A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains `mit18086_navierstokes.m`

Benjamin Seibold
Applied Mathematics
Massachusetts Institute of Technology
`www-math.mit.edu/~seibold`
`seibold@math.mit.edu`

March 31, 2008

1 Introduction

On the following pages you find a documentation for the Matlab program `mit18086_navierstokes.m`, as it is used for *Course 18.086: Computational Science and Engineering II* at the *Massachusetts Institute of Technology*. The code can be downloaded from the *Computational Science and Engineering* web page `http://www-math.mit.edu/cse`. It is also linked on the course web page `http://www-math.mit.edu/18086`.

This code shall be used for teaching and learning about incompressible, viscous flows. It is an example of a simple numerical method for solving the Navier-Stokes equations. It contains fundamental components, such as discretization on a staggered grid, an implicit viscosity step, a projection step, as well as the visualization of the solution over time. The main priorities of the code are

1. **Simplicity and compactness:** The whole code is one single Matlab file of about 100 lines.
2. **Flexibility:** The code does not use spectral methods, thus can be modified to more complex domains, boundary conditions, and flow laws.
3. **Visualization:** The evolution of the flow field is visualized while the simulation runs.
4. **Computational speed:** Full vectorization and pre-solving the arising linear systems in an initialization step results in fast time stepping.

The code provides:

- variable box sizes, grid resolution, time step, and Reynolds numbers
- implicit time stepping for the fluid viscosity

- visualization of pressure field, velocity field and streamlines
- automatic transition from central to donor-cell discretization for the nonlinear advection part
- fast computation of the time-dependent solution for small to moderate mesh sizes

The code does not provide:

- 3d
- unstructured meshes or complex geometries
- time-dependent geometries
- adaptivity, such as local mesh refinement, time step control, etc.
- non-constant viscosity or density
- higher order time stepping
- turbulence models
- time and memory efficiency for large computations

The code qualifies as a basis for modifications and extensions. The following extensions have already been applied to the code by MIT students and other users:

- external forces
- inflow and outflow boundaries
- addition of a drag region
- geometry change to a backwards facing step
- time dependent geometries
- coupling with an advection-diffusion equation

2 Incompressible Navier-Stokes Equations

We consider the incompressible Navier-Stokes equations in two space dimensions

$$u_t + p_x = -(u^2)_x - (uv)_y + \frac{1}{Re}(u_{xx} + u_{yy}) \tag{1}$$

$$v_t + p_y = -(uv)_x - (v^2)_y + \frac{1}{Re}(v_{xx} + v_{yy}) \tag{2}$$

$$u_x + v_y = 0 \tag{3}$$

on a rectangular domain $\Omega = [0, l_x] \times [0, l_y]$. The four domain boundaries are denoted ***North***, ***South***, ***West***, and ***East***. The domain is fixed in time, and we consider no-slip boundary conditions on each wall, i.e.

$$u(x, l_y) = u_N(x) \qquad v(x, l_y) = 0 \tag{4}$$
$$u(x, 0) = u_S(x) \qquad v(x, 0) = 0 \tag{5}$$
$$u(0, y) = 0 \qquad v(0, y) = v_W(y) \tag{6}$$
$$u(l_x, y) = 0 \qquad v(l_x, y) = v_E(y) \tag{7}$$

A derivation of the Navier-Stokes equations can be found in [2]. The momentum equations (1) and (2) describe the time evolution of the velocity field (u, v) under inertial and viscous forces. The pressure p is a Lagrange multiplier to satisfy the incompressibility condition (3). Note that the momentum equations are already put into a numerics-friendly form. The nonlinear terms on the right hand side equal

$$(u^2)_x + (uv)_y = uu_x + vu_y \tag{8}$$
$$(uv)_x + (v^2)_y = uv_x + vv_y \tag{9}$$

which follows by the chain rule and equation (3). The above right hand side is often written in vector form as $(\mathbf{u} \cdot \nabla)\mathbf{u}$. We choose to numerically discretize the form on the left hand side, because it is closer to a conservation form.

The incompressibility condition is not a time evolution equation, but an algebraic condition. We incorporate this condition by using a projection approach [1]: Evolve the momentum equations neglecting the pressure, then project onto the subspace of divergence-free velocity fields.

