

$$\sum_i f_{ki}^{-1} \tilde{b}_i^\dagger = \sum_i f_{ki}^{-1} \sum_{p=i_1}^{\infty} f_{ip} a_p^\dagger = a_k^\dagger + \sum_i \sum_{p=i_{n+1}}^{\infty} f_{ki}^{-1} f_{ip} a_p^\dagger.$$

Defining

$$c_{kp} = \sum_{i \le F} f_{ki}^{-1} f_{ip},$$

we can redefine

$$a_k^\dagger + \sum_i \sum_{p=i_{n+1}}^{\infty} f_{ki}^{-1} f_{ip} a_p^\dagger = a_k^\dagger + \sum_{p=i_{n+1}}^{\infty} c_{kp} a_p^\dagger = b_k^\dagger,$$

our starting point. We have shown that our general representation of a Slater determinant

$$|\tilde{c}\rangle = \prod_i \tilde{b}_i^\dagger |0\rangle = |c'\rangle = \prod_i b_i^\dagger |0\rangle,$$

with

$$b_k^\dagger = a_k^\dagger + \sum_{p=i_{n+1}}^{\infty} c_{kp} a_p^\dagger.$$

This means that we can actually write an ansatz for the ground state of the system as a linear combination of terms which contain the ansatz itself $|c\rangle$ with an admixture from an infinity of one-particle-one-hole states. The latter has important consequences when we wish to interpret the Hartree-Fock equations and their stability. We can rewrite the new representation as

$$|c'\rangle = |c\rangle + |\delta c\rangle,$$

where $|\delta c\rangle$ can now be interpreted as a small variation. If we approximate this term with contributions from one-particle-one-hole (*1p-1h*) states only, we arrive at

$$|c'\rangle = \left(1 + \sum_{ai} \delta C_{ai} a_a^\dagger a_i\right) |c\rangle.$$

In our derivation of the Hartree-Fock equations we have shown that

$$\langle \delta c | \hat{H} | c \rangle = 0,$$

which means that we have to satisfy

$$\langle c | \sum_{ai} \delta C_{ai} \left\{ a_a^\dagger a_i \right\} \hat{H} | c \rangle = 0.$$

With this as a background, we are now ready to study the stability of the Hartree-Fock equations.

Hartree-Fock in second quantization and stability of HF solution

The variational condition for deriving the Hartree-Fock equations guarantees only that the expectation value $\langle c|\hat{H}|c\rangle$ has an extreme value, not necessarily a minimum. To figure out whether the extreme value we have found is a minimum, we can use second quantization to analyze our results and find a criterion for the above expectation value to a local minimum. We will use Thouless' theorem and show that

$$\frac{\langle c'|\hat{H}|c'\rangle}{\langle c'|c'\rangle} \geq \langle c|\hat{H}|c\rangle = E_0,$$

with

$$|c'\rangle = |c\rangle + |\delta c\rangle.$$

Using Thouless' theorem we can write out $|c'\rangle$ as

$$|c'\rangle = \exp\left\{\sum_{a>F}\sum_{i\leq F}\delta C_{ai}a_a^\dagger a_i\right\}|c\rangle \tag{4.13}$$

$$= \left\{1+\sum_{a>F}\sum_{i\leq F}\delta C_{ai}a_a^\dagger a_i + \frac{1}{2!}\sum_{ab>F}\sum_{ij\leq F}\delta C_{ai}\delta C_{bj}a_a^\dagger a_i a_b^\dagger a_j + \dots\right\} \tag{4.14}$$

where the amplitudes δC are small.

The norm of $|c'\rangle$ is given by (using the intermediate normalization condition $\langle c'|c\rangle = 1$)

$$\langle c'|c'\rangle = 1 + \sum_{a>F}\sum_{i\leq F}|\delta C_{ai}|^2 + O(\delta C_{ai}^3).$$

The expectation value for the energy is now given by (using the Hartree-Fock condition)

$$\langle c'|\hat{H}|c'\rangle = \langle c|\hat{H}|c\rangle + \sum_{ab>F}\sum_{ij\leq F}\delta C_{ai}^*\delta C_{bj}\langle c|a_i^\dagger a_a\hat{H}a_b^\dagger a_j|c\rangle +$$

$$\frac{1}{2!}\sum_{ab>F}\sum_{ij\leq F}\delta C_{ai}\delta C_{bj}\langle c|\hat{H}a_a^\dagger a_i a_b^\dagger a_j|c\rangle + \frac{1}{2!}\sum_{ab>F}\sum_{ij\leq F}\delta C_{ai}^*\delta C_{bj}^*\langle c|a_j^\dagger a_b a_i^\dagger a_a\hat{H}|c\rangle + \dots$$

We have already calculated the second term on the right-hand side of the previous equation

$$\langle c|\left(\{a_i^\dagger a_a\}\hat{H}\{a_b^\dagger a_j\}\right)|c\rangle = \sum_{pq}\sum_{ijab}\delta C_{ai}^*\delta C_{bj}\langle p|\hat{h}_0|q\rangle\langle c|\left(\{a_i^\dagger a_a\}\{a_p^\dagger a_q\}\{a_b^\dagger a_j\}\right)|c\rangle \tag{4.15}$$

$$+\frac{1}{4}\sum_{pqrs}\sum_{ijab}\delta C_{ai}^*\delta C_{bj}\langle pq|\hat{v}|rs\rangle\langle c|\left(\{a_i^\dagger a_a\}\{a_p^\dagger a_q^\dagger a_s a_r\}\{a_b^\dagger a_j\}\right)|c\rangle, \tag{4.16}$$

resulting in

$$E_0\sum_{ai}|\delta C_{ai}|^2 + \sum_{ai}|\delta C_{ai}|^2(\varepsilon_a - \varepsilon_i) - \sum_{ijab}\langle aj|\hat{v}|bi\rangle\delta C_{ai}^*\delta C_{bj}.$$

$$\frac{1}{2!}\langle c|\left(\{a_j^\dagger a_b\}\{a_i^\dagger a_a\}\hat{V}_N\right)|c\rangle = \frac{1}{2!}\langle c|\left(\hat{V}_N\{a_a^\dagger a_i\}\{a_b^\dagger a_j\}\right)^\dagger|c\rangle$$

which is nothing but

$$\frac{1}{2!}\langle c|\left(\hat{V}_N\{a_a^\dagger a_i\}\{a_b^\dagger a_j\}\right)|c\rangle^* = \frac{1}{2}\sum_{ijab}(\langle ij|\hat{v}|ab\rangle)^*\delta C_{ai}^*\delta C_{bj}^*$$

or

$$\frac{1}{2}\sum_{ijab}(\langle ab|\hat{v}|ij\rangle)\delta C_{ai}^*\delta C_{bj}^*$$

where we have used the relation

$$\langle a|\hat{A}|b\rangle = (\langle b|\hat{A}^\dagger|a\rangle)^*$$

due to the hermiticity of $\hat{H}$ and $\hat{V}$.

We define two matrix elements

$$A_{ai,bj} = -\langle aj|\hat{v}bi\rangle$$

and

$$B_{ai,bj} = \langle ab|\hat{v}|ij\rangle$$

both being anti-symmetrized.

With these definitions we write out the energy as

$$\langle c'|H|c'\rangle = \left(1+\sum_{ai}|\delta C_{ai}|^2\right)\langle c|H|c\rangle + \sum_{ai}|\delta C_{ai}|^2(\varepsilon_a^{HF}-\varepsilon_i^{HF}) + \sum_{ijab}A_{ai,bj}\delta C_{ai}^*\delta C_{bj} + \tag{4.17}$$

$$\frac{1}{2}\sum_{ijab}B_{ai,bj}^*\delta C_{ai}\delta C_{bj} + \frac{1}{2}\sum_{ijab}B_{ai,bj}\delta C_{ai}^*\delta C_{bj}^* + O(\delta C_{ai}^3), \tag{4.18}$$

which can be rewritten as

$$\langle c'|H|c'\rangle = \left(1+\sum_{ai}|\delta C_{ai}|^2\right)\langle c|H|c\rangle + \Delta E + O(\delta C_{ai}^3),$$

and skipping higher-order terms we arrived

$$\frac{\langle c'|\hat{H}|c'\rangle}{\langle c'|c'\rangle} = E_0 + \frac{\Delta E}{(1+\sum_{ai}|\delta C_{ai}|^2)}.$$

We have defined

$$\Delta E = \frac{1}{2}\langle \chi|\hat{M}|\chi\rangle$$

with the vectors

$$\chi = [\delta C \quad \delta C^*]^T$$

and the matrix

$$\hat{M} = \begin{pmatrix} \Delta + A & B \\ B^* & \Delta + A^* \end{pmatrix},$$

with $\Delta_{ai,bj} = (\varepsilon_a - \varepsilon_i)\delta_{ab}\delta_{ij}$.

The condition

$$\Delta E = \frac{1}{2}\langle \chi|\hat{M}|\chi\rangle \geq 0$$

for an arbitrary vector

$$\chi = [\delta C \quad \delta C^*]^T$$

means that all eigenvalues of the matrix have to be larger than or equal zero. A necessary (but no sufficient) condition is that the matrix elements (for all ai)

$$(\varepsilon_a - \varepsilon_i)\delta_{ab}\delta_{ij} + A_{ai,bj} \geq 0.$$

This equation can be used as a first test of the stability of the Hartree-Fock equation.

Chapter 5
Many-body perturbation theory

We assume here that we are only interested in the ground state of the system and expand the exact wave function in term of a series of Slater determinants

$$|\Psi_0\rangle = |\Phi_0\rangle + \sum_{m=1}^{\infty} C_m|\Phi_m\rangle,$$

where we have assumed that the true ground state is dominated by the solution of the unperturbed problem, that is

$$\hat{H}_0|\Phi_0\rangle = W_0|\Phi_0\rangle.$$

The state $|\Psi_0\rangle$ is not normalized, rather we have used an intermediate normalization $\langle\Phi_0|\Psi_0\rangle = 1$ since we have $\langle\Phi_0|\Phi_0\rangle = 1$.

The Schroedinger equation is

$$\hat{H}|\Psi_0\rangle = E|\Psi_0\rangle,$$

and multiplying the latter from the left with $\langle\Phi_0|$ gives

$$\langle\Phi_0|\hat{H}|\Psi_0\rangle = E\langle\Phi_0|\Psi_0\rangle = E,$$

and subtracting from this equation

$$\langle\Psi_0|\hat{H}_0|\Phi_0\rangle = W_0\langle\Psi_0|\Phi_0\rangle = W_0,$$

and using the fact that the both operators $\hat{H}$ and $\hat{H}_0$ are hermitian results in

$$\Delta E = E - W_0 = \langle\Phi_0|\hat{H}_I|\Psi_0\rangle,$$

which is an exact result. We call this quantity the correlation energy.

This equation forms the starting point for all perturbative derivations. However, as it stands it represents nothing but a mere formal rewriting of Schroedinger's equation and is not of much practical use. The exact wave function $|\Psi_0\rangle$ is unknown. In order to obtain a perturbative expansion, we need to expand the exact wave function in terms of the interaction $\hat{H}_I$.

Here we have assumed that our model space defined by the operator $\hat{P}$ is one-dimensional, meaning that

$$\hat{P} = |\Phi_0\rangle\langle\Phi_0|,$$

and

$$\hat{Q} = \sum_{m=1}^{\infty} |\Phi_m\rangle\langle\Phi_m|.$$

We can thus rewrite the exact wave function as

$$|\Psi_0\rangle = (\hat{P}+\hat{Q})|\Psi_0\rangle = |\Phi_0\rangle + \hat{Q}|\Psi_0\rangle.$$

Going back to the Schrödinger equation, we can rewrite it as, adding and a subtracting a term $\omega|\Psi_0\rangle$ as

$$\left(\omega - \hat{H}_0\right)|\Psi_0\rangle = \left(\omega - E + \hat{H}_I\right)|\Psi_0\rangle,$$

where ω is an energy variable to be specified later.

We assume also that the resolvent of $\left(\omega - \hat{H}_0\right)$ exits, that is it has an inverse which defined the unperturbed Green's function as

$$\left(\omega - \hat{H}_0\right)^{-1} = \frac{1}{\left(\omega - \hat{H}_0\right)}.$$

We can rewrite Schroedinger's equation as

$$|\Psi_0\rangle = \frac{1}{\omega - \hat{H}_0}\left(\omega - E + \hat{H}_I\right)|\Psi_0\rangle,$$

and multiplying from the left with $\hat{Q}$ results in

$$\hat{Q}|\Psi_0\rangle = \frac{\hat{Q}}{\omega - \hat{H}_0}\left(\omega - E + \hat{H}_I\right)|\Psi_0\rangle,$$

which is possible since we have defined the operator $\hat{Q}$ in terms of the eigenfunctions of $\hat{H}$.

These operators commute meaning that

$$\hat{Q}\frac{1}{\left(\omega - \hat{H}_0\right)}\hat{Q} = \hat{Q}\frac{1}{\left(\omega - \hat{H}_0\right)} = \frac{\hat{Q}}{\left(\omega - \hat{H}_0\right)}.$$

With these definitions we can in turn define the wave function as

$$|\Psi_0\rangle = |\Phi_0\rangle + \frac{\hat{Q}}{\omega - \hat{H}_0}\left(\omega - E + \hat{H}_I\right)|\Psi_0\rangle.$$

This equation is again nothing but a formal rewrite of Schrödinger's equation and does not represent a practical calculational scheme. It is a non-linear equation in two unknown quantities, the energy E and the exact wave function $|\Psi_0\rangle$. We can however start with a guess for $|\Psi_0\rangle$ on the right hand side of the last equation.

The most common choice is to start with the function which is expected to exhibit the largest overlap with the wave function we are searching after, namely $|\Phi_0\rangle$. This can again be inserted in the solution for $|\Psi_0\rangle$ in an iterative fashion and if we continue along these lines we end up with

$$|\Psi_0\rangle = \sum_{i=0}^{\infty}\left\{\frac{\hat{Q}}{\omega - \hat{H}_0}\left(\omega - E + \hat{H}_I\right)\right\}^i |\Phi_0\rangle,$$

for the wave function and

$$\Delta E = \sum_{i=0}^{\infty}\langle\Phi_0|\hat{H}_I\left\{\frac{\hat{Q}}{\omega - \hat{H}_0}\left(\omega - E + \hat{H}_I\right)\right\}^i |\Phi_0\rangle,$$

which is now a perturbative expansion of the exact energy in terms of the interaction $\hat{H}_I$ and the unperturbed wave function $|\Psi_0\rangle$.

In our equations for $|\Psi_0\rangle$ and ΔE in terms of the unperturbed solutions $|\Phi_i\rangle$ we have still an undetermined parameter ω and a dependecy on the exact energy E. Not much has been gained thus from a practical computational point of view.

In Brillouin-Wigner perturbation theory it is customary to set $\omega = E$. This results in the following perturbative expansion for the energy ΔE

$$\Delta E = \sum_{i=0}^{\infty} \langle \Phi_0 | \hat{H}_I \left\{ \frac{\hat{Q}}{\omega - \hat{H}_0} \left(\omega - E + \hat{H}_I \right) \right\}^i | \Phi_0 \rangle =$$

$$\langle \Phi_0 | \left(\hat{H}_I + \hat{H}_I \frac{\hat{Q}}{E - \hat{H}_0} \hat{H}_I + \hat{H}_I \frac{\hat{Q}}{E - \hat{H}_0} \hat{H}_I \frac{\hat{Q}}{E - \hat{H}_0} \hat{H}_I + \dots \right) | \Phi_0 \rangle.$$

$$\Delta E = \sum_{i=0}^{\infty} \langle \Phi_0 | \hat{H}_I \left\{ \frac{\hat{Q}}{\omega - \hat{H}_0} \left(\omega - E + \hat{H}_I \right) \right\}^i | \Phi_0 \rangle =$$

$$\langle \Phi_0 | \left(\hat{H}_I + \hat{H}_I \frac{\hat{Q}}{E - \hat{H}_0} \hat{H}_I + \hat{H}_I \frac{\hat{Q}}{E - \hat{H}_0} \hat{H}_I \frac{\hat{Q}}{E - \hat{H}_0} \hat{H}_I + \dots \right) | \Phi_0 \rangle.$$

This expression depends however on the exact energy E and is again not very convenient from a practical point of view. It can obviously be solved iteratively, by starting with a guess for E and then solve till some kind of self-consistency criterion has been reached.

Actually, the above expression is nothing but a rewrite again of the full Schrödinger equation.

Defining $e = E - \hat{H}_0$ and recalling that $\hat{H}_0$ commutes with $\hat{Q}$ by construction and that $\hat{Q}$ is an idempotent operator $\hat{Q}^2 = \hat{Q}$. Using this equation in the above expansion for ΔE we can write the denominator

$$\hat{Q} \frac{1}{\hat{e} - \hat{Q}\hat{H}_I\hat{Q}} =$$

$$\hat{Q} \left[\frac{1}{\hat{e}} + \frac{1}{\hat{e}} \hat{Q}\hat{H}_I\hat{Q} \frac{1}{\hat{e}} + \frac{1}{\hat{e}} \hat{Q}\hat{H}_I\hat{Q} \frac{1}{\hat{e}} \hat{Q}\hat{H}_I\hat{Q} \frac{1}{\hat{e}} + \dots \right] \hat{Q}.$$

Inserted in the expression for ΔE leads to

$$\Delta E = \langle \Phi_0 | \hat{H}_I + \hat{H}_I \hat{Q} \frac{1}{E - \hat{H}_0 - \hat{Q}\hat{H}_I\hat{Q}} \hat{Q}\hat{H}_I | \Phi_0 \rangle.$$

In RS perturbation theory we set $\omega = W_0$ and obtain the following expression for the energy difference

$$\Delta E = \sum_{i=0}^{\infty} \langle \Phi_0 | \hat{H}_I \left\{ \frac{\hat{Q}}{W_0 - \hat{H}_0} \left(\hat{H}_I - \Delta E \right) \right\}^i | \Phi_0 \rangle =$$

$$\langle \Phi_0 | \left(\hat{H}_I + \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} (\hat{H}_I - \Delta E) + \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} (\hat{H}_I - \Delta E) \frac{\hat{Q}}{W_0 - \hat{H}_0} (\hat{H}_I - \Delta E) + \dots \right) | \Phi_0 \rangle.$$

Recalling that $\hat{Q}$ commutes with $\hat{H}_0$ and since ΔE is a constant we obtain that

$$\hat{Q}\Delta E | \Phi_0 \rangle = \hat{Q}\Delta E | \hat{Q}\Phi_0 \rangle = 0.$$

Inserting this results in the expression for the energy results in

$$\Delta E = \langle \Phi_0 | \left(\hat{H}_I + \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} \hat{H}_I + \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} (\hat{H}_I - \Delta E) \frac{\hat{Q}}{W_0 - \hat{H}_0} \hat{H}_I + \dots \right) | \Phi_0 \rangle.$$

We can now this expression in terms of a perturbative expression in terms of $\hat{H}_I$ where we iterate the last expression in terms of ΔE

$$\Delta E = \sum_{i=1}^{\infty} \Delta E^{(i)}.$$

We get the following expression for $\Delta E^{(i)}$

$$\Delta E^{(1)} = \langle \Phi_0 | \hat{H}_I | \Phi_0 \rangle,$$

which is just the contribution to first order in perturbation theory,

$$\Delta E^{(2)} = \langle \Phi_0 | \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} \hat{H}_I | \Phi_0 \rangle,$$

which is the contribution to second order.

$$\Delta E^{(3)} = \langle \Phi_0 | \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} \hat{H}_I \Phi_0 \rangle - \langle \Phi_0 | \hat{H}_I \frac{\hat{Q}}{W_0 - \hat{H}_0} \langle \Phi_0 | \hat{H}_I | \Phi_0 \rangle \frac{\hat{Q}}{W_0 - \hat{H}_0} \hat{H}_I | \Phi_0 \rangle,$$

being the third-order contribution.

Interpreting the correlation energy and the wave operator

In the shell-model lectures we showed that we could rewrite the exact state function for say the ground state, as a linear expansion in terms of all possible Slater determinants. That is, we define the ansatz for the ground state as

$$|\Phi_0\rangle = \left(\prod_{i \le F} \hat{a}_i^\dagger \right) |0\rangle,$$

where the index i defines different single-particle states up to the Fermi level. We have assumed that we have N fermions. A given one-particle-one-hole ($1p1h$) state can be written as

$$|\Phi_i^a\rangle = \hat{a}_a^\dagger \hat{a}_i |\Phi_0\rangle,$$

while a $2p2h$ state can be written as

$$|\Phi_{ij}^{ab}\rangle = \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i |\Phi_0\rangle,$$

and a general $ApAh$ state as

$$|\Phi_{ijk\dots}^{abc\dots}\rangle = \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_c^\dagger \dots \hat{a}_k \hat{a}_j \hat{a}_i |\Phi_0\rangle.$$

We use letters $ijkl\dots$ for states below the Fermi level and $abcd\dots$ for states above the Fermi level. A general single-particle state is given by letters $pqrs\dots$.

We can then expand our exact state function for the ground state as

$$|\Psi_0\rangle = C_0|\Phi_0\rangle + \sum_{ai} C_i^a |\Phi_i^a\rangle + \sum_{abij} C_{ij}^{ab} |\Phi_{ij}^{ab}\rangle + \dots = (C_0 + \hat{C})|\Phi_0\rangle,$$

where we have introduced the so-called correlation operator

$$\hat{C} = \sum_{ai} C_i^a \hat{a}_a^\dagger \hat{a}_i + \sum_{abij} C_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i + \dots$$

Since the normalization of Ψ_0 is at our disposal and since C_0 is by hypothesis non-zero, we may arbitrarily set $C_0 = 1$ with corresponding proportional changes in all other coefficients. Using this so-called intermediate normalization we have

$$\langle \Psi_0 | \Phi_0 \rangle = \langle \Phi_0 | \Phi_0 \rangle = 1,$$

resulting in

$$|\Psi_0\rangle = (1+\hat{C})|\Phi_0\rangle.$$

In a shell-model calculation, the unknown coefficients in $\hat{C}$ are the eigenvectors which result from the diagonalization of the Hamiltonian matrix.

How can we use perturbation theory to determine the same coefficients? Let us study the contributions to second order in the interaction, namely

$$\Delta E^{(2)} = \langle\Phi_0|\hat{H}_I\frac{\hat{Q}}{W_0-\hat{H}_0}\hat{H}_I|\Phi_0\rangle.$$

The intermediate states given by $\hat{Q}$ can at most be of a $2p-2h$ nature if we have a two-body Hamiltonian. This means that second order in the perturbation theory can have $1p-1h$ and $2p-2h$ at most as intermediate states. When we diagonalize, these contributions are included to infinite order. This means that higher-orders in perturbation theory bring in more complicated correlations.

If we limit the attention to a Hartree-Fock basis, then we have that $\langle\Phi_0|\hat{H}_I|2p-2h\rangle$ is the only contribution and the contribution to the energy reduces to

$$\Delta E^{(2)} = \frac{1}{4}\sum_{abij}\langle ij|\hat{v}|ab\rangle\frac{\langle ab|\hat{v}|ij\rangle}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b}.$$

If we compare this to the correlation energy obtained from full configuration interaction theory with a Hartree-Fock basis, we found that

$$E-E_0 = \Delta E = \sum_{abij}\langle ij|\hat{v}|ab\rangle C_{ij}^{ab},$$

where the energy E_0 is the reference energy and ΔE defines the so-called correlation energy.

We see that if we set

$$C_{ij}^{ab} = \frac{1}{4}\frac{\langle ab|\hat{v}|ij\rangle}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b},$$

we have a perfect agreement between FCI and MBPT. However, FCI includes such $2p-2h$ correlations to infinite order. In order to make a meaningful comparison we would at least need to sum such correlations to infinite order in perturbation theory.

Summing up, we can see that

- MBPT introduces order-by-order specific correlations and we make comparisons with exact calculations like FCI
- At every order, we can calculate all contributions since they are well-known and either tabulated or calculated on the fly.
- MBPT is a non-variational theory and there is no guarantee that higher orders will improve the convergence.
- However, since FCI calculations are limited by the size of the Hamiltonian matrices to diagonalize (today's most efficient codes can attach dimensionalities of ten billion basis states, MBPT can function as an approximative method which gives a straightforward (but tedious) calculation recipe.
- MBPT has been widely used to compute effective interactions for the nuclear shell-model.
- But there are better methods which sum to infinite order important correlations. Coupled cluster theory is one of these methods.

Part III
Monte Carlo Methods

Chapter 6
Variational Monte Carlo methods

Quantum Monte Carlo Motivation

We start with the variational principle. Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

$$E[H] = \langle H \rangle = \frac{\int dR \Psi_T^*(R) H(R) \Psi_T(R)}{\int dR \Psi_T^*(R) \Psi_T(R)},$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \le \langle H \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system.

The trial wave function can be expanded in the eigenstates of the hamiltonian since they form a complete set, viz.,

$$\Psi_T(R) = \sum_i a_i \Psi_i(R),$$

and assuming the set of eigenfunctions to be normalized one obtains

$$\frac{\sum_{nm} a_m^* a_n \int dR \Psi_m^*(R) H(R) \Psi_n(R)}{\sum_{nm} a_m^* a_n \int dR \Psi_m^*(R) \Psi_n(R)} = \frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \ge E_0,$$

where we used that $H(R)\Psi_n(R) = E_n \Psi_n(R)$. In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. The variational principle yields the lowest state of a given symmetry.

In most cases, a wave function has only small values in large parts of configuration space, and a straightforward procedure which uses homogenously distributed random points in configuration space will most likely lead to poor results. This may suggest that some kind of importance sampling combined with e.g., the Metropolis algorithm may be a more efficient way of obtaining the ground state energy. The hope is then that those regions of configurations space where the wave function assumes appreciable values are sampled more efficiently.

The tedious part in a VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). DMC is a way of solving exactly the many-body Schroedinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

The basic recipe in a VMC calculation consists of the following elements:

- Construct first a trial wave function $\psi_T(R,\alpha)$, for a many-body system consisting of N particles located at positions $R=(R_1,\dots,R_N)$. The trial wave function depends on α variational parameters $\alpha=(\alpha_1,\dots,\alpha_M)$.
- Then we evaluate the expectation value of the hamiltonian H
$$E[H]=\langle H\rangle=\frac{\int dR\Psi_T^*(R,\alpha)H(R)\Psi_T(R,\alpha)}{\int dR\Psi_T^*(R,\alpha)\Psi_T(R,\alpha)}.$$
- Thereafter we vary α according to some minimization algorithm and return to the first step.

With a trial wave function $\psi_T(R)$ we can in turn construct the quantum mechanical probability distribution
$$P(R)=\frac{|\psi_T(R)|^2}{\int|\psi_T(R)|^2dR}.$$
This is our new probability distribution function (PDF). The approximation to the expectation value of the Hamiltonian is now
$$E[H(\alpha)]=\frac{\int dR\Psi_T^*(R,\alpha)H(R)\Psi_T(R,\alpha)}{\int dR\Psi_T^*(R,\alpha)\Psi_T(R,\alpha)}.$$

Define a new quantity
$$E_L(R,\alpha)=\frac{1}{\psi_T(R,\alpha)}H\psi_T(R,\alpha),$$
called the local energy, which, together with our trial PDF yields
$$E[H(\alpha)]=\int P(R)E_L(R)dR\approx\frac{1}{N}\sum_{i=1}^{N}P(R_i,\alpha)E_L(R_i,\alpha)$$
with N being the number of Monte Carlo samples.

The Algorithm for performing a variational Monte Carlo calculations runs thus as this

- Initialisation: Fix the number of Monte Carlo steps. Choose an initial R and variational parameters α and calculate $|\psi_T^{\alpha}(R)|^2$.
- Initialise the energy and the variance and start the Monte Carlo calculation.
  - Calculate a trial position $R_p=R+r*step$ where r is a random variable $r\in[0,1]$.
  - Metropolis algorithm to accept or reject this move $w=P(R_p)/P(R)$.
  - If the step is accepted, then we set $R=R_p$.
  - Update averages
- Finish and compute final averages.

Observe that the jumping in space is governed by the variable *step*. This is Called brute-force sampling. Need importance sampling to get more relevant sampling, see lectures below.

Quantum Monte Carlo: hydrogen atom.

The radial Schroedinger equation for the hydrogen atom can be written as
$$-\frac{^2}{2m}\frac{\partial^2u(r)}{\partial r^2}-\left(\frac{ke^2}{r}-\frac{^2l(l+1)}{2mr^2}\right)u(r)=Eu(r),$$

or with dimensionless variables

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{u(\rho)}{\rho} + \frac{l(l+1)}{2\rho^2}u(\rho) - \lambda u(\rho) = 0,$$

with the hamiltonian

$$H = -\frac{1}{2}\frac{\partial^2}{\partial \rho^2} - \frac{1}{\rho} + \frac{l(l+1)}{2\rho^2}.$$

Use variational parameter α in the trial wave function

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho}.$$

Inserting this wave function into the expression for the local energy E_L gives

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right).$$

A simple variational Monte Carlo calculation results in

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03

We note that at $\alpha = 1$ we obtain the exact result, and the variance is zero, as it should. The reason is that we then have the exact wave function, and the action of the hamiltionan on the wave function

$$H\psi = \text{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle H^n \rangle = \frac{\int dR \Psi_T^*(R) H^n(R) \Psi_T(R)}{\int dR \Psi_T^*(R) \Psi_T(R)} = \text{constant} \times \frac{\int dR \Psi_T^*(R) \Psi_T(R)}{\int dR \Psi_T^*(R) \Psi_T(R)} = \text{constant}.$$

This gives an important information: the exact wave function leads to zero variance! Variation is then performed by minimizing both the energy and the variance.

For bosons in a harmonic oscillator-like trap we will use is a spherical (S) or an elliptical (E) harmonic trap in one, two and finally three dimensions, with the latter given by

$$V_{ext}(\mathbf{r}) = \begin{cases} \frac{1}{2}m\omega_{ho}^2 r^2 & (S) \\ \frac{1}{2}m[\omega_{ho}^2(x^2+y^2) + \omega_z^2 z^2] & (E) \end{cases} \tag{6.1}$$

where (S) stands for symmetric and

$$\hat{H} = \sum_i^N \left(\frac{-^2}{2m}\bigtriangledown_i^2 + V_{ext}(\mathbf{r}_i)\right) + \sum_{i<j}^N V_{int}(\mathbf{r}_i, \mathbf{r}_j), \tag{6.2}$$

as the two-body Hamiltonian of the system.

We will represent the inter-boson interaction by a pairwise, repulsive potential

$$V_{int}(|\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} \infty & |\mathbf{r}_i - \mathbf{r}_j| \le a \\ 0 & |\mathbf{r}_i - \mathbf{r}_j| > a \end{cases} \tag{6.3}$$

where a is the so-called hard-core diameter of the bosons. Clearly, $V_{int}(|\mathbf{r}_i - \mathbf{r}_j|)$ is zero if the bosons are separated by a distance $|\mathbf{r}_i - \mathbf{r}_j|$ greater than a but infinite if they attempt to come within a distance $|\mathbf{r}_i - \mathbf{r}_j| \leq a$.

Our trial wave function for the ground state with N atoms is given by

$$\Psi_T(\mathbf{R}) = \Psi_T(\mathbf{r}_1, \mathbf{r}_2, \dots \mathbf{r}_N, \alpha, \beta) = \prod_i g(\alpha, \beta, \mathbf{r}_i) \prod_{i<j} f(a, |\mathbf{r}_i - \mathbf{r}_j|), \tag{6.4}$$

where α and β are variational parameters. The single-particle wave function is proportional to the harmonic oscillator function for the ground state

$$g(\alpha, \beta, \mathbf{r}_i) = \exp\left[-\alpha(x_i^2 + y_i^2 + \beta z_i^2)\right]. \tag{6.5}$$

For spherical traps we have $\beta = 1$ and for non-interacting bosons ($a = 0$) we have $\alpha = 1/2a_{ho}^2$. The correlation wave function is

$$f(a, |\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} 0 & |\mathbf{r}_i - \mathbf{r}_j| \leq a \\ (1 - \frac{a}{|\mathbf{r}_i - \mathbf{r}_j|}) & |\mathbf{r}_i - \mathbf{r}_j| > a. \end{cases} \tag{6.6}$$

A simple Python code that solves the two-boson or two-fermion case in two-dimensions.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python Importing various packages from math import exp, sqrt from random import random, seed import numpy as np import matplotlib.pyplot as plt from mpl$_{toolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotli}$

Trial wave function for quantum dots in two dims def WaveFunction(r,alpha,beta): r1 = r[0,0]**2 + r[0,1]**2 r2 = r[1,0]**2 + r[1,1]**2 r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = r12/(1+beta*r12) return exp(-0.5*alpha*(r1+r2)+deno)

Local energy for quantum dots in two dims, using analytical local energy def LocalEnergy(r,alpha,beta):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) deno2 = deno*deno return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1.0/r12)

The Monte Carlo sampling with the Metropolis algo def MonteCarloSampling():

NumberMCcycles= 100000 StepSize = 1.0 positions PositionOld = np.zeros((NumberParticles,Dimension), np.double) PositionNew = np.zeros((NumberParticles,Dimension), np.double) seed for rng generator seed() start variational parameter alpha = 0.9 for ia in range(MaxVariations): alpha += .025 AlphaValues[ia] = alpha beta = 0.2 for jb in range(MaxVariations): beta += .01 BetaValues[jb] = beta energy = energy2 = 0.0 DeltaE = 0.0 Initial position for i in range(NumberParticles): for j in range(Dimension): PositionOld[i,j] = StepSize * (random() - .5) wfold = WaveFunction(PositionOld,alpha,beta)

Loop over MC MCcycles for MCcycle in range(NumberMCcycles): Trial position for i in range(NumberParticles): for j in range(Dimension): PositionNew[i,j] = PositionOld[i,j] + StepSize * (random() - .5) wfnew = WaveFunction(PositionNew,alpha,beta)

Metropolis test to see whether we accept the move if random() < wfnew**2 / wfold**2: PositionOld = PositionNew.copy() wfold = wfnew DeltaE = LocalEnergy(PositionOld,alpha,beta) energy += DeltaE energy2 += DeltaE**2

We calculate mean, variance and error ... energy /= NumberMCcycles energy2 /= NumberMCcycles variance = energy2 - energy**2 error = sqrt(variance/NumberMCcycles) Energies[ia,jb] = energy return Energies, AlphaValues, BetaValues

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 MaxVariations = 10 Energies = np.zeros((MaxVariations,MaxVariations)) AlphaValues =

np.zeros(MaxVariations) BetaValues = np.zeros(MaxVariations) (Energies, AlphaValues, BetaValues) = MonteCarloSampling()

Prepare for plots fig = plt.figure() ax = fig.gca(projection='3d') Plot the surface. X, Y = np.meshgrid(AlphaValues, BetaValues) surf = ax.plot$_{s}urface(X,Y,Energies,cmap=cm.coolwarm,linewidth=0,antialiased=False)Customizethezaxis.zmin=np.matrix(Energies).min()zmax=np.matrix(Energies).max()ax.set_{z}lim(zmin,zmax)ax.set_{x}lab$ ax.set$_{y}label(r'\beta')$ ax.set$_{z}label(r'\langle E\rangle')$ ax.zaxis.set$_{m}ajor_{l}ocator(LinearLocator(10))ax.zaxis.set_{m}ajor_{f}ormatter(FormatStrFormatter('A$ $0.5,aspect=5)plt.show()$

Quantum Monte Carlo: the helium atom

The helium atom consists of two electrons and a nucleus with charge $Z=2$. The contribution to the potential energy due to the attraction from the nucleus is

$$-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2},$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1,r_2)=-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}+\frac{ke^2}{r_{12}},$$

with the electrons separated at a distance $r_{12}=|r_1-r_2|$.

