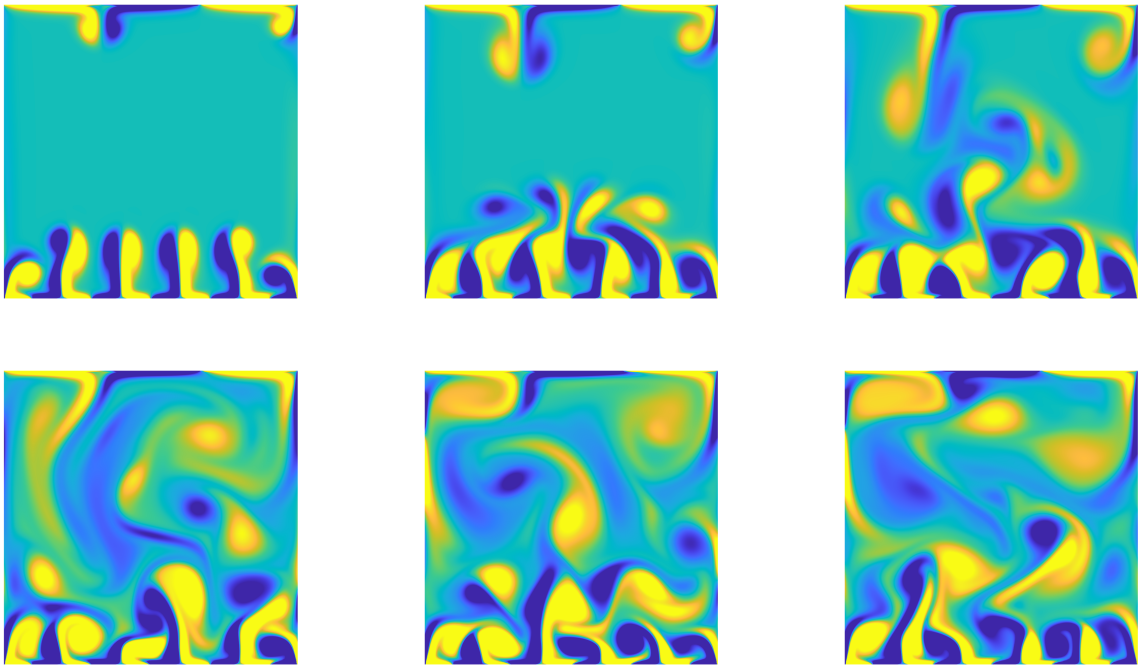
CMSE 823 Project - Spring 2021

Daniel Appelö*

Department of Computational Mathematics, Science & Engineering
Department of Mathematics,
Michigan State University, East Lansing MI 48824, USA.

January 3, 2021



THIS PROJECT IS NOT HARD BUT IT TAKES SOME TIME
GET STARTED EARLY!!!

*appeloda@msu.edu

Introduction

In this project you and your team will learn:

1. The details about one iterative or direct method for solving a sparse square system $\mathbf{Au} = \mathbf{f}$.
2. To describe and demonstrate the strength and weakness of the method.
3. To implement this method, first as a prototype in Matlab (or some other high level language) and then as a stand alone implementation in Fortran.
4. To incorporate your solver module as a part of a larger code simulating the incompressible Navier-Stokes equations in a box.
5. Make some beautiful figures and movies of fluid flows.