3 Visualization

Every fixed number of time steps the current pressure and velocity field are visualized. The pressure field is shown as a color plot with some contour lines. On top of this, the normalized velocity field is shown as a quiver plot, i.e. little arrows indicate the direction of the flow. Since in a lid driven cavity the flow rate varies significantly over different areas, outputting the normalized velocity field is a common procedure. Additionally, stream lines are shown. Those are paths particles would take in the flow if the velocity field was frozen at the current instant in time. Since the velocity field is divergence-free, the streamlines are closed. The stream lines are contour lines of the stream function q. It is a function whose orthogonal gradient is the velocity field

$$(\nabla q)^\perp = \mathbf{u} \quad \Longleftrightarrow \quad -q_y = u \text{ and } q_x = v \tag{10}$$

Applying the 2d-curl to this equation yields

$$-\Delta q = \nabla \times (\nabla q)^\perp = \nabla \times \mathbf{u} \quad \Longleftrightarrow \quad -\Delta q = -q_{yy} - q_{xx} = u_y - v_x \tag{11}$$

The stream function exists since the compatibility condition $\nabla \cdot \mathbf{u} = u_x + v_y = 0$ is satisfied.

4 Numerical Solution Approach

The general approach of the code is described in Section 6.7 in the book *Computational Science and Engineering* [4].

While u, v, p and q are the solutions to the Navier-Stokes equations, we denote the numerical approximations by capital letters. Assume we have the velocity field U^n and V^n at the nth time step (time t), and condition (3) is satisfied. We find the solution at the $(n+1)^{st}$ time step (time $t+\Delta t$) by the following three step approach:

1. **Treat nonlinear terms**
   The nonlinear terms are treated explicitly. This circumvents the solution of a nonlinear system, but introduces a CFL condition which limits the time step by a constant times the spacial resolution.
   $$\frac{U^* - U^n}{\Delta t} = -((U^n)^2)_x - (U^n V^n)_y \tag{12}$$
   $$\frac{V^* - V^n}{\Delta t} = -(U^n V^n)_x - ((V^n)^2)_y \tag{13}$$
   In Section 5 we will detail how to discretize the nonlinear terms.

2. **Implicit viscosity**
   The viscosity terms are treated implicitly. If they were treated explicitly, we would have a time step restriction proportional to the spacial discretization squared. We have no such limitation for the implicit treatment. The price to pay is two linear systems to be solved in each time step.
   $$\frac{U^{**} - U^*}{\Delta t} = \frac{1}{Re}(U^{**}_{xx} + U^{**}_{yy}) \tag{14}$$
   $$\frac{V^{**} - V^*}{\Delta t} = \frac{1}{Re}(V^{**}_{xx} + V^{**}_{yy}) \tag{15}$$

3. **Pressure correction**
   We correct the intermediate velocity field (U^{**}, V^{**}) by the gradient of a pressure P^{n+1} to enforce incompressibility.
   $$\frac{U^{n+1} - U^{**}}{\Delta t} = -(P^{n+1})_x \tag{16}$$
   $$\frac{V^{n+1} - V^{**}}{\Delta t} = -(P^{n+1})_y \tag{17}$$
   The pressure is denoted P^{n+1}, since it is only given implicitly. It is obtained by solving a linear system. In vector notation the correction equations read as
   $$\frac{1}{\Delta t}\mathbf{U}^{n+1} - \frac{1}{\Delta t}\mathbf{U}^n = -\nabla P^{n+1} \tag{18}$$
   Applying the divergence to both sides yields the linear system
   $$-\Delta P^{n+1} = -\frac{1}{\Delta t}\nabla \cdot \mathbf{U}^n \tag{19}$$
   Hence, the pressure correction step is

(a) Compute $F^n = \nabla \cdot \mathbf{U}^n$

(b) Solve Poisson equation $-\Delta P^{n+1} = -\frac{1}{\Delta t} F^n$

(c) Compute $\mathbf{G}^{n+1} = \nabla P^{n+1}$

(d) Update velocity field $\mathbf{U}^{n+1} = \mathbf{U}^n - \Delta t \mathbf{G}^{n+1}$

The question, which boundary conditions are appropriate for the Poisson equation for the pressure P, is complicated. A standard approach is to prescribe homogeneous Neumann boundary conditions for P wherever no-slip boundary conditions are prescribed for the velocity field. For the lid driven cavity problem this means that homogeneous Neumann boundary conditions are prescribed everywhere. This implies in particular that the pressure P is only defined up to a constant, which is fine, since only the gradient of P enters the momentum equation.

In addition to the solution steps, we have the visualization step, in which the stream function Q^n is computed. Similarly to the pressure is is obtained by the following steps

1. Compute $F^n = (V^n)_x - (U^n)_y$

2. Solve Poisson equation $-\Delta Q^n = -F^n$

We prescribe homogeneous Dirichlet boundary conditions.

