

Project 7 – Numerical Mathematical Software for Extreme-Scale Science and Interactive Supercomputing with JupyterLab

Due date: June 4, 2021, 12pm (midnight)

In this project, you will learn about popular scientific mathematical software libraries and tools widely used in computational science and engineering (CSE) practice. The project is also based on programming in Python, using innovative software in scientific computing for numerical optimization, and you will get familiar with the increasingly popular Jupyter notebook, which is an incredibly powerful tool for interactively developing and presenting CSE projects. Jupyter notebook integrates code and its output into a single document that combines visualizations, narrative text, mathematical equations, and other rich media. This intuitive workflow promotes iterative and rapid development, making notebooks an increasingly popular choice at the heart of contemporary CSE science, analysis, and increasingly science at large.

In this project, we will first solve the Poisson equation using state-of-the-art scientific mathematical software libraries and, second, set up Jupyter Notebooks on Euler and we will use it to solve a large-scale PDE-constrained optimization problem. These HPC software concepts will be practiced on an application of boundary control PDE problem with Dirichlet boundary conditions. You will implement a second-order finite-difference discretization scheme by which the elliptic control problem will be transcribed into a nonlinear programming problem. The full-space approach will be used, treating both control and state variables as optimization variables, resulting in a high-dimensional nonlinear programming problem.

You may do this project in groups of two or three students. In fact, we prefer that you do so.

1. An HPC Computational Software Stack for Extreme-Scale CSE

1.1. High-Level Languages for Numerical Computations

Using numerical methods and reference implementations from popular textbooks is often not sufficient for the development of serious software for large-scale applications in computational science. In general, state-of-the-art algorithms are not covered by these books and neither are the corresponding implementations suited to achieve a reasonable performance on multicores. Software packages like Mathematica [1], Maple [2], Matlab [3] provide high-level programming languages that can support the initial development of new numerical methods and the rapid prototype implementation. However, these standard packages are often not sufficient as HPC kernels, and neither are they intended for the realization of scientific software that has to deal with large-scale parallel applications on an HPC platform. A more modern example is Julia [4], which is a high-level, high-performance, dynamic programming language. While it is a general purpose language and can be used to write any application, some of its features are well suited for high-performance numerical analysis and computational science as well. It was designed with parallelism in mind, thus on top of classical multi-threading paradigm it supports asynchronous tasks and distributed computing such as MPI.

Due to the dedicated programming language of these examples, algorithms can be implemented based on a syntax which is close to mathematical notation. The expressive syntax not only allows rapid prototyping but also makes it easier to ensure the correctness of the resulting implementation. All of these high-level languages use external software libraries (e.g., BLAS [5], LAPACK [6], and other implementations, as well as libraries for numerical optimizations and PDE frameworks such as IPOPT [7] and PETSc [8]) to carry out numerical computations. Hence, they can be considered as high-level interfaces to these HPC mathematical libraries. On the one hand, Matlab and

Mathematica greatly benefit from the performance provided by external libraries. On the other hand, in general, not all features of these libraries can be exploited. For instance, these packages support only two matrix types, namely dense and sparse matrices. Other matrix types, e.g., for matrices with band or block structure, that are supported by LAPACK [6], are not available in Matlab.

In conclusion, high-level languages for numerical computations are very useful, easy to learn, and popular in computational science — they can represent a layer to HPC software libraries without exposing the application developer to the multicore complexity.

1.2. Scientific Computing Software Toolkits and Libraries

In the following text, various well-known high-performance computing mathematical software libraries that have influenced computational science and engineering are described. All parallel software toolkits can be used as an underlying layer in computational science to build successful parallel applications on multi- and many-core architectures.¹ Enabling parallel technologies in computational science must also ensure that a successful high-performance software library is fully and freely available for use throughout the scientific computing community.

Parallel Dense Matrix Software Libraries

The Basic Linear Algebra Subprogramm (BLAS) [5] specifies a set of computational routines for linear algebra operations such as vector and matrix operations. BLAS is a well-known example of a software layer which represents a de facto standard in the field of high-performance computing. All hardware vendors, such as Intel on multicores, or NVIDIA on manycores provide these BLAS routines as building blocks to ensure efficient scientific software portability.

The Linear Algebra Package, LAPACK, [6] is a software library for matrix operations such as solving systems of linear equations, least-squares problems, and eigenvalue or singular value problems. Almost all LAPACK routines make use of BLAS to perform efficient computations — LAPACK acts as an additional parallel software layer to the underlying BLAS. The motivation for the development of BLAS and LAPACK was to provide efficient and portable implementations for solving dense systems of linear equations and least-squares problems and it was initiated in the 1970s. Influenced by the changes in the hardware architectures (e.g., vector computers, cache-based processors to parallel multiprocessing architectures), it changed from the original specification to, finally, BLAS Level-3 [9] for matrix-matrix operations.

ScaLAPACK (Scalable LAPACK) is an extension of LAPACK that is targeted to multiprocessing computers with a distributed-memory hierarchy. Fundamental building blocks of ScaLAPACK are the parallel BLAS (PBLAS) and the Basic Linear Algebra Communication Subprograms (BLACS). PBLAS is a distributed-memory version of BLAS, and BLACS is a library for handling interprocessor communication. The design policy of ScaLAPACK was to have the ScaLAPACK routines similar to their LAPACK equivalents so that both libraries can be used as a computational software layer for application developers.

Parallel Sparse Matrix Software Toolkits

A number of applications give rise to large-scale sparse linear systems that are becoming exceedingly difficult to handle by standard numerical linear algebra techniques such as 3D wave propagation problems or PDE-constrained optimization problems. A number of highly efficient parallel software tools have been developed within the last fifteen years that can be used to tackle large-scale systems of linear equations or eigenvalue problems in parallel. These tools usually make heavy use of the BLAS and LAPACK routines, and all these parallel software tools now represent a de facto standard in the field of high-performance computing. The following is a list of the most widely use of the tools.

¹Some of these software packages, e.g., IPOPT, METIS, PETSc, Trilinos, or PARDISO have been selected as important tools for future HPC applications.