General Comments

- The code in the repository that you are about to clone is not large by most standards (for example at the time. when you defend your PhD thesis you will think it is small) but for a second semester graduate student it may appear large and largely incomprehensible. Don't worry about this, you don't have to understand the details as we are only focusing on providing different solvers for $\mathbf{Au} = \mathbf{f}$, this is the beautiful part with numerical linear algebra, it can be useful even if you don't fully understand the application. That said, I will provide some background to the partial differential equations solved and the discretization below and in class (you may also see something quite similar in CMSE 821 in the event that you are in that class).
- There are two reasons why I prefer to have "solve $\mathbf{Au} = \mathbf{f}$ with method X" be part of a real code rather than being a standalone assignment. First, this is what happens in real life (or at least the life of a scientist / engineer) for example when you do your first internship, or when your advisor wants you implement some new feature in some codebase. Second, for all of the possible solvers you can chose to study you will see a vast speedup compared to dense linear algebra. This will enable you to simulate more interesting flows that in turn will generate more beautiful figures. In my mind this is as good of a motivation to endure some Fortran coding as there ever were.
- Style-wise, the biggest problem of write-ups is often inappropriate or inelegant plots. Just because we live in an age where Matlab and Tecplot can generate 1000 plots easily, doesn't mean we should report all of them! In any kind of paper, only those plots that are essential for making some conclusion should be presented. All figure axes should be labeled. Whenever possible, all physical axes should be scaled the same. That is, in an $x - y$ contour plot, the aspect ratio should be unity to avoid making things look elliptical when they are really circular. Moreover, just because you can print out plot labels in microscopic-sized fonts, doesn't mean that you should! It can be difficult to set the font-size appropriately, since you may ultimately expand or shrink the plot when you paste it into the report. While time-consuming, you should develop a good strategy and set of tools to deal with this. It seems trivial, but your work will be more accessible and well-regarded if you learn to make excellent plots.
- When you have questions ask! This holds for both coding and conceptual questions. Use the office hours!

Ok enough let's get started!

Part 0. Quick-start

As soon as you find time (i.e. the first or second week of class) do the following. 1.) Start by cloning the repository `https://github.com/appelo/CMSE_823_2021` either on your workstation or on `hpcc.msu.edu` if you have an account there. 2.) Go to the `project/code` directory and compile and run the code. 3.) Plot the results using the Matlab script `pl.m`.

On my workstation the two first steps amount to:

```
git clone https://github.com/appelo/CMSE_823_2021.git
cd CMSE_823_2021
cd project
cd code
gfortran -o kalle_anka.x ins.f90 -llapack
./kalle_anka.x
```

On the high performance computing center (ICER) something like this will do the same:

```
ssh -XY appeloda@hpcc.msu.edu
// Repeat cloning the repo and change directories as above
ssh dev-amd20
module load intel
ifort -o kalle_anka.x ins.f90 -mkl
// -mkl is Intel's implementation of Lapack (and other math. stuff)
./kalle_anka.x
```

Note that it is probably best to get into the habit of submitting batch jobs. To do this you can use the slurm batch script which you can find in the same directory. To submit and monitor the jobs progress you can do:

```
sbatch insjob.sh
squeue -u your_user_name
```

If you are unsuccessful in the carrying out the above steps please let me know during office hours as soon as you can.

Part 1. Select a Method

The report for this project will be in form of a joint overleaf document where each group will fill in a section about a specific method. Once you are done with Part 0 let me know by e-mail that you are ready for Part 1 and I will add you to the editors of the overleaf document. Once added, you should fill in your name in the section title that belongs to the method you plan to study. No more than 3 persons per method. Don't make this choice too lightly, feel free to discuss it with me during office hours.

In addition to putting down your name discuss with your peers that have also selected the same method as you what are the properties of the method and fill out as much as you can in the table in the overleaf document. The abbreviations indicate if the method works for symmetric, non-symmetric, positive / negative definite or indefinite matrices. Is the method suitable for sparse matrices? Does the method work for singular matrices of the type discussed below as long as the right hand side is consistent? Don't get stuck on this, you can come back to the last category later.

Part 2. Baby Poisson

In the Navier-Stokes solver we will consider two matrices. The first one, that is used to find the velocities in the x and y directions at the next timestep, has the form

$$\kappa \mathbf{I} - \gamma \mathbf{L}_{\mathrm{D}}.$$

Here $\kappa > 0$ and $\gamma \mathbf{L}_{\mathrm{D}}$ is a scaled version of the matrix you get when discretizing Poisson's equation

$$\Delta u = f,$$

on a rectangular grid and with Dirichlet boundary conditions. The matrix $\gamma \mathbf{L}_{\mathrm{D}}$ is symmetric and negative thus the matrix $\kappa \mathbf{I} - \gamma \mathbf{L}_{\mathrm{D}}$ is spd.