5 Spacial Discretization

The spacial discretization is performed on a staggered grid with the pressure P in the cell midpoints, the velocities U placed on the vertical cell interfaces, and the velocities V placed on the horizontal cell interfaces. The stream function Q is defined on the cell corners.

Consider to have $n_x \times n_y$ cells. Figure 1 shows a staggered grid with $n_x = 5$ and $n_y = 3$. When speaking of the fields P, U and V (and Q), care has to be taken about interior and boundary points. Any point truly inside the domain is an interior point, while points on or outside boundaries are boundary points. Dark markers in Figure 1 stand for interior points, while light markers represent boundary points. The fields have the following sizes:

field quantity	interior resolution	resolution with boundary points
pressure P	$n_x \times n_y$	$(n_x + 2) \times (n_y + 2)$
velocity component U	$(n_x - 1) \times n_y$	$(n_x + 1) \times (n_y + 2)$
velocity component V	$n_x \times (n_y - 1)$	$(n_x + 2) \times (n_y + 1)$
stream function Q	$(n_x - 1) \times (n_y - 1)$	$(n_x + 1) \times (n_y + 1)$

The values at boundary points are no unknown variables. For Dirichlet boundary conditions they are prescribed, and for Neumann boundry conditions they can be expressed in term of interior points. However, boundary points of U and V are used for the finite difference approximation of the nonlinear advection terms. Note that the boundary points in the four corner are never used.

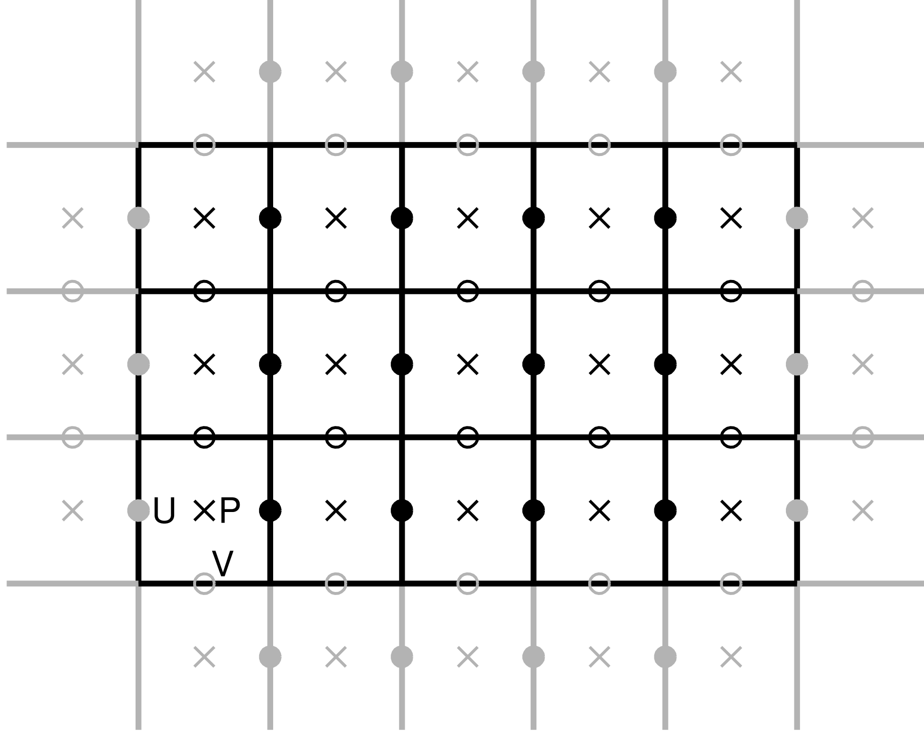

Figure 1: Staggered grid with boundary cells

5.1 Approximating derivatives

- **Second derivatives**
  Finite differences can approximate second derivatives in a grid point by a centered stencil. At an interior point $U_{i,j}$ we approximate the Laplace operator by

$$\Delta U_{i,j} = (U_{xx})_{i,j} + (U_{yy})_{i,j} \approx \frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{h_x^2} + \frac{U_{i,j-1} - 2U_{i,j} + U_{i,j+1}}{h_y^2} \quad (20)$$

  Here one or two of the neighboring points might be boundary points. The same formula holds for the component V, and for the Laplacian of the pressure P and the stream function Q. If the unknown quantity is stored in a large column vector, then the above approximation can be represented as a large sparse block matrix being applied from the left. Thus, solving the Poisson equations for P and Q, as well as solving implicitly for the viscosity terms in U and V, yields sparse linear systems to be solved, as detailed in Section 7.