- The Multifrontal Massively Parallel Solver (MUMPS) [10] is being developed at CERFACS, ENSEEIHT-IRIT, and INRIA. It is a package for solving systems of linear equations of the form $Ax = b$, where A is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS is a direct method based on a multifrontal direct factorization. It exploits both parallelism arising from sparsity in the matrix A and from dense factorizations kernels. MUMPS offers several built-in ordering algorithms to some external parallel ordering packages such as METIS [11]. The parallel version of MUMPS requires MPI for message passing and makes use of the BLAS, BLACS, and ScaLAPACK libraries.
- SuperLU [12] is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high-performance machines. It has been developed at the Computer Science Department of the University of California, Berkeley and at the National Energy Research Scientific Computing Center (NERSC). The library is written in C and is callable from either C or C++. The library routines will perform an LU decomposition with partial pivoting and triangular system solves through forward and back substitution. The matrix columns may be preordered (before factorization) either through library or user-supplied routines. SuperLU comes in three different flavors: SuperLU for sequential machines, SuperLU_MT for shared memory parallel machines, and SuperLU_DIST for highly parallel distributed-memory architectures. The parallel version of SuperLU also requires MPI for the message-passing layer and makes use of the BLAS, BLACS, and ScaLAPACK libraries.
- PARDISO [13] is a thread-safe, high-performance, robust, memory efficient, easy-to-use software for solving large sparse symmetric and nonsymmetric linear systems of equations on multicore processing architectures. The solver is also part of the Intel Math Kernel Library with interfaces to Matlab, C, C++, and Python. It is available from <https://www.pardiso-project.org/>.
- The FEAST library package [14] represents a unified framework for solving various families of eigenvalue problems and achieving accuracy, robustness, high-performance and scalability on parallel architectures. Its originality lies with a new transformative numerical approach to the traditional eigenvalue algorithm design - the FEAST algorithm. The FEAST algorithm is a general purpose eigenvalue solver which takes its inspiration from the density-matrix representation and contour integration technique in quantum mechanics. FEAST can be used for solving both standard and generalized forms of the Hermitian or non-Hermitian problems (linear or nonlinear), and it belongs to the family of contour integration eigensolvers. FEAST's main computational task consists of a numerical quadrature computation that involves solving independent linear systems along a complex contour, each with multiple right-hand sides. FEAST is both a comprehensive library package, and an easy to use software. It includes flexible reverse communication interfaces and ready to use driver interfaces for dense, banded, and sparse systems.

Software Toolkits for Parallel Large-Scale Nonlinear Optimization

IPOPT [7] is a C++ open-source software package for large-scale nonlinear optimization based on a Newton-based interior point algorithm with filter line-search method. It has been designed for equality constrained problems and treats upper and lower bounds for the variables with an interior penalty formulation as a sequence of barrier problems for a decreasing sequence of barrier parameters converging to zero. It can be shown that under mild regularity assumptions the solutions of the barrier problems will converge to the solution of the original problem. The IPOPT algorithm has been shown to be globally and superlinearly convergent under weaker assumptions than other barrier methods. IPOPT has been applied to thousands of test problems and applications and has been adopted by a widespread user community. A key advantage is its ability to use second derivative information efficiently. IPOPT makes use of PARDISO or MUMPS, so it can also require MPI for message passing and makes use of the BLAS, BLACS, and ScaLAPACK libraries. Carl Laird and Andreas Wächter are the developers of IPOPT. Wächter and Laird were awarded the 2011 J. H. Wilkinson Prize for Numerical Software for this development.

Software Toolkits for Parallel Partitioning, Load Balancing, and Data-Management Services

Parallel partitioning, load balancing, and data-management services are important enabling technologies for parallel computing. Their goal is to distribute work evenly to processors while creating decompositions that have low communication costs for applications. This distribution must be done statically as a first step in most parallel computations. For adaptive computations, where processor workloads change as computations proceed, dynamic load balancing may also be needed. Two such toolkits are, e.g., the following:

- The Zoltan toolkit [15] is a C++ open-source software collection of such data management services for parallel unstructured, adaptive, and dynamic applications, available as open-source software from the Sandia National Laboratories. The design goal is to simplify the load balancing, data movement, and unstructured communication that arise in dynamic applications such as adaptive finite element methods, particle methods, and multiphysics simulations.
- The METIS software [16] is another alternative for parallel partitioning, load balancing, and data-management services. It can provide efficient parallel partitioning of graphs with sizes up to a billion vertices, distributed over a thousand processors. It is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. METIS includes routines that are especially suited for parallel AMR computations and large-scale numerical simulations. The algorithms implemented in METIS are based on the parallel multilevel k-way graph-partitioning, adaptive repartitioning, and parallel multi-constrained partitioning schemes.

Extreme-Scale Software Framework for Multiphysics Engineering and Scientific Problems

- The Trilinos Project [17, 18] is a C++ open-source software package that represents an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific problems. A unique design feature of Trilinos is its heavy focus on other similar packages within a computational software layer stack. Trilinos has been under development at Sandia National Laboratories since 2002, and it provides uniform access to accurate, robust, and efficient solvers and tools. It also facilitates more rapid development of new libraries by providing important core functionality and software engineering processes for developers. Trilinos unifies a diverse set of libraries that have been developed at Sandia, as well as tools developed by other researchers. Trilinos has also been the development basis for other fundamental numerical algorithms, e.g., time integration methods, eigenvalue solvers, and multilevel preconditioners.
- The Portable, Extensible Toolkit for Scientific Computation (PETSc, pronounced PET-see; the S is silent) [8], is a suite of data structures and routines developed by Argonne National Laboratory for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the Message Passing Interface (MPI) standard for all message-passing communication. PETSc is the world's most widely used parallel numerical software library for PDEs and sparse matrix computations. The PETSc Core Development Group won the SIAM/ACM Prize in Computational Science and Engineering for 2015. PETSc is intended for use in large-scale application projects; many ongoing computational science projects are built around the PETSc libraries. Its careful design allows advanced users to have detailed control over the solution process. PETSc includes a large suite of parallel linear and nonlinear equation solvers that are easily used in application codes written in C, C++, Fortran, and now Python. PETSc provides many of the mechanisms needed within parallel application code, such as simple parallel matrix and vector assembly routines that allow the overlap of communication and computation. In addition, PETSc includes support for parallel distributed arrays useful for finite-difference methods.

2. Scientific Mathematical HPC Software Frameworks — The Poisson Equation [35 points]

Many books on programming languages start with a "Hello, World!" program. Readers are curious to know how fundamental tasks are expressed in the language, and printing a text to the screen can be such a task. In the world of finite-difference methods for PDEs and HPC, one of the most fundamental tasks is to solve the Poisson equation. Our counterpart to the classical "Hello, World!" program therefore is to solve the Poisson equation

$$-\Delta y(x) = f(x) \quad \text{on } \Omega = [0, 1] \times [0, 1] \quad (1)$$

$$y(x) = 0 \quad \text{on } \partial\Omega \quad (2)$$

for $x \in \Omega = [0, 1] \times [0, 1]$. The Laplace operator Δ for a two-dimensional problem is defined as

$$\Delta \equiv \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}. \quad (3)$$

Here, $y = y(x_1, x_2)$ is the unknown function, $f = f(x)$ is a prescribed function, Δ is the Laplace operator, Ω is the spatial domain, and $\partial\Omega$ is the boundary of Ω . The Poisson problem, including both the PDE $-\Delta y(x) = f(x)$ and the boundary condition $y(x) = 0$ on $\partial\Omega$, is an example of a boundary-value problem, which must be precisely stated before it makes sense to start solving it with various parallel numerical software tools for CSE.