The second matrix has the form

$$\tilde{\gamma} \mathbf{L}_{\mathrm{N}}.$$

It is the result of discretizing a Poisson equation

$$\Delta p = g, \ \ (x, y) \in \Omega, \tag{1}$$

with Neumann boundary conditions

$$\nabla p \cdot \mathbf{n} = h, \ \ (x, y) \in \partial\Omega. \tag{2}$$

Note that p is only unique up to a constant, i.e. if p is a solution to (1)-(2) then so is $p + C$ where C is any constant (recall that the derivative of a constant is zero). The discretization must inherit this property to be consistent[1] and it is therefore singular with a nullspace spanned by any constant vector.

The Dirichlet Model Problem in One Dimension

It is useful to consider a model problem before tackling Navier-Stokes. Thus, consider the Poisson equation in one dimension

$$\begin{aligned} u_{xx} &= f(x), \ \ x \in [0, 1], \\ u(0) &= h_0, \\ u(1) &= h_1. \end{aligned}$$

[1] Here this roughly means exact for constants and in general consistent means that the error in the numerical solution vanishes as the discretization size goes to zero.

Let $u_i \approx u(x_i)$ be a grid function approximating $u(x_i)$ at the gridpoints $x_i = ih_x,\ i = 0, 1, \ldots, n_x$ with a grid spacing $h_x = 1/n_x$. Using the standard second order accurate finite difference approximation we arrive at the following system of equations for the grid functions at the inner gridpoints

$$u_{i-1} - 2u_i + u_{i+1} = h_x^2 f(x_i),\ \ i = 1, \ldots, n_x - 1.$$

For $i = 1, n_x - 1$ we use the boundary conditions to replace the known values of the grid function on the two boundaries. We get

$$-2u_1 + u_2 = h_x^2 f(x_1) - h_0,$$

and

$$u_{n_x-2} - 2u_{n_x-1} = h_x^2 f(x_{n_x-1}) - h_1.$$

In this particular case we thus have that the matrix is

$$\gamma \mathbf{L}_{\mathrm{D}} = \begin{pmatrix} -2 & 1 & 0 & \ldots & & & \\ 1 & -2 & 1 & 0 & \ldots & & \\ 0 & 1 & -2 & 1 & 0 & \ldots & \\ \ldots & 0 & 1 & -2 & 1 & 0 & \ldots \\ & & \vdots & \vdots & \vdots & & \\ & & \ldots & 0 & 1 & -2 & 1 \\ & & & \ldots & 0 & 1 & -2 \end{pmatrix}$$

Multiplication of this matrix with a vector **u** is implemented in the subroutine `apply_1D_laplacian_D` in the `afuns` module in the file `baby_poisson.f90`

In the file `baby_poisson.f90` there are two options for solving the above system of equations, a direct method (LAPACK) or an iterative method (steepest descent). You can chose between the two by changing the variable `use_direct` in the `problem_setup` module. This module is also where you change the number of gridpoints and the tolerance of the iterative solver.

Task 1.

Make sure you can compile and run the `baby_poisson` code. For the Dirichlet problem make a plot or a table that has the number of gridpoints on the horizontal axis and run time on the vertical axis (you can for example use `cpu_time` to measure this). Do this experiment with the direct method, the steepest descent method with a tight tolerance, say 10^{-12} and with a tolerance that scales with the expected discretization error, i.e. $\mathrm{TOL} \sim h_x^2$. In a log-log plot you should be able to read off the complexity (the slopes of the curves) for the three different approaches. Do this task by yourself but do discuss the results with your peers.

Task 2.

Use the `iterative_solver_D` as an example and either modify it to replace steepest descent with your method or write a standalone module for your own method. Repeat the above experiment and report your findings of the complexity in the table in overleaf. If your method has a tolerance report both the complexity for the case where the tolerance is fixed and when it changes with the grid spacing. Do this task as a group.

If you are not used to compiled languages it may make sense to first implement your method in a high level language such as Matlab, Julia or Python. Once you have a debugged code you can translate it to Fortran. I find it particularly easy to translate Matlab to Fortran since the syntax is quite similar.

The Neumann Problem

The Dirichlet problem above results in a matrix that is symmetric and negative definite and as such should present no difficulties for any reasonable method. The Nuemann problem, however, does present a challenge for many methods and we will need to be a bit more careful.