The hamiltonian becomes then

$$\hat{H}=-\frac{{}^2\nabla_1^2}{2m}-\frac{{}^2\nabla_2^2}{2m}-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}+\frac{ke^2}{r_{12}},$$

and Schroedingers equation reads

$$\hat{H}\psi=E\psi.$$

All observables are evaluated with respect to the probability distribution

$$P(R)=\frac{|\psi_T(R)|^2}{\int|\psi_T(R)|^2\,dR}.$$

generated by the trial wave function. The trial wave function must approximate an exact eigenstate in order that accurate results are to be obtained.

Choice of trial wave function for Helium: Assume $r_1\to 0$.

$$E_L(R)=\frac{1}{\psi_T(R)}H\psi_T(R)=\frac{1}{\psi_T(R)}\left(-\frac{1}{2}\nabla_1^2-\frac{Z}{r_1}\right)\psi_T(R)+\text{finite terms}.$$

$$E_L(R)=\frac{1}{\mathbf{R}_T(r_1)}\left(-\frac{1}{2}\frac{d^2}{dr_1^2}-\frac{1}{r_1}\frac{d}{dr_1}-\frac{Z}{r_1}\right)\mathbf{R}_T(r_1)+\text{finite terms}$$

For small values of r_1, the terms which dominate are

$$\lim_{r_1\to 0}E_L(R)=\frac{1}{\mathbf{R}_T(r_1)}\left(-\frac{1}{r_1}\frac{d}{dr_1}-\frac{Z}{r_1}\right)\mathbf{R}_T(r_1),$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin.

This results in

$$\frac{1}{\mathbf{R}_T(r_1)}\frac{d\mathbf{R}_T(r_1)}{dr_1}=-Z,$$

and

$$\mathbf{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have

$$\frac{1}{\mathbf{R}_T(r)} \frac{d\mathbf{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Similarly, studying the case $r_{12} \to 0$ we can write a possible trial wave function as

$$\psi_T(R) = e^{-\alpha(r_1+r_2)} e^{\beta r_{12}}.$$

The last equation can be generalized to

$$\psi_T(R) = \phi(r_1)\phi(r_2)\dots\phi(r_N)\prod_{i<j} f(r_{ij}),$$

for a system with N electrons or particles.

During the development of our code we need to make several checks. It is also very instructive to compute a closed form expression for the local energy. Since our wave function is rather simple it is straightforward to find an analytic expressions. Consider first the case of the simple helium function

$$\Psi_T(r_1,r_2) = e^{-\alpha(r_1+r_2)}$$

The local energy is for this case

$$E_{L1} = (\alpha - Z)\left(\frac{1}{r_1} + \frac{1}{r_2}\right) + \frac{1}{r_{12}} - \alpha^2$$

which gives an expectation value for the local energy given by

$$\langle E_{L1} \rangle = \alpha^2 - 2\alpha\left(Z - \frac{5}{16}\right)$$

With closed form formulae we can speed up the computation of the correlation. In our case we write it as

$$\Psi_C = \exp\left\{\sum_{i<j} \frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

which means that the gradient needed for the so-called quantum force and local energy can be calculated analytically. This will speed up your code since the computation of the correlation part and the Slater determinant are the most time consuming parts in your code.

We will refer to this correlation function as Ψ_C or the *linear Pade-Jastrow*.

We can test this by computing the local energy for our helium wave function

$$\psi_T(r_1,r_2) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right),$$

with α and β as variational parameters.

The local energy is for this case

$$E_{L2} = E_{L1} + \frac{1}{2(1+\beta r_{12})^2}\left\{\frac{\alpha(r_1+r_2)}{r_{12}}(1 - \frac{r_1 r_2}{r_1 r_2}) - \frac{1}{2(1+\beta r_{12})^2} - \frac{2}{r_{12}} + \frac{2\beta}{1+\beta r_{12}}\right\}$$

It is very useful to test your code against these expressions. It means also that you don't need to compute a derivative numerically as discussed in the code example below.

For the computation of various derivatives with different types of wave functions, you will find it useful to use python with symbolic python, that is sympy, see online manual. Using sympy allows you autogenerate both Latex code as well c++, python or Fortran codes. Here you will find some simple examples. We choose the $2s$ hydrogen-orbital (not normalized) as an example

$$\phi_{2s}(r) = (Zr-2)\exp -(\frac{1}{2}Zr),$$

with $r^2 = x^2+y^2+z^2$.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sympy import symbols, diff, exp, sqrt x, y, z, Z = symbols('x y z Z') r = sqrt(x*x + y*y + z*z) r phi = (Z*r - 2)*exp(-Z*r/2) phi diff(phi, x)

This doesn't look very nice, but sympy provides several functions that allow for improving and simplifying the output.

We can improve our output by factorizing and substituting expressions

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing x, y, z, Z = symbols('x y z Z') r = sqrt(x*x + y*y + z*z) phi = (Z*r - 2)*exp(-Z*r/2) R = Symbol('r') Creates a symbolic equivalent of r print latex and c++ code print printing.latex(diff(phi, x).factor().subs(r, R)) print printing.ccode(diff(phi, x).factor().subs(r, R))

We can in turn look at second derivatives

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sympy import symbols, diff, exp, sqrt, factor, Symbol, printing x, y, z, Z = symbols('x y z Z') r = sqrt(x*x + y*y + z*z) phi = (Z*r - 2)*exp(-Z*r/2) R = Symbol('r') Creates a symbolic equivalent of r (diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().subs(r, R) Collect the Z values (diff(diff(phi, x), x) + diff(diff(phi, y), y) +diff(diff(phi, z), z)).factor().collect(Z).subs(r, R) Factorize also the r**2 terms (diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor() print printing.ccode((diff(diff(phi, x), x) + diff(diff(phi, y), y) + diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor())

With some practice this allows one to be able to check one's own calculation and translate automatically into code lines.

The Metropolis algorithm

The Metropolis algorithm , see the original article was invented by Metropolis et. al and is often simply called the Metropolis algorithm. It is a method to sample a normalized probability distribution by a stochastic process. We define $\mathbf{P}_i^{(n)}$ to be the probability for finding the system in the state i at step n. The algorithm is then

- Sample a possible new state j with some probability $T_{i\rightarrow j}$.
- Accept the new state j with probability $A_{i\rightarrow j}$ and use it as the next sample. With probability $1-A_{i\rightarrow j}$ the move is rejected and the original state i is used again as a sample.

We wish to derive the required properties of T and A such that $\mathbf{P}_i^{(n\rightarrow\infty)} \rightarrow p_i$ so that starting from any distribution, the method converges to the correct distribution. Note that the description here is for a discrete probability distribution. Replacing probabilities p_i with expressions like $p(x_i)dx_i$ will take all of these over to the corresponding continuum expressions.

The dynamical equation for $\mathbf{P}_i^{(n)}$ can be written directly from the description above. The probability of being in the state i at step n is given by the probability of being in any state j

at the previous step, and making an accepted transition to i added to the probability of being in the state i, making a transition to any state j and rejecting the move:

$$\mathbf{P}_i^{(n)} = \sum_j \left[\mathbf{P}_j^{(n-1)} T_{j\to i} A_{j\to i} + \mathbf{P}_i^{(n-1)} T_{i\to j} \left(1 - A_{i\to j}\right) \right] .$$

Since the probability of making some transition must be 1, $\sum_j T_{i\to j} = 1$, and the above equation becomes

$$\mathbf{P}_i^{(n)} = \mathbf{P}_i^{(n-1)} + \sum_j \left[\mathbf{P}_j^{(n-1)} T_{j\to i} A_{j\to i} - \mathbf{P}_i^{(n-1)} T_{i\to j} A_{i\to j} \right] .$$

For large n we require that $\mathbf{P}_i^{(n\to\infty)} = p_i$, the desired probability distribution. Taking this limit, gives the balance requirement

$$\sum_j \left[p_j T_{j\to i} A_{j\to i} - p_i T_{i\to j} A_{i\to j} \right] = 0 .$$

The balance requirement is very weak. Typically the much stronger detailed balance requirement is enforced, that is rather than the sum being set to zero, we set each term separately to zero and use this to determine the acceptance probabilities. Rearranging, the result is

$$\frac{A_{j\to i}}{A_{i\to j}} = \frac{p_i T_{i\to j}}{p_j T_{j\to i}} .$$

The Metropolis choice is to maximize the A values, that is

$$A_{j\to i} = \min\left(1, \frac{p_i T_{i\to j}}{p_j T_{j\to i}}\right) .$$

Other choices are possible, but they all correspond to multilplying $A_{i\to j}$ and $A_{j\to i}$ by the same constant smaller than unity.[1]

Having chosen the acceptance probabilities, we have guaranteed that if the $\mathbf{P}_i^{(n)}$ has equilibrated, that is if it is equal to p_i, it will remain equilibrated. Next we need to find the circumstances for convergence to equilibrium.

The dynamical equation can be written as

$$\mathbf{P}_i^{(n)} = \sum_j M_{ij} \mathbf{P}_j^{(n-1)}$$

with the matrix M given by

$$M_{ij} = \delta_{ij} \left[1 - \sum_k T_{i\to k} A_{i\to k} \right] + T_{j\to i} A_{j\to i} .$$

Summing over i shows that $\sum_i M_{ij} = 1$, and since $\sum_k T_{i\to k} = 1$, and $A_{i\to k} \leq 1$, the elements of the matrix satisfy $M_{ij} \geq 0$. The matrix M is therefore a stochastic matrix.

The Metropolis method is simply the power method for computing the right eigenvector of M with the largest magnitude eigenvalue. By construction, the correct probability distribution is a right eigenvector with eigenvalue 1. Therefore, for the Metropolis method to converge to this result, we must show that M has only one eigenvalue with this magnitude, and all other eigenvalues are smaller.

[1] The penalty function method uses just such a factor to compensate for p_i that are evaluated stochastically and are therefore noisy.

Importance sampling

We need to replace the brute force Metropolis algorithm with a walk in coordinate space biased by the trial wave function. This approach is based on the Fokker-Planck equation and the Langevin equation for generating a trajectory in coordinate space. The link between the Fokker-Planck equation and the Langevin equations are explained, only partly, in the slides below. An excellent reference on topics like Brownian motion, Markov chains, the Fokker-Planck equation and the Langevin equation is the text by Van Kampen Here we will focus first on the implementation part first.

For a diffusion process characterized by a time-dependent probability density $P(x,t)$ in one dimension the Fokker-Planck equation reads (for one particle /walker)

$$\frac{\partial P}{\partial t} = D\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x} - F\right)P(x,t),$$

where F is a drift term and D is the diffusion coefficient.

The new positions in coordinate space are given as the solutions of the Langevin equation using Euler's method, namely, we go from the Langevin equation

$$\frac{\partial x(t)}{\partial t} = DF(x(t)) + \eta,$$

with η a random variable, yielding a new position

$$y = x + DF(x)\Delta t + \xi\sqrt{\Delta t},$$

where ξ is gaussian random variable and Δt is a chosen time step. The quantity D is, in atomic units, equal to $1/2$ and comes from the factor $1/2$ in the kinetic energy operator. Note that Δt is to be viewed as a parameter. Values of $\Delta t \in [0.001, 0.01]$ yield in general rather stable values of the ground state energy.

The process of isotropic diffusion characterized by a time-dependent probability density $P(\mathbf{x},t)$ obeys (as an approximation) the so-called Fokker-Planck equation

$$\frac{\partial P}{\partial t} = \sum_i D\frac{\partial}{\partial \mathbf{x_i}}\left(\frac{\partial}{\partial \mathbf{x_i}} - \mathbf{F_i}\right)P(\mathbf{x},t),$$

where $\mathbf{F_i}$ is the i^{th} component of the drift term (drift velocity) caused by an external potential, and D is the diffusion coefficient. The convergence to a stationary probability density can be obtained by setting the left hand side to zero. The resulting equation will be satisfied if and only if all the terms of the sum are equal zero,

$$\frac{\partial^2 P}{\partial {\mathbf{x_i}}^2} = P\frac{\partial}{\partial \mathbf{x_i}}\mathbf{F_i} + \mathbf{F_i}\frac{\partial}{\partial \mathbf{x_i}}P.$$

The drift vector should be of the form $\mathbf{F} = g(\mathbf{x})\frac{\partial P}{\partial \mathbf{x}}$. Then,

$$\frac{\partial^2 P}{\partial {\mathbf{x_i}}^2} = P\frac{\partial g}{\partial P}\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2 + Pg\frac{\partial^2 P}{\partial \mathbf{x}_i^2} + g\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2.$$

The condition of stationary density means that the left hand side equals zero. In other words, the terms containing first and second derivatives have to cancel each other. It is possible only if $g = \frac{1}{P}$, which yields

$$\mathbf{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T,$$

which is known as the so-called *quantum force*. This term is responsible for pushing the walker towards regions of configuration space where the trial wave function is large, increasing the efficiency of the simulation in contrast to the Metropolis algorithm where the walker has the same probability of moving in every direction.

The Fokker-Planck equation yields a (the solution to the equation) transition probability given by the Green's function

$$G(y,x,\Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}} \exp\left(-(y-x-D\Delta t F(x))^2/4D\Delta t\right)$$

which in turn means that our brute force Metropolis algorithm

$$A(y,x) = \min(1, q(y,x))),$$

with $q(y,x) = |\Psi_T(y)|^2/|\Psi_T(x)|^2$ is now replaced by the Metropolis-Hastings algorithm as well as Hasting's article,

$$q(y,x) = \frac{G(x,y,\Delta t)|\Psi_T(y)|^2}{G(y,x,\Delta t)|\Psi_T(x)|^2}$$

Importance sampling, program elements

The general derivative formula of the Jastrow factor is (the subscript C stands for Correlation)

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_k}$$

However, with our written in way which can be reused later as

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. The function $f(r_{ij})$ will depends on the system under study. In the equations below we will keep this general form.

In the Metropolis/Hasting algorithm, the *acceptance ratio* determines the probability for a particle to be accepted at a new position. The ratio of the trial wave functions evaluated at the new and current positions is given by (OB for the onebody part)

$$R \equiv \frac{\Psi_T^{new}}{\Psi_T^{old}} = \frac{\Psi_{OB}^{new}}{\Psi_{OB}^{old}}\frac{\Psi_C^{new}}{\Psi_C^{old}}$$

Here Ψ_{OB} is our onebody part (Slater determinant or product of boson single-particle states) while Ψ_C is our correlation function, or Jastrow factor. We need to optimize the $\nabla\Psi_T/\Psi_T$ ratio and the second derivative as well, that is the $\nabla^2\Psi_T/\Psi_T$ ratio. The first is needed when we compute the so-called quantum force in importance sampling. The second is needed when we compute the kinetic energy term of the local energy.

$$\frac{\nabla\Psi}{\Psi} = \frac{\nabla(\Psi_{OB}\Psi_C)}{\Psi_{OB}\Psi_C} = \frac{\Psi_C\nabla\Psi_{OB} + \Psi_{OB}\nabla\Psi_C}{\Psi_{OB}\Psi_C} = \frac{\nabla\Psi_{OB}}{\Psi_{OB}} + \frac{\nabla\Psi_C}{\Psi_C}$$

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle \hat{K}_i \rangle = -\frac{1}{2}\frac{\langle \Psi | \nabla_i^2 | \Psi \rangle}{\langle \Psi | \Psi \rangle},$$

$$\hat{K}_i = -\frac{1}{2}\frac{\nabla_i^2 \Psi}{\Psi}.$$

The second derivative which enters the definition of the local energy is

$$\frac{\nabla^2 \Psi}{\Psi} = \frac{\nabla^2 \Psi_{OB}}{\Psi_{OB}} + \frac{\nabla^2 \Psi_C}{\Psi_C} + 2\frac{\nabla \Psi_{OB}}{\Psi_{OB}} \cdot \frac{\nabla \Psi_C}{\Psi_C}$$

We discuss here how to calculate these quantities in an optimal way,

We have defined the correlated function as

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \prod_{i<j}^{N} g(r_{ij}) = \prod_{i=1}^{N}\prod_{j=i+1}^{N} g(r_{ij}),$$

with $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ in three dimensions or $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ if we work with two-dimensional systems.

In our particular case we have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j} f(r_{ij})\right\}.$$

The total number of different relative distances r_{ij} is $N(N-1)/2$. In a matrix storage format, the relative distances form a strictly upper triangular matrix

$$\mathbf{r} \equiv \begin{pmatrix} 0 & r_{1,2} & r_{1,3} & \cdots & r_{1,N} \\ \vdots & 0 & r_{2,3} & \cdots & r_{2,N} \\ \vdots & \vdots & 0 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & r_{N-1,N} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

This applies to $\mathbf{g} = \mathbf{g}(r_{ij})$ as well.

In our algorithm we will move one particle at the time, say the *kth*-particle. This sampling will be seen to be particularly efficient when we are going to compute a Slater determinant.

We have that the ratio between Jastrow factors R_C is given by

$$R_C = \frac{\Psi_C^{\mathrm{new}}}{\Psi_C^{\mathrm{cur}}} = \prod_{i=1}^{k-1}\frac{g_{ik}^{\mathrm{new}}}{g_{ik}^{\mathrm{cur}}}\prod_{i=k+1}^{N}\frac{g_{ki}^{\mathrm{new}}}{g_{ki}^{\mathrm{cur}}}.$$

For the Pade-Jastrow form

$$R_C = \frac{\Psi_C^{\mathrm{new}}}{\Psi_C^{\mathrm{cur}}} = \frac{\exp U_{new}}{\exp U_{cur}} = \exp \Delta U,$$

where

$$\Delta U = \sum_{i=1}^{k-1}\left(f_{ik}^{\mathrm{new}} - f_{ik}^{\mathrm{cur}}\right) + \sum_{i=k+1}^{N}\left(f_{ki}^{\mathrm{new}} - f_{ki}^{\mathrm{cur}}\right)$$

One needs to develop a special algorithm that runs only through the elements of the upper triangular matrix $\mathbf{g}$ and have k as an index.

The expression to be derived in the following is of interest when computing the quantum force and the kinetic energy. It has the form

$$\frac{\nabla_i \Psi_C}{\Psi_C} = \frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_i},$$

for all dimensions and with i running over all particles.

For the first derivative only $N-1$ terms survive the ratio because the g-terms that are not differentiated cancel with their corresponding ones in the denominator. Then,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_k}.$$

An equivalent equation is obtained for the exponential form after replacing g_{ij} by $\exp(f_{ij})$, yielding:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_k},$$

with both expressions scaling as $\mathcal{O}(N)$.

Using the identity

$$\frac{\partial}{\partial x_i} g_{ij} = -\frac{\partial}{\partial x_j} g_{ij},$$

we get expressions where all the derivatives acting on the particle are represented by the *second* index of g:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_i},$$

and for the exponential case:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}.$$

For correlation forms depending only on the scalar distances r_{ij} we can use the chain rule. Noting that

$$\frac{\partial g_{ij}}{\partial x_j} = \frac{\partial g_{ij}}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x_j} = \frac{x_j - x_i}{r_{ij}}\frac{\partial g_{ij}}{\partial r_{ij}},$$

we arrive at

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial g_{ki}}{\partial r_{ki}}.$$

Note that for the Pade-Jastrow form we can set $g_{ij} \equiv g(r_{ij}) = e^{f(r_{ij})} = e^{f_{ij}}$ and

$$\frac{\partial g_{ij}}{\partial r_{ij}} = g_{ij}\frac{\partial f_{ij}}{\partial r_{ij}}.$$

Therefore,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial f_{ki}}{\partial r_{ki}},$$

where

$$\mathbf{r}_{ij} = |\mathbf{r}_j - \mathbf{r}_i| = (x_j - x_i)\mathbf{e}_1 + (y_j - y_i)\mathbf{e}_2 + (z_j - z_i)\mathbf{e}_3$$

is the relative distance.

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2 \Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}),$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2 \Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k - \mathbf{r}_i)(\mathbf{r}_k - \mathbf{r}_j)}{r_{ki}r_{kj}} f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Importance sampling, Fokker-Planck and Langevin equations

A stochastic process is simply a function of two variables, one is the time, the other is a stochastic variable X, defined by specifying

- the set $\{x\}$ of possible values for X;
- the probability distribution, $w_X(x)$, over this set, or briefly $w(x)$

The set of values $\{x\}$ for X may be discrete, or continuous. If the set of values is continuous, then $w_X(x)$ is a probability density so that $w_X(x)dx$ is the probability that one finds the stochastic variable X to have values in the range $[x, x+dx]$.

An arbitrary number of other stochastic variables may be derived from X. For example, any Y given by a mapping of X, is also a stochastic variable. The mapping may also be time-dependent, that is, the mapping depends on an additional variable t

$$Y_X(t) = f(X,t).$$

The quantity $Y_X(t)$ is called a random function, or, since t often is time, a stochastic process. A stochastic process is a function of two variables, one is the time, the other is a stochastic variable X. Let x be one of the possible values of X then

$$y(t) = f(x,t),$$

is a function of t, called a sample function or realization of the process. In physics one considers the stochastic process to be an ensemble of such sample functions.

For many physical systems initial distributions of a stochastic variable y tend to equilibrium distributions: $w(y,t) \rightarrow w_0(y)$ as $t \rightarrow \infty$. In equilibrium detailed balance constrains the transition rates

$$W(y \rightarrow y')w(y) = W(y' \rightarrow y)w_0(y),$$

where $W(y' \rightarrow y)$ is the probability, per unit time, that the system changes from a state $|y\rangle$, characterized by the value y for the stochastic variable Y , to a state $|y'\rangle$.

Note that for a system in equilibrium the transition rate $W(y' \rightarrow y)$ and the reverse $W(y \rightarrow y')$ may be very different.

Consider, for instance, a simple system that has only two energy levels $\varepsilon_0 = 0$ and $\varepsilon_1 = \Delta E$.

For a system governed by the Boltzmann distribution we find (the partition function has been taken out)

$$W(0 \rightarrow 1)\exp -(\varepsilon_0/kT) = W(1 \rightarrow 0)\exp -(\varepsilon_1/kT)$$

We get then

$$\frac{W(1\rightarrow 0)}{W(0\rightarrow 1)} = \exp-(\Delta E/kT),$$

which goes to zero when T tends to zero.

If we assume a discrete set of events, our initial probability distribution function can be given by

$$w_i(0) = \delta_{i,0},$$

and its time-development after a given time step $\Delta t = \varepsilon$ is

$$w_i(t) = \sum_j W(j\rightarrow i)w_j(t=0).$$

The continuous analog to $w_i(0)$ is

$$w(\mathbf{x}) \rightarrow \delta(\mathbf{x}),$$

where we now have generalized the one-dimensional position x to a generic-dimensional vector $\mathbf{x}$. The Kroenecker δ function is replaced by the δ distribution function $\delta(\mathbf{x})$ at $t=0$.

The transition from a state j to a state i is now replaced by a transition to a state with position $\mathbf{y}$ from a state with position $\mathbf{x}$. The discrete sum of transition probabilities can then be replaced by an integral and we obtain the new distribution at a time $t+\Delta t$ as

$$w(\mathbf{y},t+\Delta t) = \int W(\mathbf{y},t+\Delta t|\mathbf{x},t)w(\mathbf{x},t)d\mathbf{x},$$

and after m time steps we have

$$w(\mathbf{y},t+m\Delta t) = \int W(\mathbf{y},t+m\Delta t|\mathbf{x},t)w(\mathbf{x},t)d\mathbf{x}.$$

When equilibrium is reached we have

$$w(\mathbf{y}) = \int W(\mathbf{y}|\mathbf{x},t)w(\mathbf{x})d\mathbf{x},$$

that is no time-dependence. Note our change of notation for W

We can solve the equation for $w(\mathbf{y},t)$ by making a Fourier transform to momentum space. The PDF $w(\mathbf{x},t)$ is related to its Fourier transform $\tilde{w}(\mathbf{k},t)$ through

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} d\mathbf{k}\exp(i\mathbf{k}\mathbf{x})\tilde{w}(\mathbf{k},t),$$

and using the definition of the δ-function

$$\delta(\mathbf{x}) = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\mathbf{k}\exp(i\mathbf{k}\mathbf{x}),$$

we see that

$$\tilde{w}(\mathbf{k},0) = 1/2\pi.$$

We can then use the Fourier-transformed diffusion equation

$$\frac{\partial \tilde{w}(\mathbf{k},t)}{\partial t} = -D\mathbf{k}^2\tilde{w}(\mathbf{k},t),$$

with the obvious solution

$$\tilde{w}(\mathbf{k},t) = \tilde{w}(\mathbf{k},0)\exp\left[-(D\mathbf{k}^2t)\right) = \frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right].$$

With the Fourier transform we obtain

$$w(\mathbf{x},t)=\int_{-\infty}^{\infty}d\mathbf{k}\exp\left[i\mathbf{k}\mathbf{x}\right]\frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right]=\frac{1}{\sqrt{4\pi Dt}}\exp\left[-(\mathbf{x}^2/4Dt)\right],$$

with the normalization condition

$$\int_{-\infty}^{\infty}w(\mathbf{x},t)d\mathbf{x}=1.$$

The solution represents the probability of finding our random walker at position $\mathbf{x}$ at time t if the initial distribution was placed at $\mathbf{x}=0$ at $t=0$.

There is another interesting feature worth observing. The discrete transition probability W itself is given by a binomial distribution. The results from the central limit theorem state that transition probability in the limit $n\to\infty$ converges to the normal distribution. It is then possible to show that

$$W(il-jl,n\varepsilon)\to W(\mathbf{y},t+\Delta t|\mathbf{x},t)=\frac{1}{\sqrt{4\pi D\Delta t}}\exp\left[-((\mathbf{y}-\mathbf{x})^2/4D\Delta t)\right],$$

and that it satisfies the normalization condition and is itself a solution to the diffusion equation.

Let us now assume that we have three PDFs for times $t_0<t'<t$, that is $w(\mathbf{x}_0,t_0)$, $w(\mathbf{x}',t')$ and $w(\mathbf{x},t)$. We have then

$$w(\mathbf{x},t)=\int_{-\infty}^{\infty}W(\mathbf{x}.t|\mathbf{x}'.t')w(\mathbf{x}',t')d\mathbf{x}',$$

and

$$w(\mathbf{x},t)=\int_{-\infty}^{\infty}W(\mathbf{x}.t|\mathbf{x}_0.t_0)w(\mathbf{x}_0,t_0)d\mathbf{x}_0,$$

and

$$w(\mathbf{x}',t')=\int_{-\infty}^{\infty}W(\mathbf{x}'.t'|\mathbf{x}_0,t_0)w(\mathbf{x}_0,t_0)d\mathbf{x}_0.$$

We can combine these equations and arrive at the famous Einstein-Smoluchenski-Kolmogorov-Chapman (ESKC) relation

$$W(\mathbf{x}t|\mathbf{x}_0t_0)=\int_{-\infty}^{\infty}W(\mathbf{x},t|\mathbf{x}',t')W(\mathbf{x}',t'|\mathbf{x}_0,t_0)d\mathbf{x}'.$$

We can replace the spatial dependence with a dependence upon say the velocity (or momentum), that is we have

$$W(\mathbf{v},t|\mathbf{v}_0,t_0)=\int_{-\infty}^{\infty}W(\mathbf{v},t|\mathbf{v}',t')W(\mathbf{v}',t'|\mathbf{v}_0,t_0)d\mathbf{x}'.$$

We will now derive the Fokker-Planck equation. We start from the ESKC equation

$$W(\mathbf{x},t|\mathbf{x}_0,t_0)=\int_{-\infty}^{\infty}W(\mathbf{x},t|\mathbf{x}',t')W(\mathbf{x}',t'|\mathbf{x}_0,t_0)d\mathbf{x}'.$$

Define $s=t'-t_0$, $\tau=t-t'$ and $t-t_0=s+\tau$. We have then

$$W(\mathbf{x},s+\tau|\mathbf{x}_0)=\int_{-\infty}^{\infty}W(\mathbf{x},\tau|\mathbf{x}')W(\mathbf{x}',s|\mathbf{x}_0)d\mathbf{x}'.$$

Assume now that τ is very small so that we can make an expansion in terms of a small step xi, with $\mathbf{x}'=\mathbf{x}-\xi$, that is

$$W(\mathbf{x},s|\mathbf{x}_0)+\frac{\partial W}{\partial s}\tau+O(\tau^2)=\int_{-\infty}^{\infty}W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)d\mathbf{x}'.$$

We assume that $W(\mathbf{x},\tau|\mathbf{x}-\xi)$ takes non-negligible values only when ξ is small. This is just another way of stating the Master equation!!

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)=\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}+\xi,\tau|\mathbf{x})W(\mathbf{x},s|\mathbf{x}_0)\right].$$

We can then rewrite the ESKC equation as

$$\frac{\partial W}{\partial s}\tau=-W(\mathbf{x},s|\mathbf{x}_0)+\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)\int_{-\infty}^{\infty}\xi^n W(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n=0$ we get simply $W(\mathbf{x},s|\mathbf{x}_0)$ due to normalization.

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)=\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}+\xi,\tau|\mathbf{x})W(\mathbf{x},s|\mathbf{x}_0)\right].$$

We can then rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s}\tau=-W(\mathbf{x},s|\mathbf{x}_0)+\sum_{n=0}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)\int_{-\infty}^{\infty}\xi^n W(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n=0$ we get simply $W(\mathbf{x},s|\mathbf{x}_0)$ due to normalization.

We simplify the above by introducing the moments

$$M_n=\frac{1}{\tau}\int_{-\infty}^{\infty}\xi^n W(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi=\frac{\langle[\Delta x(\tau)]^n\rangle}{\tau},$$

resulting in

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s}=\sum_{n=1}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)M_n\right].$$

When $\tau\to 0$ we assume that $\langle[\Delta x(\tau)]^n\rangle\to 0$ more rapidly than τ itself if $n>2$. When τ is much larger than the standard correlation time of system then M_n for $n>2$ can normally be neglected. This means that fluctuations become negligible at large time scales.

If we neglect such terms we can rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s}=-\frac{\partial M_1 W(\mathbf{x},s|\mathbf{x}_0)}{\partial x}+\frac{1}{2}\frac{\partial^2 M_2 W(\mathbf{x},s|\mathbf{x}_0)}{\partial x^2}.$$

In a more compact form we have

$$\frac{\partial W}{\partial s}=-\frac{\partial M_1 W}{\partial x}+\frac{1}{2}\frac{\partial^2 M_2 W}{\partial x^2},$$

which is the Fokker-Planck equation! It is trivial to replace position with velocity (momentum).

Consider a particle suspended in a liquid. On its path through the liquid it will continuously collide with the liquid molecules. Because on average the particle will collide more often on the front side than on the back side, it will experience a systematic force proportional with its velocity, and directed opposite to its velocity. Besides this systematic force the particle will experience a stochastic force $\mathbf{F}(t)$. The equations of motion are

- $\frac{d\mathbf{r}}{dt} = \mathbf{v}$ and
- $\frac{d\mathbf{v}}{dt} = -\xi\mathbf{v} + \mathbf{F}$.

From hydrodynamics we know that the friction constant ξ is given by

$$\xi = 6\pi\eta a/m$$

where η is the viscosity of the solvent and a is the radius of the particle .

Solving the second equation in the previous slide we get

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}\mathbf{F}(\tau).$$

If we want to get some useful information out of this, we have to average over all possible realizations of $\mathbf{F}(t)$, with the initial velocity as a condition. A useful quantity for example is

$$\langle \mathbf{v}(t)\cdot\mathbf{v}(t)\rangle_{\mathbf{v}_0} = v_0^{-\xi 2t} + 2\int_0^t d\tau e^{-\xi(2t-\tau)}\mathbf{v}_0\cdot\langle\mathbf{F}(\tau)\rangle_{\mathbf{v}_0}$$

$$+\int_0^t d\tau' \int_0^t d\tau e^{-\xi(2t-\tau-\tau')}\langle\mathbf{F}(\tau)\cdot\mathbf{F}(\tau')\rangle_{\mathbf{v}_0}.$$

In order to continue we have to make some assumptions about the conditional averages of the stochastic forces. In view of the chaotic character of the stochastic forces the following assumptions seem to be appropriate

$$\langle\mathbf{F}(t)\rangle = 0,$$

and

$$\langle\mathbf{F}(t)\cdot\mathbf{F}(t')\rangle_{\mathbf{v}_0} = C_{\mathbf{v}_0}\delta(t-t').$$

We omit the subscript $\mathbf{v}_0$, when the quantity of interest turns out to be independent of $\mathbf{v}_0$. Using the last three equations we get

$$\langle \mathbf{v}(t)\cdot\mathbf{v}(t)\rangle_{\mathbf{v}_0} = v_0^2 e^{-2\xi t} + \frac{C_{\mathbf{v}_0}}{2\xi}(1-e^{-2\xi t}).$$

For large t this should be equal to 3kT/m, from which it follows that

$$\langle\mathbf{F}(t)\cdot\mathbf{F}(t')\rangle = 6\frac{kT}{m}\xi\delta(t-t').$$

This result is called the fluctuation-dissipation theorem .

Integrating

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}\mathbf{F}(\tau),$$

we get

$$\mathbf{r}(t) = \mathbf{r}_0 + \mathbf{v}_0\frac{1}{\xi}(1-e^{-\xi t}) + \int_0^t d\tau\int_0^\tau \tau' e^{-\xi(\tau-\tau')}\mathbf{F}(\tau'),$$

from which we calculate the mean square displacement

$$\langle(\mathbf{r}(t)-\mathbf{r}_0)^2\rangle_{\mathbf{v}_0} = \frac{v_0^2}{\xi}(1-e^{-\xi t})^2 + \frac{3kT}{m\xi^2}(2\xi t - 3 + 4e^{-\xi t} - e^{-2\xi t}).$$

For very large t this becomes

$$\langle(\mathbf{r}(t)-\mathbf{r}_0)^2\rangle = \frac{6kT}{m\xi}t$$

from which we get the Einstein relation

$$D = \frac{kT}{m\xi}$$

where we have used $\langle(\mathbf{r}(t)-\mathbf{r}_0)^2\rangle = 6Dt$.