In two space dimensions with coordinates x_1 and x_2 , we can write out the Poisson equation as

$$-\left(\frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}\right)y(x_1, x_2) = f(x_1, x_2). \quad (4)$$

The unknown y is now a function of two variables, $y = y(x_1, x_2)$, defined over a two-dimensional domain $\Omega \in \mathbb{R}^2$. The Poisson equation arises in numerous physical contexts, including heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, and water waves. Solving a boundary-value problem such as the Poisson equation using a second-order finite difference discretization, and deploying scalable mathematical algorithms and software tools for reliable parallel simulation consists of the following steps:

- Identify the computational domain (Ω), the PDE modeling the system under study, its boundary conditions, and source terms $f(x_1, x_2)$.
- Reformulate the PDE using a second-order finite-difference discretization scheme.
- Write your own code or select an appropriate mathematical software for extreme-scale computing, such as, PETSc² or Trilinos³, used to solve the discretized problem.
- The program should define the computational domain, the boundary conditions, and source terms, using the corresponding abstractions of the particular problem.

2.1. Task to Solve [35 points in total]

- Please read the paper [19] entitled "Challenges and Opportunities in CSE Research," in particular, section 2 on "Challenges and Opportunities in CSE Research," "CSE and High-Performance Computing," and "CSE Software".

²PETSc is pronounced PET-see (the S is silent), is a suite of data structures and routines for the scalable parallel solution of scientific applications modeled by partial differential equations. It supports MPI, and GPUs, as well as hybrid MPI-GPU parallelism.

³Trilinos is a collection of open-source software libraries, called packages, intended to be used as building blocks for the development of scientific applications. For more details please see, e.g., <https://en.wikipedia.org/wiki/Trilinos>

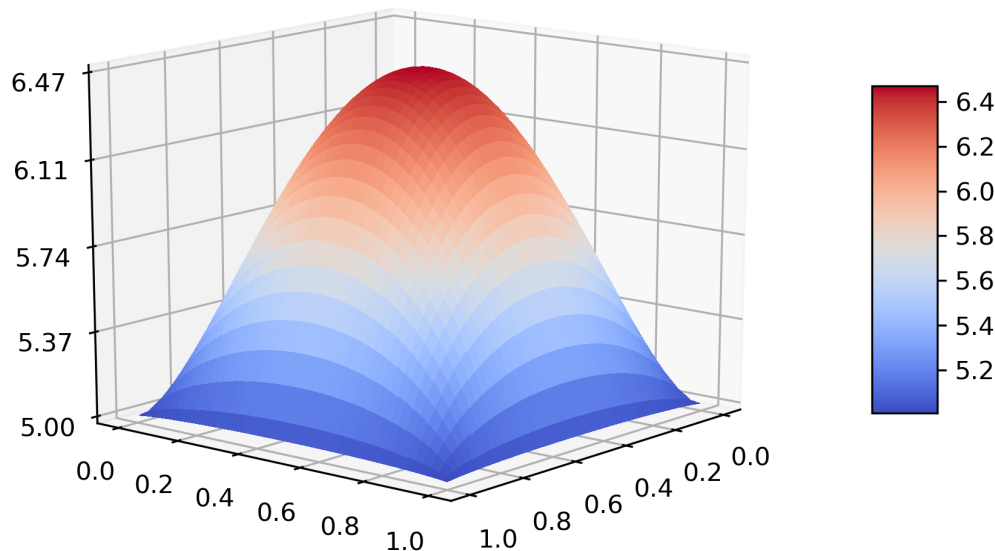


Figure 1: Solution of the Poisson equation (referred to as the forward problem) from Eqn. 5 and 6, using different RHS value in the boundary condition.

- Please view the two videos on the Moodle course webpage that are discussing the challenges posed by architecture and software environments of today's most powerful supercomputers, and even the greater complexity on the horizon from next-generation and exascale systems.
- We would like to ask you to build a code to solve the following Poisson equation:

$$-\Delta y(x) = 20 \quad \text{on } \Omega = [0, 1] \times [0, 1], \quad (5)$$

$$y(x) = 0 \quad \text{on } \partial\Omega, \quad (6)$$

using second-order finite differences in parallel and multiple cores on Euler. You may select one of the PDE mathematical software frameworks mentioned above. As an alternative, you might try to implement your own Python/Jupyter notebook discretization and solver (please refer to the second part of the project on interactive supercomputing).

- Summarize in two paragraphs your particular HPC implementation for the PDE (e.g. the motivation why you selected a particular framework, or the data structures you have used in your own solver), and report the parallel timing for various meshes below as listed in Table 1.
- Visualize the solution $y(x_1, x_2)$ of the Poisson problem for various sizes of N .

3. Interactive Supercomputing Using Jupyter Notebook [10 points]

The Jupyter notebook is an incredibly powerful tool for interactively developing and presenting CSE projects. The Jupyter notebook integrates code and its output into a single document that combines visualizations, narrative text,

Table 1: Wall-clock time (in seconds) and speedup (in brackets) using multiple cores on Euler for solving the Poisson PDE problem.

| Problem | N | Number of Euler cores | | | |
|---------|-------------------|-----------------------|---|----|----|
| | | 1 | 8 | 16 | 32 |
| Poisson | 500 ² | | | | |
| Poisson | 1000 ² | | | | |
| Poisson | 2000 ² | | | | |
| Poisson | 3000 ² | | | | |

mathematical equations, and other rich media. This intuitive workflow promotes iterative and rapid development, making notebooks an increasingly popular choice at the heart of contemporary CSE science, analysis, and increasingly science at large. In this project, we will set up Jupyter Notebooks on your local machine and on Euler and we will start using it to perform a computational science and engineering project related to large-scale PDE-constrained optimization.

ETH and CSCS support the use of Jupyter notebook for interactive supercomputing on Euler. It is the next-generation web-based user interface for Project Jupyter. It is an open-source web application that allows creation and sharing of documents containing live code, equations, visualizations and narrative text. This short introduction will help you to familiarize with the HPC libraries used in the scientific computing and set up the environment on the Euler cluster. Additionally, we will establish a remote connection to the Jupyter notebook from your web browser, giving you an interactive access to the code development.

Nowadays, the console-based approach of developing and running the code is being replaced by a qualitatively new paradigm based on interactive computing represented by the Jupyter notebook.⁴ It is providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:

- **A web application:** a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations, and their rich media output.
- **Notebook documents:** a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

A Jupyter notebook can be run locally for simple tasks, but in this project we will be running the Jupyter kernel on the Euler cluster and connect to it from your favorite web browser. This will give you all the benefits of a computing capability of the remote cluster and the interactive access on your laptop. Use the shell script `start_jupyter_nb.sh` located in folder `start_Jupyter` to start the notebook and connect to it:

```
[user@localhost]$ ./start_jupyter_nb.sh ETH_USERNAME NUM_CORES RUN_TIME MEM_PER_CORE
```

This shell script starts a Jupyter notebook in a batch job on Euler and connects your local browser with it. The parameters are the following:⁵

- **ETH_USERNAME:** Your username for the Euler cluster
- **NUM_CORES:** Number of cores to be used on the cluster (maximum: 36)
- **RUN_TIME:** Runtime limit for the Jupyter notebook on the cluster (HH:MM)

⁴<https://jupyter-notebook.readthedocs.io/en/stable/notebook.html>

⁵https://scicomp.ethz.ch/wiki/Jupyter_on_Euler_and_Leonhard_Open

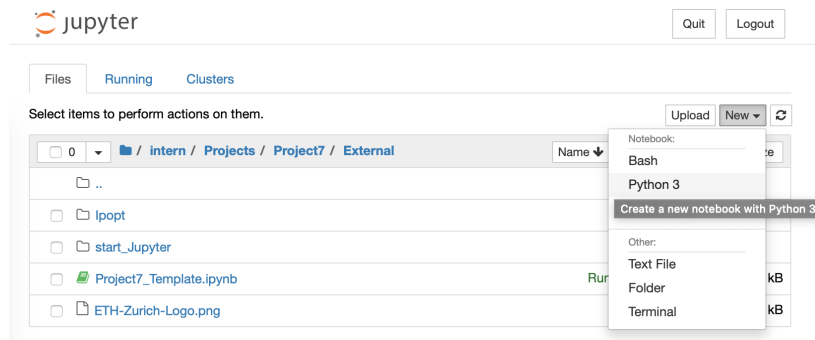


Figure 2: New Jupyter notebook.