Consider the problem

$$\begin{aligned} u_{xx} &= f(x), \ x \in [0,1], \\ u_x(0) &= g_0, \\ u_x(1) &= g_1. \end{aligned}$$

Again, let $u_i \approx u(x_i)$ be a grid function approximating $u(x_i)$ at the gridpoints $x_i = ih_x, \ i = 0, 1, \ldots, n_x$ with the same grid spacing $h_x = 1/n_x$. Again we get the equations in **all** the gridpoints (note that the linear system is larger than before)

$$u_{i-1} - 2u_i + u_{i+1} = h_x^2 f(x_i), \ i = 0, \ldots, n_x.$$

In this case we use the boundary conditions to eliminate the so called "ghost values" u_{-1} and u_{n_x+1}. Approximating the boundary conditions by a second order accurate finite difference we find

$$u_1 - u_{-1} = 2h_x g_0,$$

and

$$u_{n_x+1} - u_{n_x-1} = 2h_x g_1.$$

Inserting this in the approximation of the PDE

$$u_{-1} - 2u_0 + u_1 = h_x^2 f(x_0),$$

we obtain

$$-2u_0 + 2u_1 = h_x^2 f(x_0) + 2h_x g_0.$$

We could use this equation as it is but it is better for most methods to rescale the equation by 1/2 so that the matrix remains symmetric

$$-u_0 + u_1 = \frac{1}{2} h_x^2 f(x_0) + h_x g_0.$$

Repeating this procedure on the right boundary we get a matrix

$$\tilde{\gamma}\mathbf{L}_{\mathrm{N}} = \begin{pmatrix} -1 & 1 & 0 & \ldots & & & \\ 1 & -2 & 1 & 0 & \ldots & & \\ 0 & 1 & -2 & 1 & 0 & \ldots & \\ \ldots & 0 & 1 & -2 & 1 & 0 & \ldots \\ & & \vdots & \vdots & \vdots & & \\ & & \ldots & 0 & 1 & -2 & 1 \\ & & & \ldots & 0 & 1 & -1 \end{pmatrix}$$

Multiplication of this matrix with a vector **u** is implemented in the subroutine `apply_1D_laplacian_N` in the `afuns` module in the file `baby_poisson.f90`

The row sum of each row in this matrix is zero so multiplication of it onto any constant vector will return the zero vector (the definition of a null vector). Being a singular matrix we cannot directly apply Gauss elimination but we first have to pin down the constant. One way to do this is to introduce a new variable, say m, representing the mean of the solution and to insist that this mean is zero. We thus get one new equation for the mean being equal to zero

$$u_0 + u_1 + \cdots u_{n_x-1} + u_{n_x} = 0,$$

and (being good mathematicians) we may add $m = 0$ to all the other equations. Tagging on the new equation to the bottom of the matrix we can form the system

$$\begin{pmatrix} \tilde{\gamma}\mathbf{L}_{\mathrm{N}} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ m \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{f}} \\ 0 \end{pmatrix}. \tag{3}$$

This system i non-singular and can be factored by a standard dense **PLU** factorization / solve (e.g. `dgetrf` / `dgetrs`). Note however that the system is not definite so some iterative methods will not necessarily be applicable.

Task 3.

Incorporate the above technique in `baby_poisson.f90` so that you can solve the Neumann problem. Note that the average of the exact solution used in the program may not be zero so you might have to adjust the error computation to account for this.

Task 4.

The steepest descent method for $\mathbf{Ax} = \mathbf{b}$ starts with initializing $\mathbf{x}_0 = 0$, $\mathbf{r}_0 = \mathbf{b}$ followed by the iteration

$$\begin{aligned} \alpha &= \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_i^T \mathbf{A}\mathbf{x}_i} \\ \mathbf{x}_{i+1} &= \mathbf{x}_i + \alpha \mathbf{r}_i \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - \alpha \mathbf{A}\mathbf{x}_i \end{aligned}$$

This algorithm can be applied directly to the Neumann problem (there is no problem to multiply with a singular matrix) but what does it converge to?

It turns out that the steepest descent method converges to the minimum length solution so if you use the exact solution $u(x) = e^{-x}$ in `baby_poisson.f90` you should actually compute the error against

$$\tilde{u} = e^{-x} + c,$$

where

$$c = \arg\min \int_0^1 (e^{-x} + c)^2 dx.$$

Add the steepest descent method to the `iterative_solver_N` module and repeat the steps in Task 2.