Code example for two electrons in a quantum dots

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings No energy minimization from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl$_{t}oolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,Form$

Read name of output file from command line if len(sys.argv) == 2: outfilename = sys.argv[1] else: print(': Name of output file must be given as command line argument.') outfile = open(outfilename,'w')

Trial wave function for the 2-electron quantum dot in two dims def WaveFunction(r,alpha,beta): r1 = r[0,0]**2 + r[0,1]**2 r2 = r[1,0]**2 + r[1,1]**2 r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = r12/(1+beta*r12) return exp(-0.5*alpha*(r1+r2)+deno)

Local energy for the 2-electron quantum dot in two dims, using analytical local energy def LocalEnergy(r,alpha,beta):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) deno2 = deno*deno return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1.0/r12)

Setting up the quantum force for the two-electron quantum dot, recall that it is a vector def QuantumForce(r,alpha,beta):

qforce = np.zeros((NumberParticles,Dimension), np.double) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) qforce[0,:] = -2*r[0,:]*alpha*(r[0,:]-r[1,:])*deno*deno/r12 qforce[1,:] = -2*r[1,:]*alpha*(r[1,:]-r[0,:])*deno*deno/r12 return qforce

The Monte Carlo sampling with the Metropolis algo jit decorator tells Numba to compile this function. The argument types will be inferred by Numba when function is called. @jit() def MonteCarloSampling():

NumberMCcycles= 100000 Parameters in the Fokker-Planck simulation of the quantum force D = 0.5 TimeStep = 0.05 positions PositionOld = np.zeros((NumberParticles,Dimension), np.double) PositionNew = np.zeros((NumberParticles,Dimension), np.double) Quantum force QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double) QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

seed for rng generator seed() start variational parameter loops, two parameters here alpha = 0.9 for ia in range(MaxVariations): alpha += .025 AlphaValues[ia] = alpha beta = 0.2 for jb in range(MaxVariations): beta += .01 BetaValues[jb] = beta energy = energy2 = 0.0 DeltaE = 0.0 Initial position for i in range(NumberParticles): for j in range(Dimension): PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep) wfold = WaveFunction(PositionOld,alpha,beta) QuantumForceOld = QuantumForce(PositionOld,alpha, beta)

Loop over MC MCcycles for MCcycle in range(NumberMCcycles): Trial position moving one particle at the time for i in range(NumberParticles): for j in range(Dimension): PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+ QuantumForceOld[i,j]*TimeStep*D wfnew = WaveFunction(PositionNew,alpha,beta) QuantumForceNew = QuantumForce(PositionNew,alpha, beta) GreensFunction = 0.0 for j in range(Dimension): GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])* (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])- PositionNew[i,j]+PositionOld[i,j])

GreensFunction = exp(GreensFunction) ProbabilityRatio = GreensFunction*wfnew**2/wfold**2 Metropolis-Hastings test to see whether we accept the move if random() <= ProbabilityRatio: for j in range(Dimension): PositionOld[i,j] = PositionNew[i,j] QuantumForceOld[i,j] = QuantumForceNew[i,j] wfold = wfnew DeltaE = LocalEnergy(PositionOld,alpha,beta) energy += DeltaE energy2 += DeltaE**2 We calculate mean, variance and error (no blocking applied) energy /= NumberMCcycles energy2 /= NumberMCcycles variance = energy2 - energy**2 error = sqrt(variance/NumberMCcycles) Energies[ia,jb] = energy outfile.write('return Energies, AlphaValues, BetaValues

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 MaxVariations = 10 Energies = np.zeros((MaxVariations,MaxVariations)) AlphaValues = np.zeros(MaxVariations) BetaValues = np.zeros(MaxVariations) (Energies, AlphaValues, BetaValues) = MonteCarloSampling() outfile.close() Prepare for plots fig = plt.figure() ax = fig.gca(projection='3d') Plot the surface. X, Y = np.meshgrid(AlphaValues, BetaValues) surf = ax.plot$_{s}urface(X,Y,Energies,cmap = cm.coolwarm,linewidth = 0,antialiased = False)Customizethezaxis.zmin = np.matrix(Energies).min()zmax = np.matrix(Energies).max()ax.set_{z}lim(zmin,zmax)ax.set_{x}label(r'$α') ax.set$_{y}label(r'$β') ax.set$_{z}label(r'\langle E\rangle$') ax.zaxis.set$_{m}ajor_{l}ocator(LinearLocator(10))ax.zaxis.set_{m}ajor_{f}ormatter(FormatStrFormatter('Addacolorbarwhichm$
$0.5,aspect = 5)plt.show()$

Bringing the gradient optmization.

The simple one-particle case in a harmonic oscillator trap

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python Gradient descent stepping with analytical derivative import numpy as np from scipy.optimize import minimize def DerivativeE(x): return x-1.0/(4*x*x*x);

def Energy(x): return x*x*0.5+1.0/(8*x*x); x0 = 1.0 eta = 0.1 Niterations = 100

for iter in range(Niterations): gradients = DerivativeE(x0) x0 -= eta*gradients

print(x0)

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl$_{t}oolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,FormatStrFormatterimportsysfromm$

Trial wave function for the 2-electron quantum dot in two dims def WaveFunction(r,alpha): r1 = r[0,0]**2 + r[0,1]**2 r2 = r[1,0]**2 + r[1,1]**2 return exp(-0.5*alpha*(r1+r2))

Local energy for the 2-electron quantum dot in two dims, using analytical local energy def LocalEnergy(r,alpha):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha

Derivate of wave function ansatz as function of variational parameters def DerivativeWFansatz(r,alpha):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) WfDer = -(r1+r2) return WfDer

Setting up the quantum force for the two-electron quantum dot, recall that it is a vector def QuantumForce(r,alpha):

qforce = np.zeros((NumberParticles,Dimension), np.double) qforce[0,:] = -2*r[0,:]*alpha qforce[1,:] = -2*r[1,:]*alpha return qforce

Computing the derivative of the energy and the energy jit decorator tells Numba to compile this function. The argument types will be inferred by Numba when function is called. @jit def EnergyMinimization(alpha):

NumberMCcycles= 1000 Parameters in the Fokker-Planck simulation of the quantum force D = 0.5 TimeStep = 0.05 positions PositionOld = np.zeros((NumberParticles,Dimension), np.double) PositionNew = np.zeros((NumberParticles,Dimension), np.double) Quantum force QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double) QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

seed for rng generator seed() energy = 0.0 DeltaE = 0.0 EnergyDer = 0.0 DeltaPsi = 0.0 DerivativePsiE = 0.0 Initial position for i in range(NumberParticles): for j in range(Dimension): PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep) wfold = WaveFunction(PositionOld,alpha) QuantumForceOld = QuantumForce(PositionOld,alpha)

Loop over MC MCcycles for MCcycle in range(NumberMCcycles): Trial position moving one particle at the time for i in range(NumberParticles): for j in range(Dimension): PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+ QuantumForceOld[i,j]*TimeStep*D wfnew = WaveFunction(PositionNew,alpha) QuantumForceNew = QuantumForce(PositionNew,alpha) GreensFunction = 0.0 for j in range(Dimension): GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])* (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])- PositionNew[i,j]+PositionOld[i,j])

GreensFunction = exp(GreensFunction) ProbabilityRatio = GreensFunction*wfnew**2/wfold**2 Metropolis-Hastings test to see whether we accept the move if random() <= ProbabilityRatio: for j in range(Dimension): PositionOld[i,j] = PositionNew[i,j] QuantumForceOld[i,j] = QuantumForceNew[i,j] wfold = wfnew DeltaE = LocalEnergy(PositionOld,alpha) DeltaPsi = DerivativeWFansatz(PositionOld,alpha) energy += DeltaE DerivativePsiE += DeltaPsi*DeltaE

We calculate mean, variance and error (no blocking applied) energy /= NumberMCcycles DerivativePsiE /= NumberMCcycles DeltaPsi /= NumberMCcycles EnergyDer = 2*(DerivativePsiE-DeltaPsi*energy) return energy, EnergyDer

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 guess for variational parameters x0 = 1.5 Set up iteration using stochastic gradient method Energy =0 ; EnergyDer = 0 Energy, EnergyDer = EnergyMinimization(x0) print(Energy, EnergyDer)

eta = 0.01 Niterations = 100

for iter in range(Niterations): gradients = EnergyDer x0 -= eta*gradients Energy, EnergyDer = EnergyMinimization(x0)

print(x0)

VMC for fermions: Efficient calculation of Slater determinants

The potentially most time-consuming part is the evaluation of the gradient and the Laplacian of an N-particle Slater determinant.

We have to differentiate the determinant with respect to all spatial coordinates of all particles. A brute force differentiation would involve $N\cdot d$ evaluations of the entire determinant which would even worsen the already undesirable time scaling, making it $Nd\cdot O(N^3)\sim O(d\cdot N^4)$.

This poses serious hindrances to the overall efficiency of our code.

The efficiency can be improved however if we move only one electron at the time. The Slater determinant matrix $\hat{D}$ is defined by the matrix elements

$$d_{ij}=\phi_j(x_i)$$

where $\phi_j(\mathbf{r}_i)$ is a single particle wave function. The columns correspond to the position of a given particle while the rows stand for the various quantum numbers.

What we need to realize is that when differentiating a Slater determinant with respect to some given coordinate, only one row of the corresponding Slater matrix is changed.

Therefore, by recalculating the whole determinant we risk producing redundant information. The solution turns out to be an algorithm that requires to keep track of the *inverse* of the Slater matrix.

Let the current position in phase space be represented by the $(N \cdot d)$-element vector $\mathbf{r}^{\text{old}}$ and the new suggested position by the vector $\mathbf{r}^{\text{new}}$.

The inverse of $\hat{D}$ can be expressed in terms of its cofactors C_{ij} and its determinant (this our notation for a determinant) $|\hat{D}|$:

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|} \tag{6.7}$$

Notice that the interchanged indices indicate that the matrix of cofactors is to be transposed.

If $\hat{D}$ is invertible, then we must obviously have $\hat{D}^{-1}\hat{D} = \mathbf{1}$, or explicitly in terms of the individual elements of $\hat{D}$ and $\hat{D}^{-1}$:

$$\sum_{k=1}^{N} d_{ik} d_{kj}^{-1} = \delta_{ij} \tag{6.8}$$

Consider the ratio, which we shall call R, between $|\hat{D}(\mathbf{r}^{\text{new}})|$ and $|\hat{D}(\mathbf{r}^{\text{old}})|$. By definition, each of these determinants can individually be expressed in terms of the i-th row of its cofactor matrix

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) C_{ij}(\mathbf{r}^{\text{old}})} \tag{6.9}$$

Suppose now that we move only one particle at a time, meaning that $\mathbf{r}^{\text{new}}$ differs from $\mathbf{r}^{\text{old}}$ by the position of only one, say the i-th, particle . This means that $\hat{D}(\mathbf{r}^{\text{new}})$ and $\hat{D}(\mathbf{r}^{\text{old}})$ differ only by the entries of the i-th row. Recall also that the i-th row of a cofactor matrix $\hat{C}$ is independent of the entries of the i-th row of its corresponding matrix $\hat{D}$. In this particular case we therefore get that the i-th row of $\hat{C}(\mathbf{r}^{\text{new}})$ and $\hat{C}(\mathbf{r}^{\text{old}})$ must be equal. Explicitly, we have:

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\, j \in \{1, \dots, N\} \tag{6.10}$$

Inserting this into the numerator of eq. (6.9) and using eq. (6.7) to substitute the cofactors with the elements of the inverse matrix, we get:

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}})} \tag{6.11}$$

Now by eq. (6.8) the denominator of the rightmost expression must be unity, so that we finally arrive at:

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}) \tag{6.12}$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\text{new}}), \dots, \phi_N(\mathbf{r}_i^{\text{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\text{old}})$.

If the new position $\mathbf{r}^{\text{new}}$ is accepted, then the inverse matrix can by suitably updated by an algorithm having a time scaling of $O(N^2)$. This algorithm goes as follows. First we update all but the i-th column of $\hat{D}^{-1}$. For each column $j \neq i$, we first calculate the quantity:

$$S_j = (\hat{D}(\mathbf{r}^{\text{new}}) \times \hat{D}^{-1}(\mathbf{r}^{\text{old}}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}}) d_{lj}^{-1}(\mathbf{r}^{\text{old}}) \tag{6.13}$$

The new elements of the j-th column of $\hat{D}^{-1}$ are then given by:

$$d_{kj}^{-1}(\mathbf{r}^{\mathrm{new}}) = d_{kj}^{-1}(\mathbf{r}^{\mathrm{old}}) - \frac{S_j}{R} d_{ki}^{-1}(\mathbf{r}^{\mathrm{old}}) \quad \begin{array}{l} \forall \;\; k \in \{1,\dots,N\} \\ j \neq i \end{array} \tag{6.14}$$

Finally the elements of the i-th column of $\hat{D}^{-1}$ are updated simply as follows:

$$d_{ki}^{-1}(\mathbf{r}^{\mathrm{new}}) = \frac{1}{R} d_{ki}^{-1}(\mathbf{r}^{\mathrm{old}}) \quad \forall \;\; k \in \{1,\dots,N\} \tag{6.15}$$

We see from these formulas that the time scaling of an update of $\hat{D}^{-1}$ after changing one row of $\hat{D}$ is $O(N^2)$.

The scheme is also applicable for the calculation of the ratios involving derivatives. It turns out that differentiating the Slater determinant with respect to the coordinates of a single particle $\mathbf{r}_i$ changes only the i-th row of the corresponding Slater matrix.

The gradient and the Laplacian.

The gradient and the Laplacian can therefore be calculated as follows:

$$\frac{\nabla_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i \phi_j(\mathbf{r}_i) d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2 |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

Thus, to calculate all the derivatives of the Slater determinant, we only need the derivatives of the single particle wave functions ($\nabla_i \phi_j(\mathbf{r}_i)$ and $\nabla_i^2 \phi_j(\mathbf{r}_i)$) and the elements of the corresponding inverse Slater matrix ($\hat{D}^{-1}(\mathbf{r}_i)$). A calculation of a single derivative is by the above result an $O(N)$ operation. Since there are $d \cdot N$ derivatives, the time scaling of the total evaluation becomes $O(d \cdot N^2)$. With an $O(N^2)$ updating algorithm for the inverse matrix, the total scaling is no worse, which is far better than the brute force approach yielding $O(d \cdot N^4)$.

Important note: In most cases you end with closed form expressions for the single-particle wave functions. It is then useful to calculate the various derivatives and make separate functions for them.

The Slater determinant takes the form

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = \frac{1}{\sqrt{4!}} \left| \begin{array}{cccc} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{array} \right|.$$

The Slater determinant as written is zero since the spatial wave functions for the spin up and spin down states are equal. But we can rewrite it as the product of two Slater determinants, one for spin up and one for spin down.

We can rewrite it as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = |\uparrow|(1,2)|\downarrow|(3,4) - |\uparrow|(1,3)|\downarrow|(2,4)$$

$$-|\uparrow|(1,4)|\downarrow|(3,2) + |\uparrow|(2,3)|\downarrow|(1,4) - |\uparrow|(2,4)|\downarrow|(1,3)$$

$$+|\uparrow|(3,4)|\downarrow|(1,2),$$

where we have defined

$$|\uparrow|(1,2) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) \end{vmatrix},$$

and

$$|\downarrow|(3,4) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

We want to avoid to sum over spin variables, in particular when the interaction does not depend on spin.

It can be shown, see for example Moskowitz and Kalos, Int. J. Quantum Chem. **20** 1107 (1981), that for the variational energy we can approximate the Slater determinant as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) \propto |\uparrow|(1,2)|\downarrow|(3,4),$$

or more generally as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\ldots\mathbf{r}_N) \propto |\uparrow||\downarrow|,$$

where we have the Slater determinant as the product of a spin up part involving the number of electrons with spin up only (2 for beryllium and 5 for neon) and a spin down part involving the electrons with spin down.

This ansatz is not antisymmetric under the exchange of electrons with opposite spins but it can be shown (show this) that it gives the same expectation value for the energy as the full Slater determinant.

As long as the Hamiltonian is spin independent, the above is correct. It is rather straightforward to see this if you go back to the equations for the energy discussed earlier this semester.

We will thus factorize the full determinant $|\hat{D}|$ into two smaller ones, where each can be identified with $\uparrow$ and $\downarrow$ respectively:

$$|\hat{D}| = |\hat{D}|_\uparrow \cdot |\hat{D}|_\downarrow$$

The combined dimensionality of the two smaller determinants equals the dimensionality of the full determinant. Such a factorization is advantageous in that it makes it possible to perform the calculation of the ratio R and the updating of the inverse matrix separately for $|\hat{D}|_\uparrow$ and $|\hat{D}|_\downarrow$:

$$\frac{|\hat{D}|^{\mathrm{new}}}{|\hat{D}|^{\mathrm{old}}} = \frac{|\hat{D}|^{\mathrm{new}}_\uparrow}{|\hat{D}|^{\mathrm{old}}_\uparrow} \cdot \frac{|\hat{D}|^{\mathrm{new}}_\downarrow}{|\hat{D}|^{\mathrm{old}}_\downarrow}$$

This reduces the calculation time by a constant factor. The maximal time reduction happens in a system of equal numbers of $\uparrow$ and $\downarrow$ particles, so that the two factorized determinants are half the size of the original one.

Consider the case of moving only one particle at a time which originally had the following time scaling for one transition:

$$O_R(N) + O_{\mathrm{inverse}}(N^2)$$

For the factorized determinants one of the two determinants is obviously unaffected by the change so that it cancels from the ratio R.

Therefore, only one determinant of size $N/2$ is involved in each calculation of R and update of the inverse matrix. The scaling of each transition then becomes:

$$O_R(N/2) + O_{\mathrm{inverse}}(N^2/4)$$

and the time scaling when the transitions for all N particles are put together:

$$O_R(N^2/2) + O_{\mathrm{inverse}}(N^3/4)$$

which gives the same reduction as in the case of moving all particles at once.

Computing the ratios discussed above requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N\times N$ matrix by Gaussian elimination has a complexity of order of $\mathscr{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. We will use the expression

$$d_{kj}^{-1}(\mathbf{x^{new}}) = \begin{cases} d_{kj}^{-1}(\mathbf{x^{old}}) - \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N} d_{il}(\mathbf{x^{new}}) d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j \neq i \\ \frac{d_{ki}^{-1}(\mathbf{x^{old}})}{R}\sum_{l=1}^{N} d_{il}(\mathbf{x^{old}}) d_{lj}^{-1}(\mathbf{x^{old}}) & \text{if } j = i \end{cases}$$

This equation scales as $O(N^2)$. The evaluation of the determinant of an $N\times N$ matrix by standard Gaussian elimination requires $\mathbf{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by standard LU decomposition methods.

Expectation value of the kinetic energy.

The expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle\hat{K}_i\rangle = -\frac{1}{2}\frac{\langle\Psi|\nabla_i^2|\Psi\rangle}{\langle\Psi|\Psi\rangle},$$

$$K_i = -\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}. \tag{6.16}$$

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2(\Psi_D\Psi_C)}{\Psi_D\Psi_C} = \frac{\nabla\cdot[\nabla(\Psi_D\Psi_C)]}{\Psi_D\Psi_C} = \frac{\nabla\cdot[\Psi_C\nabla\Psi_D + \Psi_D\nabla\Psi_C]}{\Psi_D\Psi_C} \\ &= \frac{\nabla\Psi_C\cdot\nabla\Psi_D + \Psi_C\nabla^2\Psi_D + \nabla\Psi_D\cdot\nabla\Psi_C + \Psi_D\nabla^2\Psi_C}{\Psi_D\Psi_C}\end{aligned} \tag{6.17}$$

$$\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C} \tag{6.18}$$

The second derivative of the Jastrow factor divided by the Jastrow factor (the way it enters the kinetic energy) is

$$\left[\frac{\nabla^2\Psi_C}{\Psi_C}\right]_x = 2\sum_{k=1}^{N}\sum_{i=1}^{k-1}\frac{\partial^2 g_{ik}}{\partial x_k^2} + \sum_{k=1}^{N}\left(\sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}\right)^2$$

But we have a simple form for the function, namely

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

and it is easy to see that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj}) + \sum_{j\neq k}\left(f''(r_{kj}) + \frac{2}{r_{kj}}f'(r_{kj})\right)$$

Using

$$f(r_{ij}) = \frac{ar_{ij}}{1+\beta r_{ij}},$$

and $g'(r_{kj}) = dg(r_{kj})/dr_{kj}$ and $g''(r_{kj}) = d^2g(r_{kj})/dr_{kj}^2$ we find that for particle k we have

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}\frac{a}{(1+\beta r_{ki})^2}\frac{a}{(1+\beta r_{kj})^2} + \sum_{j\neq k}\left(\frac{2a}{r_{kj}(1+\beta r_{kj})^2} - \frac{2a\beta}{(1+\beta r_{kj})^3}\right)$$

The gradient and Laplacian can be calculated as follows:

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

and

$$\frac{\nabla_i^2|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i^2 d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i^2\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r})$$

The gradient for the determinant is

$$\frac{\nabla_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N}\nabla_i d_{ij}(\mathbf{r})\, d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N}\nabla_i\phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}).$$

We have

$$\Psi_C = \prod_{i<j} g(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\},$$

the gradient needed for the quantum force and local energy is easy to compute. We get for particle k

$$\frac{\nabla_k\Psi_C}{\Psi_C} = \sum_{j\neq k}\frac{\mathbf{r}_{kj}}{r_{kj}}\frac{a}{(1+\beta r_{kj})^2},$$

which is rather easy to code. Remember to sum over all particles when you compute the local energy.

We need to compute the ratio between wave functions, in particular for the Slater determinants.

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\mathrm{new}})\, d_{ji}^{-1}(\mathbf{r}^{\mathrm{old}}) = \sum_{j=1}^{N}\phi_j(\mathbf{r}_i^{\mathrm{new}})\, d_{ji}^{-1}(\mathbf{r}^{\mathrm{old}})$$

What this means is that in order to get the ratio when only the i-th particle has been moved, we only need to calculate the dot product of the vector $(\phi_1(\mathbf{r}_i^{\mathrm{new}}), \ldots, \phi_N(\mathbf{r}_i^{\mathrm{new}}))$ of single particle wave functions evaluated at this new position with the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position. Such an operation has a time scaling of $O(N)$. The only extra thing we need to do is to maintain the inverse matrix $\hat{D}^{-1}(\mathbf{x}^{\mathrm{old}})$.

6.1 Gradient Methods

Top-down start

- We will start with a top-down view, with a simple harmonic oscillator problem in one dimension as case.
- Thereafter we continue with implementing the simplest possible steepest descent approach to our two-electron problem with an electrostatic (Coulomb) interaction. Our code includes also importance sampling. The simple Python code here illustrates the basic elements which need to be included in our own code.
- Then we move on to the mathematical description of various gradient methods.

Motivation

Our aim with this part of the project is to be able to

- find an optimal value for the variational parameters using only some few Monte Carlo cycles
- use these optimal values for the variational parameters to perform a large-scale Monte Carlo calculation

To achieve this will look at methods like *Steepest descent* and the *conjugate gradient method*. Both these methods allow us to find the minima of a multivariable function like our energy (function of several variational parameters). Alternatively, you can always use Newton's method. In particular, since we will normally have one variational parameter, Newton's method can be easily used in finding the minimum of the local energy.

Simple example and demonstration

Let us illustrate what is needed in our calculations using a simple example, the harmonic oscillator in one dimension. For the harmonic oscillator in one-dimension we have a trial wave function and probability

$$\psi_T(x;\alpha) = \exp-(\frac{1}{2}\alpha^2x^2),$$

which results in a local energy

$$\frac{1}{2}\left(\alpha^2+x^2(1-\alpha^4)\right).$$

We can compare our numerically calculated energies with the exact energy as function of α

$$\overline{E}[\alpha] = \frac{1}{4}\left(\alpha^2+\frac{1}{\alpha^2}\right).$$

Simple example and demonstration

The derivative of the energy with respect to α gives

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = \frac{1}{2}\alpha - \frac{1}{2\alpha^3}$$

and a second derivative which is always positive (meaning that we find a minimum)

$$\frac{d^2\langle E_L[\alpha]\rangle}{d\alpha^2} = \frac{1}{2} + \frac{3}{2\alpha^4}$$

The condition

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 0,$$

gives the optimal $\alpha = 1$, as expected.

*Exercise : Find the local energy for the harmonic oscillator

a) Derive the local energy for the harmonic oscillator in one dimension and find its expectation value.

b) Show also that the optimal value of optimal $\alpha = 1$

c) Repeat the above steps in two dimensions for N bosons or electrons. What is the optimal value of α?

Variance in the simple model

We can also minimize the variance. In our simple model the variance is

$$\sigma^2[\alpha] = \frac{1}{4}\left(1 + (1-\alpha^4)^2\frac{3}{4\alpha^4}\right) - \overline{E}^2.$$

which yields a second derivative which is always positive.

Computing the derivatives

In general we end up computing the expectation value of the energy in terms of some parameters $\alpha_0, \alpha_1, \dots, \alpha_n$ and we search for a minimum in this multi-variable parameter space. This leads to an energy minimization problem *where we need the derivative of the energy as a function of the variational parameters*.

In the above example this was easy and we were able to find the expression for the derivative by simple derivations. However, in our actual calculations the energy is represented by a multi-dimensional integral with several variational parameters. How can we can then obtain the derivatives of the energy with respect to the variational parameters without having to resort to expensive numerical derivations?

Expressions for finding the derivatives of the local energy

To find the derivatives of the local energy expectation value as function of the variational parameters, we can use the chain rule and the hermiticity of the Hamiltonian.

Let us define

$$\bar{E}_\alpha = \frac{d\langle E_L[\alpha]\rangle}{d\alpha}.$$

as the derivative of the energy with respect to the variational parameter α (we limit ourselves to one parameter only). In the above example this was easy and we obtain a simple expression for the derivative. We define also the derivative of the trial function (skipping the subindex T) as

$$\bar{\psi}_\alpha = \frac{d\psi[\alpha]\rangle}{d\alpha}.$$

Derivatives of the local energy

The elements of the gradient of the local energy are then (using the chain rule and the hermiticity of the Hamiltonian)

$$\bar{E}_\alpha = 2\left(\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} E_L[\alpha]\rangle - \langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle\right).$$

From a computational point of view it means that you need to compute the expectation values of

$$\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} E_L[\alpha]\rangle,$$

and

$$\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle$$

*Exercise : General expression for the derivative of the energy

a) Show that

$$\bar{E}_\alpha = 2\left(\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} E_L[\alpha]\rangle - \langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle\right).$$

b) Find the corresponding expression for the variance.

Python program for 2-electrons in 2 dimensions

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings Added energy minimization with gradient descent using fixed step size To do: replace with optimization codes from scipy and/or use stochastic gradient descent from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl$_{toolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,FormatStrFormatterimportsys}$

Trial wave function for the 2-electron quantum dot in two dims def WaveFunction(r,alpha,beta): r1 = r[0,0]**2 + r[0,1]**2 r2 = r[1,0]**2 + r[1,1]**2 r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = r12/(1+beta*r12) return exp(-0.5*alpha*(r1+r2)+deno)

Local energy for the 2-electron quantum dot in two dims, using analytical local energy def LocalEnergy(r,alpha,beta):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) deno2 = deno*deno return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1.0/r12)

Derivate of wave function ansatz as function of variational parameters def DerivativeW-Fansatz(r,alpha,beta):

WfDer = np.zeros((2), np.double) r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) deno2 = deno*deno WfDer[0] = -0.5*(r1+r2) WfDer[1] = -r12*r12*deno2 return WfDer

Setting up the quantum force for the two-electron quantum dot, recall that it is a vector def QuantumForce(r,alpha,beta):

qforce = np.zeros((NumberParticles,Dimension), np.double) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) qforce[0,:] = -2*r[0,:]*alpha*(r[0,:]-r[1,:])*deno*deno/r12 qforce[1,:] = -2*r[1,:]*alpha*(r[1,:]-r[0,:])*deno*deno/r12 return qforce

Computing the derivative of the energy and the energy def EnergyMinimization(alpha, beta):

NumberMCcycles= 10000 Parameters in the Fokker-Planck simulation of the quantum force D = 0.5 TimeStep = 0.05 positions PositionOld = np.zeros((NumberParticles,Dimension), np.double) PositionNew = np.zeros((NumberParticles,Dimension), np.double) Quantum force QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double) QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

seed for rng generator seed() energy = 0.0 DeltaE = 0.0 EnergyDer = np.zeros((2), np.double) DeltaPsi = np.zeros((2), np.double) DerivativePsiE = np.zeros((2), np.double) Initial position for i in range(NumberParticles): for j in range(Dimension): PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep) wfold = WaveFunction(PositionOld,alpha,beta) QuantumForceOld = QuantumForce(PositionOld,alpha, beta)

Loop over MC MCcycles for MCcycle in range(NumberMCcycles): Trial position moving one particle at the time for i in range(NumberParticles): for j in range(Dimension): PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+ QuantumForce-Old[i,j]*TimeStep*D wfnew = WaveFunction(PositionNew,alpha,beta) QuantumForceNew = QuantumForce(PositionNew,alpha, beta) GreensFunction = 0.0 for j in range(Dimension): GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])* (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])- PositionNew[i,j]+PositionOld[i,j])

GreensFunction = exp(GreensFunction) ProbabilityRatio = GreensFunction*wfnew**2/wfold**2 Metropolis-Hastings test to see whether we accept the move if random() <= ProbabilityRatio: for j in range(Dimension): PositionOld[i,j] = PositionNew[i,j] QuantumForceOld[i,j] = QuantumForceNew[i,j] wfold = wfnew DeltaE = LocalEnergy(PositionOld,alpha,beta) DerPsi = DerivativeWFansatz(PositionOld,alpha,beta) DeltaPsi += DerPsi energy += DeltaE DerivativePsiE += DerPsi*DeltaE

We calculate mean values energy /= NumberMCcycles DerivativePsiE /= NumberMCcycles DeltaPsi /= NumberMCcycles EnergyDer = 2*(DerivativePsiE-DeltaPsi*energy) return energy, EnergyDer

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 guess for variational parameters alpha = 0.9 beta = 0.2 Set up iteration using gradient descent method Energy = 0 EDerivative = np.zeros((2), np.double) eta = 0.01 Niterations = 50 for iter in range(Niterations): Energy, EDerivative = EnergyMinimization(alpha,beta) alphagradient = EDerivative[0] betagradient = EDerivative[1] alpha -= eta*alphagradient beta -= eta*betagradient

print(alpha, beta) print(Energy, EDerivative[0], EDerivative[1])

Using Broyden's algorithm in scipy

The following function uses the above described BFGS algorithm. Here we have defined a function which calculates the energy and a function which computes the first derivative.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python 2-electron VMC code for 2dim quantum dot with importance sampling Using gaussian rng for new positions and Metropolis- Hastings Added energy minimization using the BFGS algorithm, see p. 136 of https://www.springer.com/it/book/9780387303031 from math import exp, sqrt from random import random, seed, normalvariate import numpy as np import matplotlib.pyplot as plt from mpl$_{toolkits.mplot3dimportAxes3Dfrommatplotlibimportcmfrommatplotlib.tickerimportLinearLocator,Form}$

Trial wave function for the 2-electron quantum dot in two dims def WaveFunction(r,alpha,beta): r1 = r[0,0]**2 + r[0,1]**2 r2 = r[1,0]**2 + r[1,1]**2 r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = r12/(1+beta*r12) return exp(-0.5*alpha*(r1+r2)+deno)

Local energy for the 2-electron quantum dot in two dims, using analytical local energy def LocalEnergy(r,alpha,beta):

r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) deno2 = deno*deno return 0.5*(1-alpha*alpha)*(r1 + r2) +2.0*alpha + 1.0/r12+deno2*(alpha*r12-deno2+2*beta*deno-1.0/r12)

Derivate of wave function ansatz as function of variational parameters def DerivativeWFansatz(r,alpha,beta):

WfDer = np.zeros((2), np.double) r1 = (r[0,0]**2 + r[0,1]**2) r2 = (r[1,0]**2 + r[1,1]**2) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) deno2 = deno*deno WfDer[0] = -0.5*(r1+r2) WfDer[1] = -r12*r12*deno2 return WfDer

Setting up the quantum force for the two-electron quantum dot, recall that it is a vector def QuantumForce(r,alpha,beta):

qforce = np.zeros((NumberParticles,Dimension), np.double) r12 = sqrt((r[0,0]-r[1,0])**2 + (r[0,1]-r[1,1])**2) deno = 1.0/(1+beta*r12) qforce[0,:] = -2*r[0,:]*alpha*(r[0,:]-r[1,:])*deno*deno/r12 qforce[1,:] = -2*r[1,:]*alpha*(r[1,:]-r[0,:])*deno*deno/r12 return qforce

Computing the derivative of the energy and the energy def EnergyDerivative(x0):

Parameters in the Fokker-Planck simulation of the quantum force D = 0.5 TimeStep = 0.05 NumberMCcycles= 10000 positions PositionOld = np.zeros((NumberParticles,Dimension), np.double) PositionNew = np.zeros((NumberParticles,Dimension), np.double) Quantum force QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double) QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

energy = 0.0 DeltaE = 0.0 alpha = x0[0] beta = x0[1] EnergyDer = 0.0 DeltaPsi = 0.0 DerivativePsiE = 0.0 Initial position for i in range(NumberParticles): for j in range(Dimension): PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep) wfold = WaveFunction(PositionOld,alpha,beta) QuantumForceOld = QuantumForce(PositionOld,alpha, beta)

Loop over MC MCcycles for MCcycle in range(NumberMCcycles): Trial position moving one particle at the time for i in range(NumberParticles): for j in range(Dimension): PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+ QuantumForceOld[i,j]*TimeStep*D wfnew = WaveFunction(PositionNew,alpha,beta) QuantumForceNew = QuantumForce(PositionNew,alpha, beta) GreensFunction = 0.0 for j in range(Dimension): GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])* (D*TimeStep*0.5*(QuantumForceOld[i,j]-QuantumForceNew[i,j])- PositionNew[i,j]+PositionOld[i,j])

GreensFunction = exp(GreensFunction) ProbabilityRatio = GreensFunction*wfnew**2/wfold**2 Metropolis-Hastings test to see whether we accept the move if random() <= ProbabilityRatio: for j in range(Dimension): PositionOld[i,j] = PositionNew[i,j] QuantumForceOld[i,j] = QuantumForceNew[i,j] wfold = wfnew DeltaE = LocalEnergy(PositionOld,alpha,beta) DerPsi = DerivativeWFansatz(PositionOld,alpha,beta) DeltaPsi += DerPsi energy += DeltaE DerivativePsiE += DerPsi*DeltaE

We calculate mean values energy /= NumberMCcycles DerivativePsiE /= NumberMCcycles DeltaPsi /= NumberMCcycles EnergyDer = 2*(DerivativePsiE-DeltaPsi*energy) return EnergyDer

Computing the expectation value of the local energy def Energy(x0): Parameters in the Fokker-Planck simulation of the quantum force D = 0.5 TimeStep = 0.05 positions PositionOld = np.zeros((NumberParticles,Dimension), np.double) PositionNew = np.zeros((NumberParticles,Dimension) np.double) Quantum force QuantumForceOld = np.zeros((NumberParticles,Dimension), np.double) QuantumForceNew = np.zeros((NumberParticles,Dimension), np.double)

energy = 0.0 DeltaE = 0.0 alpha = x0[0] beta = x0[1] NumberMCcycles= 10000 Initial position for i in range(NumberParticles): for j in range(Dimension): PositionOld[i,j] = normalvariate(0.0,1.0)*sqrt(TimeStep) wfold = WaveFunction(PositionOld,alpha,beta) QuantumForceOld = QuantumForce(PositionOld,alpha, beta)

Loop over MC MCcycles for MCcycle in range(NumberMCcycles): Trial position moving one particle at the time for i in range(NumberParticles): for j in range(Dimension): PositionNew[i,j] = PositionOld[i,j]+normalvariate(0.0,1.0)*sqrt(TimeStep)+ QuantumForceOld[i,j]*TimeStep*D wfnew = WaveFunction(PositionNew,alpha,beta) QuantumForceNew = QuantumForce(PositionNew,alpha, beta) GreensFunction = 0.0 for j in range(Dimension): GreensFunction += 0.5*(QuantumForceOld[i,j]+QuantumForceNew[i,j])* (D*TimeStep*0.5*(QuantumForceOld[i,j]- QuantumForceNew[i,j])- PositionNew[i,j]+PositionOld[i,j])

GreensFunction = exp(GreensFunction) ProbabilityRatio = GreensFunction*wfnew**2/wfold**2 Metropolis-Hastings test to see whether we accept the move if random() <= ProbabilityRatio: for j in range(Dimension): PositionOld[i,j] = PositionNew[i,j] QuantumForceOld[i,j] = QuantumForceNew[i,j] wfold = wfnew DeltaE = LocalEnergy(PositionOld,alpha,beta) energy += DeltaE

We calculate mean values energy /= NumberMCcycles return energy

Here starts the main program with variable declarations NumberParticles = 2 Dimension = 2 seed for rng generator seed() guess for variational parameters x0 = np.array([0.9,0.2]) Using Broydens method res = minimize(Energy, x0, method='BFGS', jac=EnergyDerivative, options='gtol': 1e-4,'disp': True) print(res.x)

Note that the **minimize** function returns the finale values for the variable $\alpha = x0[0]$ and $\beta = x0[1]$ in the array x.