- MEM_PER_CORE: Memory limit in MB per core

Before running the script, make sure of the following:

- You source the global definitions in your Euler's `$HOME/.bashrc` file

```
. /etc/bashrc
```

- You are using alias `euler` in your local `$HOME/.ssh/config` file, i.e., you are able to connect to Euler as

```
[user@localhost]$ ssh euler
```

Otherwise, you will have to change the variable `CHOSTNAME` in the script.

- You have put the IPOPT libraries in the following directory

```
[user@eu-login]$ ls $HOME/lecture/Project7/Ipopt/lib
```

and created the Python package repository (see section 4.1.1) at

```
[user@eu-login]$ ls $HOME/python
```

It is strongly recommended to use the paths as above, otherwise you will have to set the variables `LD_LIBRARY_PATH` and `PYTHONPATH` in the script accordingly.

Please note that when you finish working with the Jupyter notebook, you need to click on the "Quit" or "Logout" button in your browser. "Quit" will stop the batch job running on Euler, "Logout" will just log you out from the session but it will not stop the batch job (you need to login to the cluster, identify the job with `bjobs`, and then kill it with the `kill` command, using the jobid as a parameter). Afterwards you also need to clean up the SSH tunnel that is running in the background.

3.1. Jupyter Notebook — A Toy Example

In order to start working with the Jupyter notebook you need to create a new notebook. A new notebook may be created at any time, either from the dashboard (see Figure 2), or using the *File - New* menu option from within an active notebook. The new notebook is created within the same directory and will open in a new browser tab. It will also be reflected as a new entry in the notebook list on the dashboard. Be sure to create a Python 3 notebook, as shown in Figure 2. An opened notebook has exactly one interactive session connected to a kernel, which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel. The notebook consists of a sequence of cells. A cell is a multiline text input field, and its contents can be executed by

using Shift-Enter, or by clicking the "Play" button on the toolbar. The execution behavior of a cell is determined by the cell's type. There are three types of cells: code cells, markdown cells, and raw cells. Every cell starts off being a code cell, but its type can be changed by using a drop-down on the toolbar (which will be "Code", initially), or via keyboard shortcuts. The Jupyter notebook aims to support the latest versions of popular browsers; please use one of the supported browsers: Chrome, Safari, Firefox.

3.2. Task to Solve [10 points]

Your first task is to explore Python's numerical libraries (NumPy, SciPy) and verify that you are able to run the computational kernels using multiple threads. These libraries use optimized implementation of BLAS, taking advantage of cache memory and assembler implementation. But many architectures now have a BLAS that also takes advantage of a multicore machine. If your numpy/scipy is compiled using one of these, then `dot()` will be computed in parallel (if this is faster) without you doing anything. Similarly for other matrix operations, like inversion, singular value decomposition, determinant, and so on.⁶

- Pick some computational kernel that can benefit from multi-threading provided by Python's numerical libraries
- Verify that the library scales with the number of the CPU threads. Note that depending on the underlying BLAS library, you will have to set different variables controlling multi-threading (OMP_NUM_THREADS: OpenMP, OPENBLAS_NUM_THREADS: OpenBLAS, MKL_NUM_THREADS: MKL, VECLIB_MAXIMUM_THREADS: Accelerate, NUMEXPR_NUM_THREADS: Numexpr).
- Describe your Toy example (code, mathematical expression, etc.) and report the scaling of your Python example using a Jupyter notebook.

Do not forget to start the Jupyter notebook in Euler's allocation with sufficient computational resources!

4. Jupyter Notebook — Parallel PDE-Constrained Optimization [40 points]

4.1. Large-Scale Parallel Nonlinear Optimization with IPOPT

In this project we will use Jupyter notebook and IPOPT [7] to solve a PDE-constrained optimization problem. IPOPT is an open-source software package for large-scale non-linear optimization. It is designed to find (local) solutions of mathematical optimization problems of the form

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) \quad (7a)$$

$$\text{subject to} \quad g_L \leq g(x) \leq g_U, \quad (7b)$$

$$x_L \leq x \leq x_U, \quad (7c)$$

where the objective function f is a mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the equality and inequality constraints are defined as $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Note that equality constraints can be formulated in the above formulation by setting the corresponding components of g_L and g_U to the same value.

Such optimization problems arise in a number of important engineering, financial, scientific, and medical applications. IPOPT implements an interior-point line-search filter method [20], which is particularly suitable for large problems with up to millions of variables and constraints, assuming that the Jacobian matrix of constraint function is sparse. It is important to keep in mind that the algorithm is only trying to find a local minimizer of the problem; if the problem is nonconvex, many stationary points with different objective function values might exist, and it depends on the starting

⁶<https://scipy.github.io/old-wiki/pages/ParallelProgramming>

point and algorithmic choices which particular one the method converges to. In general, the computational effort during the optimization with IPOPT is typically concentrated in the solution of linear systems, or in the computation of the problem functions and their derivatives, depending on the particular application. Thus an efficient linear solver, optimized linear algebra kernels and code implementing the problem functions and their gradient and Hessian is of paramount importance. Consequently, in order to compile IPOPT, certain third party code is required (such as the linear algebra routines: e.g., optimized Intel MKL⁷ BLAS/LAPACK and a high-performance linear solver: e.g. PARDISO). Those codes are usually available under different conditions/licenses. The MKL library is already available on Euler cluster (after loading the appropriate module) and PARDISO licence is required and can be obtained for free for academic purposes at <https://www.pardiso-project.org>.

An interface between your application and IPOPT is usually established by a set of the callback routines you need to implement. These routines provide information about your problem, e.g. (7), such as the number of variables, functions for evaluating the objective and constraints, their first- and second-order derivatives, etc. This means you have to work out derivatives, including the sparsity structure of the problem, and provide gradient/Hessian of the objective function f and Jacobian/Hessians of the constraints g .

We will be using a Python interface⁸ in this project, meaning you will have to write a Python script following certain rules established by the interface, e.g., structure of the object and signature of the methods implementing your problem. Please refer to the official documentation⁹ for details.