Task 5.

How will you solve the Neumann problem using your method? For some of the methods you can use the extended system (on matrix-free form) directly as they can handle indefinite systems. For other methods you may be able to use the method almost unchanged as long as you make sure that the iterates (and maybe other vectors) stay consistent with the right hand side. Then again, for the FFT which uses the exact form of the eigenvectors of the you will have to explicitly account for the new eigenvectors.

At this point you should be have a grasp of the added difficulty with solving the Neumann problem if your method is designed for positive definite systems. If it is read up on how to best handle this in your method. Some good references are [3, 4, 6, 1, 2, 7]. IN particular for the FFT method [3] is good and for multigrid the standard introductory reference is [2].

Part 3. Incompressible Navier-Stokes

The incompressible Navier-Stokes equations arise from the conservation of momentum and mass followed by boundary and initial conditions:

$$\begin{aligned} \partial\mathbf{u}/\partial t + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \nu\Delta\mathbf{u} + \mathbf{F}, &\qquad x \in \Omega,\ t > 0, \\ \nabla\cdot\mathbf{u} = 0, &\qquad x \in \bar{\Omega},\ t > 0 \\ B(\mathbf{u}, p) = \mathbf{g}, &\qquad x \in \partial\Omega,\ t > 0, \\ \mathbf{u}(\mathbf{x}, 0) = \mathbf{f}(\mathbf{x}), &\qquad x \in \Omega,\ t = 0. \end{aligned}$$

Here $\mathbf{u} = (u, v)$ is the velocity, p is the kinematic pressure (scaled with the reciprocal of the constant density), ν is the kinematic viscosity and $\mathbf{F}$ is a force per unit volume (which we take to be zero). We consider a rectangular domain in two dimensions. We also assume that incompressibility is satisfied for the initial data.

The equations that we will solve is a discretization of the equations in the so called "velocity-pressure" formulation which is obtained by taking the divergence of the momentum equation.

$$\begin{aligned} \partial\mathbf{u}/\partial t + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \nu\Delta\mathbf{u} + F, &\qquad x \in \Omega,\ t > 0, \\ \Delta p + J(\nabla\mathbf{u}) - \alpha\nabla\cdot\mathbf{u} = \nabla\cdot\mathbf{F}, &\qquad x \in \Omega,\ t > 0, \\ B(\mathbf{u}, p) = \mathbf{g}, &\qquad x \in \partial\Omega,\ t > 0, \\ \nabla\cdot\mathbf{u} = 0, &\qquad x \in \partial\Omega,\ t > 0 \\ \mathbf{u}(\mathbf{x}, 0) = \mathbf{f}(\mathbf{x}), &\qquad x \in \Omega,\ t = 0. \end{aligned}$$

Note that we have added added "zero", $\alpha\nabla\cdot\mathbf{u}$, which will clean out any numerically created divergence. In two dimensions the term $J(\nabla\mathbf{u})$ is $J(\nabla\mathbf{u}) = u_x^2 + v_y^2 + 2u_y v_x$.

You can already see where the Poisson problem for the pressure comes from but can you see how we will get two more Poisson-like problems?

Finite Difference Discretization

To discretize the above equations we use the finite difference discretization by Henshaw and Petersson[2] [5]. The discretization is already implemented in the file `ins.f90` and as mentioned before

[2] `http://www.overtureframework.org/publications/henshawPetersson2001.pdf`

you don't really have to know the details. The systems of equations we will solve are essentially the two dimensional versions of the equations from Part 2 but let us outline briefly the discretization.

In the equations

$$\begin{aligned} \partial \mathbf{u}/\partial t + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \nu\Delta\mathbf{u} + F, \qquad & x \in \Omega,\ t > 0, \\ \Delta p + J(\nabla\mathbf{u}) - \alpha\nabla\cdot\mathbf{u} = \nabla\cdot\mathbf{F}, \qquad & x \in \Omega,\ t > 0, \end{aligned}$$

spatial derivatives are replaced with centered finite differences on the same rectilinear grid for both velocities and pressure.