Brief reminder on Newton-Raphson's method

Let us quickly remind ourselves how we derive the above method.

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method requires the evaluation of both the function f and its derivative f' at arbitrary points. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we normally discourage the use of this method.

The equations

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots.$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x) + (s-x)f'(x) \approx 0,$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}.$$

Having in mind an iterative procedure, it is natural to start iterating with

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Simple geometric interpretation

The above is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n, f(x_n))$ crosses the x-axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally

Extending to more than one variable

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

$$\begin{aligned} f_1(x_1,x_2) &= 0 \\ f_2(x_1,x_2) &= 0, \end{aligned}$$

which we Taylor expand to obtain

$$\begin{aligned} 0 = f_1(x_1+h_1, x_2+h_2) &= f_1(x_1,x_2) + h_1\partial f_1/\partial x_1 + h_2\partial f_1/\partial x_2 + \dots \\ 0 = f_2(x_1+h_1, x_2+h_2) &= f_2(x_1,x_2) + h_1\partial f_2/\partial x_1 + h_2\partial f_2/\partial x_2 + \dots \end{aligned}.$$

Defining the Jacobian matrix $\hat{J}$ we have

$$\hat{J} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{pmatrix},$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^{n} \\ x_2^{n} \end{pmatrix} + \begin{pmatrix} h_1^{n} \\ h_2^{n} \end{pmatrix},$$

where we have defined

$$\begin{pmatrix} h_1^{n} \\ h_2^{n} \end{pmatrix} = -\hat{\mathbf{J}}^{-1} \begin{pmatrix} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{pmatrix}.$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case $\hat{J}$ is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations. In our case, the Jacobian matrix is given by the Hessian that represents the second derivative of cost function.

Steepest descent

The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

with $\gamma_k > 0$.

For γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum.

More on Steepest descent

The previous observation is the basis of the method of steepest descent, which is also referred to as just gradient descent (GD). One starts with an initial guess $\mathbf{x}_0$ for a minimum of F and computes new approximations according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad k \geq 0.$$

The parameter γ_k is often referred to as the step length or the learning rate within the context of Machine Learning.

The ideal

Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a global minimum of the function F. In general we do not know if we are in a global or local minimum. In the special case when F is a convex function, all local minima are also global minima, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement. However the method in this form has some severe limitations:

In machine learing we are often faced with non-convex high dimensional cost functions with many local minima. Since GD is deterministic we will get stuck in a local minimum, if the method converges, unless we have a very good intial guess. This also implies that the scheme is sensitive to the chosen initial condition.

Note that the gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$ which makes it expensive to compute numerically.

The sensitiveness of the gradient descent

The gradient descent method is sensitive to the choice of learning rate γ_k. This is due to the fact that we are only guaranteed that $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$ for sufficiently small γ_k. The problem is to determine an optimal learning rate. If the learning rate is chosen too small the method will take a long time to converge and if it is too large we can experience erratic behavior.

Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD), see below.

Convex functions

Ideally we want our cost/loss function to be convex(concave).

First we give the definition of a convex set: A set C in $\mathbb{R}^n$ is said to be convex if, for all x and y in C and all $t \in (0,1)$, the point $(1t)x + ty$ also belongs to C. Geometrically this means that every point on the line segment connecting x and y is in C as discussed below.

The convex subsets of $\mathbb{R}$ are the intervals of $\mathbb{R}$. Examples of convex sets of $\mathbb{R}^2$ are the regular polygons (triangles, rectangles, pentagons, etc...).

Convex function

Convex function: Let $X \subset \mathbb{R}^n$ be a convex set. Assume that the function $f : X \to \mathbb{R}$ is continuous, then f is said to be convex if

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

for all $x_1, x_2 \in X$ and for all $t \in [0,1]$. If $\leq$ is replaced with a strict inequaltiy in the definition, we demand $x_1 \neq x_2$ and $t \in (0,1)$ then f is said to be strictly convex. For a single variable function, convexity means that if you draw a straight line connecting $f(x_1)$ and $f(x_2)$, the value of the function on the interval $[x_1, x_2]$ is always below the line as illustrated below.

Conditions on convex functions

In the following we state first and second-order conditions which ensures convexity of a function f. We write D_f to denote the domain of f, i.e the subset of R^n where f is defined. For more details and proofs we refer to: S. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press.

First order condition.

Suppose f is differentiable (i.e $\nabla f(x)$ is well defined for all x in the domain of f). Then f is convex if and only if D_f is a convex set and

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

holds for all $x, y \in D_f$. This condition means that for a convex function the first order Taylor expansion (right hand side above) at any point a global under estimator of the function. To convince yourself you can make a drawing of $f(x) = x^2 + 1$ and draw the tangent line to $f(x)$ and note that it is always below the graph.

Second order condition.

Assume that f is twice differentiable, i.e the Hessian matrix exists at each point in D_f. Then f is convex if and only if D_f is a convex set and its Hessian is positive semi-definite for all $x \in D_f$. For a single-variable function this reduces to $f''(x) \geq 0$. Geometrically this means that f has nonnegative curvature everywhere.

This condition is particularly useful since it gives us an procedure for determining if the function under consideration is convex, apart from using the definition.

More on convex functions

The next result is of great importance to us and the reason why we are going on about convex functions. In machine learning we frequently have to minimize a loss/cost function in order to find the best parameters for the model we are considering.

Ideally we want the global minimum (for high-dimensional models it is hard to know if we have local or global minimum). However, if the cost/loss function is convex the following result provides invaluable information:

Any minimum is global for convex functions.

Consider the problem of finding $x \in \mathbb{R}^n$ such that $f(x)$ is minimal, where f is convex and differentiable. Then, any point x^* that satisfies $\nabla f(x^*) = 0$ is a global minimum.

This result means that if we know that the cost/loss function is convex and we are able to find a minimum, we are guaranteed that it is a global minimum.

Some simple problems

1. Show that $f(x) = x^2$ is convex for $x \in \mathbb{R}$ using the definition of convexity. Hint: If you re-write the definition, f is convex if the following holds for all $x, y \in D_f$ and any $\lambda \in [0,1]$ $\lambda f(x) + (1-\lambda) f(y) - f(\lambda x + (1-\lambda) y) \geq 0$.
2. Using the second order condition show that the following functions are convex on the specified domain.
   - $f(x) = e^x$ is convex for $x \in \mathbb{R}$.
   - $g(x) = -\ln(x)$ is convex for $x \in (0, \infty)$.

3. Let $f(x) = x^2$ and $g(x) = e^x$. Show that $f(g(x))$ and $g(f(x))$ is convex for $x \in \mathbb{R}$. Also show that if $f(x)$ is any convex function than $h(x) = e^{f(x)}$ is convex.
4. A norm is any function that satisfy the following properties
   - $f(\alpha x) = |\alpha| f(x)$ for all $\alpha \in \mathbb{R}$.
   - $f(x+y) \leq f(x) + f(y)$
   - $f(x) \leq 0$ for all $x \in \mathbb{R}^n$ with equality if and only if $x = 0$

Using the definition of convexity, try to show that a function satisfying the properties above is convex (the third condition is not needed to show this).

Standard steepest descent

Before we proceed, we would like to discuss the approach called the **standard Steepest descent**, which again leads to us having to be able to compute a matrix. It belongs to the class of Conjugate Gradient methods (CG).

The success of the CG method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{A}\hat{x} = \hat{b}.$$

In the iterative process we end up with a problem like

$$\hat{r} = \hat{b} - \hat{A}\hat{x},$$

where $\hat{r}$ is the so-called residual or error in the iterative process.

When we have found the exact solution, $\hat{r} = 0$.

Gradient method

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{b},$$

with the constraint that the matrix $\hat{A}$ is positive definite and symmetric. This defines also the Hessian and we want it to be positive definite.

Steepest descent method

We denote the initial guess for $\hat{x}$ as $\hat{x}_0$. We can assume without loss of generality that

$$\hat{x}_0 = 0,$$

or consider the system

$$\hat{A}\hat{z} = \hat{b} - \hat{A}\hat{x}_0,$$

instead.

Steepest descent method

One can show that the solution $\hat{x}$ is also the unique minimizer of the quadratic form

$$f(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{x}, \quad \hat{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{r}_1$ (see below for definition) to be the gradient of f at $\hat{x} = \hat{x}_0$, which equals

$$\hat{A}\hat{x}_0 - \hat{b},$$

and $\hat{x}_0 = 0$ it is equal $-\hat{b}$.

Final expressions

We can compute the residual iteratively as

$$\hat{r}_{k+1} = \hat{b} - \hat{A}\hat{x}_{k+1},$$

which equals

$$\hat{b} - \hat{A}(\hat{x}_k + \alpha_k\hat{r}_k),$$

or

$$(\hat{b} - \hat{A}\hat{x}_k) - \alpha_k\hat{A}\hat{r}_k,$$

which gives

$$\alpha_k = \frac{\hat{r}_k^T\hat{r}_k}{\hat{r}_k^T\hat{A}\hat{r}_k}$$

leading to the iterative scheme

$$\hat{x}_{k+1} = \hat{x}_k - \alpha_k\hat{r}_k,$$

Steepest descent example

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
import numpy as np import numpy.linalg as la

import scipy.optimize as sopt

import matplotlib.pyplot as pt from mpl$_{toolkits.mplot3dimportaxes3d}$

def f(x): return 0.5*x[0]**2 + 2.5*x[1]**2

def df(x): return np.array([x[0], 5*x[1]])

fig = pt.figure() ax = fig.gca(projection="3d")

xmesh, ymesh = np.mgrid[-2:2:50j,-2:2:50j] fmesh = f(np.array([xmesh, ymesh])) ax.plot$_{surface(xmesh,ymesh,fmesh)}$

And then as countor plot

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
pt.axis("equal") pt.contour(xmesh, ymesh, fmesh) guesses = [np.array([2, 2./5])]

Find guesses

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python x = guesses[-1] s = -df(x)

Run it!

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python def f1d(alpha): return f(x + alpha*s)

alpha$_opt = sopt.golden(f1d)next_guess = x + alpha_opt * sguesses.append(next_guess)print(next_guess)$

What happened?

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python pt.axis("equal") pt.contour(xmesh, ymesh, fmesh, 50) it$_array = np.array(guesses)pt.plot(it_array.T[0], it_array.T[1], "x-$ ")

Conjugate gradient method

In the CG method we define so-called conjugate directions and two vectors $\hat{s}$ and $\hat{t}$ are said to be conjugate if

$$\hat{s}^T\hat{A}\hat{t} = 0.$$

The philosophy of the CG method is to perform searches in various conjugate directions of our vectors $\hat{x}_i$ obeying the above criterion, namely

$$\hat{x}_i^T\hat{A}\hat{x}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\hat{s}$ is conjugate to $\hat{t}$, then $\hat{t}$ is conjugate to $\hat{s}$.

Conjugate gradient method

An example is given by the eigenvectors of the matrix

$$\hat{v}_i^T\hat{A}\hat{v}_j = \lambda\hat{v}_i^T\hat{v}_j,$$

which is zero unless $i = j$.

Conjugate gradient method

Assume now that we have a symmetric positive-definite matrix $\hat{A}$ of size $n\times n$. At each iteration $i+1$ we obtain the conjugate direction of a vector

$$\hat{x}_{i+1} = \hat{x}_i + \alpha_i\hat{p}_i.$$

We assume that $\hat{p}_i$ is a sequence of n mutually conjugate directions. Then the $\hat{p}_i$ form a basis of R^n and we can expand the solution $\hat{A}\hat{x} = \hat{b}$ in this basis, namely

$$\hat{x} = \sum_{i=1}^{n} \alpha_i \hat{p}_i.$$

Conjugate gradient method

The coefficients are given by

$$\mathbf{Ax} = \sum_{i=1}^{n} \alpha_i \mathbf{Ap}_i = \mathbf{b}.$$

Multiplying with $\hat{p}_k^T$ from the left gives

$$\hat{p}_k^T \hat{A}\hat{x} = \sum_{i=1}^{n} \alpha_i \hat{p}_k^T \hat{A}\hat{p}_i = \hat{p}_k^T \hat{b},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\hat{p}_k^T \hat{b}}{\hat{p}_k^T \hat{A}\hat{p}_k}$$

Conjugate gradient method and iterations

If we choose the conjugate vectors $\hat{p}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\hat{x}$. We want to regard the conjugate gradient method as an iterative method. This will us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for $\hat{x}$ as $\hat{x}_0$. We can assume without loss of generality that

$$\hat{x}_0 = 0,$$

or consider the system

$$\hat{A}\hat{z} = \hat{b} - \hat{A}\hat{x}_0,$$

instead.

Conjugate gradient method

One can show that the solution $\hat{x}$ is also the unique minimizer of the quadratic form

$$f(\hat{x}) = \frac{1}{2}\hat{x}^T \hat{A}\hat{x} - \hat{x}^T \hat{x}, \quad \hat{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{p}_1$ to be the gradient of f at $\hat{x} = \hat{x}_0$, which equals

$$\hat{A}\hat{x}_0 - \hat{b},$$

and $\hat{x}_0 = 0$ it is equal $-\hat{b}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Conjugate gradient method

Let $\hat{r}_k$ be the residual at the k-th step:

$$\hat{r}_k = \hat{b} - \hat{A}\hat{x}_k.$$

Note that $\hat{r}_k$ is the negative gradient of f at $\hat{x} = \hat{x}_k$, so the gradient descent method would be to move in the direction $\hat{r}_k$. Here, we insist that the directions $\hat{p}_k$ are conjugate to each other, so we take the direction closest to the gradient $\hat{r}_k$ under the conjugacy constraint. This gives the following expression

$$\hat{p}_{k+1} = \hat{r}_k - \frac{\hat{p}_k^T \hat{A}\hat{r}_k}{\hat{p}_k^T \hat{A}\hat{p}_k}\hat{p}_k.$$

Conjugate gradient method

We can also compute the residual iteratively as

$$\hat{r}_{k+1} = \hat{b} - \hat{A}\hat{x}_{k+1},$$

which equals

$$\hat{b} - \hat{A}(\hat{x}_k + \alpha_k \hat{p}_k),$$

or

$$(\hat{b} - \hat{A}\hat{x}_k) - \alpha_k \hat{A}\hat{p}_k,$$

which gives

$$\hat{r}_{k+1} = \hat{r}_k - \hat{A}\hat{p}_k,$$

Broyden–Fletcher–Goldfarb–Shanno algorithm

The optimization problem is to minimize $f(\mathbf{x})$ where $\mathbf{x}$ is a vector in R^n, and f is a differentiable scalar function. There are no constraints on the values that $\mathbf{x}$ can take.

The algorithm begins at an initial estimate for the optimal value $\mathbf{x}_0$ and proceeds iteratively to get a better estimate at each stage.

The search direction p_k at stage k is given by the solution of the analogue of the Newton equation

$$B_k\mathbf{p}_k = -\nabla f(\mathbf{x}_k),$$

where B_k is an approximation to the Hessian matrix, which is updated iteratively at each stage, and $\nabla f(\mathbf{x}_k)$ is the gradient of the function evaluated at x_k. A line search in the direction p_k is then used to find the next point x_{k+1} by minimising

$$f(\mathbf{x}_k + \alpha\mathbf{p}_k),$$

over the scalar $\alpha > 0$.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above.

The underlying idea of SGD comes from the observation that a given function, which we want to minimize, can almost always be written as a sum over n data points $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\beta) = \sum_{i=1}^n c_i(\mathbf{x}_i, \beta).$$

Computation of gradients

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_i^n \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

Stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

SGD example

As an example, suppose we have 10 data points $(\mathbf{x}_1, \cdots, \mathbf{x}_{10})$ and we choose to have $M = 5$ minibathces, then each minibatch contains two data points. In particular we have $B_1 = (\mathbf{x}_1, \mathbf{x}_2), \cdots, B_5 = (\mathbf{x}_9, \mathbf{x}_{10})$. Note that if you choose $M = 1$ you have only a single batch with all data points and on the other extreme, you may choose $M = n$ resulting in a minibatch for each datapoint, i.e $B_k = \mathbf{x}_k$.

The idea is now to approximate the gradient by replacing the sum over all data points with a sum over the data points in one the minibatches picked at random in each gradient descent step

$$\nabla_\beta C(\beta) = \sum_{i=1}^n \nabla_\beta c_i(\mathbf{x}_i, \beta) \to \sum_{i\in B_k}^n \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

The gradient step

Thus a gradient descent step now looks like

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibathces (n/M) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches, as exemplified in the code below.

Simple example code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
import numpy as np

n = 100 100 datapoints M = 5 size of each minibatch m = int(n/M) number of minibatches n$_e pochs = 10 number of epochs$

j = 0 for epoch in range(1,n$_e pochs+1): foriinrange(m): k = np.random.randint(m) Pickthek-thminibatchatrandomComputethegrad$
1

Taking the gradient only on a subset of the data has two important benefits. First, it introduces randomness which decreases the chance that our opmization scheme gets stuck in a local minima. Second, if the size of the minibatches are small relative to the number of datapoints ($M < n$), the computation of the gradient is much cheaper since we sum over the datapoints in the $k-th$ minibatch and not all n datapoints.

When do we stop?

A natural question is when do we stop the search for a new minimum? One possibility is to compute the full gradient after a given number of epochs and check if the norm of the gradient is smaller than some threshold and stop if true. However, the condition that the gradient is zero is valid also for local minima, so this would only tell us that we are close to a local/global minimum. However, we could also evaluate the cost function at this point, store the result and continue the search. If the test kicks in at a later stage we can compare the values of the cost function and keep the β that gave the lowest value.

Slightly different approach

Another approach is to let the step length γ_j depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all.

As an example, let $e = 0, 1, 2, 3, \cdots$ denote the current epoch and let $t_0, t_1 > 0$ be two fixed numbers. Furthermore, let $t = e \cdot m + i$ where m is the number of minibatches and $i = 0, \cdots, m-1$. Then the function

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1}$$

goes to zero as the number of epochs gets large. I.e. we start with a step length $\gamma_j(0;t_0,t_1) = t_0/t_1$ which decays in *time* t.

In this way we can fix the number of epochs, compute β and evaluate the cost function at the end. Repeating the computation will give a different result since the scheme is random by design. Then we pick the final β that gives the lowest value of the cost function.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python import numpy as np

def step$_length(t,t0,t1) : returnt0/(t+t1)$

n = 100 100 datapoints M = 5 size of each minibatch m = int(n/M) number of minibatches n$_epochs = 500numberofepochst0 = 1.0t1 = 10$

gamma$_j = t0/t1 j = 0forepochinrange(1,n_epochs+1) : foriinrange(m) : k = np.random.randint(m)Pickthek-thminibatchatrandomComputethegradientusingthedatainminibatchBkComputenewsuggestionforbetat = epoch*m+igamma_j = step_length(t,t0,t1) j+ = 1$

print("gamma$_jafter$

Program for stochastic gradient

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python Importing various packages from math import exp, sqrt from random import random, seed import numpy as np import matplotlib.pyplot as plt from sklearn.linear$_modelimportSGDRegressor$

x = 2*np.random.rand(100,1) y = 4+3*x+np.random.randn(100,1)

xb = np.c$_[np.ones((100,1)),x]theta_linreg = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y)print("Owninversion")print(theta_linreg)sgdreg = SGDRegressor(n_iter = 50, penalty = None, eta0 = 0.1)sgdreg.fit(x,y.ravel())print("sgdregfromscikit")print(sgdreg.intercept_,sgdreg.coef_)$

theta = np.random.randn(2,1)

eta = 0.1 Niterations = 1000 m = 100

for iter in range(Niterations): gradients = 2.0/m*xb.T.dot(xb.dot(theta)-y) theta -= eta*gradients print("theta frm own gd") print(theta)

xnew = np.array([[0],[2]]) xbnew = np.c$_[np.ones((2,1)),xnew]ypredict = xbnew.dot(theta)ypredict2 = xbnew.dot(theta_linreg)$

n$_epochs = 50t0,t1 = 5,50m = 100deflearning_schedule(t) : returnt0/(t+t1)$

theta = np.random.randn(2,1)

for epoch in range(n$_epochs) : foriinrange(m) : random_index = np.random.randint(m)xi = xb[random_index : random_index+1]yi = y[random_index : random_index+1]gradients = 2*xi.T.dot(xi.dot(theta)-yi)eta = learning_schedule(epoch*m+i)theta = theta-eta*gradientsprint("thetafromownsdg")print(theta)$

plt.plot(xnew, ypredict, "r-") plt.plot(xnew, ypredict2, "b-") plt.plot(x, y ,'ro') plt.axis([0,2.0,0, 15.0]) plt.xlabel(r'x') plt.ylabel(r'y') plt.title(r'Random numbers ') plt.show()

Using gradient descent methods, limitations

- **Gradient descent (GD) finds local minima of our function**. Since the GD algorithm is deterministic, if it converges, it will converge to a local minimum of our energy function. Because in ML we are often dealing with extremely rugged landscapes with many local minima, this can lead to poor performance.
- **GD is sensitive to initial conditions**. One consequence of the local nature of GD is that initial conditions matter. Depending on where one starts, one will end up at a different local minima. Therefore, it is very important to think about how one initializes the training process. This is true for GD as well as more complicated variants of GD.

- **Gradients are computationally expensive to calculate for large datasets**. In many cases in statistics and ML, the energy function is a sum of terms, with one term for each data point. For example, in linear regression, $E \propto \sum_{i=1}^{n}(y_i - \mathbf{w}^T \cdot \mathbf{x}_i)^2$; for logistic regression, the square error is replaced by the cross entropy. To calculate the gradient we have to sum over *all* n data points. Doing this at every GD step becomes extremely computationally expensive. An ingenious solution to this, is to calculate the gradients using small subsets of the data called “mini batches”. This has the added benefit of introducing stochasticity into our algorithm.
- **GD is very sensitive to choices of learning rates**. GD is extremely sensitive to the choice of learning rates. If the learning rate is very small, the training process take an extremely long time. For larger learning rates, GD can diverge and give poor results. Furthermore, depending on what the local landscape looks like, we have to modify the learning rates to ensure convergence. Ideally, we would *adaptively* choose the learning rates to match the landscape.
- **GD treats all directions in parameter space uniformly.** Another major drawback of GD is that unlike Newton’s method, the learning rate for GD is the same in all directions in parameter space. For this reason, the maximum learning rate is set by the behavior of the steepest direction and this can significantly slow down training. Ideally, we would like to take large steps in flat directions and small steps in steep directions. Since we are exploring rugged landscapes where curvatures change, this requires us to keep track of not only the gradient but second derivatives. The ideal scenario would be to calculate the Hessian but this proves to be too computationally expensive.
- GD can take exponential time to escape saddle points, even with random initialization. As we mentioned, GD is extremely sensitive to initial condition since it determines the particular local minimum GD would eventually reach. However, even with a good initialization scheme, through the introduction of randomness, GD can still take exponential time to escape saddle points.

Codes from numerical recipes

You can however use codes we have adapted from the text Numerical Recipes in C++, see chapter 10.7. Here we present a program, which you also can find at the webpage of the course we use the functions **dfpmin** and **lnsrch**. This is a variant of the Broyden et al algorithm discussed in the previous slide.

- The program uses the harmonic oscillator in one dimensions as example.
- The program does not use armadillo to handle vectors and matrices, but employs rather my own vector-matrix class. These auxiliary functions, and the main program *model.cpp* can all be found under the program link here.

Below we show only excerpts from the main program. For the full program, see the above link.

Finding the minimum of the harmonic oscillator model in one dimension

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ // Main function begins here int main() int n, iter; double gtol, fret; double alpha; n = 1; // reserve space in memory for vectors containing the variational // parameters Vector g(n), p(n); cout « "Read in guess for alpha" « endl; cin » alpha; gtol = 1.0e-5; // now call dfmin and compute the minimum p(0) = alpha; dfpmin(p, n, gtol, iter, fret, Efunction, dEfunction); cout « "Value of energy minimum = " « fret « endl; cout « "Number of iterations = " « iter « endl; cout « "Value of alpha at minimum = " « p(0) « endl; return 0; // end of main program

Functions to observe

The functions **Efunction** and **dEfunction** compute the expectation value of the energy and its derivative. They use the the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) It uses the first derivatives only. The BFGS algorithm has proven good performance even for non-smooth optimizations. These functions need to be changed when you want to your own derivatives.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ // this function defines the expectation value of the local energy double Efunction(Vector x) double value = x(0)*x(0)*0.5+1.0/(8*x(0)*x(0)); return value; // end of function to evaluate

// this function defines the derivative of the energy void dEfunction(Vector x, Vector g) g(0) = x(0)-1.0/(4*x(0)*x(0)*x(0)); // end of function to evaluate

You need to change these functions in order to compute the local energy for your system. I used 1000 cycles per call to get a new value of $\langle E_L[\alpha]\rangle$. When I compute the local energy I also compute its derivative. After roughly 10-20 iterations I got a converged result in terms of α.

Chapter 7
Resampling Techniques, Bootstrap and Blocking

Why resampling methods ?

Statistical analysis.

- Our simulations can be treated as *computer experiments*. This is particularly the case for Monte Carlo methods
- The results can be analysed with the same statistical tools as we would use analysing experimental data.
- As in all experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., possible sources for errors.

Statistical analysis

- As in other experiments, many numerical experiments have two classes of errors:
  - Statistical errors
  - Systematical errors
- Statistical errors can be estimated using standard tools from statistics
- Systematical errors are method specific and must be treated differently from case to case.

Statistics, wrapping up from last week

Let us analyze the problem by splitting up the correlation term into partial sums of the form:

$$f_d = \frac{1}{n-d}\sum_{k=1}^{n-d}(x_k-\bar{x}_n)(x_{k+d}-\bar{x}_n)$$

The correlation term of the error can now be rewritten in terms of f_d

$$\frac{2}{n}\sum_{k<l}(x_k-\bar{x}_n)(x_l-\bar{x}_n)=2\sum_{d=1}^{n-1}f_d$$

The value of f_d reflects the correlation between measurements separated by the distance d in the sample samples. Notice that for $d=0$, f is just the sample variance, $\mathrm{var}(x)$. If we divide f_d by $\mathrm{var}(x)$, we arrive at the so called *autocorrelation function*

$$\kappa_d=\frac{f_d}{\mathrm{var}(x)}$$

which gives us a useful measure of pairwise correlations starting always at 1 for $d=0$.

Statistics, final expression

The sample error can now be written in terms of the autocorrelation function:

$$\begin{aligned}\mathrm{err}_X^2 &= \frac{1}{n}\mathrm{var}(x)+\frac{2}{n}\cdot\mathrm{var}(x)\sum_{d=1}^{n-1}\frac{f_d}{\mathrm{var}(x)}\\ &= \left(1+2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\mathrm{var}(x)\\ &= \frac{\tau}{n}\cdot\mathrm{var}(x)\end{aligned}\tag{7.1}$$

and we see that err_X can be expressed in terms the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*:

$$\tau=1+2\sum_{d=1}^{n-1}\kappa_d\tag{7.2}$$

Statistics, effective number of correlations

For a correlation free experiment, τ equals 1.

We can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes:

$$n_{\mathrm{eff}}=\frac{n}{\tau}$$

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\mathrm{err}_X^2\approx\mathrm{var}(x)/n$ to be less than the true sample error. The estimate of the error will be too *good*. On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large.

Can we understand this? Time Auto-correlation Function

The so-called time-displacement autocorrelation $\phi(t)$ for a quantity $\mathbf{M}$ is given by

$$\phi(t) = \int dt' \left[\mathbf{M}(t') - \langle \mathbf{M} \rangle\right] \left[\mathbf{M}(t'+t) - \langle \mathbf{M} \rangle\right],$$

which can be rewritten as

$$\phi(t) = \int dt' \left[\mathbf{M}(t')\mathbf{M}(t'+t) - \langle \mathbf{M} \rangle^2\right],$$

where $\langle \mathbf{M} \rangle$ is the average value and $\mathbf{M}(t)$ its instantaneous value. We can discretize this function as follows, where we used our set of computed values $\mathbf{M}(t)$ for a set of discretized times (our Monte Carlo cycles corresponding to moving all electrons?)

$$\phi(t) = \frac{1}{t_{\max} - t} \sum_{t'=0}^{t_{\max}-t} \mathbf{M}(t')\mathbf{M}(t'+t) - \frac{1}{t_{\max} - t} \sum_{t'=0}^{t_{\max}-t} \mathbf{M}(t') \times \frac{1}{t_{\max} - t} \sum_{t'=0}^{t_{\max}-t} \mathbf{M}(t'+t).$$

Time Auto-correlation Function

One should be careful with times close to $t_{\max}$, the upper limit of the sums becomes small and we end up integrating over a rather small time interval. This means that the statistical error in $\phi(t)$ due to the random nature of the fluctuations in $\mathbf{M}(t)$ can become large.

One should therefore choose $t \ll t_{\max}$.

Note that the variable $\mathbf{M}$ can be any expectation values of interest.

The time-correlation function gives a measure of the correlation between the various values of the variable at a time t' and a time $t'+t$. If we multiply the values of $\mathbf{M}$ at these two different times, we will get a positive contribution if they are fluctuating in the same direction, or a negative value if they fluctuate in the opposite direction. If we then integrate over time, or use the discretized version of, the time correlation function $\phi(t)$ should take a non-zero value if the fluctuations are correlated, else it should gradually go to zero. For times a long way apart the different values of $\mathbf{M}$ are most likely uncorrelated and $\phi(t)$ should be zero.

Time Auto-correlation Function

We can derive the correlation time by observing that our Metropolis algorithm is based on a random walk in the space of all possible spin configurations. Our probability distribution function $\hat{\mathbf{w}}(t)$ after a given number of time steps t could be written as

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^{\mathbf{t}} \hat{\mathbf{w}}(0),$$

with $\hat{\mathbf{w}}(0)$ the distribution at $t = 0$ and $\hat{\mathbf{W}}$ representing the transition probability matrix. We can always expand $\hat{\mathbf{w}}(0)$ in terms of the right eigenvectors of $\hat{\mathbf{v}}$ of $\hat{\mathbf{W}}$ as

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i \hat{\mathbf{v}}_i,$$

resulting in

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t \hat{\mathbf{w}}(0) = \hat{\mathbf{W}}^t \sum_i \alpha_i \hat{\mathbf{v}}_i = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i,$$

with λ_i the i^{th} eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

Time Auto-correlation Function

If we assume that λ_0 is the largest eigenvector we see that in the limit $t \to \infty$, $\hat{\mathbf{w}}(t)$ becomes proportional to the corresponding eigenvector $\hat{\mathbf{v}}_0$. This is our steady state or final distribution.

We can relate this property to an observable like the mean energy. With the probabilty $\hat{\mathbf{w}}(t)$ (which in our case is the squared trial wave function) we can write the expectation values as

$$\langle \mathbf{M}(t) \rangle = \sum_\mu \hat{\mathbf{w}}(t)_\mu \mathbf{M}_\mu,$$

or as the scalar of a vector product

$$\langle \mathbf{M}(t) \rangle = \hat{\mathbf{w}}(t) \mathbf{m},$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathbf{M}_\mu$ in its various microstates μ.

Time Auto-correlation Function

We rewrite this relation as

$$\langle \mathbf{M}(t) \rangle = \hat{\mathbf{w}}(t) \mathbf{m} = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i \mathbf{m}_i.$$

If we define $m_i = \hat{\mathbf{v}}_i \mathbf{m}_i$ as the expectation value of $\mathbf{M}$ in the i^{th} eigenstate we can rewrite the last equation as

$$\langle \mathbf{M}(t) \rangle = \sum_i \lambda_i^t \alpha_i m_i.$$

Since we have that in the limit $t \to \infty$ the mean value is dominated by the the largest eigenvalue λ_0, we can rewrite the last equation as

$$\langle \mathbf{M}(t) \rangle = \langle \mathbf{M}(\infty) \rangle + \sum_{i \neq 0} \lambda_i^t \alpha_i m_i.$$

We define the quantity

$$\tau_i = -\frac{1}{log \lambda_i},$$

and rewrite the last expectation value as

$$\langle \mathbf{M}(t) \rangle = \langle \mathbf{M}(\infty) \rangle + \sum_{i \neq 0} \alpha_i m_i e^{-t/\tau_i}.$$

Time Auto-correlation Function

The quantities τ_i are the correlation times for the system. They control also the auto-correlation function discussed above. The longest correlation time is obviously given by the second largest eigenvalue τ_1, which normally defines the correlation time discussed above. For large times, this is the only correlation time that survives. If higher eigenvalues of the transition matrix are well separated from λ_1 and we simulate long enough, τ_1 may well define the correlation time. In other cases we may not be able to extract a reliable result for τ_1. Coming back to the time correlation function $\phi(t)$ we can present a more general definition in terms of the mean magnetizations $\langle \mathbf{M}(t)\rangle$. Recalling that the mean value is equal to $\langle \mathbf{M}(\infty)\rangle$ we arrive at the expectation values

$$\phi(t) = \langle \mathbf{M}(0) - \mathbf{M}(\infty)\rangle\langle \mathbf{M}(t) - \mathbf{M}(\infty)\rangle,$$

resulting in

$$\phi(t) = \sum_{i,j\neq 0} m_i\alpha_i m_j\alpha_j e^{-t/\tau_i},$$

which is appropriate for all times.

Correlation Time

If the correlation function decays exponentially

$$\phi(t) \sim \exp{(-t/\tau)}$$

then the exponential correlation time can be computed as the average

$$\tau_{\mathrm{exp}} = -\langle \frac{t}{log|\frac{\phi(t)}{\phi(0)}|}\rangle.$$

If the decay is exponential, then

$$\int_0^{\infty} dt\phi(t) = \int_0^{\infty} dt\phi(0)\exp{(-t/\tau)} = \tau\phi(0),$$

which suggests another measure of correlation

$$\tau_{\mathrm{int}} = \sum_k \frac{\phi(k)}{\phi(0)},$$

called the integrated correlation time.

Resampling methods: Jackknife and Bootstrap

Two famous resampling methods are the **independent bootstrap** and **the jackknife**.

The jackknife is a special case of the independent bootstrap. Still, the jackknife was made popular prior to the independent bootstrap. And as the popularity of the independent bootstrap soared, new variants, such as **the dependent bootstrap**.

The Jackknife and independent bootstrap work for independent, identically distributed random variables. If these conditions are not satisfied, the methods will fail. Yet, it should be said that if the data are independent, identically distributed, and we only want to estimate the variance of $\overline{X}$ (which often is the case), then there is no need for bootstrapping.

Resampling methods: Jackknife

The Jackknife works by making many replicas of the estimator $\widehat{\theta}$. The jackknife is a resampling method, we explained that this happens by scrambling the data in some way. When using the jackknife, this is done by systematically leaving out one observation from the vector of observed values $\hat{x} = (x_1, x_2, \cdots, X_n)$. Let $\hat{x}_i$ denote the vector

$$\hat{x}_i = (x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n),$$

which equals the vector $\hat{x}$ with the exception that observation number i is left out. Using this notation, define $\widehat{\theta}_i$ to be the estimator $\widehat{\theta}$ computed using X_i.