- **First derivatives**
  A first derivative in a grid point can be approximated by a centered stencil.

$$(U_x)_{i,j} \approx \frac{U_{i+1,j} - U_{i-1,j}}{2h_x} \quad (21)$$

This, however, can yield instabilities, as shown in many textbooks on numerical analysis. Here the staggered grid comes into play. Assume, we are not interested in the value of U_x in the position of $U_{i,j}$, but instead we want the value in the middle between the points $U_{i+1,j}$ and $U_{i,j}$. Then the approximation

$$(U_x)_{i+\frac{1}{2},j} \approx \frac{U_{i+1,j} - U_{i,j}}{h_x} \tag{22}$$

is a stable centered approximation to U_x in the middle between the two points. In the staggered grid this position happens to be the position of $P_{i,j}$.

- **Pressure correction**
  In the pressure correction, the approximation of first derivatives on a staggered grid works out perfectly:
  1. The divergence of the velocity field $F = \nabla \cdot \mathbf{U}$ computes U_x and V_y. Both values then live in the cell centers, i.e. they can be added directly.
  2. For the pressure Poisson equation $-\Delta P = -\frac{1}{\Delta t}F$, both sides are defined in the cell centers, and $-\Delta P$ can be approximated as described above.
  3. The gradient of the pressure $\mathbf{G} = \nabla P$ requires the computation of P_x and P_y. Again, a look at Figure 1 indicates that the former then live at the position of U, while the latter live at the positions of V. Exactly where both are needed to update the velocity field.
- **Stream function**
  Similarly, the staggered grid works fine for the stream function. Both V_x and U_y live on the cell corners, so the Poisson right hand side $F = V_x - U_y$ and thus the stream function Q lives at the cell corners.
- **Nonlinear terms (central differencing)**
  The nonlinear terms are the only place where the discretization on the staggered grid does not work directly. For instance, the product UV is not directly defined, since U and V live in different positions. The solution is to take the arguments backwards: For updating U, we need $(U^2)_x$ and $(UV)_y$. If the flow in each time step is comparably slow, we wish to use the same centered staggered derivatives as before. This requires U^2 to be defined in the cell centers, and UV to the defined in the cell corners. We obtain these quantities by interpolating two neighboring values.

$$(U^2)_{i+\frac{1}{2},j} = \left(\frac{U_{i,j} + U_{i+1,j}}{2}\right)^2 \tag{23}$$

$$U_{i,j+\frac{1}{2}} = \frac{U_{i,j} + U_{i,j+1}}{2} \tag{24}$$

$$V_{i+\frac{1}{2},j} = \frac{V_{i,j} + V_{i+1,j}}{2} \tag{25}$$

  Analogously for the component V. The index notation lets this idea look more complicated than it is. All we do is create new data between two points by averaging. Using an overbar with superscript h to indicate a horizontally averaged

quantity, and an overbar with superscript v to indicate a vertically averaged quantity, we can write the update for the nonlinear terms as

$$\frac{U^* - U}{\Delta t} = -((\bar{U}^h)^2)_x - (\bar{U}^v \bar{V}^h)_y \tag{26}$$

$$\frac{V^* - V}{\Delta t} = -(\bar{U}^v \bar{V}^h)_x - ((\bar{V}^v)^2)_y \tag{27}$$

– **Nonlinear terms (upwinding)**

The above centered differencing is appropriate if quantities are not transported too far in each time step. For faster flows or larger time steps, the discretization shall be closer to an upwinding approach. We implement a smooth transition between centered differencing and upwinding using a parameter $\gamma \in [0, 1]$. We define it as

$$\gamma = \min\left(1.2 \cdot \Delta t \cdot \max\left(\max_{i,j} |U_{i,j}|, \max_{i,j} |V_{i,j}|\right), 1\right) \tag{28}$$

The value of gamma is the maximum fraction of a cells which information can travel in one time step, multiplied by 1.2, and capped by 1. The factor of 1.2 is taken from the experience that often times tending a bit more towards upwinding can be advantageous for accuracy [3].