4.1.1. Installation of IPOPT on Euler

First, you need to get the IPOPT libraries (C/C++ dynamic library). We are providing a precompiled IPOPT library (v3.13.0) for Euler using MKL BLAS/LAPACK (v2018.1) and PARDISO (v6.2) linear solver. Be sure to use the following modules on Euler when using the library in order to avoid runtime errors:

```
[user@eu-login]$ module load new
[user@eu-login]$ module load gcc/6.2.0
[user@eu-login]$ module load mkl/2018.1
```

Copy the directory with the IPOPT libraries to a location according to your preference (e.g. you can use the default location of the IPOPT library in the HPC Lab git repository `$HOME/lecture/Project7/Ipopt`), update the prefix variable in the `Ipopt/lib/pkgconfig/ipopt.pc` file to point to the library's location, and set the environment variables accordingly:

```
[user@eu-login]$ export LD_LIBRARY_PATH=$HOME/lecture/Project7/Ipopt/lib:$LD_LIBRARY_PATH
[user@eu-login]$ export PKG_CONFIG_PATH=$HOME/lecture/Project7/Ipopt/lib/pkgconfig/:
➔ $PKG_CONFIG_PATH
```

The IPOPT's Python interface is a Python package you will need to install on the Euler's file system. Since you don't have root privilege to install a system-wide package, you can create your personal Python package repository, for example in your `$HOME` directory:

```
[user@eu-login]$ mkdir $HOME/python
```

which has to contain a given subdirectory structure:

```
[user@eu-login]$ cd $HOME/python
[user@eu-login]$ mkdir -p lib64/python3.6/site-packages
```

In order to make Python aware of the local package repository, you should set the environment variable `PYTHONPATH` and add it to your `$HOME/.bashrc` file:

```
export PYTHONPATH=$HOME/python/lib64/python3.6/site-packages/
```

⁷<https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html>

⁸<https://pypi.org/project/ipopt/>

⁹<https://pythonhosted.org/ipopt/>

Note that we are using python3, make sure you are using the version shown below:

```
[user@eu-login]$ module load python/3.6.1
```

Finally, download the Python package `ipopt` from <https://pypi.org/project/ipopt/>, extract the archive, and install it (together with other automatically installed dependencies) to your local python repository created previously (`$HOME/python`):

```
[user@eu-login]$ cd ipopt-0.1.9  
[user@eu-login]$ python setup.py install --prefix ${HOME}/python
```

We also need to make sure Python is aware of the MKL libraries, we need to set the environment variable `LD_PRELOAD` as following (add this also to your `$HOME/.bashrc`):

```
[user@eu-login]$ export LD_PRELOAD=$MKLR00T/lib/intel64/libmkl_sequential.so:$MKLR00T/lib/  
↪ intel64/libmkl_core.so
```

If you haven't obtained your PARDISO license before please do so now from <https://www.pardiso-project.org/> by providing your user name on Euler and an email address. You need to paste your license key that you will receive by email to the file `$HOME/pardiso.lic`, for example:

```
[user@eu-login]$ echo "5C7FF596266D99895D8EFEB426F13EB3CF363F931DE83F433FC9A308" >> $HOME/  
↪ pardiso.lic
```

Now, you should be able to call the IPOPT library from Python, after importing the module `import ipopt`. You can test it by running one of the examples provided with the IPOPT's interface:

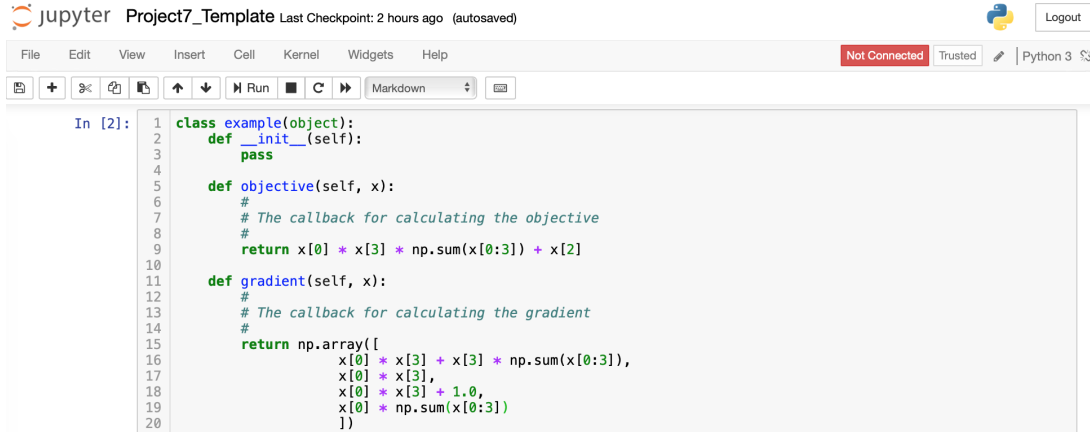
```
[user@eu-login]$ cd ipopt-0.1.9/test/  
[user@eu-login]$ python examplehs071.py
```

4.1.2. Local Installation of IPOPT

If you want to also use IPOPT locally on your laptop, you will need to compile the library for your architecture, supplying also a BLAS/LAPACK and a linear solver of your choice. If you would like to opt for this variant and try to build it yourself (also including the dependencies), you can study the installation guidelines for IPOPT¹⁰ and ask the TA for help if you run into any difficulties. The library that is provided to you at Euler has been compiled using the following configure options:

```
INSTALL_DIR=/cluster/home/<username>/Ipopt  
  
#intel MKL 32 bit sequential version  
MKL_OPTIONS="-m64 -I${MKLR00T}/include"  
MKL_LIBS="-L${MKLR00T}/lib/intel64 -Wl,--no-as-needed -lmkl_intel_lp64 -lmkl_sequential -  
↪ lmkl_core -lpthread -lm -ldl"  
  
ADD_CFLAGS="-fopenmp -fPIC -fno-common -fexceptions " \  
ADD_CXXFLAGS="-fopenmp -fPIC -fno-common -fexceptions " \  
ADD_FFLAGS="-fopenmp -fPIC -fexceptions " \  
../configure --disable-java F77=gfortran CC=gcc CXX=g++ --with-pic \  
--prefix="${INSTALL_DIR}" --enable-shared \  
--with-pardiso=" -L${HOME}/lib -lpardiso600-GNU720-X86-64 -lgfortran -lquadmath"
```

¹⁰<https://coin-or.github.io/Ipopt/INSTALL.html>



```

In [2]: 1 class example(object):
2         def __init__(self):
3             pass
4
5         def objective(self, x):
6             #
7             # The callback for calculating the objective
8             return x[0] * x[3] * np.sum(x[0:3]) + x[2]
9
10        def gradient(self, x):
11            #
12            # The callback for calculating the gradient
13            #
14            return np.array([
15                x[0] * x[3] + x[3] * np.sum(x[0:3]),
16                x[0] * x[3],
17                x[0] * x[3] + 1.0,
18                x[0] * np.sum(x[0:3])
19            ])
20

```

Figure 3: A Python class implementing the example problem (8).