Resampling methods: Jackknife estimator

To get an estimate for the bias and standard error of $\widehat{\theta}$, use the following estimators for each component of $\widehat{\theta}$

$$\widehat{\mathrm{Bias}}(\widehat{\theta}, \theta) = (n-1)\left(-\widehat{\theta} + \frac{1}{n}\sum_{i=1}^{n}\widehat{\theta}_i\right) \qquad \text{and} \qquad \widehat{\sigma}^2_{\widehat{\theta}} = \frac{n-1}{n}\sum_{i=1}^{n}(\widehat{\theta}_i - \frac{1}{n}\sum_{j=1}^{n}\widehat{\theta}_j)^2.$$

Jackknife code example

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from numpy import * from numpy.random import randint, randn from time import time

def jackknife(data, stat): n = len(data);t = zeros(n); inds = arange(n); t0 = time() 'jackknifing' by leaving out an observation for each i for i in range(n): t[i] = stat(delete(data,i))

analysis print("Runtime: print("original bias std. error") print("

return t

Returns mean of data samples def stat(data): return mean(data)

mu, sigma = 100, 15 datapoints = 10000 x = mu + sigma*random.randn(datapoints) jackknife returns the data sample t = jackknife(x, stat)

Resampling methods: Bootstrap

Bootstrapping is a nonparametric approach to statistical inference that substitutes computation for more traditional distributional assumptions and asymptotic results. Bootstrapping offers a number of advantages:

1. The bootstrap is quite general, although there are some cases in which it fails.
2. Because it does not require distributional assumptions (such as normally distributed errors), the bootstrap can provide more accurate inferences when the data are not well behaved or when the sample size is small.
3. It is possible to apply the bootstrap to statistics with sampling distributions that are difficult to derive, even asymptotically.
4. It is relatively simple to apply the bootstrap to complex data-collection plans (such as stratified and clustered samples).

Resampling methods: Bootstrap background

Since $\widehat{\theta} = \widehat{\theta}(\hat{X})$ is a function of random variables, $\widehat{\theta}$ itself must be a random variable. Thus it has a pdf, call this function $p(\hat{t})$. The aim of the bootstrap is to estimate $p(\hat{t})$ by the relative frequency of $\widehat{\theta}$. You can think of this as using a histogram in the place of $p(\hat{t})$. If the relative frequency closely resembles $p(t)$, then using numerics, it is straight forward to estimate all the interesting parameters of $p(\hat{t})$ using point estimators.

Resampling methods: More Bootstrap background

In the case that $\widehat{\theta}$ has more than one component, and the components are independent, we use the same estimator on each component separately. If the probability density function of X_i, $p(x)$, had been known, then it would have been straight forward to do this by:

1. Drawing lots of numbers from $p(x)$, suppose we call one such set of numbers $(X_1^*, X_2^*, \cdots, X_n^*)$.
2. Then using these numbers, we could compute a replica of $\widehat{\theta}$ called $\widehat{\theta}^*$.

By repeated use of (1) and (2), many estimates of $\widehat{\theta}$ could have been obtained. The idea is to use the relative frequency of $\widehat{\theta}^*$ (think of a histogram) as an estimate of $p(\hat{t})$.

Resampling methods: Bootstrap approach

But unless there is enough information available about the process that generated $X_1, X_2, \cdots, X_n$, $p(x)$ is in general unknown. Therefore, Efron in 1979 asked the question: What if we replace $p(x)$ by the relative frequency of the observation X_i; if we draw observations in accordance with the relative frequency of the observations, will we obtain the same result in some asymptotic sense? The answer is yes.

Instead of generating the histogram for the relative frequency of the observation X_i, just draw the values $(X_1^*, X_2^*, \cdots, X_n^*)$ with replacement from the vector $\hat{X}$.

Resampling methods: Bootstrap steps

The independent bootstrap works like this:

1. Draw with replacement n numbers for the observed variables $\hat{x} = (x_1, x_2, \cdots, x_n)$.
2. Define a vector $\hat{x}^*$ containing the values which were drawn from $\hat{x}$.
3. Using the vector $\hat{x}^*$ compute $\widehat{\theta}^*$ by evaluating $\widehat{\theta}$ under the observations $\hat{x}^*$.
4. Repeat this process k times.

When you are done, you can draw a histogram of the relative frequency of $\widehat{\theta}^*$. This is your estimate of the probability distribution $p(t)$. Using this probability distribution you can estimate any statistics thereof. In principle you never draw the histogram of the relative frequency of $\widehat{\theta}^*$. Instead you use the estimators corresponding to the statistic of interest. For example, if you are interested in estimating the variance of $\widehat{\theta}$, apply the etsimator $\widehat{\sigma}^2$ to the values $\widehat{\theta}^*$.

Code example for the Bootstrap method

The following code starts with a Gaussian distribution with mean value $\mu = 100$ and variance $\sigma = 15$. We use this to generate the data used in the bootstrap analysis. The bootstrap analysis returns a data set after a given number of bootstrap operations (as many as we have data points). This data set consists of estimated mean values for each bootstrap operation. The histogram generated by the bootstrap method shows that the distribution for these mean values is also a Gaussian, centered around the mean value $\mu = 100$ but with standard deviation $\sigma/\sqrt{n}$, where n is the number of bootstrap samples (in this case the same as the number of original data points). The value of the standard deviation is what we expect from the central limit theorem.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python

from numpy import * from numpy.random import randint, randn from time import time from scipy.stats import norm import matplotlib.pyplot as plt

Returns mean of bootstrap samples def stat(data): return mean(data)

Bootstrap algorithm def bootstrap(data, statistic, R): t = zeros(R); n = len(data); inds = arange(n); t0 = time()

non-parametric bootstrap for i in range(R): t[i] = statistic(data[randint(0,n,n)])

analysis print("Runtime: print("original bias std. error") print("mean(t), std(t))) return t

mu, sigma = 100, 15 datapoints = 10000 x = mu + sigma*random.randn(datapoints) bootstrap returns the data sample t = bootstrap(x, stat, datapoints) the histogram of the bootstrapped data t = bootstrap(x, stat, datapoints) the histogram of the bootstrapped data n, binsboot, patches = plt.hist(t, bins=50, density='true',histtype='bar', color='red', alpha=0.75)

add a 'best fit' line y = norm.pdf(binsboot, mean(t), std(t)) lt = plt.plot(binsboot, y, 'r–', linewidth=1) plt.xlabel('Smarts') plt.ylabel('Probability') plt.axis([99.5, 100.6, 0, 3.0]) plt.grid(True)

plt.show()

Resampling methods: Blocking

The blocking method was made popular by Flyvbjerg and Pedersen (1989) and has become one of the standard ways to estimate $V(\widehat{\theta})$ for exactly one $\widehat{\theta}$, namely $\widehat{\theta} = \overline{X}$.

Assume $n = 2^d$ for some integer $d > 1$ and $X_1, X_2, \cdots, X_n$ is a stationary time series to begin with. Moreover, assume that the time series is asymptotically uncorrelated. We switch to vector notation by arranging $X_1, X_2, \cdots, X_n$ in an n-tuple. Define:

$$\hat{X} = (X_1, X_2, \cdots, X_n).$$

The strength of the blocking method is when the number of observations, n is large. For large n, the complexity of dependent bootstrapping scales poorly, but the blocking method does not, moreover, it becomes more accurate the larger n is.

Blocking Transformations

We now define blocking transformations. The idea is to take the mean of subsequent pair of elements from X and form a new vector X_1. Continuing in the same way by taking the mean of subsequent pairs of elements of X_1 we obtain X_2, and so on. Define X_i recursively by:

$$\begin{aligned}(X_0)_k &\equiv (X)_k \\ (X_{i+1})_k &\equiv \frac{1}{2}\Big((X_i)_{2k-1} + (X_i)_{2k}\Big) \qquad \text{for all} \qquad 1 \leq i \leq d-1 \qquad (7.3)\end{aligned}$$

The quantity X_k is subject to k **blocking transformations**. We now have d vectors $X_0, X_1, \cdots, X_{d-1}$ containing the subsequent averages of observations. It turns out that if the components of X is a stationary time series, then the components of X_i is a stationary time series for all $0 \leq i \leq d-1$

We can then compute the autocovariance, the variance, sample mean, and number of observations for each i. Let $\gamma_i, \sigma_i^2, \overline{X}_i$ denote the autocovariance, variance and average of the elements of X_i and let n_i be the number of elements of X_i. It follows by induction that $n_i = n/2^i$.

Blocking Transformations

Using the definition of the blocking transformation and the distributive property of the covariance, it is clear that since $h = |i - j|$ we can define

$$\begin{aligned}\gamma_{k+1}(h) &= cov\left((X_{k+1})_i, (X_{k+1})_j\right) \\ &= \frac{1}{4}cov\left((X_k)_{2i-1} + (X_k)_{2i}, (X_k)_{2j-1} + (X_k)_{2j}\right) \\ &= \frac{1}{2}\gamma_k(2h) + \frac{1}{2}\gamma_k(2h+1) \; \mathrm{h} = 0 \qquad (7.4) \\ &= \frac{1}{4}\gamma_k(2h-1) + \frac{1}{2}\gamma_k(2h) + \frac{1}{4}\gamma_k(2h+1) \quad \text{else} \qquad (7.5)\end{aligned}$$

The quantity $\hat{X}$ is asymptotic uncorrelated by assumption, $\hat{X}_k$ is also asymptotic uncorrelated. Let's turn our attention to the variance of the sample mean $V(\overline{X})$.

Blocking Transformations, getting there

We have

$$V(\overline{X}_k) = \frac{\sigma_k^2}{n_k} + \underbrace{\frac{2}{n_k}\sum_{h=1}^{n_k-1}\left(1-\frac{h}{n_k}\right)\gamma_k(h)}_{\equiv e_k} = \frac{\sigma_k^2}{n_k} + e_k \quad \text{if} \quad \gamma_k(0) = \sigma_k^2. \tag{7.6}$$

The term e_k is called the **truncation error**:

$$e_k = \frac{2}{n_k}\sum_{h=1}^{n_k-1}\left(1-\frac{h}{n_k}\right)\gamma_k(h). \tag{7.7}$$

We can show that $V(\overline{X}_i) = V(\overline{X}_j)$ for all $0 \le i \le d-1$ and $0 \le j \le d-1$.

Blocking Transformations, final expressions

We can then wrap up

$$\begin{aligned} n_{j+1}\overline{X}_{j+1} &= \sum_{i=1}^{n_{j+1}} (\hat{X}_{j+1})_i = \frac{1}{2}\sum_{i=1}^{n_j/2} (\hat{X}_j)_{2i-1} + (\hat{X}_j)_{2i} \\ &= \frac{1}{2}\left[(\hat{X}_j)_1 + (\hat{X}_j)_2 + \cdots + (\hat{X}_j)_{n_j}\right] = \underbrace{\frac{n_j}{2}}_{=n_{j+1}} \overline{X}_j = n_{j+1}\overline{X}_j. \end{aligned} \tag{7.8}$$

By repeated use of this equation we get $V(\overline{X}_i) = V(\overline{X}_0) = V(\overline{X})$ for all $0 \le i \le d-1$. This has the consequence that

$$V(\overline{X}) = \frac{\sigma_k^2}{n_k} + e_k \qquad \text{for all} \qquad 0 \le k \le d-1. \tag{7.9}$$

Flyvbjerg and Petersen demonstrated that the sequence $\{e_k\}_{k=0}^{d-1}$ is decreasing, and conjecture that the term e_k can be made as small as we would like by making k (and hence d) sufficiently large. The sequence is decreasing (Master of Science thesis by Marius Jonsson, UiO 2018). It means we can apply blocking transformations until e_k is sufficiently small, and then estimate $V(\overline{X})$ by $\widehat{\sigma}_k^2/n_k$.

For an elegant solution and proof of the blocking method, see the recent article of Marius Jonsson (former MSc student of the Computational Physics group).

Chapter 8
Optimization and Vectorization

Optimization and profiling

Till now we have not paid much attention to speed and possible optimization possibilities inherent in the various compilers. We have compiled and linked as

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ c++ -c mycode.cpp c++ -o mycode.exe mycode.o

For Fortran replace with for example **gfortran** or **ifort**. This is what we call a flat compiler option and should be used when we develop the code. It produces normally a very large and slow code when translated to machine instructions. We use this option for debugging and for establishing the correct program output because every operation is done precisely as the user specified it.

It is instructive to look up the compiler manual for further instructions by writing

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ man c++

More on optimization

We have additional compiler options for optimization. These may include procedure inlining where performance may be improved, moving constants inside loops outside the loop, identify potential parallelism, include automatic vectorization or replace a division with a reciprocal and a multiplication if this speeds up the code.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ c++ -O3 -c mycode.cpp c++ -O3 -o mycode.exe mycode.o

This (other options are -O2 or -Ofast) is the recommended option.

Optimization and profiling

It is also useful to profile your program under the development stage. You would then compile with

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ c++ -pg -O3 -c mycode.cpp c++ -pg -O3 -o mycode.exe mycode.o

After you have run the code you can obtain the profiling information via

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ gprof mycode.exe > ProfileOutput

When you have profiled properly your code, you must take out this option as it slows down performance. For memory tests use valgrind. An excellent environment for all these aspects, and much more, is Qt creator.

Optimization and debugging

Adding debugging options is a very useful alternative under the development stage of a program. You would then compile with

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ c++ -g -O0 -c mycode.cpp c++ -g -O0 -o mycode.exe mycode.o

This option generates debugging information allowing you to trace for example if an array is properly allocated. Some compilers work best with the no optimization option **-O0**.

Other optimization flags.

Depending on the compiler, one can add flags which generate code that catches integer overflow errors. The flag **-ftrapv** does this for the CLANG compiler on OS X operating systems.

Other hints

In general, irrespective of compiler options, it is useful to

- avoid if tests or call to functions inside loops, if possible.
- avoid multiplication with constants inside loops if possible

Here is an example of a part of a program where specific operations lead to a slower code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ k = n-1; for (i = 0; i < n; i++) a[i] = b[i] +c*d; e = g[k];

A better code is

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ temp = c*d; for (i = 0; i < n; i++) a[i] = b[i] + temp; e = g[n-1];

Here we avoid a repeated multiplication inside a loop. Most compilers, depending on compiler flags, identify and optimize such bottlenecks on their own, without requiring any particular action by the programmer. However, it is always useful to single out and avoid code examples like the first one discussed here.

Vectorization and the basic idea behind parallel computing

Present CPUs are highly parallel processors with varying levels of parallelism. The typical situation can be described via the following three statements.

- Pursuit of shorter computation time and larger simulation size gives rise to parallel computing.
- Multiple processors are involved to solve a global problem.
- The essence is to divide the entire computation evenly among collaborative processors. Divide and conquer.

Before we proceed with a more detailed discussion of topics like vectorization and parallelization, we need to remind ourselves about some basic features of different hardware models.

A rough classification of hardware models

- Conventional single-processor computers are named SISD (single-instruction-single-data) machines.
- SIMD (single-instruction-multiple-data) machines incorporate the idea of parallel processing, using a large number of processing units to execute the same instruction on different data.
- Modern parallel computers are so-called MIMD (multiple-instruction-multiple-data) machines and can execute different instruction streams in parallel on different data.

Shared memory and distributed memory

One way of categorizing modern parallel computers is to look at the memory configuration.

- In shared memory systems the CPUs share the same address space. Any CPU can access any data in the global memory.
- In distributed memory systems each CPU has its own memory.

The CPUs are connected by some network and may exchange messages.

Different parallel programming paradigms

- **Task parallelism**: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations represent a typical situation. Integration is another. However this paradigm is of limited use.
- **Data parallelism**: use of multiple threads (e.g. one or more threads per processor) to dissect loops over arrays etc. Communication and synchronization between processors are often hidden, thus easy to program. However, the user surrenders much control to a specialized compiler. Examples of data parallelism are compiler-based parallelization and OpenMP directives.

Different parallel programming paradigms

- **Message passing**: all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.
- This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult.

What is vectorization?

Vectorization is a special case of **Single Instructions Multiple Data** (SIMD) to denote a single instruction stream capable of operating on multiple data elements in parallel. We can think of vectorization as the unrolling of loops accompanied with SIMD instructions.

Vectorization is the process of converting an algorithm that performs scalar operations (typically one operation at the time) to vector operations where a single operation can refer to many simultaneous operations. Consider the following example

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (i = 0; i < n; i++) a[i] = b[i] + c[i];

If the code is not vectorized, the compiler will simply start with the first element and then perform subsequent additions operating on one address in memory at the time.

Number of elements that can acted upon

A SIMD instruction can operate on multiple data elements in one single instruction. It uses the so-called 128-bit SIMD floating-point register. In this sense, vectorization adds some form of parallelism since one instruction is applied to many parts of say a vector.

The number of elements which can be operated on in parallel range from four single-precision floating point data elements in so-called Streaming SIMD Extensions and two double-precision floating-point data elements in Streaming SIMD Extensions 2 to sixteen byte operations in a 128-bit register in Streaming SIMD Extensions 2. Thus, vector-length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

IN summary, our instructions operate on 128 bit (16 byte) operands

- 4 floats or ints
- 2 doubles
- Data paths 128 bits vide for vector unit

Number of elements that can acted upon, examples

We start with the simple scalar operations given by

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (i = 0; i < n; i++) a[i] = b[i] + c[i];

If the code is not vectorized and we have a 128-bit register to store a 32 bits floating point number, it means that we have 3×32 bits that are not used. For the first element we have

0	1	2	3
a[0]=	not used	not used	not used
b[0]+	not used	not used	not used
c[0]	not used	not used	not used

We have thus unused space in our SIMD registers. These registers could hold three additional integers.

Operation counts for scalar operation

The code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (i = 0; i < n; i++) a[i] = b[i] + c[i];

has for n repeats

1. one load for $c[i]$ in address 1
2. one load for $b[i]$ in address 2
3. add $c[i]$ and $b[i]$ to give $a[i]$
4. store $a[i]$ in address 2

Number of elements that can acted upon, examples

If we vectorize the code, we can perform, with a 128-bit register four simultaneous operations, that is we have

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (i = 0; i < n; i+=4) a[i] = b[i] + c[i]; a[i+1] = b[i+1] + c[i+1]; a[i+2] = b[i+2] + c[i+2]; a[i+3] = b[i+3] + c[i+3];

displayed here as

0	1	2	3
a[0]=	a[1]=	a[2]=	a[3]=
b[0]+	b[1]+	b[2]+	b[3]+
c[0]	c[1]	c[2]	c[3]

Four additions are now done in a single step.

Number of operations when vectorized

For $n/4$ repeats assuming floats or integers

1. one vector load for $c[i]$ in address 1
2. one load for $b[i]$ in address 2
3. add $c[i]$ and $b[i]$ to give $a[i]$
4. store $a[i]$ in address 2

A simple test case with and without vectorization

We implement these operations in a simple c++ program that computes at the end the norm of a vector.

[numbers=none,fontsize=,baselinestretch=0.95] include <cstdlib> include <iostream> include <cmath> include <iomanip> include "time.h"

using namespace std; // note use of namespace int main (int argc, char* argv[]) // read in dimension of square matrix int n = atoi(argv[1]); double s = 1.0/sqrt((double) n); double *a, *b, *c; // Start timing clock$_t start, finish; start = clock(); //Allocatespaceforthevectorstobeuseda = newdouble[n]; b = newdouble[n]; c = newdouble[n]; //Defineparallelregion//Setupvaluesforvectorsaandbfor(inti = 0; i < n; i++) doubleangle = 2.0 * M_PI * i/((double)n); a[i] = s * (sin(angle) + cos(angle)); b[i] = s * sin(2.0 * angle); c[i] = 0.0; //Thenperform 0; i < n; i++) c[i] += a[i] + b[i]; //Computenowthenorm - 2doubleNorm2 = 0.0; for(inti = 0; i < n; i++) Norm2 += c[i] * c[i]; finish = clock(); doubletimeused = (double)(finish - start)/(CLOCKS_PER_SEC); cout << setiosflags(ios :: showpoint | ios :: uppercase); cout << setprecision(10) << setw(20) << "Timeusedfornormcomputation = " << timeused << endl; cout << "Norm - 2 = " << Norm2 << endl; //Freeupspacedelete[]a; delete[]b; delete[]c; return 0;$

Compiling with and without vectorization

We can compile and link without vectorization using the clang c++ compiler

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ clang -o novec.x vecexample.cpp

and with vectorization (and additional optimizations)

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ clang++ -O3 -Rpass=loop-vectorize -o vec.x vecexample.cpp

The speedup depends on the size of the vectors. In the example here we have run with 10^7 elements. The example here was run on an IMac17.1 with OSX El Capitan (10.11.4) as operating system and an Intel i5 3.3 GHz CPU.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ Compphys: hjensen$./vec.x 10000000 Timeusedfornormcomputation = 0.04720500000 Compphys : hjensen$ ./novec.x 10000000 Time used for norm computation=0.03311700000

This particular C++ compiler speeds up the above loop operations with a factor of 1.5 Performing the same operations for 10^9 elements results in a smaller speedup since reading from main memory is required. The non-vectorized code is seemingly faster.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ Compphys: hjensen$./vec.x 1000000000 Time used for norm computation=58.41391100 Compphys: hjensen$ ./novec.x 1000000000 Time used for norm computation=46.51295300

We will discuss these issues further in the next slides.

Compiling with and without vectorization using clang

We can compile and link without vectorization with clang compiler

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ clang++ -o -fno-vectorize novec.x vecexample.cpp

and with vectorization

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ clang++ -O3 -Rpass=loop-vectorize -o vec.x vecexample.cpp

We can also add vectorization analysis, see for example

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ clang++ -O3 -Rpass-analysis=loop-vectorize -o vec.x vecexample.cpp

or figure out if vectorization was missed

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ clang++ -O3 -Rpass-missed=loop-vectorize -o vec.x vecexample.cpp

Automatic vectorization and vectorization inhibitors, criteria

Not all loops can be vectorized, as discussed in Intel's guide to vectorization

An important criteria is that the loop counter n is known at the entry of the loop.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (int j = 0; j < n; j++) a[j] = cos(j*1.0);

The variable n does need to be known at compile time. However, this variable must stay the same for the entire duration of the loop. It implies that an exit statement inside the loop cannot be data dependent.

Automatic vectorization and vectorization inhibitors, exit criteria

An exit statement should in general be avoided. If the exit statement contains data-dependent conditions, the loop cannot be vectorized. The following is an example of a non-vectorizable loop

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (int j = 0; j < n; j++) a[j] = cos(j*1.0); if (a[j] < 0) break;

Avoid loop termination conditions and opt for a single entry loop variable n. The lower and upper bounds have to be kept fixed within the loop.

Automatic vectorization and vectorization inhibitors, straight-line code

SIMD instructions perform the same type of operations multiple times. A **switch** statement leads thus to a non-vectorizable loop since different statemens cannot branch. The following code can however be vectorized since the **if** statement is implemented as a masked assignment.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (int j = 0; j < n; j++) double x = cos(j*1.0); if (x > 0) a[j] = x*sin(j*2.0); else a[j] = 0.0;

These operations can be performed for all data elements but only those elements which the mask evaluates as true are stored. In general, one should avoid branches such as **switch**, **go to**, or **return** statements or **if** constructs that cannot be treated as masked assignments.

Automatic vectorization and vectorization inhibitors, nested loops

Only the innermost loop of the following example is vectorized

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) a[i][j] += b[i][j];

The exception is if an original outer loop is transformed into an inner loop as the result of compiler optimizations.

Automatic vectorization and vectorization inhibitors, function calls

Calls to programmer defined functions ruin vectorization. However, calls to intrinsic functions like $\sin x$, $\cos x$, $\exp x$ etc are allowed since they are normally efficiently vectorized. The following example is fully vectorizable

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (int i = 0; i < n; i++) a[i] = log10(i)*cos(i);

Similarly, **inline** functions defined by the programmer, allow for vectorization since the function statements are glued into the actual place where the function is called.

Automatic vectorization and vectorization inhibitors, data dependencies

One has to keep in mind that vectorization changes the order of operations inside a loop. A so-called read-after-write statement with an explicit flow dependency cannot be vectorized. The following code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ double b = 15.; for (int i = 1; i < n; i++) a[i] = a[i-1] + b;

is an example of flow dependency and results in wrong numerical results if vectorized. For a scalar operation, the value $a[i-1]$ computed during the iteration is loaded into the right-hand side and the results are fine. In vector mode however, with a vector length of four, the values $a[0]$, $a[1]$, $a[2]$ and $a[3]$ from the previous loop will be loaded into the right-hand side and produce wrong results. That is, we have [fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ a[1] = a[0] + b; a[2] = a[1] + b; a[3] = a[2] + b; a[4] = a[3] + b;

and if the two first iterations are executed at the same by the SIMD instruction, the value of say $a[1]$ could be used by the second iteration before it has been calculated by the first iteration, leading thereby to wrong results.

Automatic vectorization and vectorization inhibitors, more data dependencies

On the other hand, a so-called write-after-read statement can be vectorized. The following code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ double b = 15.; for (int i = 1; i < n; i++) a[i-1] = a[i] + b;

is an example of flow dependency that can be vectorized since no iteration with a higher value of i can complete before an iteration with a lower value of i. However, such code leads to problems with parallelization.

Automatic vectorization and vectorization inhibitors, memory stride

For C++ programmers it is also worth keeping in mind that an array notation is preferred to the more compact use of pointers to access array elements. The compiler can often not tell if it is safe to vectorize the code.

When dealing with arrays, you should also avoid memory stride, since this slows down considerably vectorization. When you access array element, write for example the inner loop to vectorize using unit stride, that is, access successively the next array element in memory, as shown here

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) a[i][j] += b[i][j];

Memory management

The main memory contains the program data

1. Cache memory contains a copy of the main memory data
2. Cache is faster but consumes more space and power. It is normally assumed to be much faster than main memory
3. Registers contain working data only
   - Modern CPUs perform most or all operations only on data in register
4. Multiple Cache memories contain a copy of the main memory data
   - Cache items accessed by their address in main memory
   - L1 cache is the fastest but has the least capacity
   - L2, L3 provide intermediate performance/size tradeoffs

Loads and stores to memory can be as important as floating point operations when we measure performance.

Memory and communication

1. Most communication in a computer is carried out in chunks, blocks of bytes of data that move together
2. In the memory hierarchy, data moves between memory and cache, and between different levels of cache, in groups called lines
    - Lines are typically 64-128 bytes, or 8-16 double precision words
    - Even if you do not use the data, it is moved and occupies space in the cache

Many of these performance features are not captured in most programming languages.

Measuring performance

How do we measure performance? What is wrong with this code to time a loop?
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]text clock$_t start, finish; start = clock(); for(int j = 0; j < i; j++) a[j] = b[j] + b[j] * c[j]; finish = clock(); doubletimeused = (double)(finish - start)/(CLOCKS_PER_SEC);$

Problems with measuring time

1. Timers are not infinitely accurate
2. All clocks have a granularity, the minimum time that they can measure
3. The error in a time measurement, even if everything is perfect, may be the size of this granularity (sometimes called a clock tick)
4. Always know what your clock granularity is
5. Ensure that your measurement is for a long enough duration (say 100 times the **tick**)

Problems with cold start

What happens when the code is executed? The assumption is that the code is ready to execute. But

1. Code may still be on disk, and not even read into memory.
2. Data may be in slow memory rather than fast (which may be wrong or right for what you are measuring)
3. Multiple tests often necessary to ensure that cold start effects are not present
4. Special effort often required to ensure data in the intended part of the memory hierarchy.

Problems with smart compilers

1. If the result of the computation is not used, the compiler may eliminate the code
2. Performance will look impossibly fantastic

3. Even worse, eliminate some of the code so the performance looks plausible
4. Ensure that the results are (or may be) used.

Problems with interference

1. Other activities are sharing your processor
   - Operating system, system demons, other users
   - Some parts of the hardware do not always perform with exactly the same performance
2. Make multiple tests and report
3. Easy choices include
   - Average tests represent what users might observe over time

Problems with measuring performance

1. Accurate, reproducible performance measurement is hard
2. Think carefully about your experiment:
3. What is it, precisely, that you want to measure?
4. How representative is your test to the situation that you are trying to measure?

Thomas algorithm for tridiagonal linear algebra equations

$$\begin{pmatrix} b_0 & c_0 & & & \\ a_0 & b_1 & c_1 & & \\ & & \ddots & & \\ & & a_{m-3} & b_{m-2} & c_{m-2} \\ & & & a_{m-2} & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{m-2} \\ f_{m-1} \end{pmatrix}$$

Thomas algorithm, forward substitution

The first step is to multiply the first row by a_0/b_0 and subtract it from the second row. This is known as the forward substitution step. We obtain then

$$a_i = 0,$$

$$b_i = b_i - \frac{a_{i-1}}{b_{i-1}} c_{i-1},$$

and

$$f_i = f_i - \frac{a_{i-1}}{b_{i-1}} f_{i-1}.$$

At this point the simplified equation, with only an upper triangular matrix takes the form

$$\begin{pmatrix} b_0 & c_0 & & & \\ & b_1 & c_1 & & \\ & & \ddots & & \\ & & & b_{m-2} & c_{m-2} \\ & & & & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{m-2} \\ f_{m-1} \end{pmatrix}$$

Thomas algorithm, backward substitution

The next step is the backward substitution step. The last row is multiplied by c_{N-3}/b_{N-2} and subtracted from the second to last row, thus eliminating c_{N-3} from the last row. The general backward substitution procedure is

$$c_i = 0,$$

and

$$f_{i-1} = f_{i-1} - \frac{c_{i-1}}{b_i} f_i$$

All that ramains to be computed is the solution, which is the very straight forward process of

$$x_i = \frac{f_i}{b_i}$$

Thomas algorithm and counting of operations (floating point and memory)

Operation	Floating Point
Memory Reads	$14(N-2)$
Memory Writes	$4(N-2)$
Subtractions	$3(N-2)$
Multiplications	$3(N-2)$
Divisions	$4(N-2)$

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ // Forward substitution // Note that we can simplify by precalculating a[i-1]/b[i-1] for (int i=1; i < n; i++) b[i] = b[i] - (a[i-1]*c[i-1])/b[i-1]; f[i] = g[i] - (a[i-1]*f[i-1])/b[i-1]; x[n-1] = f[n-1] / b[n-1]; // Backwards substitution for (int i = n-2; i >= 0; i–) f[i] = f[i] - c[i]*f[i+1]/b[i+1]; x[i] = f[i]/b[i];

Example: Transpose of a matrix

[numbers=none,fontsize=,baselinestretch=0.95] include <cstdlib> include <iostream> include <cmath> include <iomanip> include "time.h"

using namespace std; // note use of namespace int main (int argc, char* argv[]) // read in dimension of square matrix int n = atoi(argv[1]); double **A, **B; // Allocate space for the two matrices A = new double*[n]; B = new double*[n]; for (int i = 0; i < n; i++) A[i] = new double[n]; B[i] = new double[n]; // Set up values for matrix A for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) A[i][j] = cos(i*1.0)*sin(j*3.0); clock$_t start, finish; start = clock(); //Thencomputethetransposefor(int i = 0; i < n; i++) for(int j = 0; j < n; j++) B[i][j] = A[j][i];$

finish = clock(); double timeused = (double) (finish - start)/(CLOCKS$_PER_SEC); cout << setiosflags(ios :: showpoint|ios :: uppercase); cout << setprecision(10) << setw(20) << "Timeusedforsettinguptransposeofmatrix = " << timeused << endl;$

// Free up space for (int i = 0; i < n; i++) delete[] A[i]; delete[] B[i]; delete[] A; delete[] B; return 0;

Matrix-matrix multiplication

This the matrix-matrix multiplication code with plain c++ memory allocation. It computes at the end the Frobenius norm.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]text include <cstdlib> include <iostream> include <cmath> include <iomanip> include "time.h"

using namespace std; // note use of namespace int main (int argc, char* argv[]) // read in dimension of square matrix int n = atoi(argv[1]); double s = 1.0/sqrt((double) n); double **A, **B, **C; // Start timing clock$_t start, finish; start = clock(); //AllocatespaceforthetwomatricesA = newdouble * [n]; B = newdouble * [n]; C = newdouble * [n]; for(int i = 0; i < n; i++) A[i] = newdouble[n]; B[i] = newdouble[n]; C[i] = newdouble[n]$
$0; i < n; i++) for(int j = 0; j < n; j++) doubleangle = 2.0 * M_PI * i * j/((double)n); A[i][j] = s * (sin(angle) + cos(angle)); B[j][i] = A[i][j]; //T$
$matrixmultiplicationfor(int i = 0; i < n; i++) for(int j = 0; j < n; j++) doublesum = 0.0; for(int k = 0; k < n; k++) sum+ = B[i][k] * A[k][j]; C[i$
$0.0; for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) Fsum+ = C[i][j] * C[i][j]; Fsum = sqrt(Fsum); finish = clock(); doubletimeused = (double)(finish - start)/(CLOCKS_PER_SEC); cout << setiosflags(ios :: showpoint|ios :: uppercase); cout << setprecision(10) << setw(20) << "Timeusedformatrix - matrixmultiplication = " << timeused << endl; cout << "Frobeniusnorm = " << Fsum << endl; //Freeupspacefor(int i = 0; i < n; i++) delete[]A[i]; delete[]B[i]; delete[]C[i]; delete[]A; delete[]B; delete[]C; return0;$

How do we define speedup? Simplest form

- Speedup measures the ratio of performance between two objects
- Versions of same code, with different number of processors
- Serial and vector versions
- Try different programing languages, c++ and Fortran
- Two algorithms computing the **same** result

How do we define speedup? Correct baseline

The key is choosing the correct baseline for comparison

- For our serial vs. vectorization examples, using compiler-provided vectorization, the baseline is simple; the same code, with vectorization turned off
  - For parallel applications, this is much harder:
    - Choice of algorithm, decomposition, performance of baseline case etc.

Parallel speedup

For parallel applications, speedup is typically defined as

- Speedup $= T_1/T_p$

Here T_1 is the time on one processor and T_p is the time using p processors.

- Can the speedup become larger than p? That means using p processors is more than p times faster than using one processor.

Speedup and memory

The speedup on p processors can be greater than p if memory usage is optimal! Consider the case of a memorybound computation with M words of memory

- If M/p fits into cache while M does not, the time to access memory will be different in the two cases:
- T_1 uses the main memory bandwidth
- T_p uses the appropriate cache bandwidth

Upper bounds on speedup

Assume that almost all parts of a code are perfectly parallelizable (fraction f). The remainder, fraction $(1-f)$ cannot be parallelized at all.

That is, there is work that takes time W on one process; a fraction f of that work will take time Wf/p on p processors.

- What is the maximum possible speedup as a function of f?

Amdahl's law

On one processor we have

$$T_1 = (1-f)W + fW = W$$

On p processors we have

$$T_p = (1-f)W + \frac{fW}{p},$$

resulting in a speedup of

$$\frac{T_1}{T_p} = \frac{W}{(1-f)W + fW/p}$$

As p goes to infinity, fW/p goes to zero, and the maximum speedup is

$$\frac{1}{1-f},$$

meaning that if if $f = 0.99$ (all but 1% parallelizable), the maximum speedup is $1/(1-.99) = 100$!

Chapter 9
Parallelization with MPI and OpenMPI

How much is parallelizable

If any non-parallel code slips into the application, the parallel performance is limited.

In many simulations, however, the fraction of non-parallelizable work is 10^{-6} or less due to large arrays or objects that are perfectly parallelizable.

Today's situation of parallel computing

- Distributed memory is the dominant hardware configuration. There is a large diversity in these machines, from MPP (massively parallel processing) systems to clusters of off-the-shelf PCs, which are very cost-effective.
- Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware. It is primarily used for the distributed memory systems, but can also be used on shared memory systems.
- Modern nodes have nowadays several cores, which makes it interesting to use both shared memory (the given node) and distributed memory (several nodes with communication). This leads often to codes which use both MPI and OpenMP.

Our lectures will focus on both MPI and OpenMP.

Overhead present in parallel computing

- **Uneven load balance**: not all the processors can perform useful work at all time.
- **Overhead of synchronization**
- **Overhead of communication**
- **Extra computation due to parallelization**

Due to the above overhead and that certain parts of a sequential algorithm cannot be parallelized we may not achieve an optimal parallelization.

Parallelizing a sequential algorithm

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This is the difficult part,
- Distribute the global work and data among *P* processors.