The linear combination between centered differencing and upwinding is implemented in the following way. In corresponence to the averaged quantities, we define differenced quantities

$$(\tilde{U}^h)_{i+\frac{1}{2},j} = \frac{U_{i+1,j} - U_{i,j}}{2} \tag{29}$$

$$(\tilde{U}^v)_{i,j+\frac{1}{2}} = \frac{U_{i,j+1} - U_{i,j}}{2} \tag{30}$$

and analogously for $\tilde{V}^h$ and $\tilde{V}^v$. Using the linear combination, equations (26) and (27) generalize to

$$\frac{U^* - U}{\Delta t} = -\left((\bar{U}^h)^2 - \gamma|\bar{U}^h|\tilde{U}^h\right)_x - \left((\bar{U}^v \bar{V}^h) - \gamma|\bar{V}^h|\tilde{U}^v\right)_y \tag{31}$$

$$\frac{V^* - V}{\Delta t} = -\left((\bar{U}^v \bar{V}^h) - \gamma|\bar{U}^v|\tilde{V}^h\right)_x - \left((\bar{V}^v)^2 - \gamma|\bar{V}^v|\tilde{V}^v\right)_y \tag{32}$$

One can check the approximation values, as for instance for the U^2 term:

$$\left((\bar{U}^h)^2 - \gamma|\bar{U}^h|\tilde{U}^h\right)_{i+\frac{1}{2},j} = |\bar{U}^h| \left(|\bar{U}^h| - \gamma\tilde{U}^h\right)_{i+\frac{1}{2},j} \tag{33}$$

$$= \bar{U}^h \begin{cases} \left(\frac{1-\gamma}{2}\right) U_{i+1,j} + \left(\frac{1+\gamma}{2}\right) U_{i,j} & \text{if } \bar{U}^h \geq 0 \\ \left(\frac{1+\gamma}{2}\right) U_{i+1,j} + \left(\frac{1-\gamma}{2}\right) U_{i,j} & \text{if } \bar{U}^h < 0 \end{cases} \tag{34}$$

One can easily see that this becomes averaged central differencing for $\gamma = 0$ and conservative upwinding for $\gamma = 1$.

5.2 Boundary conditions

In the lid driven cavity problem we provide Dirichlet boundary conditions for U and V, and Neumann boundary conditions for P. Applying the correct boundary conditions requires care, since for the staggered grid some points lie on a boundary while others have a boundary between them. At the points that lie on the boundary the value is directly prescribed, such as U at the west and east boundary, and V at the north and south boundary. For U at the north and south boundary and for V at west and east boundary, one has to define a value between two data points. The idea is the same as above: average the values of two points. For instance, the north boundary lies between points with velocity U. Let the two points below respectively above the boundary be $U_{i,j}$ and $U_{i,j+1}$, and let the prescribed boundary value be U_N. Then the boundary condition is

$$\frac{U_{i,j} + U_{i,j+1}}{2} = U_N \quad \Longleftrightarrow \quad U_{i,j} + U_{i,j+1} = 2U_N \tag{35}$$

Analogously at the south boundary, and for V at the west and east boundary.

The normal derivative $\frac{\partial P}{\partial \mathbf{n}}$ at the boundary is defined using two points which have the boundary in their middle. For instance, at the north boundary, prescribing homogeneous Neumann boundary conditions yields

$$\frac{P_{i,j+1} - P_{i,j}}{h_y} = 0 \quad \Longleftrightarrow \quad P_{i,j+1} = P_{i,j} \tag{36}$$

The stream function is defined on cell boundary points. Thus all its boundary points lie on the domain boundary. We prescibe homogeneous Dirichlet boundary conditions, which can be prescribed directly.

6 Solving the Linear Systems

The pressure correction and the implicit discretization of the viscosity terms requires linear systems to be solved in every time step. Additionally, the computation of the stream function requires another system to be solved whenever the data is plotted. Since neither geometry nor discretization change with time, the corresponding system matrices remain the same in every step. This means that all matrices can be constructed in an initialization step. Of course, one would would even wish to compute the inverse matrices in an initialization step, but even for medium grid resolutions these could not be saved, since they are full matrices. Of the many possible approaches to do at least some work in the initialization, we propose the following three:

- Use Fourier methods based on the fast Fourier transform in the solution step. Initialize memory and constants in the setup phase.
- Compute good preconditioners in the setup phase. Candidates are ILU or multigrid. Save the preconditioners as sparse matrices and use them in the solution phase.
- Use elimination with reordering to compute the inverse matrices exactly, but in a comparably sparse format.

In the code the third approach is implemented. Since the matrices are symmetric positive definite, the sparse Cholesky decomposition can be used.