4.2. JupyterLab — Example of Using IPOPT for a Small Optimization Problem

An example of how to formulate and solve a small problem using IPOPT is provided to you in a Jupyter notebook named `Project7_Template`. The problem on hand is to find $x = [x_0, x_1, x_2, x_3]$ such that

$$\underset{x}{\text{minimize}} \quad f(x) := \left(x_0 \cdot x_3 \cdot \sum_{i=0}^2 x_i \right) + x_2 \quad (8a)$$

$$\text{subject to} \quad x^T x = 40, \quad (8b)$$

$$\prod_{i=0}^3 x_i \geq 25, \quad (8c)$$

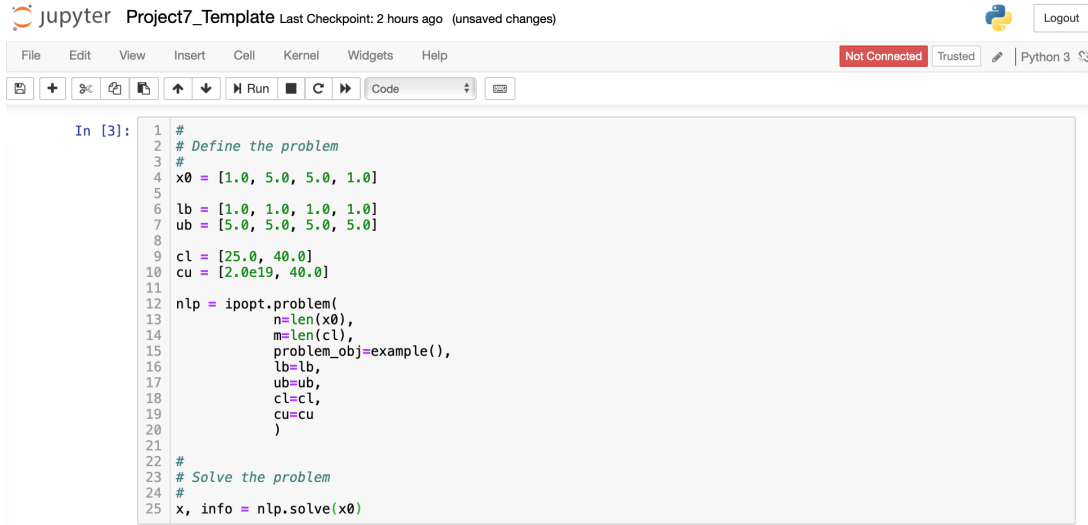
$$1 \leq x \leq 5. \quad (8d)$$

In order to solve the problem, we need to create a Python class which implements routines that evaluate the following:

- objective function $f(x)$,
- objective function gradient $\nabla_x f(x)$,
- constraints $g(x)$ (8b) and (8c),
- Jacobian of constraints $\nabla_x g(x)$,
- Hessian of Lagrangian $\nabla_x^2 f(x) + \lambda^T \nabla_x^2 g(x)$.

You can find an implementation of such a class in one of the notebook cells, as shown in Figure 3, illustrating definition of the class, implementation of the objective function, and its gradient. Note that name of the class methods, their signature, and type of the return value is given by the IPOPT's interface and need to be strictly respected by the user. Please see the documentation for more information.¹¹ The call of IPOPT library is shown in Figure 4. Apart from the class implementing the problem, you need to provide the initial point x_0 and the bounds for the variables lb, ub and constraints cl, cu . We also need to provide the size of the vector of optimization variables n and number of constraints m . The solution, i.e., the local minimizer given the problem and the starting point, is returned, together with additional information (e.g. the values of the Lagrange multipliers at the optimum). Next, you can continue in your notebook to analyze the solution, create the visualizations or do a performance analysis. It's up to you to explore and familiarize with the library so that you can solve yourself more interesting problem in the next part of this project.

¹¹<https://pythonhosted.org/ipopt/>



```

1 #
2 # Define the problem
3 #
4 x0 = [1.0, 5.0, 5.0, 1.0]
5
6 lb = [1.0, 1.0, 1.0, 1.0]
7 ub = [5.0, 5.0, 5.0, 5.0]
8
9 cl = [25.0, 40.0]
10 cu = [2.0e19, 40.0]
11
12 nlp = ipopt.problem(
13     n=len(x0),
14     m=len(cl),
15     problem_obj=example(),
16     lb=lb,
17     ub=ub,
18     cl=cl,
19     cu=cu
20 )
21
22 #
23 # Solve the problem
24 #
25 x, info = nlp.solve(x0)

```

Figure 4: IPOPT library call.

4.3. JupyterLab — Large-Scale Parallel PDE-Constrained Optimization with IPOPT

4.3.1. PDE-Constrained Optimization

One of the outstanding challenges of computational sciences and engineering is large-scale nonlinear parameter estimation governed by partial differential equations. These inverse problems are known as PDE-constrained optimization problems and are significantly more difficult to solve than PDE forward problems. PDE-constrained optimization refers to the optimization of systems governed by partial differential equations. These are known as *inverse problems* and typically characterize very large-scale optimizations, where the *forward problem* or *simulation problem* appears as one part within the optimizations. The forward problem usually characterizes applications in which the *control variables* of the PDE — initial conditions, domain sources, material coefficients, or boundary conditions (as we will use it in this project) — are known, and the *state variables* are determined by solving the PDE. The *inverse* or optimization problem reverses the process: here one tries to determine some control variables given performance goals in the form of an objective function and possibly inequality and equality constraints on the behavior of the system. Since the behavior of the system is modeled by a PDE, they appear as (usually equality) constraints in the optimization problem and it will be referred to as the *state equations*.

Let \mathbf{y} represent the state variables, \mathbf{u} the control variables, \mathcal{J} the objective function, \mathbf{c} the PDE state equations, and \mathbf{h} the inequality constraints. The PDE-constrained optimization problem can be stated as

$$\min_{\mathbf{y}, \mathbf{u}} \quad \mathcal{J}(\mathbf{y}, \mathbf{u}), \quad (9a)$$

$$\text{s.t.} \quad \mathbf{c}(\mathbf{y}, \mathbf{u}) = 0, \quad (9b)$$

$$\mathbf{h}(\mathbf{y}, \mathbf{u}) \geq 0. \quad (9c)$$

Many engineering and science problems — in such diverse areas as aerodynamics, atmospheric sciences, image registration, medicine, structural-fluid interactions, and chemical process industry — can be expressed in the form of a PDE-constrained problem. The common difficulty is that the PDE solution is just a subproblem associated with the optimization problem. Moreover, the inverse problem is often ill-posed despite the well-posedness of the forward problem, and the inverse problem can have numerous local solutions. For these reasons the optimization problem is often significantly more difficult to solve than the simulation problem.