Strategies

- Develop codes locally, run with some few processes and test your codes. Do benchmarking, timing and so forth on local nodes, for example your laptop or PC.
- When you are convinced that your codes run correctly, you can start your production runs on available supercomputers.

How do I run MPI on a PC/Laptop? MPI

To install MPI is rather easy on hardware running unix/linux as operating systems, follow simply the instructions from the OpenMPI website. See also subsequent slides. When you have made sure you have installed MPI on your PC/laptop,

- Compile with mpicxx/mpic++ or mpif90

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ Compile and link mpic++ -O3 -o nameofprog.x nameofprog.cpp run code with for example 8 processes using mpirun/mpiexec mpiexec -n 8 ./nameofprog.x

Can I do it on my own PC/laptop? OpenMP installation

If you wish to install MPI and OpenMP on your laptop/PC, we recommend the following:

- For OpenMP, the compile option **-fopenmp** is included automatically in recent versions of the C++ compiler and Fortran compilers. For users of different Linux distributions, simply use the available C++ or Fortran compilers and add the above compiler instructions, see also code examples below.
- For OS X users however, install **libomp**

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ brew install libomp

and compile and link as

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ c++ -o <name executable> <name program.cpp> -lomp

Installing MPI

For linux/ubuntu users, you need to install two packages (alternatively use the synaptic package manager)

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ sudo apt-get install libopenmpi-dev sudo apt-get install openmpi-bin

For OS X users, install brew (after having installed xcode and gcc, needed for the gfortran compiler of openmpi) and then install with brew

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ brew install openmpi

When running an executable (code.x), run as

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ mpirun -n 10 ./code.x

where we indicate that we want the number of processes to be 10.

Installing MPI and using Qt

With openmpi installed, when using Qt, add to your .pro file the instructions here

You may need to tell Qt where openmpi is stored.

What is Message Passing Interface (MPI)?

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C/C++ or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library.

MPI programs should be able to run on all possible machines and run all MPI implementetations without change.

An MPI computation is a collection of processes communicating with messages.

Going Parallel with MPI

Task parallelism: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations or numerical integration are examples of this.

MPI is a message-passing library where all the routines have corresponding C/C++-binding

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ MPI$_{C}$*ommand*$_{n}$*ame*

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

[numbers=none,fontsize=,baselinestretch=0.95] MPI$_{C}$*OMMAND*$_{N}$*AME*

MPI is a library

MPI is a library specification for the message passing interface, proposed as a standard.

- independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use. Designed for high performance.

Insert communication and synchronization functions where necessary.

Bindings to MPI routines

MPI is a message-passing library where all the routines have corresponding C/C++-binding

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ MPI$_{C}$*ommand*$_{n}$*ame*

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

[numbers=none,fontsize=,baselinestretch=0.95] MPI$_{C}$*OMMAND*$_{N}$*AME*

The discussion in these slides focuses on the C++ binding.

Communicator

- A group of MPI processes with a name (context).
- Any process is identified by its rank. The rank is only meaningful within a particular communicator.
- By default the communicator contains all the MPI processes.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ MPI$_{C}OMM_{W}ORLD$

- Mechanism to identify subset of processes.
- Promotes modular design of parallel libraries.

Some of the most important MPI functions

- *MPI_Init* - initiate an MPI computation
- *MPI_Finalize* - terminate the MPI computation and clean up
- *MPI_Comm_size* - how many processes participate in a given MPI communicator?
- *MPI_Comm_rank* - which one am I? (A number between 0 and size-1.)
- *MPI_Send* - send a message to a particular process within an MPI communicator
- *MPI_Recv* - receive a message from a particular process within an MPI communicator
- *MPI_reduce* or *MPI_Allreduce*, send and receive messages

The first MPI C/C++ program

Let every process write "Hello world" (oh not this program again!!) on the standard output.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ using namespace std; include <mpi.h> include <iostream> int main (int nargs, char* args[]) int numprocs, my$_{r}ank; //MPIinitializationsMPI_{I}nit(nargs, args); MPI_{C}omm_{s}ize(MPI_{C}OMM_{W}ORLD, numprocs); MPI_{C}omm_{r}ank(MPI_{C}OM$ $"Helloworld, Ihaverank" << my_{r}ank << "outof" << numprocs << endl; //EndMPIMPI_{F}inalize();$

The Fortran program

[numbers=none,fontsize=,baselinestretch=0.95] PROGRAM hello INCLUDE "mpif.h" INTEGER:: size, my$_{r}ank, ierr$

CALL MPI$_{I}NIT(ierr)CALLMPI_{C}OMM_{S}IZE(MPI_{C}OMM_{W}ORLD, size, ierr)CALLMPI_{C}OMM_{R}ANK(MPI_{C}OMM_{W}ORLD, my_{r}ank, ierr)WR$

END PROGRAM hello

Note 1

- The output to screen is not ordered since all processes are trying to write to screen simultaneously.

- It is the operating system which opts for an ordering.
- If we wish to have an organized output, starting from the first process, we may rewrite our program as in the next example.

Ordered output with MPIBarrier

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
int main (int nargs, char* args[]) int numprocs, my$_{r}ank, i; MPI_{I}nit(nargs, args); MPI_{C}omm_{s}ize(MPI_{C}OMM_{W}ORLD, numprocs); MPI$
$0; i < numprocs; i++)MPI_{B}arrier(MPI_{C}OMM_{W}ORLD); if(i == my_{r}ank)cout << ''Helloworld, Ihaverank'' << my_{r}ank << ''outof'' << nump$

Note 2

- Here we have used the *MPI_Barrier* function to ensure that that every process has completed its set of instructions in a particular order.
- A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here *MPI_COMM_WORLD*) have called *MPI_Barrier*.
- The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like *MPI_ALLREDUCE* to be discussed later, have the same property; that is, no process can exit the operation until all processes have started.

However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

Ordered output

[numbers=none,fontsize=,baselinestretch=0.95] ..... int numprocs, my$_{r}ank, flag; MPI_{S}tatus status; MPI_{I}nit(nargs, args); MPI_{C}$
$0)MPI_{R}ecv(flag, 1, MPI_{I}NT, my_{r}ank-1, 100, MPI_{C}OMM_{W}ORLD, status); cout << ''Helloworld, Ihaverank'' << my_{r}ank <<$
$''outof'' << numprocs << endl; if(my_{r}ank < numprocs-1)MPI_{S}end(my_{r}ank, 1, MPI_{I}NT, my_{r}ank+1, 100, MPI_{C}OMM_{W}ORLD); MPI_{F}inalize()$

Note 3

The basic sending of messages is given by the function *MPI_SEND*, which in C/C++ is defined as

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
int MPI$_{S}end(void * buf, int count, MPI_{D}atatype datatype, int dest, int tag, MPI_{C}omm comm)$

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable **buf** is the variable we wish to send while **count** is the number of variables we are passing. If we are passing only a single value, this should be 1.

If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

Note 4

Once you have sent a message, you must receive it on another task. The function *MPI_RECV* is similar to the send call.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ int MPI$_Recv(void*buf,intcount,MPI_Datatypedatatype,intsource,inttag,MPI_Commcomm,MPI_Status*status)$

The arguments that are different from those in MPI_SEND are **buf** which is the name of the variable where you will be storing the received data, **source** which replaces the destination in the send command. This is the return ID of the sender.

Finally, we have used *MPI_Status_status*, where one can check if the receive was completed.

The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.

Numerical integration in parallel

Integrating π.

- The code example computes π using the trapezoidal rules.
- The trapezoidal rule

$$I = \int_a^b f(x)dx \approx h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f(b)/2\right).$$

Click on this link for the full program.

Dissection of trapezoidal rule with MPI_reduce

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ // Trapezoidal rule and numerical integration usign MPI using namespace std; include <mpi.h> include <iostream>

// Here we define various functions called by the main program

double int$_function(double);doubletrapezoidal_rule(double,double,int,double(*)(double));$

// Main function begins here int main (int nargs, char* args[]) int n, local$_n, numprocs, my_rank; doublea, b, h, local_a, local_b, to$

Dissection of trapezoidal rule

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
// MPI initializations MPI$_{Init}(nargs,args);MPI_{C}omm_{s}ize(MPI_{C}OMM_{W}ORLD,numprocs);MPI_{C}omm_{r}ank(MPI_{C}OMM_{W}ORLD,my_{r}ank);ti$
$MPI_{W}time();//Fixedvaluesfora,bandna=0.0;b=1.0;n=1000;h=(b-a)/n;//histhesameforallprocesseslocal_{n}=$
$n/numprocs;//makesuren>numprocs,elseintegerdivisiongiveszero//Lengthofeachprocess'intervalof//integration=$
$local_{n}*h.local_{a}=a+my_{r}ank*local_{n}*h;local_{b}=local_{a}+local_{n}*h;$

Integrating with MPI

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
total$_{sum}=0.0;local_{s}um=trapezoidal_{r}ule(local_{a},local_{b},local_{n},int_{f}unction);MPI_{R}educe(local_{s}um,total_{s}um,1,MPI_{D}OUBLE,MPI_{S}UM,0,M$
$MPI_{W}time();total_{t}ime=time_{e}nd-time_{s}tart;if(my_{r}ank==0)cout<<''Trapezoidalrule=''<<total_{s}um<<endl;cout<<''Time=''<<t$

How do I use MPI_reduce*?*

Here we have used
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
MPI$_{r}educe(void*senddata,void*resultdata,intcount,MPI_{D}atatypedatatype,MPI_{O}p,introot,MPI_{C}ommcomm)$
The two variables $senddata$ and $resultdata$ are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable $count$ represents the total dimensionality, 1 in case of just one variable, while $MPI\_Datatype$ defines the type of variable which is sent and received.
The new feature is MPI_Op. It defines the type of operation we want to do.

More on MPI_Reduce

In our case, since we are summing the rectangle contributions from every process we define $MPI\_Op = MPI\_SUM$. If we have an array or matrix we can search for the largest og smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINOC. If we want the product we write MPI_PROD.
$MPI\_Allreduce$ is defined as
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
MPI$_{A}llreduce(void*senddata,void*resultdata,intcount,MPI_{D}atatypedatatype,MPI_{O}p,MPI_{C}ommcomm)$

Dissection of trapezoidal rule

We use *MPI_reduce* to collect data from each process. Note also the use of the function *MPI_Wtime*.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
// this function defines the function to integrate double int$_{function(doublex)doublevalue=4./(1.+x*x);returnvalue;//endoffu}$

Dissection of trapezoidal rule

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
// this function defines the trapezoidal rule double trapezoidal$_{rule(doublea,doubleb,intn,double(*func)(double))doubletrapez_{s}}$

The quantum dot program for two electrons

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
// Variational Monte Carlo for atoms with importance sampling, slater det // Test case for 2-electron quantum dot, no classes using Mersenne-Twister RNG include "mpi.h" include <cmath> include <random> include <string> include <iostream> include <fstream> include <iomanip> include "vectormatrixclass.h"

using namespace std; // output file as global variable ofstream ofile; // the step length and its squared inverse for the second derivative // Here we define global variables used in various functions // These can be changed by using classes int Dimension = 2; int NumberParticles = 2; // we fix also the number of electrons to be 2

// declaration of functions

// The Mc sampling for the variational Monte Carlo void MonteCarloSampling(int, double , double , Vector);

// The variational wave function double WaveFunction(Matrix , Vector);

// The local energy double LocalEnergy(Matrix , Vector);

// The quantum force void QuantumForce(Matrix , Matrix , Vector);

// inline function for single-particle wave function inline double SPwavefunction(double r, double alpha) return exp(-alpha*r*0.5);

// inline function for derivative of single-particle wave function inline double DerivativeSPwavefunction(double r, double alpha) return -r*alpha;

// function for absolute value of relative distance double RelativeDistance(Matrix r, int i, int j) double r$_{i}j=0;for(intk=0;k<Dimension;k++)r_{i}j+=(r(i,k)-r(j,k))*(r(i,k)-r(j,k));returnsqrt(r_{i}j);$

// inline function for derivative of Jastrow factor inline double JastrowDerivative(Matrix r, double beta, int i, int j, int k) return (r(i,k)-r(j,k))/(RelativeDistance(r, i, j)*pow(1.0+beta*RelativeDistance(r, i, j),2));

// function for square of position of single particle double singleparticle$_{pos}2(Matrixr,inti)doubler_{s}ingle_{p}article=0;for(int$

void lnsrch(int n, Vector xold, double fold, Vector g, Vector p, Vector x, double *f, double stpmax, int *check, double (*func)(Vector p));

void dfpmin(Vector p, int n, double gtol, int *iter, double *fret, double(*func)(Vector p), void (*dfunc)(Vector p, Vector g));

static double sqrarg; define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)

static double maxarg1,maxarg2; define FMAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ? (maxarg1) : (maxarg2))

// Begin of main program

int main(int argc, char* argv[])

// MPI initializations int NumberProcesses, MyRank, NumberMCsamples; MPI$_{Init}(argc,argv);MPI_{C}omm_{s}ize(MPI_{C}OMM$
$MPI_{W}time();if(MyRank==0argc<=1)cout<<"BadUsage:"<<argv[0]<<"Readalsooutputfileonsamelineandnumberof MonteCarlocy$
$0argc>2)stringfilename=argv[1];//firstcommandlineargumentafternameofprogramNumberMCsamples=atoi(argv[2]);stringfileout=f$
$NumberMCsamples*NumberProcesses;//Loopovervariationalparametersfor(doublealpha=0.5;alpha<=1.5;alpha+=$
$0.1)for(doublebeta=0.1;beta<=0.5;beta+=0.05)VariationalParameters(0)=alpha;//valueofalphaVariationalParameters(1)=beta;/$
$MPI_{W}time();doubleTotalTime=EndTime-StartTime;if(MyRank==0)cout<<"Time="<<TotalTime<<$
$"onnumberofprocessors:"<<NumberProcesses<<endl;if(MyRank==0)ofile.close();//closeoutputfile//EndMPIMPI_{F}inalize();return$

// Monte Carlo sampling with the Metropolis algorithm

void MonteCarloSampling(int NumberMCsamples, double cumulative$_{e},doublecumulative_{e}2,VectorVariationalParameters)$

// Initialize the seed and call the Mersienne algo std::random$_{d}evicerd;std::mt19937_{6}4gen(rd());//Setuptheuniformdistributi$
$[[0,1]std::uniform_{r}eal_{d}istribution<double>UniformNumberGenerator(0.0,1.0);std::normal_{d}istribution<$
$double>Normaldistribution(0.0,1.0);//diffusionconstantfromSchroedingerequationdoubleD=0.5;doubletimestep=$
$0.05;//wefixthetimestepforthegaussiandeviate//allocatematriceswhichcontainthepositionoftheparticlesMatrixOldPosition(NumberParticle$
$0.0;doubleEnergySquared=0.0;doubleDeltaE=0.0;//initialtrialpositionsfor(inti=0;i<NumberParticles;i+$
$+)for(intj=0;j<Dimension;j++)OldPosition(i,j)=Normaldistribution(gen)*sqrt(timestep);doubleOldWaveFunction=$
$WaveFunction(OldPosition,VariationalParameters);QuantumForce(OldPosition,OldQuantumForce,VariationalParameters);//loopovermon$
$1;cycles<=NumberMCsamples;cycles++)//newpositionfor(inti=0;i<NumberParticles;i++)for(intj=0;j<Dimension;j++)//gau$
$Energy/NumberMCsamples;cumulative_{e}2=EnergySquared/NumberMCsamples;//endMonteCarloSamplingfunction$

// Function to compute the squared wave function and the quantum force

double WaveFunction(Matrix r, Vector VariationalParameters) double wf = 0.0; // full Slater determinant for two particles, replace with Slater det for more particles wf = SPwavefunction(singleparticle$_{pos}2(r,0),$
$SPwavefunction(singleparticle_{p}os2(r,1),VariationalParameters(0));//contributionfromJastrowfactorfor(inti=0;i<$
$NumberParticles-1;i++)for(intj=i+1;j<NumberParticles;j++)wf*=exp(RelativeDistance(r,i,j)/((1.0+VariationalParameters(1)$

// Function to calculate the local energy without numerical derivation of kinetic energy

double LocalEnergy(Matrix r, Vector VariationalParameters)

// compute the kinetic and potential energy from the single-particle part // for a many-electron system this has to be replaced by a Slater determinant // The absolute value of the interparticle length Matrix length(NumberParticles, NumberParticles); // Set up interparticle distance for (int i = 0; i < NumberParticles-1; i++) for(int j = i+1; j < NumberParticles; j++) length(i,j) = RelativeDistance(r, i, j); length(j,i) = length(i,j); double KineticEnergy = 0.0; // Set up kinetic energy from Slater and Jastrow terms for (int i = 0; i < NumberParticles; i++) for (int k = 0; k < Dimension; k++) double sum1 = 0.0; for(int j = 0; j < NumberParticles; j++) if (j != i) sum1 += JastrowDerivative(r, VariationalParameters(1), i, j, k); KineticEnergy += (sum1+DerivativeSPwavefunction(r(i,k),VariationalParameters(0)))*(sum1+DerivativeSPwavefunction(r(i,k),Va
KineticEnergy += -2*VariationalParameters(0)*NumberParticles; for (int i = 0; i < NumberParticles-1; i++) for (int j = i+1; j < NumberParticles; j++) KineticEnergy += 2.0/(pow(1.0 + VariationalParameters(1)*length(i,j),2))*(1.0/length(i,j)-2*VariationalParameters(1)/(1+VariationalParameters(1)*le
); KineticEnergy *= -0.5; // Set up potential energy, external potential + eventual electron-electron repulsion double PotentialEnergy = 0; for (int i = 0; i < NumberParticles; i++) double DistanceSquared = singleparticle$_{pos}2(r,i);PotentialEnergy+=0.5*DistanceSquared;//spenergyHOpart,noteithastheoscillator$
$electronrepulsionfor(inti=0;i<NumberParticles-1;i++)for(intj=i+1;j<NumberParticles;j++)PotentialEnergy+=1.0/length(i,j$
$KineticEnergy+PotentialEnergy;returnLocalE;$

// Compute the analytical expression for the quantum force void QuantumForce(Matrix r, Matrix qforce, Vector VariationalParameters) // compute the first derivative for (int i

= 0; i < NumberParticles; i++) for (int k = 0; k < Dimension; k++) // single-particle part, replace with Slater det for larger systems double sppart = DerivativeSPwavefunction(r(i,k),VariationalParameters(0)); // Jastrow factor contribution double Jsum = 0.0; for (int j = 0; j < NumberParticles; j++) if (j != i) Jsum += JastrowDerivative(r, VariationalParameters(1), i, j, k); qforce(i,k) = 2.0*(Jsum+sppart);   // end of QuantumForce function

define ITMAX 200 define EPS 3.0e-8 define TOLX (4*EPS) define STPMX 100.0

void dfpmin(Vector p, int n, double gtol, int *iter, double *fret, double(*func)(Vector p), void (*dfunc)(Vector p, Vector g))

int check,i,its,j; double den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test; Vector dg(n), g(n), hdg(n), pnew(n), xi(n); Matrix hessian(n,n);

fp=(*func)(p); (*dfunc)(p,g); for (i = 0;i < n;i++) for (j = 0; j< n;j++) hessian(i,j)=0.0; hessian(i,i)=1.0; xi(i) = -g(i); sum += p(i)*p(i); stpmax=STPMX*FMAX(sqrt(sum),(double)n); for (its=1;its<=ITMAX;its++) *iter=its; lnsrch(n,p,fp,g,xi,pnew,fret,stpmax,check,func); fp = *fret; for (i = 0; i< n;i++) xi(i)=pnew(i)-p(i); p(i)=pnew(i); test=0.0; for (i = 0;i< n;i++) temp=fabs(xi(i))/FMAX(fabs(p(i)),1.0); if (temp > test) test=temp; if (test < TOLX) return; for (i=0;i<n;i++) dg(i)=g(i); (*dfunc)(p,g); test=0.0; den=FMAX(*fret,1.0); for (i=0;i<n;i++) temp=fabs(g(i))*FMAX(fabs(p(i)),1.0)/den; if (temp > test) test=temp; if (test < gtol) return; for (i=0;i<n;i++) dg(i)=g(i)-dg(i); for (i=0;i<n;i++) hdg(i)=0.0; for (j=0;j<n;j++) hdg(i) += hessian(i,j)*dg(j); fac=fae=sumdg=sumxi=0.0; for (i=0;i<n;i++) fac += dg(i)*xi(i); fae += dg(i)*hdg(i); sumdg += SQR(dg(i)); sumxi += SQR(xi(i)); if (fac*fac > EPS*sumdg*sumxi) fac=1.0/fac; fad=1.0/fae; for (i=0;i<n;i++) dg(i)=fac*xi(i)-fad*hdg(i); for (i=0;i<n;i++) for (j=0;j<n;j++) hessian(i,j) += fac*xi(i)*xi(j) -fad*hdg(i)*hdg(j)+fae*dg(i)*dg(j); for (i=0;i<n;i++) xi(i)=0.0; for (j=0;j<n;j++) xi(i) -= hessian(i,j)*g(j); cout « "too many iterations in dfpmin" « endl; undef ITMAX undef EPS undef TOLX undef STPMX

define ALF 1.0e-4 define TOLX 1.0e-7

void lnsrch(int n, Vector xold, double fold, Vector g, Vector p, Vector x, double *f, double stpmax, int *check, double (*func)(Vector p)) int i; double a,alam,alam2,alamin,b,disc,f2,fold2,rhs1,rhs2,slope,sum,tem
test,tmplam;

*check=0; for (sum=0.0,i=0;i<n;i++) sum += p(i)*p(i); sum=sqrt(sum); if (sum > stpmax) for (i=0;i<n;i++) p(i) *= stpmax/sum; for (slope=0.0,i=0;i<n;i++) slope += g(i)*p(i); test=0.0; for (i=0;i<n;i++) temp=fabs(p(i))/FMAX(fabs(xold(i)),1.0); if (temp > test) test=temp; alamin=TOLX/test; alam=1.0; for (;;) for (i=0;i<n;i++) x(i)=xold(i)+alam*p(i); *f=(*func)(x); if (alam < alamin) for (i=0;i<n;i++) x(i)=xold(i); *check=1; return; else if (*f <= fold+ALF*alam*slope) return; else if (alam == 1.0) tmplam = -slope/(2.0*(*f-fold-slope)); else rhs1 = *f-fold-alam*slope; rhs2=f2-fold2-alam2*slope; a=(rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2); b=(-alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2); if (a == 0.0) tmplam = -slope/(2.0*b); else disc=b*b-3.0*a*slope; if (disc<0.0) cout « "Roundoff problem in lnsrch." « endl; else tmplam=(-b+sqrt(disc))/(3.0*a); if (tmplam>0.5*alam) tmplam=0.5*alam; alam2=alam; f2 = *f; fold2=fold; alam=FMAX(tmplam,0.1*alam);   undef ALF undef TOLX

What is OpenMP

- OpenMP provides high-level thread programming
- Multiple cooperating threads are allowed to run simultaneously
- Threads are created and destroyed dynamically in a fork-join pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread

  – In the beginning of a parallel region, a team of new threads is spawned

- The newly spawned threads work simultaneously with the master thread
- At the end of a parallel region, the new threads are destroyed

Many good tutorials online and excellent textbook

1. Using OpenMP, by B. Chapman, G. Jost, and A. van der Pas
2. Many tutorials online like OpenMP official site

Getting started, things to remember

- Remember the header file

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <omp.h>

- Insert compiler directives in C++ syntax as

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp...

- Compile with for example *c++ -fopenmp code.cpp*
- Execute
  - Remember to assign the environment variable **OMP NUM THREADS**
  - It specifies the total number of threads inside a parallel region, if not otherwise overwritten

OpenMP syntax

- Mostly directives

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp construct [clause ...]

- Some functions and types

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <omp.h>

- Most apply to a block of code
- Specifically, a **structured block**
- Enter at top, exit at bottom only, exit(), abort() permitted

Different OpenMP styles of parallelism

OpenMP supports several different ways to specify thread parallelism

- General parallel regions: All threads execute the code, roughly as if you made a routine of that region and created a thread to run that code
- Parallel loops: Special case for loops, simplifies data parallel code
- Task parallelism, new in OpenMP 3
- Several ways to manage thread coordination, including Master regions and Locks
- Memory model for shared data

General code structure

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <omp.h> main () int var1, var2, var3; /* serial code */ /* ... */ /* start of a parallel region */ pragma omp parallel private(var1, var2) shared(var3) /* ... */ /* more serial code */ /* ... */ /* another parallel region */ pragma omp parallel /* ... */

Parallel region

- A parallel region is a block of code that is executed by a team of threads
- The following compiler directive creates a parallel region

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel ...

- Clauses can be added at the end of the directive
- Most often used clauses:
  - **default(shared)** or **default(none)**
  - **public(list of variables)**
  - **private(list of variables)**

Hello world, not again, please!

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <omp.h> include <cstdio> int main (int argc, char *argv[]) int th$_{i}d, nthreads; pragma omp parallel private(th_{i}d) share$

Hello world, yet another variant

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
include <cstdio> include <omp.h> int main(int argc, char *argv[]) omp$_set_num_threads(4); pragmaompparallelintid = omp_get_t$

Variables declared outside of the parallel region are shared by all threads If a variable like **id** is declared outside of the

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
pragma omp parallel,

it would have been shared by various the threads, possibly causing erroneous output

- Why? What would go wrong? Why do we add possibly?

Important OpenMP library routines

- **int omp get num threads ()**, returns the number of threads inside a parallel region
- **int omp get thread num ()**, returns the a thread for each thread inside a parallel region
- **void omp set num threads (int)**, sets the number of threads to be used
- **void omp set nested (int)**, turns nested parallelism on/off

Private variables

Private clause can be used to make thread- private versions of such variables:

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
pragma omp parallel private(id) int id = omp$_get_thread_num(); cout << ''Mythreadnum'' << id << endl;$

- What is their value on entry? Exit?
- OpenMP provides ways to control that
- Can use default(none) to require the sharing of each variable to be described

Master region

It is often useful to have only one thread execute some of the code in a parallel region. I/O statements are a common example

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
pragma omp parallel pragma omp master int id = omp$_get_thread_num(); cout << ''Mythreadnum'' << id << endl;$

Parallel for loop

- Inside a parallel region, the following compiler directive can be used to parallelize a for-loop:

  [fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp for

- Clauses can be added, such as
  - **schedule(static, chunk size)**
  - **schedule(dynamic, chunk size)**
  - **schedule(guided, chunk size)** (non-deterministic allocation)
  - **schedule(runtime)**
  - **private(list of variables)**
  - **reduction(operator:variable)**
  - **nowait**

Parallel computations and loops

OpenMP provides an easy way to parallelize a loop
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel for for (i=0; i<n; i++) c[i] = a[i];
OpenMP handles index variable (no need to declare in for loop or make private)
Which thread does which values? Several options.

Scheduling of loop computations

We can let the OpenMP runtime decide. The decision is about how the loop iterates are scheduled and OpenMP defines three choices of loop scheduling:

1. Static: Predefined at compile time. Lowest overhead, predictable
2. Dynamic: Selection made at runtime
3. Guided: Special case of dynamic; attempts to reduce overhead

Example code for loop scheduling

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <omp.h> define CHUNKSIZE 100 define N 1000 int main (int argc, char *argv[])

int i, chunk; float a[N], b[N], c[N]; for (i=0; i < N; i++) a[i] = b[i] = i * 1.0; chunk = CHUNKSIZE; pragma omp parallel shared(a,b,c,chunk) private(i) pragma omp for schedule(dynamic,chunk) for (i=0; i < N; i++) c[i] = a[i] + b[i]; /* end of parallel region */

Example code for loop scheduling, guided instead of dynamic

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ include <omp.h> define CHUNKSIZE 100 define N 1000 int main (int argc, char *argv[]) int i, chunk; float a[N], b[N], c[N]; for (i=0; i < N; i++) a[i] = b[i] = i * 1.0; chunk = CHUNKSIZE; pragma omp parallel shared(a,b,c,chunk) private(i) pragma omp for schedule(guided,chunk) for (i=0; i < N; i++) c[i] = a[i] + b[i]; /* end of parallel region */

More on Parallel for loop

- The number of loop iterations cannot be non-deterministic; break, return, exit, goto not allowed inside the for-loop
- The loop index is private to each thread
- A reduction variable is special
  - During the for-loop there is a local private copy in each thread
  - At the end of the for-loop, all the local copies are combined together by the reduction operation
- Unless the nowait clause is used, an implicit barrier synchronization will be added at the end by the compiler

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ // pragma omp parallel and pragma omp for
can be combined into
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel for

What can happen with this loop?

What happens with code like this
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel for for (i=0; i<n; i++) sum += a[i]*a[i];
All threads can access the **sum** variable, but the addition is not atomic! It is important to avoid race between threads. So-called reductions in OpenMP are thus important for performance and for obtaining correct results. OpenMP lets us indicate that a variable is used for a reduction with a particular operator. The above code becomes

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ sum = 0.0; pragma omp parallel for reduction(+:sum) for (i=0; i<n; i++) sum += a[i]*a[i];

Inner product

$$\sum_{i=0}^{n-1} a_i b_i$$

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ int i; double sum = 0.; /* allocating and initializing arrays */ /* ... */ pragma omp parallel for default(shared) private(i) reduction(+:sum) for (i=0; i<N; i++) sum += a[i]*b[i];

Different threads do different tasks

Different threads do different tasks independently, each section is executed by one thread.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel pragma omp sections pragma omp section funcA (); pragma omp section funcB (); pragma omp section funcC ();

Single execution

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp single ...

The code is executed by one thread only, no guarantee which thread

Can introduce an implicit barrier at the end

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp master ...

Code executed by the master thread, guaranteed and no implicit barrier at the end.

Coordination and synchronization

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp barrier

Synchronization, must be encountered by all threads in a team (or none)

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp ordered a block of codes

is another form of synchronization (in sequential order). The form [fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp critical a block of codes

and

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp atomic single assignment statement

is more efficient than

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp critical

Data scope

- OpenMP data scope attribute clauses:
  - **shared**
  - **private**
  - **firstprivate**
  - **lastprivate**
  - **reduction**

What are the purposes of these attributes

- define how and which variables are transferred to a parallel region (and back)
- define which variables are visible to all threads in a parallel region, and which variables are privately allocated to each thread

Some remarks

- When entering a parallel region, the **private** clause ensures each thread having its own new variable instances. The new variables are assumed to be uninitialized.
- A shared variable exists in only one memory location and all threads can read and write to that address. It is the programmer's responsibility to ensure that multiple threads properly access a shared variable.
- The **firstprivate** clause combines the behavior of the private clause with automatic initialization.
- The **lastprivate** clause combines the behavior of the private clause with a copy back (from the last loop iteration or section) to the original variable outside the parallel region.

Parallelizing nested for-loops

- Serial code

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (i=0; i<100; i++) for (j=0; j<100; j++) a[i][j] = b[i][j] + c[i][j];

- Parallelization

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel for private(j) for (i=0; i<100; i++) for (j=0; j<100; j++) a[i][j] = b[i][j] + c[i][j];

- Why not parallelize the inner loop? to save overhead of repeated thread forks-joins
- Why must **j** be private? To avoid race condition among the threads

Nested parallelism

When a thread in a parallel region encounters another parallel construct, it may create a new team of threads and become the master of the new team.
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel num$_t hreads(4)/*....*/pragmaompparallelnum_t hreads(2)//$

Parallel tasks

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp task pragma omp parallel shared(p$_v ec)private(i)pragmaompsinglefor(i=0;i<N;i++)doubler = random_n umber();$

Common mistakes

Race condition
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ int nthreads; pragma omp parallel shared(nthreads) nthreads = omp$_{get_n um_t hreads}();$
Deadlock
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel ... pragma omp critical ... pragma omp barrier

Not all computations are simple

Not all computations are simple loops where the data can be evenly divided among threads without any dependencies between threads

An example is finding the location and value of the largest element in an array
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ for (i=0; i<n; i++) if (x[i] > maxval) maxval = x[i]; maxloc = i;

Not all computations are simple, competing threads

All threads are potentially accessing and changing the same values, **maxloc** and **maxval**.

1. OpenMP provides several ways to coordinate access to shared values

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp atomic

1. Only one thread at a time can execute the following statement (not block). We can use the critical option

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp critical

1. Only one thread at a time can execute the following block

Atomic may be faster than critical but depends on hardware

How to find the max value using OpenMP

Write down the simplest algorithm and look carefully for race conditions. How would you handle them? The first step would be to parallelize as
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel for for (i=0; i<n; i++) if (x[i] > maxval) maxval = x[i]; maxloc = i;

Then deal with the race conditions

Write down the simplest algorithm and look carefully for race conditions. How would you handle them? The first step would be to parallelize as
[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++ pragma omp parallel for for (i=0; i<n; i++) pragma omp critical if (x[i] > maxval) maxval = x[i]; maxloc = i;

Exercise: write a code which implements this and give an estimate on performance. Perform several runs, with a serial code only with and without vectorization and compare the serial code with the one that uses OpenMP. Run on different archictectures if you can.

What can slow down OpenMP performance?

Give it a thought!

What can slow down OpenMP performance?

Performance poor because we insisted on keeping track of the maxval and location during the execution of the loop.

- We do not care about the value during the execution of the loop, just the value at the end.

This is a common source of performance issues, namely the description of the method used to compute a value imposes additional, unnecessary requirements or properties

Idea: Have each thread find the maxloc in its own data, then combine and use temporary arrays indexed by thread number to hold the values found by each thread

Find the max location for each thread

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
int maxloc[MAX$_THREADS], mloc; doublemaxval[MAX_THREADS], mval; pragmaompparallelshared(maxval, maxloc)intid = omp_get_thread$

Combine the values from each thread

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]c++
pragma omp flush (maxloc,maxval) pragma omp master int nt = omp$_get_num_threads(); mloc = maxloc[0]; mval = maxval[0]; for(inti = 1; i < nt; i++)if(maxval[i] > mval)mval = maxval[i]; mloc = maxloc[i];$

Note that we let the master process perform the last operation.