7 Implementation in Matlab

We take advantage of the Matlab data structures and save the field quantities as matrices. Each quantity is stored without boundary points, yielding matrices of the following sizes.

quantity	matrix size
U	$(n_x-1)\times(n_y+1)$
V	$(n_x+1)\times(n_y-1)$
P	$n_x\times n_y$
Q	$(n_x-1)\times(n_y-1)$

In solving for P and Q, as well as for the viscosity part for U and V, the boundary conditions are brought to the right hand side. The only time boundary values are used explicitly is when evaluating the nonlinear terms. Here, we extend the matrices U and V to matrices containing also the boundary points.

7.1 Updates in matrix form

Approximating derivatives can be done in matrix form using the command `diff`. If P is a matrix of values living in the cell centers (size $n_x \times n_y$), then `diff(P)/hx` yields a matrix approximating P_x (size $(n_x-1)\times n_y$) whose values live on the vertical cell edges. Analogously, `diff(P')'/hy` yields a matrix approximating P_y (size $n_x \times (n_y-1)$) whose values live on the horizontal cell edges. Similarly, we define the function `avg`, which averages in the horizontal direction, i.e. `avg(P)` yields an $(n_x-1)\times n_y$ matrix whose values are horizontally averaged, and `avg(P')'` yields an $n_x\times(n_y-1)$ matrix whose values are vertically averaged. Using `diff` and `avg`, we can write the updating steps as follows:

- **Treat nonlinear terms**
  First we extend the velocity matrices by rows and columns of boundary points.

  ```
  Ue = [uW;U;uE]; Ue = [2*uS'-Ue(:,1) Ue 2*uN'-Ue(:,end)];
  Ve = [vS' V vN']; Ve = [2*vW-Ve(1,:);Ve;2*vE-Ve(end,:)];
  ```

  The boundary points which fall onto a boundary are concatenated directly. The values of the boundary points half a meshsize outside of the boundary are computed using relation (35). These extended matrices U^e and V^e are used to proceed further. Compute $(UV)_x$ and $(UV)_y$ by averaging U vertically and V horizontally, and using the transition parameter γ:

  ```
  Ua = avg(Ue')'; Ud = diff(Ue')'/2;
  Va = avg(Ve);   Vd = diff(Ve)/2;
  UVx = diff(Ua.*Va-gamma*abs(Ua).*Vd)/hx;
  UVy = diff((Ua.*Va-gamma*Ud.*abs(Va))')'/hy;
  ```

Compute $(U^2)_x$ and $(V^2)_y$ by averaging U horizontally and V vertically:

```
Ua = avg(Ue(:,2:end-1));   Ud = diff(Ue(:,2:end-1))/2;
Va = avg(Ve(2:end-1,:)')'; Vd = diff(Ve(2:end-1,:)')'/2;
U2x = diff(Ua.^2-gamma*abs(Ua).*Ud)/hx;
V2y = diff((Va.^2-gamma*abs(Va).*Vd)')'/hy;
```

Update values of interior points:

```
U = U-dt*(UVy(2:end-1,:)+U2x);
V = V-dt*(UVx(:,2:end-1)+V2y);
```

- **Pressure correction**
  Compute divergence of the velocity field $\nabla \cdot \mathbf{U}$:

  ```
  diff(U(:,2:end-1))/hx+diff(V(2:end-1,:)')'/hy
  ```

  Update values of interior points by $-\nabla P$:

  ```
  U(2:end-1,2:end-1) = U(2:end-1,2:end-1)-diff(P)/hx;
  V(2:end-1,2:end-1) = V(2:end-1,2:end-1)-diff(P')'/hy;
  ```

- **Stream function**
  Compute curl of the velocity field $\nabla \times \mathbf{U}$:

  ```
  diff(U')'/hy-diff(V)/hx
  ```

7.2 Linear Solves

To access the field quantities by a linear combination we transform them into a long column vector using the command `reshape`. Multiplying a large sparse block matrix from the left yields the desired linear combination. Afterward the column vector is reshaped back into the appropriate matrix size. These steps are done whenever linear systems are solved:

- **Implicit viscosity**
  Consider interior points only. Add appropriate boundary values (for implicit viscosity step). Reshape. Solve linear system. Reshape and update interior points:

  ```
  rhs = reshape(U(2:end-1,2:end-1)+Ubc,[],1);
  u(peru) = Ru\(Rut\rhs(peru));
  U(2:end-1,2:end-1) = reshape(u,nx-1,ny);
  ```