A sticky point when using this type of collocated grid is how to specify (numerical) boundary conditions for the pressure (the continuous pressure does not have a natural boundary condition). Suppose that the boundary conditions for the velocities are of Dirichlet type $\mathbf{u} = \mathbf{g}$, so called no-slip boundary conditions. Then the momentum equation becomes

$$\partial \mathbf{u}/\partial t + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \nu\Delta\mathbf{u}, \ \ x \in \partial\Omega,\ t > 0.$$

Now, since time is tangential to the boundary we may differentiate the boundary conditions in time

$$\mathbf{u}_t = \mathbf{g}_t$$

to find

$$\partial \mathbf{g}/\partial t + (\mathbf{g}\cdot\nabla)\mathbf{u} + \nabla p = \nu\Delta\mathbf{u}, \ \ x \in \partial\Omega,\ t > 0.$$

Extracting the normal component of the pressure we find

$$\partial p/\partial n = \mathbf{n}\cdot(-\partial \mathbf{g}/\partial t + (\mathbf{g}\cdot\nabla)\mathbf{u} + \nu\Delta\mathbf{u}), \ \ x \in \partial\Omega,\ t > 0.$$

This is not a new boundary condition, no new information was added since we did not exploit $\nabla\cdot\mathbf{u} = 0$. This is the most "natural" form of the Neuman boundary condition for pressure but it is notoriously difficult to discretize in a stable way.

A better boundary condition can be obtained by first recalling that vector calculus tells us that

$$\Delta\mathbf{u} = \nabla(\nabla\cdot\mathbf{u}) - \nabla\times\nabla\times\mathbf{u},$$

which together with the "new information" that $\nabla\cdot\mathbf{u} = 0$ gives the so called curl-curl boundary condition

$$\partial p/\partial n = \mathbf{n}\cdot(-\partial \mathbf{g}/\partial t + (\mathbf{g}\cdot\nabla)\mathbf{u} - \nu\nabla\times\nabla\times\mathbf{u}), \ \ x \in \partial\Omega,\ t > 0.$$

This boundary condition uses new information and enforces incompressibility on the boundary and is stable and does not cause a time step restriction. This is what is implemented in the code you will use.

Timestepping

In the code `ins.f90` we treat the viscous term in the momentum equation, $\mathbf{L}_I$, with the trapezoidal rule and the (nonlinear) explicit term, $\mathbf{L}_E$, using Adams-Bashforth. We set the forcing to zero. This leads to a timestepping scheme that has the form

$$\frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t} = \frac{3}{2}L_E\mathbf{U}^n - \frac{1}{2}L_E\mathbf{U}^{n-1} + \frac{1}{2}(L_I\mathbf{U}^{n+1} + L_I\mathbf{U}^n).$$

Rearranging the equations we find

$$(\frac{\mathbf{I}}{\Delta t} - \frac{1}{2}L_I)\mathbf{U}^{n+1} = \frac{\mathbf{U}^n}{\Delta t} + \frac{3}{2}L_E\mathbf{U}^n - \frac{1}{2}L_E\mathbf{U}^{n-1} + \frac{1}{2}L_I\mathbf{U}^n.$$

Denote by $\mathbf{L}_\mathrm{D}$ the matrix corresponding to a centered finite difference discretization of $u_{xx} + u_{yy}$ with Dirichlet boundary conditions and using uniform grid spacing h_x and h_y and with ordering of the degrees of freedom being fast in x and arranged with u and v being the slowest changing index. Then we will have to solve the two systems

$$(\frac{\mathbf{I}}{\Delta t} - \frac{\nu}{2}\mathbf{L}_\mathrm{D})\mathbf{W}^{n+1} = \hat{\mathbf{h}}.$$

Where $\mathbf{W}$ is the grid function for either u or v and $\hat{\mathbf{h}}$ incorporates the right hand side from above and boundary conditions. When using a direct dense method we extract this whole matrix as follow (see `ins.f90`)

```
do i = 1,(nx-1)*(ny-1)
   uvec(i) = 1.0_dp
   call apply_velocity_laplacian(vvec,uvec,nx,ny,hx,hy)
   uvec(i) = 0.0_dp
   LapUV(:,i) = -0.5_dp*nu*vvec
   LapUV(i,i) = LapUV(i,i) + 1.0_dp/k
end do
```

This matrix is then factored by the call `CALL DGETRF(sys_size_uv,sys_size_uv,LapUV,sys_size_uv,ipiv_uv,INFO)` before the timestepping starts. Once the timestepping starts the call `CALL DGETRS('N',sys_size_uv,1,Lapuv,sys_size_uv,IPIV_uv,uvec,sys_size_uv,INFO)` performs the actual solve that gives new values for the x and y-velocities.