Matrix-matrix multiplication

This code computes the norm of a vector using OpenMp

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]text
// OpenMP program to compute vector norm by adding two other vectors include <cstdlib> include <iostream> include <cmath> include <iomanip> include <omp.h> include <ctime>

using namespace std; // note use of namespace int main (int argc, char* argv[]) // read in dimension of vector int n = atoi(argv[1]); double *a, *b, *c; int i; int thread$_num; doublewtime, Norm2, s, angle; cout << ”Performadditionoftwovectorsandcomputethenorm-2.” << endl; omp_set_num_threads(4); thread_num = omp_get_max_threads(); cout << ”Thenumberofprocessorsavailable = ” << omp_get_num_procs() << endl; cout << ”Thenumberofthreadsavailable = ” << thread_num << endl; cout << ”Thematrixordern = ” << n << endl;$

s = 1.0/sqrt((double) n); wtime = omp$_get_wtime();//Allocatespaceforthevectorstobeuseda = newdouble[n];b =$
$newdouble[n];c = newdouble[n];//Defineparallelregionpragmaompparallelfordefault(shared)private(angle,i)reduction(+:$
$Norm2)//Setupvaluesforvectorsaandbfor(i = 0;i < n;i++)angle = 2.0*M_PI*i/((double)n);a[i] = s*(sin(angle)+cos(angle));b[i] = s*si$
$0;i < n;i++)c[i]+ = a[i]+b[i];//Computenowthenorm-2Norm2 = 0.0;for(i = 0;i < n;i++)Norm2+ = c[i]*c[i];//endparallelregionwtime =$
$omp_get_wtime()-wtime;cout << setiosflags(ios::showpoint|ios::uppercase);cout << setprecision(10) << setw(20) <<$
$"Timeusedfornorm-2computation = " << wtime << endl;cout << "Norm-2 = " << Norm2 << endl;//Freeupspacedelete[]a;delete[]b;de$

Matrix-matrix multiplication

This the matrix-matrix multiplication code with plain c++ memory allocation using OpenMP

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]text // Matrix-matrix multiplication and Frobenius norm of a matrix with OpenMP include <cst-dlib> include <iostream> include <cmath> include <iomanip> include <omp.h> include <ctime>

using namespace std; // note use of namespace int main (int argc, char* argv[]) // read in dimension of square matrix int n = atoi(argv[1]); double **A, **B, **C; int i, j, k; int thread$_num;doublewtime,Fsum,s,angle;cout << "ComputematrixproductC = A*BandFrobeniusnorm." <<$
$endl;omp_set_num_threads(4);thread_num = omp_get_max_threads();cout << "Thenumberofprocessorsavailable = " <<$
$omp_get_num_procs() << endl;cout << "Thenumberofthreadsavailable = " << thread_num << endl;cout << "Thematrixordern =$
$" << n << endl;$

s = 1.0/sqrt((double) n); wtime = omp$_get_wtime();//AllocatespaceforthetwomatricesA = newdouble*$
$[n];B = newdouble*[n];C = newdouble*[n];for(i = 0;i < n;i++)A[i] = newdouble[n];B[i] = newdouble[n];C[i] = newdouble[n];//Definepara$
$Fsum)//SetupvaluesformatrixAandBandzeromatrixCfor(i = 0;i < n;i++)for(j = 0;j < n;j++)angle = 2.0*M_PI*i*j/((double)n);A[i][j$
$matrixmultiplicationfor(i = 0;i < n;i++)for(j = 0;j < n;j++)C[i][j] = 0.0;for(k = 0;k < n;k++)C[i][j]+ = A[i][k]*B[k][j];//Computeno$
$0.0;for(i = 0;i < n;i++)for(j = 0;j < n;j++)Fsum+ = C[i][j]*C[i][j];Fsum = sqrt(Fsum);//endparallelregionandlettingonlyonethreadp$
$omp_get_wtime()-wtime;cout << setiosflags(ios::showpoint|ios::uppercase);cout << setprecision(10) << setw(20) <<$
$"Timeusedformatrix-matrixmultiplication = " << wtime << endl;cout << "Frobeniusnorm = " << Fsum <<$
$endl;//Freeupspacefor(inti = 0;i < n;i++)delete[]A[i];delete[]B[i];delete[]C[i];delete[]A;delete[]B;delete[]C;return0;$

Part IV
Machine Learning

Chapter 10
Neural Networks

Introduction

Artificial neural networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and each connection is represented by a weight variable.

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{10.1}$$

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \dots, x_n$ received by n other neurons.

Conceptually, it is helpful to divide neural networks into four categories:

1. general purpose neural networks for supervised learning,
2. neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs),
3. neural networks for sequential data such as Recurrent Neural Networks (RNNs), and
4. neural networks for unsupervised learning such as Deep Boltzmann Machines.

In natural science, DNNs and CNNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum physics. Various quantum phase transitions can be detected and studied using DNNs and CNNs, topological phases, and even non-equilibrium many-body localization. Representing quantum states as DNNs quantum state tomography are among some of the impressive achievements to reveal the potential of DNNs to facilitate the study of quantum systems.

In quantum information theory, it has been shown that one can perform gate decompositions with the help of neural.

The applications are not limited to the natural sciences. There is a plethora of applications in essentially all disciplines, from the humanities to life science and medicine.

An artificial neural network (ANN), is a computational model that consists of layers of connected neurons, or nodes or units. We will refer to these interchangeably as units or nodes, and sometimes as neurons.

It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. A wide variety of different ANNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

Neural networks (also called neural nets) are neural-inspired nonlinear models for supervised learning. As we will see, neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods we discussed earlier.

Feed-forward neural networks.

The feed-forward neural network (FFNN) was the first and simplest type of ANNs that were devised. In this network, the information moves in only one direction: forward through the layers.

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionally, each arrow corresponds to a weight variable (figure to come). We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN.

Convolutional Neural Network.

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation. (figure to come)

Convolutional neural networks emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected only to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition.

Recurrent neural networks.

So far we have only mentioned ANNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

Other types of networks.

There are many other kinds of ANNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer (”linear” here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN. They are however usually treated as a separate type of NN due the unusual activation functions.

Multilayer perceptrons

One uses often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions.

Such networks are often called *multilayer perceptrons* (MLPs).

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**.

Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values.

The output y is produced via the activation function f

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z),$$

This function receives x_i as inputs. Here the activation $z = (\sum_{i=1}^{n} w_i x_i + b_i)$. In an FFNN of such neurons, the *inputs* x_i are the *outputs* of the neurons in the preceding layer. Furthermore, an MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

First, for each node i in the first hidden layer, we calculate a weighted sum z_i^1 of the input coordinates x_j,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{10.2}$$

Here b_i is the so-called bias which is normally needed in case of zero activation weights or inputs. How to fix the biases and the weights will be discussed below. The value of z_i^1 is the argument to the activation function f_i of each node i, The variable M stands for all possible inputs to a given node i in the first layer. We define the output y_i^1 of all neurons in layer 1 as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{10.3}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f. In general, we could assume in the more general case that different layers have different activation functions. In this case we would identify these functions with a superscript l for the l-th layer,

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{10.4}$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained.

The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{10.5}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{10.6}$$

where we have substituted y_k^1 with the inputs x_k. Finally, the ANN output reads

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{10.7}$$

$$= f_3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{10.8}$$

We can generalize this expression to an MLP with l hidden layers. The complete functional form is,

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right] \tag{10.9}$$

which illustrates a basic property of MLPs: The only independent variables are the input values x_n.

This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\hat{x} \in \mathbb{R}^n \rightarrow \hat{y} \in \mathbb{R}^m$.

Furthermore, the flexibility and universality of an MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \tag{10.10}$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

We can introduce a more convenient notation for the activations in an A NN.

Additionally, we can represent the biases and activations as layer-wise column vectors $\hat{b}_l$ and $\hat{y}_l$, so that the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively.

We have that W_l is an $N_{l-1} \times N_l$ matrix, while $\hat{b}_l$ and $\hat{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 (assuming three nodes for simplicity) as

$$\hat{y}_2 = f_2(\mathrm{W}_2\hat{y}_1 + \hat{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \tag{10.11}$$

The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \tag{10.12}$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\mathrm{W}_l\hat{y}_{l-1}$ we move forward one layer.

Activation functions.

A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). As described in, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions Typical examples are the logistic *Sigmoid*

$$f(x) = \frac{1}{1+e^{-x}},$$

and the *hyperbolic tangent* function

$$f(x) = \tanh(x)$$

The *sigmoid* function are more biologically plausible because the output of inactive neurons are zero. Such activation function are called *one-sided*. However, it has been shown that the hyperbolic tangent performs better than the sigmoid for training MLPs. has become the most popular for *deep neural networks*

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python """The sigmoid function (or the logistic curve) is a function that takes any real number, z, and outputs a number (0,1). It is useful in neural networks for assigning weights on a relative scale. The value z is the weighted sum of parameters involved in the learning algorithm."""

import numpy import matplotlib.pyplot as plt import math as mt z = numpy.arange(-5, 5, .1) sigma$_{fn} = numpy.vectorize(lambdaz : 1/(1+numpy.exp(-z)))sigma =$ $sigma_f n(z)$

fig = plt.figure() ax = fig.add$_{subplot}(111)ax.plot(z,sigma)ax.set_y lim([-0.1,1.1])ax.set_x lim([-5,5])ax.grid(True)ax.set_x label('z')ax.s$

plt.show()

"""Step Function""" z = numpy.arange(-5, 5, .02) step$_{fn} = numpy.vectorize(lambdaz : 1.0ifz >=$ $0.0else0.0)step = step_f n(z)$

fig = plt.figure() ax = fig.add$_{subplot}(111)ax.plot(z,step)ax.set_y lim([-0.5,1.5])ax.set_x lim([-5,5])ax.grid(True)ax.set_x label('z')ax.se$

plt.show()

"""Sine Function""" z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1) t = numpy.sin(z)

fig = plt.figure() ax = fig.add$_{subplot}(111)ax.plot(z,t)ax.set_y lim([-1.0,1.0])ax.set_x lim([-2*mt.pi,2*$ $mt.pi])ax.grid(True)ax.set_x label('z')ax.set_t itle('sinefunction')$

plt.show()

"""Plots a graph of the squashing function used by a rectified linear unit""" z = numpy.arange(-2, 2, .1) zero = numpy.zeros(len(z)) y = numpy.max([zero, z], axis=0)

fig = plt.figure() ax = fig.add$_{subplot}(111)ax.plot(z,y)ax.set_y lim([-2.0,2.0])ax.set_x lim([-2.0,2.0])ax.grid(True)ax.set_x label('z')ax.se$

plt.show()

The multilayer perceptron (MLP)

The multilayer perceptron is a very popular, and easy to implement approach, to deep learning. It consists of

1. A neural network with one or more layers of nodes between the input and the output nodes.
2. The multilayer network structure, or architecture, or topology, consists of an input layer, one or more hidden layers, and one output layer.
3. The input nodes pass values to the first hidden layer, its nodes pass the information on to the second and so on till we reach the output layer.

As a convention it is normal to call a network with one layer of input units, one layer of hidden units and one layer of output units as a two-layer network. A network with two layers of hidden units is called a three-layer network etc etc.

For an MLP network there is no direct connection between the output nodes/neurons/units and the input nodes/neurons/units. Hereafter we will call the various entities of a layer for nodes. There are also no connections within a single layer.

The number of input nodes does not need to equal the number of output nodes. This applies also to the hidden layers. Each layer may have its own number of nodes and activation functions.

The hidden layers have their name from the fact that they are not linked to observables and as we will see below when we define the so-called activation $\hat{z}$, we can think of this as a basis expansion of the original inputs $\hat{x}$. The difference however between neural networks and say linear regression is that now these basis functions (which will correspond to the weights in the network) are learned from data. This results in an important difference between neural networks and deep learning approaches on one side and methods like logistic regression or linear regression and their modifications on the other side.

From one to many layers, the universal approximation theorem.

A neural network with only one layer, what we called the simple perceptron, is best suited if we have a standard binary model with clear (linear) boundaries between the outcomes. As such it could equally well be replaced by standard linear regression or logistic regression. Networks with one or more hidden layers approximate systems with more complex boundaries.

As stated earlier, an important theorem in studies of neural networks, restated without proof here, is the universal approximation theorem.

It states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of real functions. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters. It is the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators.

Deriving the back propagation code for a multilayer perceptron model

As we have seen now in a feed forward network, we can express the final output of our network in terms of basic matrix-vector multiplications. The unknowwn quantities are our weights w_{ij} and we need to find an algorithm for changing them so that our errors are as small as possible. This leads us to the famous back propagation algorithm.

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights?

To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\hat{W}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - t_i\right)^2,$$

where the t_is are our n targets (the values we want to reproduce), while the outputs of the network after having propagated all inputs $\hat{x}$ are given by y_i. Below we will demonstrate how the basic equations arising from the back propagation algorithm can be modified in order to study classification problems with K classes.

With our definition of the targets $\hat{t}$, the outputs of the network $\hat{y}$ and the inputs $\hat{x}$ we define now the activation z_j^l of node/neuron/unit j of the l-th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the forward passes/outputs $\hat{a}^{l-1}$ from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

where b_k^l are the biases from layer l. Here M_{l-1} represents the total number of nodes/neurons/units of layer $l-1$. The figure here illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\hat{z}^l = \left(\hat{W}^l\right)^T \hat{a}^{l-1} + \hat{b}^l.$$

With the activation values $\hat{z}^l$ we can in turn define the output of layer l as $\hat{a}^l = f(\hat{z}^l)$ where f is our activation function. In the examples here we will use the sigmoid function discussed in our logistic regression lectures. We will also use the same activation function f for all layers and their nodes. It means we have

$$a_j^l = f(z_j^l) = \frac{1}{1+\exp-(z_j^l)}.$$

From the definition of the activation z_j^l we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1},$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1-a_j^l) = f(z_j^l)(1-f(z_j^l)).$$

With these definitions we can now compute the derivative of the cost function in terms of the weights.

Let us specialize to the output layer $l = L$. Our cost function is

$$\mathscr{C}(\hat{W}^L) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(a_i^L - t_i\right)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial \mathscr{C}(\hat{W}^L)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right)\frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1-a_j^L)a_k^{L-1},$$

We have thus

$$\frac{\partial \mathscr{C}(\hat{W}^L)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right)a_j^L(1-a_j^L)a_k^{L-1},$$

Defining

$$\delta_j^L = a_j^L(1-a_j^L)\left(a_j^L - t_j\right) = f'(z_j^L)\frac{\partial \mathscr{C}}{\partial (a_j^L)},$$

and using the Hadamard product of two vectors we can write this as

$$\hat{\delta}^L = f'(\hat{z}^L) \circ \frac{\partial \mathscr{C}}{\partial (\hat{a}^L)}.$$

This is an important expression. The second term on the right handside measures how fast the cost function is changing as a function of the jth output activation. If, for example, the cost function doesn't depend much on a particular output node j, then δ_j^L will be small, which is what we would expect. The first term on the right, measures how fast the activation function f is changing at a given activation value z_j^L.

Notice that everything in the above equations is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it is only a small additional overhead to compute $f'(z_j^L)$. The exact form of the derivative with respect to the output depends on the form of the cost function. However, provided the cost function is known there should be little trouble in calculating

$$\frac{\partial \mathscr{C}}{\partial (a_j^L)}$$

With the definition of δ_j^L we have a more compact definition of the derivative of the cost function in terms of the weights, namely

$$\frac{\partial \mathscr{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}.$$

It is also easy to see that our previous equation can be written as

$$\delta_j^L = \frac{\partial \mathscr{C}}{\partial z_j^L} = \frac{\partial \mathscr{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L},$$

which can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L, namely

$$\delta_j^L = \frac{\partial \mathscr{C}}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathscr{C}}{\partial b_j^L},$$

That is, the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

We have now three equations that are essential for the computations of the derivatives of the cost function at the output layer. These equations are needed to start the algorithm and they are

The starting equations.

$$\frac{\partial \mathscr{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{10.13}$$

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathscr{C}}{\partial (a_j^L)}, \tag{10.14}$$

and

$$\delta_j^L = \frac{\partial \mathscr{C}}{\partial b_j^L}, \tag{10.15}$$

An interesting consequence of the above equations is that when the activation a_k^{L-1} is small, the gradient term, that is the derivative of the cost function with respect to the weights, will also tend to be small. We say then that the weight learns slowly, meaning that it changes slowly when we minimize the weights via say gradient descent. In this case we say the system learns slowly.

Another interesting feature is that is when the activation function, represented by the sigmoid function here, is rather flat when we move towards its end values 0 and 1 (see the above Python codes). In these cases, the derivatives of the activation function will also be close to zero, meaning again that the gradients will be small and the network learns slowly again.

We need a fourth equation and we are set. We are going to propagate backwards in order to the determine the weights and biases. In order to do so we need to represent the error in the layer before the final one $L-1$ in terms of the errors in the final output layer.

We have that (replacing L with a general layer l)

$$\delta_j^l = \frac{\partial \mathscr{C}}{\partial z_j^l}.$$

We want to express this in terms of the equations for layer $l+1$. Using the chain rule and summing over all k entries we have

$$\delta_j^l = \sum_k \frac{\partial \mathscr{C}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l},$$

and recalling that

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1} a_i^l + b_j^{l+1},$$

with M_l being the number of nodes in layer l, we obtain

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l),$$

This is our final equation.

We are now ready to set up the algorithm for back propagation and learning the weights and biases.

Setting up the Back propagation algorithm

The four equations provide us with a way of computing the gradient of the cost function. Let us write this out in the form of an algorithm.

First, we set up the input data $\hat{x}$ and the activations $\hat{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^1$.

Secondly, we perform then the feed forward till we reach the output layer and compute all $\hat{z}_l$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^l$ for $l = 2, 3, \ldots, L$.

Thereafter we compute the ouput error $\hat{\delta}^L$ by computing all

$$\delta_j^L = f'(z_j^L) \frac{\partial \mathscr{C}}{\partial (a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \ldots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \dots, 2$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow = w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathscr{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

The parameter η is the learning parameter discussed in connection with the gradient descent methods. Here it is convenient to use stochastic gradient descent (see the examples below) with mini-batches with an outer loop that steps through multiple epochs of training.

Setting up a Multi-layer perceptron model for classification

We are now gong to develop an example based on the MNIST data base. This is a classification problem and we need to use our cross-entropy function we discussed in connection with logistic regression. The cross-entropy defines our cost function for the classificaton problems with neural networks.

In binary classification with two classes $(0,1)$ we define the logistic/sigmoid function as the probability that a particular input is in class 0 or 1. This is possible because the logistic function takes any input from the real numbers and inputs a number between 0 and 1, and can therefore be interpreted as a probability. It also has other nice properties, such as a derivative that is simple to calculate.

For an input a from the hidden layer, the probability that the input x is in class 0 or 1 is just. We let θ represent the unknown weights and biases to be adjusted by our equations). The variable x represents our activation values z. We have

$$P(y=0 \mid \hat{x}, \hat{\theta}) = \frac{1}{1+\exp{(-\hat{x})}},$$

and

$$P(y=1 \mid \hat{x}, \hat{\theta}) = 1 - P(y=0 \mid \hat{x}, \hat{\theta}),$$

where $y \in \{0,1\}$ and $\hat{\theta}$ represents the weights and biases of our network.

Our cost function is given as (see the Logistic regression lectures)

$$\mathscr{C}(\hat{\theta}) = -\ln P(\mathscr{D} \mid \hat{\theta}) = -\sum_{i=1}^n y_i \ln[P(y_i = 0)] + (1-y_i)\ln[1-P(y_i=0)] = \sum_{i=1}^n \mathscr{L}_i(\hat{\theta}).$$

This last equality means that we can interpret our *cost* function as a sum over the *loss* function for each point in the dataset $\mathscr{L}_i(\hat{\theta})$. The negative sign is just so that we can think about our algorithm as minimizing a positive number, rather than maximizing a negative number.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$y = 5 \quad \rightarrow \quad \hat{y} = (0,0,0,0,0,1,0,0,0,0)$, and

$y = 1 \quad \rightarrow \quad \hat{y} = (0,1,0,0,0,0,0,0,0,0),$

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset (numbers from 0 to 9)..

If $\hat{x}_i$ is the i-th input (image), y_{ic} refers to the c-th component of the i-th output vector $\hat{y}_i$. The probability of $\hat{x}_i$ being in class c will be given by the softmax function:

$$P(y_{ic} = 1 \mid \hat{x}_i, \hat{\theta}) = \frac{\exp\left((\hat{a}_i^{hidden})^T \hat{w}_c\right)}{\sum_{c'=0}^{C-1} \exp\left((\hat{a}_i^{hidden})^T \hat{w}_{c'}\right)},$$

which reduces to the logistic function in the binary case. The likelihood of this C-class classifier is now given as:

$$P(\mathscr{D} \mid \hat{\theta}) = \prod_{i=1}^{n} \prod_{c=0}^{C-1} [P(y_{ic} = 1)]^{y_{ic}}.$$

Again we take the negative log-likelihood to define our cost function:

$$\mathscr{C}(\hat{\theta}) = -\log P(\mathscr{D} \mid \hat{\theta}).$$

See the logistic regression lectures for a full definition of the cost function.

The back propagation equations need now only a small change, namely the definition of a new cost function. We are thus ready to use the same equations as before!

Example: binary classification problem.

As an example of the above, relevant for project 2 as well, let us consider a binary class. As discussed in our logistic regression lectures, we defined a cost function in terms of the parameters β as

$$\mathscr{C}(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i \log p(y_i|x_i,\hat{\beta}) + (1-y_i) \log 1 - p(y_i|x_i,\hat{\beta}) \right),$$

where we had defined the logistic (sigmoid) function

$$p(y_i = 1|x_i,\hat{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$

and

$$p(y_i = 0|x_i,\hat{\beta}) = 1 - p(y_i = 1|x_i,\hat{\beta}).$$

The parameters $\hat{\beta}$ were defined using a minimization method like gradient descent or Newton-Raphson's method.

Now we replace x_i with the activation z_i^l for a given layer l and the outputs as $y_i = a_i^l = f(z_i^l)$, with z_i^l now being a function of the weights w_{ij}^l and biases b_i^l. We have then

$$a_i^l = y_i = \frac{\exp(z_i^l)}{1 + \exp(z_i^l)},$$

with

$$z_i^l = \sum_j w_{ij}^l a_j^{l-1} + b_i^l,$$

where the superscript $l-1$ indicates that these are the outputs from layer $l-1$. Our cost function at the final layer $l = L$ is now

$$\mathscr{C}(\hat{W}) = -\sum_{i=1}^{n} \left(t_i \log a_i^L + (1-t_i)\log\left(1-a_i^L\right)\right),$$

where we have defined the targets t_i. The derivatives of the cost function with respect to the output a_i^L are then easily calculated and we get

$$\frac{\partial \mathscr{C}(\hat{W})}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L(1-a_i^L)}.$$

In case we use another activation function than the logistic one, we need to evaluate other derivatives.

The Softmax function.

In case we employ the more general case given by the Softmax equation, we need to evaluate the derivative of the activation function with respect to the activation z_i^l, that is we need

$$\frac{\partial f(z_i^l)}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l} a_k^{l-1}.$$

For the Softmax function we have

$$f(z_i^l) = \frac{\exp(z_i^l)}{\sum_{m=1}^{K}\exp(z_m^l)}.$$

Its derivative with respect to z_j^l gives

$$\frac{\partial f(z_i^l)}{\partial z_j^l} = f(z_i^l)\left(\delta_{ij} - f(z_j^l)\right),$$

which in case of the simply binary model reduces to having $i = j$.

Developing a code for doing neural networks with back propagation.

One can identify a set of key steps when using neural networks to solve supervised learning problems:

1. Collect and pre-process data
2. Define model and architecture
3. Choose cost function and optimizer
4. Train the model
5. Evaluate model performance on test data
6. Adjust hyperparameters (if necessary, network architecture)

Collect and pre-process data.

Here we will be using the MNIST dataset, which is readily available through the **scikit-learn** package. You may also find it for example here. The *MNIST* (Modified National Institute of Standards and Technology) database is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST dataset consists of

70 000 images of size 28×28 pixels, each labeled from 0 to 9. The scikit-learn dataset we will use consists of a selection of 1797 images of size 8×8 collected and processed from this database.

To feed data into a feed-forward neural network we need to represent the inputs as a design/feature matrix $X = (n_{inputs}, n_{features})$. Each row represents an *input*, in this case a handwritten digit, and each column represents a *feature*, in this case a pixel. The correct answers, also known as *labels* or *targets* are represented as a 1D array of integers $Y = (n_{inputs}) = (5, 3, 1, 8, \ldots)$.

As an example, say we want to build a neural network using supervised learning to predict Body-Mass Index (BMI) from measurements of height (in m) and weight (in kg). If we have measurements of 5 people the design/feature matrix could be for example:

$$X = \begin{bmatrix} 1.85\&81 \\ 1.71\&65 \\ 1.95\&103 \\ 1.55\&42 \\ 1.63\&56 \end{bmatrix},$$

and the targets would be:

$$Y = (23.7, 22.2, 27.1, 17.5, 21.1)$$

Since each input image is a 2D matrix, we need to flatten the image (i.e. "unravel" the 2D matrix into a 1D array) to turn the data into a design/feature matrix. This means we lose all spatial information in the image, such as locality and translational invariance. More complicated architectures such as Convolutional Neural Networks can take advantage of such information, and are most commonly applied when analyzing images.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python import necessary packages import numpy as np import matplotlib.pyplot as plt from sklearn import datasets

ensure the same random numbers appear every time np.random.seed(0)

display images in notebook plt.rcParams['figure.figsize'] = (12,12)

download MNIST dataset digits = datasets.load$_{d}igits()$

define inputs and labels inputs = digits.images labels = digits.target

print("inputs = (n$_{i}nputs, pixel_{w}idth, pixel_{h}eight) = " + str(inputs.shape))print("labels = (n_{i}nputs) = " + str(labels.shape))$

flatten the image the value -1 means dimension is inferred from the remaining dimensions: 8x8 = 64 n$_{i}nputs = len(inputs)inputs = inputs.reshape(n_{i}nputs, -1)print("X = (n_{i}nputs, n_{f}eatures) = " + str(inputs.shape))$

choose some random images to display indices = np.arange(n$_{i}nputs)random_{i}ndices = np.random.choice(indices, size = 5)$

for i, image in enumerate(digits.images[random$_{i}ndices]) : plt.subplot(1, 5, i+1)plt.axis('off')plt.imshow(image, cmap = plt.cm.gray_{r}, interpolation =' nearest')plt.title("Label : plt.show()$

Train and test datasets.

Performing analysis before partitioning the dataset is a major error, that can lead to incorrect conclusions.

We will reserve 80% of our dataset for training and 20% for testing.

It is important that the train and test datasets are drawn randomly from our dataset, to ensure no bias in the sampling. Say you are taking measurements of weather data to predict the weather in the coming 5 days. You don’t want to train your model on measurements taken from the hours 00.00 to 12.00, and then test it on data collected from 12.00 to 24.00.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sklearn.model$_{s}electionimporttrain_{t}est_{s}plit$

one-liner from scikit-learn library train$_{s}ize = 0.8test_{s}ize = 1 - train_{s}izeX_{t}rain, X_{t}est, Y_{t}rain, Y_{t}est = train_{t}est_{s}plit(inputs, labels, train_{s}ize = train_{s}ize, test_{s}ize = test_{s}ize)$

equivalently in numpy def train$_{t}est_{s}plit_{n}umpy(inputs, labels, train_{s}ize, test_{s}ize) : n_{i}nputs = len(inputs)inputs_{s}huffled = inputs.copy()labels_{s}huffled = labels.copy()$

np.random.shuffle(inputs$_{s}huffled)np.random.shuffle(labels_{s}huffled)$

train$_{e}nd = int(n_{i}nputs * train_{s}ize)X_{t}rain, X_{t}est = inputs_{s}huffled[: train_{e}nd], inputs_{s}huffled[train_{e}nd :]Y_{t}rain, Y_{t}est = labels_{s}huffled[: train_{e}nd], labels_{s}huffled[train_{e}nd :]$

return X$_{t}rain, X_{t}est, Y_{t}rain, Y_{t}est$

X$_{t}rain, X_{t}est, Y_{t}rain, Y_{t}est = train_{t}est_{s}plit_{n}umpy(inputs, labels, train_{s}ize, test_{s}ize)$

print("Number of training images: " + str(len(X$_{t}rain)))print("Numberoftestimages : " + str(len(X_{t}est)))$

Define model and architecture.

Our simple feed-forward neural network will consist of an *input* layer, a single *hidden* layer and an *output* layer. The activation y of each neuron is a weighted sum of inputs, passed through an activation function. In case of the simple perceptron model we have

$$z = \sum_{i=1}^{n} w_i a_i,$$

$$y = f(z),$$

where f is the activation function, a_i represents input from neuron i in the preceding layer and w_i is the weight to input i. The activation of the neurons in the input layer is just the features (e.g. a pixel value).

The simplest activation function for a neuron is the *Heaviside* function:

$$f(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

A feed-forward neural network with this activation is known as a *perceptron*. For a binary classifier (i.e. two classes, 0 or 1, dog or not-dog) we can also use this in our output layer. This activation can be generalized to k classes (using e.g. the *one-against-all* strategy), and we call these architectures *multiclass perceptrons*.

However, it is now common to use the terms Single Layer Perceptron (SLP) (1 hidden layer) and Multilayer Perceptron (MLP) (2 or more hidden layers) to refer to feed-forward neural networks with any activation function.

Typical choices for activation functions include the sigmoid function, hyperbolic tangent, and Rectified Linear Unit (ReLU). We will be using the sigmoid function $\sigma(x)$:

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}},$$

which is inspired by probability theory (see logistic regression) and was most commonly used until about 2011. See the discussion below concerning other activation functions.

Layers.

- Input

Since each input image has 8x8 = 64 pixels or features, we have an input layer of 64 neurons.

- Hidden layer

We will use 50 neurons in the hidden layer receiving input from the neurons in the input layer. Since each neuron in the hidden layer is connected to the 64 inputs we have 64x50 = 3200 weights to the hidden layer.

- Output

If we were building a binary classifier, it would be sufficient with a single neuron in the output layer, which could output 0 or 1 according to the Heaviside function. This would be an example of a *hard* classifier, meaning it outputs the class of the input directly. However, if we are dealing with noisy data it is often beneficial to use a *soft* classifier, which outputs the probability of being in class 0 or 1.

For a soft binary classifier, we could use a single neuron and interpret the output as either being the probability of being in class 0 or the probability of being in class 1. Alternatively we could use 2 neurons, and interpret each neuron as the probability of being in each class.

Since we are doing multiclass classification, with 10 categories, it is natural to use 10 neurons in the output layer. We number the neurons $j = 0, 1, ..., 9$. The activation of each output neuron j will be according to the *softmax* function:

$$P(\text{class } j \mid \text{input } \hat{a}) = \frac{\exp\left(\hat{a}^T \hat{w}_j\right)}{\sum_{c=0}^{9} \exp\left(\hat{a}^T \hat{w}_c\right)},$$

i.e. each neuron j outputs the probability of being in class j given an input from the hidden layer $\hat{a}$, with $\hat{w}_j$ the weights of neuron j to the inputs. The denominator is a normalization factor to ensure the outputs (probabilities) sum up to 1. The exponent is just the weighted sum of inputs as before:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

Since each neuron in the output layer is connected to the 50 inputs from the hidden layer we have 50x10 = 500 weights to the output layer.

Weights and biases.

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Setting all weights to zero means all neurons give the same output, making the network useless.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias unit has an output of 1, and a weight to each neuron j, b_j:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

The bias weights $\hat{b}$ are often initialized to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python building our neural network

n$_{i}nputs, n_{f}eatures = X_{t}rain.shapen_{h}idden_{n}eurons = 50n_{c}ategories = 10$

we make the weights normally distributed using numpy.random.randn

weights and bias in the hidden layer hidden$_{w}eights = np.random.randn(n_{f}eatures, n_{h}idden_{n}eurons)hidden_{b}ias = np.zeros(n_{h}idden_{n}eurons) + 0.01$

weights and bias in the output layer output$_{w}eights = np.random.randn(n_{h}idden_{n}eurons, n_{c}ategories)output_{b}ias = np.zeros(n_{c}ategories) + 0.01$

Feed-forward pass.

Denote F the number of features, H the number of hidden neurons and C the number of categories. For each input image we calculate a weighted sum of input features (pixel values) to each neuron j in the hidden layer l:

$$z_j^l = \sum_{i=1}^{F} w_{ij}^l x_i + b_j^l,$$

this is then passed through our activation function

$$a_j^l = f(z_j^l).$$

We calculate a weighted sum of inputs (activations in the hidden layer) to each neuron j in the output layer:

$$z_j^L = \sum_{i=1}^{H} w_{ij}^L a_i^l + b_j^L.$$

Finally we calculate the output of neuron j in the output layer using the softmax function:

$$a_j^L = \frac{\exp(z_j^L)}{\sum_{c=0}^{C-1} \exp(z_c^L)}.$$

Matrix multiplications.

Since our data has the dimensions $X = (n_{inputs}, n_{features})$ and our weights to the hidden layer have the dimensions $W_{hidden} = (n_{features}, n_{hidden})$, we can easily feed the network all our training data in one go by taking the matrix product

$$XW^h = (n_{inputs}, n_{hidden}),$$

and obtain a matrix that holds the weighted sum of inputs to the hidden layer for each input image and each hidden neuron. We also add the bias to obtain a matrix of weighted sums to the hidden layer Z^h:

$$\hat{z}^l = \hat{X}\hat{W}^l + \hat{b}^l,$$

meaning the same bias (1D array with size equal number of hidden neurons) is added to each input image. This is then passed through the activation:

$$\hat{a}^l = f(\hat{z}^l).$$

This is fed to the output layer:

$$\hat{z}^L = \hat{a}^L \hat{W}^L + \hat{b}^L.$$

Finally we receive our output values for each image and each category by passing it through the softmax function:

$$output = softmax(\hat{z}^L) = (n_{inputs}, n_{categories}).$$

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
setup the feed-forward pass, subscript h = hidden layer

def sigmoid(x): return 1/(1 + np.exp(-x))

def feed$_forward(X): weightedsumofinputstothehiddenlayerz_h = np.matmul(X, hidden_weights) + hidden_biasactivationinthehiddenlayera$
$sigmoid(z_h)$

weighted sum of inputs to the output layer z$_o = np.matmul(a_h, output_weights) + output_biassoftmaxoutputaxis0holdseachinputand$
$np.exp(z_o) probabilities = exp_term/np.sum(exp_term, axis = 1, keepdims = True)$

return probabilities

probabilities = feed$_forward(X_train)print(\text{"}probabilities = (n_inputs, n_categories) = \text{"} + str(probabilities.shape))print(\text{"}probabilitythat$
$\text{"} + str(probabilities[0]))print(\text{"}probabilitiessumupto: \text{"} + str(probabilities[0].sum()))print()$

we obtain a prediction by taking the class with the highest likelihood def predict(X): probabilities = feed$_forward(X)returnnp.argmax(probabilities, axis = 1)$

predictions = predict(X$_train)print(\text{"}predictions = (n_inputs) = \text{"} + str(predictions.shape))print(\text{"}predictionforimage0:$
$\text{"} + str(predictions[0]))print(\text{"}correctlabelforimage0: \text{"} + str(Y_train[0]))$

Choose cost function and optimizer.

To measure how well our neural network is doing we need to introduce a cost function. We will call the function that gives the error of a single sample output the *loss* function, and the function that gives the total error of our network across all samples the *cost* function. A typical choice for multiclass classification is the *cross-entropy* loss, also known as the negative log likelihood.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$$y = 5 \quad \rightarrow \quad \hat{y} = (0,0,0,0,0,1,0,0,0,0),$$

$$y = 1 \quad \rightarrow \quad \hat{y} = (0,1,0,0,0,0,0,0,0,0),$$

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset.

Let y_{ic} denote the c-th component of the i-th one-hot vector. We define the cost function $\mathcal{C}$ as a sum over the cross-entropy loss for each point $\hat{x}_i$ in the dataset.

In the one-hot representation only one of the terms in the loss function is non-zero, namely the probability of the correct category c' (i.e. the category c' such that $y_{ic'} = 1$). This means that the cross entropy loss only punishes you for how wrong you got the correct label. The probability of category c is given by the softmax function. The vector $\hat{\theta}$ represents the parameters of our network, i.e. all the weights and biases.

Optimizing the cost function.

The network is trained by finding the weights and biases that minimize the cost function. One of the most widely used classes of methods is *gradient descent* and its generalizations. The idea behind gradient descent is simply to adjust the weights in the direction where the gradient of the cost function is large and negative. This ensures we flow toward a *local* minimum of the cost function. Each parameter θ is iteratively adjusted according to the rule

$$\theta_{i+1} = \theta_i - \eta \nabla \mathscr{C}(\theta_i),$$

where η is known as the *learning rate*, which controls how big a step we take towards the minimum. This update can be repeated for any number of iterations, or until we are satisfied with the result.

A simple and effective improvement is a variant called *Batch Gradient Descent*. Instead of calculating the gradient on the whole dataset, we calculate an approximation of the gradient on a subset of the data called a *minibatch*. If there are N data points and we have a minibatch size of M, the total number of batches is N/M. We denote each minibatch B_k, with $k = 1,2,...,N/M$. The gradient then becomes:

$$\nabla \mathscr{C}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla \mathscr{L}_i(\theta) \quad \rightarrow \quad \frac{1}{M}\sum_{i \in B_k} \nabla \mathscr{L}_i(\theta),$$

i.e. instead of averaging the loss over the entire dataset, we average over a minibatch.