- **Pressure correction**
  Compute right hand side in matrix form. Reshape. Solve linear system. Reshape back and update interior points:

```
rhs = reshape(diff(U(:,2:end-1))/hx+diff(V(2:end-1,:)')'/hy,[],1);
p(perp) = -Rp\(Rpt\rhs(perp));
P = reshape(p,nx,ny);
U(2:end-1,2:end-1) = U(2:end-1,2:end-1)-diff(P)/hx;
V(2:end-1,2:end-1) = V(2:end-1,2:end-1)-diff(P')'/hy;
```

- **Stream function**
  Compute right hand side in matrix form. Reshape. Solve linear system. Reshape back:

  ```
  rhs = reshape(diff(U')'/hy-diff(V)/hx,[],1);
  q(perq) = Rq\(Rqt\rhs(perq));
  Q = zeros(nx+1,ny+1);
  Q(2:end-1,2:end-1) = reshape(q,nx-1,ny-1);
  ```

  For visualization purposes, the homogeneous Dirichlet boundary points are included in the matrix Q.

7.3 System matrices

The Laplace operator with appropriate boundary conditions is discretized in the system matrices `Lp`, `Lu`, `Lv` and `Lq`. The matrices act on the corresponding field quantities P, U, V and Q, all in column vector shape. Since we use the sparse Cholesky presolve step, solving the system may look cumbersome. It is not. In principle (yet not efficient), the pressure correction could be computed as simple as

```
p = Lp\rhs;
```

All information required is contained in the four matrices themselves. They are constructed in the initialization step. Since the unknowns are reshaped into a long column vector row by row, we can construct the block matrices using the `kron` command: Let `K1x` be a tridiagonal matrix approximating $-\frac{\partial^2}{\partial x^2}$, and `K1y` a matrix approximating $-\frac{\partial^2}{\partial y^2}$, both in one space dimension. Then the two dimensional Laplace matrix, approximating $-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}$ is obtained by

```
kron(speye(ny),K1x)+kron(K1y,speye(nx))
```

This is it. Only care has to be taken to use the correct boundary conditions in the 1d matrices. We provide the function `K1` which generates the matrix

$$K_1 = \frac{1}{h^2} \begin{pmatrix} a_{11} & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & a_{11} \end{pmatrix} \tag{37}$$

The choice of the parameter a_{11} allows three kinds of boundary conditions, all of which can be derived by inserting the boundary relations into a three point Laplace stencil:

a_{11}	type of boundary condition
1	Neumann
2	Dirichlet for point on the boundary
3	Dirichlet for boundary between two points

This leaves us with one small detail: We prescribe Neumann boundary conditions for P. Hence, the matrix `Lp` is of corank 1. Its nullspace is $N(\texttt{Lp}) = \text{span}(\vec{e})$, where $\vec{e} = (1, \ldots, 1)^T$. Since `Lp` is symmetric, its range is $R(\texttt{Lp}) = \{\vec{b} : \vec{e}^T\vec{b} = 0\}$. The right hand side is in the range of `Lp` by construction of the projection step. In order to perform a Cholesky factorization, we modify `Lp` so that it becomes a regular matrix, while still yielding a correct solution for the pressure. This is achieved by adding 1 to the last entry. The proof is as follows.

Let $\vec{w} = (0, \ldots, 0, 1)^T$. Thus $\vec{w}\vec{w}^T$ is a matrix with its only nonzero entry being a 1 in the last position. Let A be the Neumann-Poisson matrix, i.e. $A\vec{e} = 0$. The new modified Poisson matrix is $B = A + \vec{w}\vec{w}^T$. Let $\vec{u}$ be an arbitrary vector. Then

$$B\vec{u} = \underbrace{A\vec{u}}_{\perp\vec{e}} + (\vec{w}^T\vec{u})\underbrace{\vec{w}}_{\not\perp\vec{e}} \ . \tag{38}$$

This sum can only equal 0 if $\vec{u} = 0$. Hence B has full rank. It remains to show that for any right hand side $\vec{b}$, which is admissible (i.e. $\vec{b} \perp \vec{e}$), the solution to $B\vec{u} = \vec{b}$ satisfies $A\vec{u} = \vec{b}$. First note that due to (38), we have $B^{-1}\vec{w} = \vec{e}$. Hence, we get

$$AB^{-1} = (B - \vec{w}\vec{w}^T)B^{-1} = I - \vec{w}\vec{w}^T B^{-1} = I - \vec{w}\vec{e}^T \ ,$$

and consequently $AB^{-1}\vec{b} = \vec{b}$.