The size, complexity, and infinite-dimensional nature of PDE-constrained optimization problems present significant challenges for general-purpose optimization algorithms, and parallel HPC implementations are typically necessary to

cope with the numerical challenges. When the infinite-dimensional conditions in (9) are discretized, one can represent the discretized PDE-constrained optimization problem by

$$\min_{y,u} \mathcal{J}(y, u), \quad (10a)$$

$$\text{s.t.} \quad c(y, u) = 0, \quad (10b)$$

where $y \in \mathbb{R}^n$, $u \in \mathbb{R}^m$ are the discrete state and control variables, $\mathcal{J} \in \mathbb{R}$ is the objective function, and $c \in \mathbb{R}^m$ are the discretized state equations.¹² Using adjoint variables $\lambda \in \mathbb{R}^n$, one can define the Lagrangian function by $\mathcal{L} := \mathcal{J}(y, u) + \lambda^T c(y, u)$. The first-order optimality conditions require that the gradient of the Lagrangian vanishes:

$$\begin{Bmatrix} \partial_y \mathcal{L} \\ \partial_u \mathcal{L} \\ \partial_\lambda \mathcal{L} \end{Bmatrix} = \begin{Bmatrix} g_y + J_y^T \lambda \\ g_u + J_u^T \lambda \\ c \end{Bmatrix} = 0. \quad (11)$$

Here, $g_y \in \mathbb{R}^n$ and $g_u \in \mathbb{R}^m$ are the gradients of \mathcal{J} with respect to the states and control variables respectively; $J_y \in \mathbb{R}^{n \times n}$ is the Jacobian of the state equations with respect to the state variables; and $J_u \in \mathbb{R}^{n \times m}$ is the Jacobian of the state equations with respect to the control variables. A Newton step on the optimality conditions gives the following linear system:

$$\begin{bmatrix} W_{yy} & W_{yu} & J_y^T \\ W_{uy} & W_{uu} & J_u^T \\ J_y & J_u & 0 \end{bmatrix} \begin{Bmatrix} p_y \\ p_u \\ \lambda_+ \end{Bmatrix} = \begin{Bmatrix} -g_y \\ -g_u \\ c \end{Bmatrix}. \quad (12)$$

Here, $W \in \mathbb{R}^{(n+m) \times (n+m)}$ is the Hessian of the Lagrangian \mathcal{L} , it involves second derivatives of both \mathcal{J} and c , and it is block-partioned according to state and control variables. $p_y \in \mathbb{R}^n$ is the search direction in the y variables, $p_u \in \mathbb{R}^m$ is the search direction in the u variables, and $\lambda \in \mathbb{R}^n$ is the update in the Lagrangian multipliers.

In this "HPC Lab for CSE" project, we are aiming to solve a boundary control problem. The goal is to determine a boundary control $u(x_1, x_2)$ on $\partial\Omega = [0, 1] \times [0, 1]$, that minimizes the functional with a given weight $\alpha \geq 0$

$$F(y, u) = \frac{1}{2} \int_{\Omega} (y(x) - y^d(x))^2 dx + \frac{\alpha}{2} \int_{\partial\Omega} (u(x) - u^d(x))^2 dx. \quad (13)$$

subject to the state equations

$$-\Delta y(x) + d(x, y(x)) = 0, \quad \text{for } x \in \Omega \quad (14)$$

$$y(x) = b(x, u(x)), \quad \text{for } x \in \partial\Omega, \quad (15)$$

and the inequality constraints on state and control

$$S(x, y(x)) \leq 0, \quad \text{for } x \in \Omega, \quad (16)$$

$$C(x, u(x)) \leq 0, \quad \text{for } x \in \partial\Omega. \quad (17)$$

4.3.2. Discretization

As in the forward problem (in our case the Poisson example from Eqn. ?? and ??), the discretization is restricted to the Laplacian operator Δ on a unit square $\bar{\Omega} = (0, 1) \times (0, 1)$ considering Dirichlet boundary conditions. The domain can be discretized using a standard mesh with $N \in \mathbb{N}_+$ points and the step size $h = 1/(N + 1)$, obtaining the mesh points

$$x_{i,j} = (ih, jh), \quad 0 \leq i, j \leq N + 1. \quad (18)$$

¹²Let us omit the nonequality constraints \mathbf{h} in what follows in order to simplify the notation for better readability.

The approximation of the state and control values, $y(x_{i,j})$ and $u(x_{i,j})$ are denoted as $y_{i,j}$ and $u_{i,j}$, respectively. The discretized cost function, considering the functional defined in (13) becomes

$$F^h(y, u) = \frac{h^2}{2} \sum_{\Omega} (y_{i,j} - y_{i,j}^d)^2 + \frac{\alpha h}{2} \sum_{\partial\Omega} (u_{i,j} - u_{i,j}^d)^2. \quad (19)$$

Similarly, the discretized state equation (14) (in our case the Poisson equation from ??) becomes a standard 5-point stencil operator

$$G_{i,j}^h = 4y_{i,j} - y_{i+1,j} - y_{i-1,j} - y_{i,j+1} - y_{i,j-1} + h^2 d_{i,j}. \quad (20)$$

Assuming the Dirichlet boundary conditions, the equation (15) reduces to identity $y_{i,j} = u_{i,j}$ on the boundary $\partial\Omega$. Finally, the state and control inequality take the form

$$S^h(x_{i,j}, y_{i,j}) \leq 0, \quad \text{for } (i, j) \in \Omega, \quad (21)$$

$$C^h(x_{i,j}, u_{i,j}) \leq 0, \quad \text{for } (i, j) \in \partial\Omega. \quad (22)$$

4.3.3. Optimization Problem

Having defined the objective function, constraints, and the variable bounds we can formulate the optimization problem in form (7) to be solved. This reads

$$\underset{y, u}{\text{minimize}} F^h(y, u) \quad (23a)$$

$$\text{subject to } g_L \leq G^h(y) \leq g_U, \quad (23b)$$

$$u_L \leq u \leq u_U, \quad (23c)$$

$$y_L \leq y \leq y_U. \quad (23d)$$

4.3.4. Task to Solve [40 points]

1. Use the Jupyter notebook to implement and solve the PDE-constrained optimization problem (24) using IPOPT. To get you started you can find a demo notebook `Project7_Template.ipynb` provided with this assignment, which demonstrates a basic usage of the IPOPT interface and some useful features of the Jupyter notebook. Consider the following parameters for the boundary control problem on $\Omega = [0, 1] \times [0, 1]$:

$$-\Delta y(x) = 20 \quad \text{on } \Omega \quad (24)$$

$$y(x) \leq 3.5 \quad \text{on } \Omega \quad (25)$$

$$y^d \equiv 3 + 5x_1(x_1 - 1)x_2(x_2 - 1) \quad \text{on } \Omega \quad (26)$$

$$y(x) = u(x) \quad \text{on } \partial\Omega \quad (27)$$

$$0 \leq u(x) \leq 10 \quad \text{on } \partial\Omega \quad (28)$$

$$u^d \equiv 0 \quad \text{on } \partial\Omega \quad (29)$$

$$\alpha = 0.01 \quad \text{on } \partial\Omega \quad (30)$$

$$(31)$$

using second-order finite differences. We would like to ask you to build a Jupyter notebook code and to solve the PDE-constrained optimization problem in parallel using multiple cores on Euler. You should use IPOPT to solve this PDE-constrained optimization problem in parallel. [25 points]

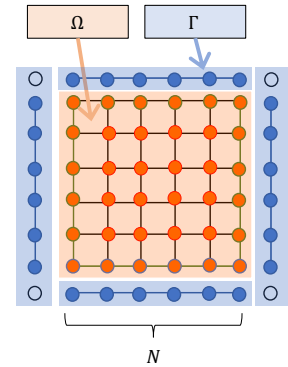


Figure 5: Discretization of the domain.