This has two important benefits:

1. Introducing stochasticity decreases the chance that the algorithm becomes stuck in a local minima.
2. It significantly speeds up the calculation, since we do not have to use the entire dataset to calculate the gradient.

The various optmization methods, with codes and algorithms, are discussed in our lectures on Gradient descent approaches.

Regularization.

It is common to add an extra term to the cost function, proportional to the size of the weights. This is equivalent to constraining the size of the weights, so that they do not grow out of control. Constraining the size of the weights means that the weights cannot grow arbitrarily large to fit the training data, and in this way reduces *overfitting*.

We will measure the size of the weights using the so called *L2-norm*, meaning our cost function becomes:

$$\mathscr{C}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathscr{L}_i(\theta) \quad \rightarrow \quad \frac{1}{N}\sum_{i=1}^{N} \mathscr{L}_i(\theta) + \lambda ||\hat{w}||_2^2 = \frac{1}{N}\sum_{i=1}^{N} \mathscr{L}(\theta) + \lambda \sum_{ij} w_{ij}^2,$$

i.e. we sum up all the weights squared. The factor λ is known as a regularization parameter.

In order to train the model, we need to calculate the derivative of the cost function with respect to every bias and weight in the network. In total our network has $(64+1)\times 50 = 3250$ weights in the hidden layer and $(50+1)\times 10 = 510$ weights to the output layer ($+1$ for the bias), and the gradient must be calculated for every parameter. We use the *backpropagation* algorithm discussed above. This is a clever use of the chain rule that allows us to calculate the gradient efficently.

Matrix multiplication.

To more efficently train our network these equations are implemented using matrix operations. The error in the output layer is calculated simply as, with $\hat{t}$ being our targets,

$$\delta_L = \hat{t} - \hat{y} = (n_{inputs}, n_{categories}).$$

The gradient for the output weights is calculated as

$$\nabla W_L = \hat{a}^T \delta_L = (n_{hidden}, n_{categories}),$$

where $\hat{a} = (n_{inputs}, n_{hidden})$. This simply means that we are summing up the gradients for each input. Since we are going backwards we have to transpose the activation matrix.

The gradient with respect to the output bias is then

$$\nabla \hat{b}_L = \sum_{i=1}^{n_{inputs}} \delta_L = (n_{categories}).$$

The error in the hidden layer is

$$\Delta_h = \delta_L W_L^T \circ f'(z_h) = \delta_L W_L^T \circ a_h \circ (1 - a_h) = (n_{inputs}, n_{hidden}),$$

where $f'(a_h)$ is the derivative of the activation in the hidden layer. The matrix products mean that we are summing up the products for each neuron in the output layer. The symbol $\circ$ denotes the *Hadamard product*, meaning element-wise multiplication.

This again gives us the gradients in the hidden layer:

$$\nabla W_h = X^T \delta_h = (n_{features}, n_{hidden}),$$

$$\nabla b_h = \sum_{i=1}^{n_{inputs}} \delta_h = (n_{hidden}).$$

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python
to categorical turns our integer vector into a onehot representation from sklearn.metrics import accuracy$_{score}$

one-hot in numpy def to$_{c}ategorical_{n}umpy(integer_{v}ector) : n_{i}nputs = len(integer_{v}ector)n_{c}ategories = np.max(integer_{v}ector) + 1onehot_{v}ector = np.zeros((n_{i}nputs, n_{c}ategories))onehot_{v}ector[range(n_{i}nputs), integer_{v}ector] = 1$

return onehot$_{v}ector$

Y$_{t}rain_{o}nehot, Y_{t}est_{o}nehot = to_{c}ategorical(Y_{t}rain), to_{c}ategorical(Y_{t}est)Y_{t}rain_{o}nehot, Y_{t}est_{o}nehot = to_{c}ategorical_{n}umpy(Y_{t}rain), to_{c}ategorical$

def feed$_{f}orward_{t}rain(X) : weightedsumofinputstothehiddenlayerz_{h} = np.matmul(X, hidden_{w}eights) + hidden_{b}iasactivationinthehiddenla$ $sigmoid(z_h)$

weighted sum of inputs to the output layer z$_{o} = np.matmul(a_{h}, output_{w}eights) + output_{b}iassoftmaxoutputaxis0holdseachinputand$ $np.exp(z_{o})probabilities = exp_{t}erm/np.sum(exp_{t}erm, axis = 1, keepdims = True)$

for backpropagation need activations in hidden and output layers return a$_{h}, probabilities$

def backpropagation(X, Y): a$_{h}, probabilities = feed_{f}orward_{t}rain(X)$

error in the output layer error$_{o}utput = probabilities - Yerrorinthehiddenlayererror_{h}idden = np.matmul(error_{o}utput, output_{w}eights.T$ $a_h * (1 - a_h)$

gradients for the output layer output$_{w}eights_{g}radient = np.matmul(a_{h}.T, error_{o}utput)output_{b}ias_{g}radient = np.sum(error_{o}utput, axis = 0)$

gradient for the hidden layer hidden$_{w}eights_{g}radient = np.matmul(X.T, error_{h}idden)hidden_{b}ias_{g}radient = np.sum(error_{h}idden, axis = 0)$

return output$_{w}eights_{g}radient, output_{b}ias_{g}radient, hidden_{w}eights_{g}radient, hidden_{b}ias_{g}radient$

print("Old accuracy on training data: " + str(accuracy$_{s}core(predict(X_{t}rain), Y_{t}rain)))$

eta = 0.01 lmbd = 0.01 for i in range(1000): calculate gradients dWo, dBo, dWh, dBh = backpropagation(X$_{t}rain, Y_{t}rain_{o}nehot)$

regularization term gradients dWo += lmbd * output$_{w}eightsdWh+=lmbd*hidden_{w}eights$

update weights and biases output$_{w}eights-=eta*dWooutput_{b}ias-=eta*dBohidden_{w}eights-=eta*dWhhidden_{b}ias-=eta*dBh$

print("New accuracy on training data: " + str(accuracy$_{s}core(predict(X_{t}rain),Y_{t}rain)))$

Improving performance.

As we can see the network does not seem to be learning at all. It seems to be just guessing the label for each image. In order to obtain a network that does something useful, we will have to do a bit more work.

The choice of *hyperparameters* such as learning rate and regularization parameter is hugely influential for the performance of the network. Typically a *grid-search* is performed, wherein we test different hyperparameters separated by orders of magnitude. For example we could test the learning rates $\eta = 10^{-6}, 10^{-5}, ..., 10^{-1}$ with different regularization parameters $\lambda = 10^{-6}, ..., 10^{-0}$.

Next, we haven't implemented minibatching yet, which introduces stochasticity and is though to act as an important regularizer on the weights. We call a feed-forward + backward pass with a minibatch an *iteration*, and a full training period going through the entire dataset (n/M batches) an *epoch*.

If this does not improve network performance, you may want to consider altering the network architecture, adding more neurons or hidden layers. Andrew Ng goes through some of these considerations in this video. You can find a summary of the video here.

Full object-oriented implementation.

It is very natural to think of the network as an object, with specific instances of the network being realizations of this object with different hyperparameters. An implementation using Python classes provides a clean structure and interface, and the full implementation of our neural network is given below.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python class NeuralNetwork: def $_{i}nit_{(self,X_{d}ata,Y_{d}ata,n_{h}idden_{n}eurons=50,n_{c}ategories=10,epochs=10,batch_{s}ize=100,eta=0.1,lmbd=0.0):}$

self.X$_{d}ata_{f}ull=X_{d}ataself.Y_{d}ata_{f}ull=Y_{d}ata$

self.n$_{i}nputs=X_{d}ata.shape[0]self.n_{f}eatures=X_{d}ata.shape[1]self.n_{h}idden_{n}eurons=n_{h}idden_{n}euronsself.n_{c}ategories=n_{c}ategories$

self.epochs = epochs self.batch$_{s}ize=batch_{s}izeself.iterations=self.n_{i}nputs//self.batch_{s}izeself.eta=etaself.lmbd=lmbd$

self.create$_{b}iases_{a}nd_{w}eights()$

def create$_{b}iases_{a}nd_{w}eights(self):self.hidden_{w}eights=np.random.randn(self.n_{f}eatures,self.n_{h}idden_{n}eurons)self.hidden_{b}ias=np.zeros(self.n_{h}idden_{n}eurons)+0.01$

self.output$_{w}eights=np.random.randn(self.n_{h}idden_{n}eurons,self.n_{c}ategories)self.output_{b}ias=np.zeros(self.n_{c}ategories)+0.01$

def feed$_{f}orward(self):feed-forwardfortrainingself.z_{h}=np.matmul(self.X_{d}ata,self.hidden_{w}eights)+self.hidden_{b}iasself.a_{h}=sigmoid(self.z_{h})$

self.z$_{o}=np.matmul(self.a_{h},self.output_{w}eights)+self.output_{b}ias$

exp$_{t}erm=np.exp(self.z_{o})self.probabilities=exp_{t}erm/np.sum(exp_{t}erm,axis=1,keepdims=True)$

def feed$_{f}orward_{o}ut(self,X):feed-forwardforoutputz_{h}=np.matmul(X,self.hidden_{w}eights)+self.hidden_{b}iasa_{h}=sigmoid(z_{h})$

z$_o = np.matmul(a_h, self.output_weights) + self.output_bias$

exp$_term = np.exp(z_o)probabilities = exp_term/np.sum(exp_term, axis = 1, keepdims = True)returnprobabilities$

def backpropagation(self): error$_output = self.probabilities - self.Y_data error_hidden = np.matmul(error_output, self.output_weights.T$
$self.a_h * (1 - self.a_h)$

self.output$_weights_gradient = np.matmul(self.a_h.T, error_output)self.output_bias_gradient = np.sum(error_output, axis =$
$0)$

self.hidden$_weights_gradient = np.matmul(self.X_data.T, error_hidden)self.hidden_bias_gradient = np.sum(error_hidden, axis =$
$0)$

if self.lmbd > 0.0: self.output$_weights_gradient + = self.lmbd * self.output_weightsself.hidden_weights_gradient + =$
$self.lmbd * self.hidden_weights$

self.output$_weights - = self.eta * self.output_weights_gradientself.output_bias - = self.eta * self.output_bias_gradientself.hidden_weights - =$
$self.eta * self.hidden_weights_gradientself.hidden_bias - = self.eta * self.hidden_bias_gradient$

def predict(self, X): probabilities = self.feed$_forward_out(X)returnnp.argmax(probabilities, axis = 1)$

def predict$_probabilities(self, X) : probabilities = self.feed_forward_out(X)returnprobabilities$

def train(self): data$_indices = np.arange(self.n_inputs)$

for i in range(self.epochs): for j in range(self.iterations): pick datapoints with replacement chosen$_datapoints = np.random.choice(data_indices, size = self.batch_size, replace = False)$

minibatch training data self.X$_data = self.X_data_full[chosen_datapoints]self.Y_data = self.Y_data_full[chosen_datapoints]$

self.feed$_forward()self.backpropagation()$

Evaluate model performance on test data.

To measure the performance of our network we evaluate how well it does it data it has never seen before, i.e. the test data. We measure the performance of the network using the *accuracy* score. The accuracy is as you would expect just the number of images correctly labeled divided by the total number of images. A perfect classifier will have an accuracy score of 1.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(\hat{y}_i = y_i)}{n},$$

where I is the indicator function, 1 if $\hat{y}_i = y_i$ and 0 otherwise.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python epochs = 100 batch$_size = 100$

dnn = NeuralNetwork(X$_train, Y_train_onehot, eta = eta, lmbd = lmbd, epochs = epochs, batch_size = batch_size, n_hidden_neurons =$
$n_hidden_neurons, n_categories = n_categories)dnn.train()test_predict = dnn.predict(X_test)$

accuracy score from scikit library print("Accuracy score on test set: ", accuracy$_score(Y_test, test_predict))$

equivalent in numpy def accuracy$_score_numpy(Y_test, Y_pred) : returnnp.sum(Y_test == Y_pred)/len(Y_test)$

print("Accuracy score on test set: ", accuracy$_score_numpy(Y_test, test_predict))$

Adjust hyperparameters.

We now perform a grid search to find the optimal hyperparameters for the network. Note that we are only using 1 layer with 50 neurons, and human performance is estimated to be around 98% (2% error rate).

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python eta$_vals = np.logspace(-5, 1, 7)lmbd_vals = np.logspace(-5, 1, 7)storethemodelsforlateruseDNN_numpy = np.zeros((len(eta_vals), len(lmbd_val$
$object)$

grid search for i, eta in enumerate(eta$_vals) : forj, lmbdinenumerate(lmbd_vals) : dnn = NeuralNetwork(X_train, Y_train_onehot, eta =$
$eta, lmbd = lmbd, epochs = epochs, batch_size = batch_size, n_hidden_neurons = n_hidden_neurons, n_categories = n_categories)dnn.train()$

DNN$_numpy[i][j] = dnn$

test$_p redict = dnn.predict(X_t est)$

print("Learning rate = ", eta) print("Lambda = ", lmbd) print("Accuracy score on test set: ", accuracy$_s core(Y_t est, test_p redict))print()$

Visualization.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python visual representation of grid search uses seaborn heatmap, you can also do this with matplotlib imshow import seaborn as sns

sns.set()

train$_a ccuracy = np.zeros((len(eta_v als), len(lmbd_v als)))test_a ccuracy = np.zeros((len(eta_v als), len(lmbd_v als)))$

for i in range(len(eta$_v als)) : forjinrange(len(lmbd_v als)) : dnn = DNN_n umpy[i][j]$

train$_p red = dnn.predict(X_t rain)test_p red = dnn.predict(X_t est)$

train$_a ccuracy[i][j] = accuracy_s core(Y_t rain, train_p red)test_a ccuracy[i][j] = accuracy_s core(Y_t est, test_p red)$

fig, ax = plt.subplots(figsize = (10, 10)) sns.heatmap(train$_a ccuracy, annot = True, ax = ax, cmap = $ $''viridis'')ax.set_t itle(''TrainingAccuracy'')ax.set_y label(''\eta$") ax.set$_x label(''\lambda$") plt.show()

fig, ax = plt.subplots(figsize = (10, 10)) sns.heatmap(test$_a ccuracy, annot = True, ax = ax, cmap = $ $''viridis'')ax.set_t itle(''TestAccuracy'')ax.set_y label(''\eta$") ax.set$_x label(''\lambda$") plt.show()

scikit-learn implementation.

scikit-learn focuses more on traditional machine learning methods, such as regression, clustering, decision trees, etc. As such, it has only two types of neural networks: Multi Layer Perceptron outputting continuous values, *MPLRegressor*, and Multi Layer Perceptron outputting labels, *MLPClassifier*. We will see how simple it is to use these classes.

scikit-learn implements a few improvements from our neural network, such as early stopping, a varying learning rate, different optimization methods, etc. We would therefore expect a better performance overall.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from sklearn.neural$_n etworkimportMLPClassifierstoremodelsforlateruseDNN_s cikit = np.zeros((len(eta_v als), len(lmbd_v als)), dtype = object)$

for i, eta in enumerate(eta$_v als) : forj, lmbdinenumerate(lmbd_v als) : dnn = MLPClassifier(hidden_l ayer_s izes = (n_h idden_n eurons), activation =' logistic', alpha = lmbd, learning_r ate_i nit = eta, max_i ter = epochs)dnn.fit(X_t rain, Y_t rain)$

DNN$_s cikit[i][j] = dnn$

print("Learning rate = ", eta) print("Lambda = ", lmbd) print("Accuracy score on test set: ", dnn.score(X$_t est, Y_t est))print()$

Visualization.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python optional visual representation of grid search uses seaborn heatmap, could probably do this in matplotlib import seaborn as sns

sns.set()

train$_a ccuracy = np.zeros((len(eta_v als), len(lmbd_v als)))test_a ccuracy = np.zeros((len(eta_v als), len(lmbd_v als)))$

for i in range(len(eta$_v als)) : forjinrange(len(lmbd_v als)) : dnn = DNN_s cikit[i][j]$

train$_p red = dnn.predict(X_t rain)test_p red = dnn.predict(X_t est)$

train$_a ccuracy[i][j] = accuracy_s core(Y_t rain, train_p red)test_a ccuracy[i][j] = accuracy_s core(Y_t est, test_p red)$

fig, ax = plt.subplots(figsize = (10, 10)) sns.heatmap(train$_{a}ccuracy, annot = True, ax = ax, cmap =$ $''viridis'')ax.set_{t}itle(''TrainingAccuracy'')ax.set_{y}label(''\eta$") ax.set$_{x}label(''\lambda$") plt.show()

fig, ax = plt.subplots(figsize = (10, 10)) sns.heatmap(test$_{a}ccuracy, annot = True, ax = ax, cmap =$ $''viridis'')ax.set_{t}itle(''TestAccuracy'')ax.set_{y}label(''\eta$") ax.set$_{x}label(''\lambda$") plt.show()

Building neural networks in Tensorflow and Keras

Now we want to build on the experience gained from our neural network implementation in NumPy and scikit-learn and use it to construct a neural network in Tensorflow. Once we have constructed a neural network in NumPy and Tensorflow, building one in Keras is really quite trivial, though the performance may suffer.

In our previous example we used only one hidden layer, and in this we will use two. From this it should be quite clear how to build one using an arbitrary number of hidden layers, using data structures such as Python lists or NumPy arrays.

Tensorflow is an open source library machine learning library developed by the Google Brain team for internal use. It was released under the Apache 2.0 open source license in November 9, 2015.

Tensorflow is a computational framework that allows you to construct machine learning models at different levels of abstraction, from high-level, object-oriented APIs like Keras, down to the C++ kernels that Tensorflow is built upon. The higher levels of abstraction are simpler to use, but less flexible, and our choice of implementation should reflect the problems we are trying to solve.

Tensorflow uses so-called graphs to represent your computation in terms of the dependencies between individual operations, such that you first build a Tensorflow *graph* to represent your model, and then create a Tensorflow *session* to run the graph.

In this guide we will analyze the same data as we did in our NumPy and scikit-learn tutorial, gathered from the MNIST database of images. We will give an introduction to the lower level Python Application Program Interfaces (APIs), and see how we use them to build our graph. Then we will build (effectively) the same graph in Keras, to see just how simple solving a machine learning problem can be.

To install tensorflow on Unix/Linux systems, use pip as

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python pip3 install tensorflow

and/or if you use **anaconda**, just write (or install from the graphical user interface) (current release of CPU-only TensorFlow)

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python conda create -n tf tensorflow conda activate tf

To install the current release of GPU TensorFlow

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python conda create -n tf-gpu tensorflow-gpu conda activate tf-gpu

Keras is a high level neural network that supports Tensorflow, CTNK and Theano as backends. If you have Anaconda installed you may run the following command

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python conda install keras

You can look up the instructions here for more information.

We will to a large extent use **keras** in this course.

Let us look again at the MINST data set.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python import necessary packages import numpy as np import matplotlib.pyplot as plt import tensorflow as tf from sklearn import datasets

ensure the same random numbers appear every time np.random.seed(0)

display images in notebook plt.rcParams['figure.figsize'] = (12,12)

download MNIST dataset digits = datasets.load$_digits()$

define inputs and labels inputs = digits.images labels = digits.target

print("inputs = (n$_inputs, pixel_width, pixel_height) = " + str(inputs.shape))print("labels = (n_inputs) = " + str(labels.shape))$

flatten the image the value -1 means dimension is inferred from the remaining dimensions: 8x8 = 64 n$_inputs = len(inputs)inputs = inputs.reshape(n_inputs, -1)print("X = (n_inputs, n_features) = " + str(inputs.shape))$

choose some random images to display indices = np.arange(n$_inputs)random_indices = np.random.choice(indices, size = 5)$

for i, image in enumerate(digits.images[random$_indices]) : plt.subplot(1,5,i+1)plt.axis('off')plt.imshow(image, cmap = plt.cm.gray_r, interpolation =' nearest')plt.title("Label : plt.show()$

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python from tensorflow.keras.layers import Input from tensorflow.keras.models import Sequential This allows appending layers to existing models from tensorflow.keras.layers import Dense This allows defining the characteristics of a particular layer from tensorflow.keras import optimizers This allows using whichever optimiser we want (sgd,adam,RMSprop) from tensorflow.keras import regularizers This allows using whichever regularizer we want (l1,l2,l1$_l2)fromtensorflow.keras.utilsimpo$

from sklearn.model$_selectionimporttrain_test_split$

one-hot representation of labels labels = to$_categorical(labels)$

split into train and test data train$_size = 0.8test_size = 1 - train_sizeX_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size = train_size, test_size = test_size)$

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python

epochs = 100 batch$_size = 100n_neurons_layer1 = 100n_neurons_layer2 = 50n_categories = 10eta_vals = np.logspace(-5,1,7)lmbd_vals = np.logspace(-5,1,7)defcreate_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_categories, eta, lmbd) : model = Sequential()model.add(Dense(n_neurons_layer1, activation =' sigmoid', kernel_regularizer = regularizers.l2(lmbd)))model.add(Dense(n_neurons_l$ $sigmoid', kernel_regularizer = regularizers.l2(lmbd)))model.add(Dense(n_categories, activation =' softmax'))$

sgd = optimizers.SGD(lr=eta) model.compile(loss='categorical$_crossentropy', optimizer = sgd, metrics = ['accuracy'])$

return model

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python DNN$_keras = np.zeros((len(eta_vals), len(lmbd_vals)), dtype = object)$

for i, eta in enumerate(eta$_vals) : forj, lmbdinenumerate(lmbd_vals) : DNN = create_neural_network_keras(n_neurons_layer1, n_neurons_layer$ $eta, lmbd = lmbd)DNN.fit(X_train, Y_train, epochs = epochs, batch_size = batch_size, verbose = 0)scores = DNN.evaluate(X_test, Y_test)$

DNN$_keras[i][j] = DNN$

print("Learning rate = ", eta) print("Lambda = ", lmbd) print("Test accuracy: print()

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python optional visual representation of grid search uses seaborn heatmap, could probably do this in matplotlib import seaborn as sns

sns.set()

train$_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))$

for i in range(len(eta$_vals)) : forjinrange(len(lmbd_vals)) : DNN = DNN_keras[i][j]$

train$_accuracy[i][j] = DNN.evaluate(X_train, Y_train)[1]test_accuracy[i][j] = DNN.evaluate(X_test, Y_test)[1]$

fig, ax = plt.subplots(figsize = (10, 10)) sns.heatmap(train$_accuracy, annot = True, ax = ax, cmap = "viridis")ax.set_title("TrainingAccuracy")ax.set_ylabel("\eta$") ax.set$_xlabel("\lambda$") plt.show()

fig, ax = plt.subplots(figsize = (10, 10)) sns.heatmap(test$_accuracy, annot = True, ax = ax, cmap = "viridis")ax.set_title("TestAccuracy")ax.set_ylabel("\eta$") ax.set$_xlabel("\lambda$") plt.show()

The Breast Cancer Data, now with Keras.

[fontsize=,linenos=false,mathescape,baselinestretch=1.0,fontfamily=tt,xleftmargin=7mm]python

import tensorflow as tf from tensorflow.keras.layers import Input from tensorflow.keras.models import Sequential This allows appending layers to existing models from tensorflow.keras.layers import Dense This allows defining the characteristics of a particular layer from tensorflow.keras import optimizers This allows using whichever optimiser we want (sgd,adam,RMSprop) from tensorflow.keras import regularizers This allows using whichever regularizer we want (l1,l2,l1$_{l}2)fromtensorflow.keras.utilsimportto_{c}ategoricalThisallowsusingcategoricalcrossentropyasthecostfunctionimportnumpyasnpimpo$

"""Load breast cancer dataset"""

np.random.seed(0) create same seed for random number every time

cancer=load$_{b}reast_{c}ancer()Downloadbreastcancerdataset$

inputs=cancer.data Feature matrix of 569 rows (samples) and 30 columns (parameters) outputs=cancer.target Label array of 569 rows (0 for benign and 1 for malignant) labels=cancer.feature$_{n}ames[0:30]$

print('The content of the breast cancer dataset is:') Print information about the datasets print(labels) print('————————————-') print("inputs = " + str(inputs.shape)) print("outputs = " + str(outputs.shape)) print("labels = "+ str(labels.shape))

x=inputs Reassign the Feature and Label matrices to other variables y=outputs

Visualisation of dataset (for correlation analysis)

plt.figure() plt.scatter(x[:,0],x[:,2],s=40,c=y,cmap=plt.cm.Spectral) plt.xlabel('Mean radius',fontweight='bold') plt.ylabel('Mean perimeter',fontweight='bold') plt.show()

plt.figure() plt.scatter(x[:,5],x[:,6],s=40,c=y, cmap=plt.cm.Spectral) plt.xlabel('Mean compactness',fontweight='bold') plt.ylabel('Mean concavity',fontweight='bold') plt.show()

plt.figure() plt.scatter(x[:,0],x[:,1],s=40,c=y,cmap=plt.cm.Spectral) plt.xlabel('Mean radius',fontweight='bold') plt.ylabel('Mean texture',fontweight='bold') plt.show()

plt.figure() plt.scatter(x[:,2],x[:,1],s=40,c=y,cmap=plt.cm.Spectral) plt.xlabel('Mean perimeter',fontweight='bold') plt.ylabel('Mean compactness',fontweight='bold') plt.show()

Generate training and testing datasets

Select features relevant to classification (texture,perimeter,compactness and symmetery) and add to input matrix

temp1=np.reshape(x[:,1],(len(x[:,1]),1)) temp2=np.reshape(x[:,2],(len(x[:,2]),1)) X=np.hstack((temp1,temp2)) temp=np.reshape(x[:,5],(len(x[:,5]),1)) X=np.hstack((X,temp)) temp=np.reshape(x[:,8],(len(x[:,8]),1)) X=np.hstack((X,temp))

X$_{t}rain,X_{t}est,y_{t}rain,y_{t}est = splitter(X,y,test_{s}ize = 0.1)Splitdatasetsintotrainingandtesting$

y$_{t}rain = to_{c}ategorical(y_{t}rain)Convertlabelstocategoricalwhenusingcategoricalcrossentropyy_{t}est = to_{c}ategorical(y_{t}est)$

del temp1,temp2,temp

Define tunable parameters"

eta=np.logspace(-3,-1,3) Define vector of learning rates (parameter to SGD optimiser) lamda=0.01 Define hyperparameter n$_{l}ayers = 2Definenumberofhiddenlayersinthemodeln_{n}euron = np.logspace(0,3,4,dtype = int)Definenumberofneuronsperlayerepochs = 100Numberofreiterationsovertheinputdatabatch_{s}ize = 100Numberofsamplespergradientupdat$

"""Define function to return Deep Neural Network model"""

def NN$_{m}odel(inputsize,n_{l}ayers,n_{n}euron,eta,lamda):model = Sequential()foriinrange(n_{l}ayers):Runlooptoaddhiddenlayerstothemodeli$ $0):Firstlayerrequiresinputdimensionsmodel.add(Dense(n_{n}euron,activation =' relu',kernel_{r}egularizer = regularizers.l2(lamda),input_{d}im = inputsize))else:Subsequentlayersarecapableofautomaticshapeinferencingmodel.add(Dense(n_{n}euron,activation =' relu',kernel_{r}egularizer = regularizers.l2(lamda)))model.add(Dense(2,activation =' softmax'))2outputs - orderedanddisordered(softmaxfor$ $optimizers.SGD(lr = eta)model.compile(loss =' categorical_{c}rossentropy',optimizer = sgd,metrics = ['accuracy'])returnmodel$

Train$_{a}ccuracy = np.zeros((len(n_{n}euron),len(eta)))Definematricestostoreaccuracyscoresasafunction Test_{a}ccuracy = np.zeros((len(n_{n}euron),len(eta)))oflearningrateandnumberofhiddenneuronsfor$

for i in range(len(n$_{n}euron)):runloopsoverhiddenneuronsandlearningratestocalculateforjinrange(len(eta)):accuracyscoresDNN_{m}odel = NN_{m}odel(X_{t}rain.shape[1],n_{l}ayers,n_{n}euron[i],eta[j],lamda)DNN_{m}odel.fit(X_{t}rain,y_{t}rain,epochs =$

$epochs, batch_size = batch_size, verbose = 1)Train_accuracy[i, j] = DNN_model.evaluate(X_train, y_train)[1]Test_accuracy[i, j] = DNN_model.evaluate(X_test, y_test)[1]$

def plot$_data(x, y, data, title = None):$

plot results fontsize=16

fig = plt.figure() ax = fig.add$_subplot(111)cax = ax.matshow(data, interpolation =' nearest', vmin = 0, vmax = 1)$

cbar=fig.colorbar(cax) cbar.ax.set$_ylabel('accuracy(cbar.set_ticks([0, .2, .4, 0.6, 0.8, 1.0])cbar.set_ticklabels(['0$

put text on matrix elements for i, x$_valinenumerate(np.arange(len(x))) : for j, y_valinenumerate(np.arange(len(y))) : c =$ "0:.1f

ax.text(x$_val, y_val, c, va =' center', ha =' center')$

convert axis vaues to to string labels x=[str(i) for i in x] y=[str(i) for i in y]

ax.set$_xticklabels(['']+x)ax.set_yticklabels(['']+y)$

ax.set$_xlabel('$

mathrmlearning

rate$', fontsize = fontsize)ax.set_ylabel('$

mathrmhidden

neurons$', fontsize = fontsize)iftitleisnotNone : ax.set_title(title)$

plt.tight$_layout()$

plt.show()

plot$_data(eta, n_neuron, Train_accuracy,' training')plot_data(eta, n_neuron, Test_accuracy,' testing')$

Fine-tuning neural network hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network topology (how neurons/nodes are interconnected), but even in a simple FFNN you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, the stochastic gradient optmized and much more. How do you know what combination of hyperparameters is the best for your task?

- You can use grid search with cross-validation to find the right hyperparameters.

However,since there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space.

- You can use randomized search.
- Or use tools like Oscar, which implements more complex algorithms to help you find a good set of hyperparameters quickly.

Hidden layers.

For many problems you can start with just one or two hidden layers and it will work just fine. For the MNIST data set you ca easily get a high accuracy using just one hidden layer with a few hundred neurons. You can reach for this data set above 98% accuracy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time.

For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task.

Which activation function should I use?

The Back propagation algorithm we derived above works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent (GD) step.

Unfortunately for us, the gradients often get smaller and smaller as the algorithm progresses down to the first hidden layers. As a result, the GD update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is known in the literature as **the vanishing gradients problem**.

In other cases, the opposite can happen, namely the the gradients can grow bigger and bigger. The result is that many of the layers get large updates of the weights the algorithm diverges. This is the **exploding gradients problem**, which is mostly encountered in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients, different layers may learn at widely different speeds

Is the Logistic activation function (Sigmoid) our choice?

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it.

A paper titled Understanding the Difficulty of Training Deep Feedforward Neural Networks by Xavier Glorot and Yoshua Bengio found that the problems with the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using a normal distribution with a mean of 0 and a standard deviation of 1.

They showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

The derivative of the Logistic funtion.

Looking at the logistic activation function, when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction.

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

The RELU function family.

The ReLU activation function suffers from a problem known as the dying ReLUs: during training, some neurons effectively die, meaning they stop outputting anything other than 0.

In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, nowadays practitioners use a variant of the ReLU function, such as the leaky ReLU discussed above or the so-called exponential linear unit (ELU) function

$$ELU(z) = \begin{cases} \alpha\left(\exp(z) - 1\right) & z < 0, \\ z & z \geq 0. \end{cases}$$

Which activation function should we use?

In general it seems that the ELU activation function is better than the leaky ReLU function (and its variants), which is better than ReLU. ReLU performs better than $\tanh$ which in turn performs better than the logistic function.

If runtime performance is an issue, then you may opt for the leaky ReLU function over the ELU function If you don't want to tweak yet another hyperparameter, you may just use the default α of 0.01 for the leaky ReLU, and 1 for ELU. If you have spare time and computing power, you can use cross-validation or bootstrap to evaluate other activation functions.

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants).

It is a bit faster to compute than other activation functions, and the gradient descent optimization does in general not get stuck.

For the output layer:

- For classification the softmax activation function is generally a good choice for classification tasks (when the classes are mutually exclusive).
- For regression tasks, you can simply use no activation function at all.

Batch Normalization.

Batch Normalization aims to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

The technique consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer. In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs over the current mini-batch, from this the name batch normalization.

Dropout.

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily dropped out, meaning it will be entirely ignored during this training step, but it may be active during the next step.

The hyperparameter p is called the dropout rate, and it is typically set to 50%. After training, the neurons are not dropped anymore. It is viewed as one of the most popular regularization techniques.

Gradient Clipping.

A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for recurrent neural networks).

This technique is called Gradient Clipping.

In general however, Batch Normalization is preferred.

A top-down perspective on Neural networks

The first thing we would like to do is divide the data into two or three parts. A training set, a validation or dev (development) set, and a test set. The test set is the data on which we want to make predictions. The dev set is a subset of the training data we use to check how well we are doing out-of-sample, after training the model on the training dataset. We use the validation error as a proxy for the test error in order to make tweaks to our model. It is crucial that we do not use any of the test data to train the algorithm. This is a cardinal sin in ML. Then:

- Estimate optimal error rate
- Minimize underfitting (bias) on training data set.
- Make sure you are not overfitting.

If the validation and test sets are drawn from the same distributions, then a good performance on the validation set should lead to similarly good performance on the test set.

However, sometimes the training data and test data differ in subtle ways because, for example, they are collected using slightly different methods, or because it is cheaper to collect data in one way versus another. In this case, there can be a mismatch between the training and test data. This can lead to the neural network overfitting these small differences between the test and training sets, and a poor performance on the test set despite having a good performance on the validation set. To rectify this, Andrew Ng suggests making two validation or dev sets, one constructed from the training data and one constructed from the test data. The difference between the performance of the algorithm on these two validation sets quantifies the train-test mismatch. This can serve as another important diagnostic when using DNNs for supervised learning.

Limitations of supervised learning with deep networks.

Like all statistical methods, supervised learning using neural networks has important limitations. This is especially important when one seeks to apply these methods, especially to physics problems. Like all tools, DNNs are not a universal solution. Often, the same or better performance on a task can be achieved by using a few hand-engineered features (or even a collection of random features).

Here we list some of the important limitations of supervised neural network based models.

- **Need labeled data**. All supervised learning methods, DNNs for supervised learning require labeled data. Often, labeled data is harder to acquire than unlabeled data (e.g. one must pay for human experts to label images).
- **Supervised neural networks are extremely data intensive.** DNNs are data hungry. They perform best when data is plentiful. This is doubly so for supervised methods where the data must also be labeled. The utility of DNNs is extremely limited if data is hard to acquire or the datasets are small (hundreds to a few thousand samples). In this case, the performance of other methods that utilize hand-engineered features can exceed that of DNNs.
- **Homogeneous data.** Almost all DNNs deal with homogeneous data of one type. It is very hard to design architectures that mix and match data types (i.e. some continuous variables, some discrete variables, some time series). In applications beyond images, video, and language, this is often what is required. In contrast, ensemble models like random forests or gradient-boosted trees have no difficulty handling mixed data types.
- **Many problems are not about prediction.** In natural science we are often interested in learning something about the underlying distribution that generates the data. In this case, it is often difficult to cast these ideas in a supervised learning setting. While the problems are related, it is possible to make good predictions with a *wrong* model. The model might or might not be useful for understanding the underlying science.

Some of these remarks are particular to DNNs, others are shared by all supervised learning methods. This motivates the use of unsupervised methods which in part circumvent these problems.

Part V
Quantum Computing