The matrix corresponding to the pressure discretization is extracted in a similar fashion and stored in `LapP` and the extended system is stored in `LapPBig`. Also for the pressure there is a factorization call to `DGETRF` before the timestepping loop and solves `DGETRS` inside the timestepping loop.

Task 6.

Now find the comment `! Do necessary precomputations / allocations here for your code` and the comment `! !!! YOUR CODE GOES HERE !!!!` in the code and make a plan for how you will add your solvers (using your method) from Part 2. Start with the Dirichlet problem and modify your implementation so that it works for the matrix $(\frac{\mathbf{I}}{\Delta t} - \frac{\nu}{2}\mathbf{L}_\mathrm{D})$. If your method requires that the matrix is on the "no h_x, h_y" form (like the FFT method) you may just rescale the righthand side.

Make sure that you get the same (up to the tolerance of the iterative method) solution for your method and for the direct dense method for the Dirichlet (velocity) problem before you proceed to the Neumann (pressure) problem.

Now that you (finally) have a fast (or at least different) new Navier-Stokes solver carry out some timing studies for a few different grid resolutions and start writing up your part of the report on overleaf.

Note that:

1. you should keep the ratio of the grid spacing in space and time roughly the same as in the original code. A rule of thumb is that

$$\frac{\max |\mathbf{u}| \Delta t}{\min h_x, h_y} \approx \text{constant},$$

   for the method to be stable.

2. Also note that when ν is small you need to resolve the small eddies produced by the flow by decreasing h_x, h_y.

Part 4. Colorful Fluid Dynamics

The original solver is setup to simulate flow within a rectangular box with aspect ratio L_x/L_y (length in x times height in y). The top wall can move with a constant speed (to the right), giving the classic "lid driven cavity" flow. If set to zero, then the flow would need to be somehow forced–either through a nonzero initial condition or through the addition of body force on the right-hand-side of the equations in order to produce a non-trival flow.

Task 7

Do something fun with your solver. Here are some suggestions, some of these are quick, some may take considerably longer! Feel free to ignore all these suggestions and come up with something on your own!

1. Initialize one or more Taylor vortices in your box and watch them do interesting things.

2. Increase the Reynolds number for the lid driven cavity. Is there a Reynolds number beyond which no steady flow can be found?

3. Add a body force to the right-hand-side of the momentum equations to create some interesting flow.

4. *Two-dimensional turbulence*. Initialize your domain with Fourier coefficients of equal magnitude, but randomly chosen phases. Excite only a portion of wavenumbers (I suggest exciting up to about $|\alpha|, |\beta| < N/4$). Allow the disturbances to evolve in time and observe especially the long-time behavior. Time permitting, try initializing your run with different random-number seeds.

5. *Dynamics of a co-rotating vortex pair.* Initialize two Taylor vortices with core size c, equal maximum velocity (unity), separated by a distance d. Note that c and d will be relative to your box size, L. Vary the core-size and separation distance and see how close they need to be before they will merge to form a single (wobbly) vortex. Note that because your solution is bounded, there will be an influence from the boundary conditions so make your box as big as you can. The impact can be assessed by making c and d small relative to the box size.

6. Solve an additional equation using by extending the finite-volume scheme for a *passive scalar* such as concentration of a non-reacting mixture. That is, solve the two-dimensional advection diffusion equation. Use it to investigate mixing in the lid driven cavity (or another flow).

References

[1] A. Björck. *Numerical methods in matrix computations*, volume 59. Springer, 2015.

[2] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.

[3] James W Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[4] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2nd edition, 1989.

[5] William D Henshaw and N Anders Petersson. A split-step scheme for the incompressible Navier-Stokes equations. In *Numerical simulations of incompressible flows*, pages 108–125. World Scientific, 2003.

[6] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[7] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid*. Elsevier, 2000.