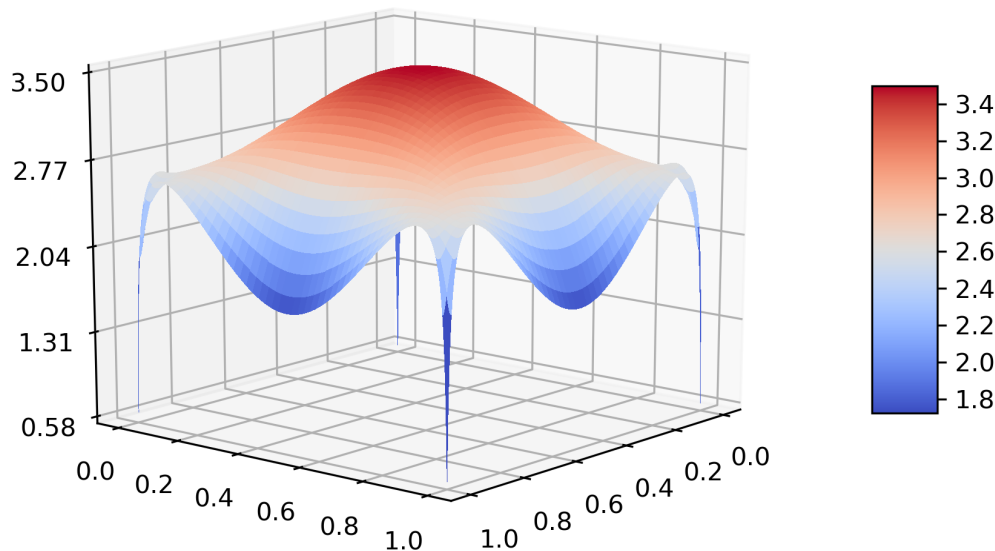


Figure 6: Solution of the PDE-constrained optimization equation (referred to as the inverse problem) from Eqn 24 to 31

2. Can you think of an appropriate initial point for the optimizer? [2.5 points]
3. Visualize the result $y(x)$ on Ω and on $\partial\Omega$ using Jupyter notebook. [2.5 points]
4. IPOPT is a package for solving general NLP problems, and as such evaluates the Hessian and Jacobian of constraints in every iteration. Is this necessary for problem (23)? Consult the IPOPT documentation to see if you can solve the problem more efficiently¹³. [5 points]
5. Analyze the performance of the optimization procedure for different discretization sizes of N , and observe scaling with different number of threads. Please report these numbers using Table 2. What are your findings? Add this analysis to your Jupyter notebook (be sure to start the Jupyter notebook with sufficient resources). [15 points]
6. The optimizer needs to control only the u variables, but we ask you to also optimize the state variables y by including them in the optimization vector. What would be the consequences of letting the optimizer control only the u variables and handling y explicitly given the controls u ? What is the consequence for the Hessian and Jacobian of the problem when solving the reduced space problem (in terms of dimensionality and sparsity)? [20 bonus points]

5. Task: Quality of the Report [15 Points]

Each project will have 100 points (out of which 15 points will be given to the general quality of the written report).

¹³https://coin-or.github.io/Ipopt/OPTIONS.html#OPT_NLP

Table 2: Wall-clock time (in seconds) and speedup (in brackets) using multiple cores on Euler for solving the PDE-constrained optimization problem.

| Problem | N | Number of Euler cores | | | |
|-----------------|----------|-----------------------|---|----|----|
| | | 1 | 8 | 16 | 32 |
| Inverse Poisson | 500^2 | | | | |
| Inverse Poisson | 1000^2 | | | | |
| Inverse Poisson | 2000^2 | | | | |
| Inverse Poisson | 3000^2 | | | | |

Additional Reading

- Andreas Wächter. Short Tutorial: Getting Started With Ipopt in 90 Minutes. [online]
- Helmut Mauer and Hans D. Mittelmann. Optimization Techniques for Solving Elliptic Control Problems with Control and State Constraints: Part 1. Boundary Control. [online]
- The Jupyter Notebook. [online]

Additional Notes and Submission Details

Collect all your source code, results and figures in a Jupyter notebook. Be sure it also contains your name and summary of your answers, results, and observations for all exercises. When you are satisfied with your notebook, upload it together with the PDF version (export the notebook as a PDF file) to Moodle. Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and observations for all exercises by writing an extended Latex report. Use the Latex template provided on the webpage and upload the Latex summary as a PDF to Moodle.

- Your submission should be a gzipped tar archive, formatted like `project_number_lastname_firstname.zip` or `project_number_lastname_firstname.tgz`. It should contain
 - all the source codes of your solutions;
 - your write-up with your name `project_number_lastname_firstname.pdf`.
- Submit your `.zip/.tgz` through Moodle.

References

- [1] S. Wolfram. *The Mathematica Book 5*. Cambridge University Press, 2008.
- [2] M.B. Monagan, K.M. Heal and Labahn K.O. Geddes, S.M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 12 Advanced Programming Guide*. Maplesoft, Waterloo, Ontario, Canada, 2008.
- [3] MATLAB. *Version Release 2019b*. The MathWorks Inc., Natick, Massachusetts, 2019.
- [4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

- [5] Jack J. Dongarra, J. Du Croz, Sven Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 1(14):1–17, March 1988.
- [6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 3 edition, 2000.
- [7] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.
- [8] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. <https://www.mcs.anl.gov/petsc>, 2019.
- [9] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.
- [10] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Softw.*, 45(1), February 2019.
- [11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
- [12] Xiaoye S. Li. An overview of Superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302 – 325, September 2005.
- [13] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23(1):158–179, 2006.
- [14] Brendan Gavin, Agnieszka Miedlar, and Eric Polizzi. Feast eigensolver for nonlinear eigenvalue problems. *Journal of Computational Science*, 27:107 – 117, 2018.
- [15] S. Bhowmick, E. Boman, K. Devine, A. Gebremedhin, B. Hendrickson, P. Hovland, T. Munson, and A. Pothen. Combinatorial Algorithms Enabling Computational Science: Tales from the Front. In *Journal of Physics: Conference Series*, pages 453–457, 46 (2006).
- [16] Dominique LaSalle, Md Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. Improving graph partitioning for modern graphs and architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, New York, NY, USA, 2015. Association for Computing Machinery.
- [17] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, et al. An overview of the Trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.
- [18] Katherine J. Evans, Mitchell T. Young, Benjamin S. Collins, Seth R. Johnson, Andrey V. Prokopenko, and Michael A. Heroux. Interfaces to trilinos in preparation for exascale developments.
- [19] Ulrich Rüde, Karen Willcox, Lois Curfman McInnes, and Hans De Sterck. Research and education in computational science and engineering. *SIAM Review*, 60(3):707–754, 2018.

- [20] Andreas Wächter and Lorenz T. Biegler. Line search filter methods for nonlinear programming: motivation and global convergence. *SIAM J. Optim.*, 16(1):1–31 (electronic), 2005.