7.4 Presolve with sparse Cholesky

Instead of solving the full system

```
p = Lp\rhs;
```

in each time step, we compute a Cholesky matrix in an initialization step. A plain Cholesky matrix would have many nonzero entries. We use a reordering of the unknowns to minimize the amount of fill-in due to elimination steps. A good permutation of the unknown with respect to elimination for symmetric matrices is obtained by the command `symamd`. The index permutation is stored in a permutation vector. The Cholesky decomposition of the reordered (rows and columns) system matrix is computed. Finally, to prevent Matlab from allocating memory in the solution phase we store also the transposed Cholesky matrix to gain extra speed.

```
perp = symamd(Lp);
Rp = chol(Lp(perp,perp));
Rpt = Rp';
```

In the solution step, the system is solved by a backward and a forward solution step, applied to the reordered right hand side. The inverse reordering is obtained by applying the permutation vector to the left hand side:

```
p(perp) = -Rp\(Rpt\rhs(perp));
```

8 Matlab Code

The code can be downloaded from `http://www-math.mit.edu/18086` and `http://www-math.mit.edu/cse`

```
function mit18086_navierstokes
%MIT18086_NAVIERSTOKES
%    Solves the incompressible Navier-Stokes equations in a
%    rectangular domain with prescribed velocities along the
%    boundary. The solution method is finite differencing on
%    a staggered grid with implicit diffusion and a Chorin
%    projection method for the pressure.
%    Visualization is done by a colormap-isoline plot for
%    pressure and normalized quiver and streamline plot for
%    the velocity field.
%    The standard setup solves a lid driven cavity problem.

% 07/2007 by Benjamin Seibold
%            http://www-math.mit.edu/~seibold/
% Feel free to modify for teaching and learning.
%-----------------------------------------------------------------------
Re = 1e2;     % Reynolds number
dt = 1e-2;    % time step
tf = 4e-0;    % final time
lx = 1;       % width of box
ly = 1;       % height of box
nx = 90;      % number of x-gridpoints
ny = 90;      % number of y-gridpoints
nsteps = 10;  % number of steps with graphic output
%-----------------------------------------------------------------------
nt = ceil(tf/dt); dt = tf/nt;
x = linspace(0,lx,nx+1); hx = lx/nx;
y = linspace(0,ly,ny+1); hy = ly/ny;
[X,Y] = meshgrid(y,x);
%-----------------------------------------------------------------------
% initial conditions
U = zeros(nx-1,ny); V = zeros(nx,ny-1);
% boundary conditions
uN = x*0+1;    vN = avg(x)*0;
uS = x*0;      vS = avg(x)*0;
uW = avg(y)*0; vW = y*0;
uE = avg(y)*0; vE = y*0;
%-----------------------------------------------------------------------
Ubc = dt/Re*([2*uS(2:end-1)' zeros(nx-1,ny-2) 2*uN(2:end-1)']/hx^2+...
      [uW;zeros(nx-3,ny);uE]/hy^2);
Vbc = dt/Re*([vS' zeros(nx,ny-3) vN']/hx^2+...
      [2*vW(2:end-1);zeros(nx-2,ny-1);2*vE(2:end-1)]/hy^2);

fprintf('initialization')
Lp = kron(speye(ny),K1(nx,hx,1))+kron(K1(ny,hy,1),speye(nx));
Lp(1,1) = 3/2*Lp(1,1);
perp = symamd(Lp); Rp = chol(Lp(perp,perp)); Rpt = Rp';
Lu = speye((nx-1)*ny)+dt/Re*(kron(speye(ny),K1(nx-1,hx,2))+...
     kron(K1(ny,hy,3),speye(nx-1)));
peru = symamd(Lu); Ru = chol(Lu(peru,peru)); Rut = Ru';
```

```
Lv = speye(nx*(ny-1))+dt/Re*(kron(speye(ny-1),K1(nx,hx,3))+...
     kron(K1(ny-1,hy,2),speye(nx)));
perv = symamd(Lv); Rv = chol(Lv(perv,perv)); Rvt = Rv';
Lq = kron(speye(ny-1),K1(nx-1,hx,2))+kron(K1(ny-1,hy,2),speye(nx-1));
perq = symamd(Lq); Rq = chol(Lq(perq,perq)); Rqt = Rq';

fprintf(', time loop\n--20%%--40%%--60%%--80%%-100%%\n')
for k = 1:nt
   % treat nonlinear terms
   gamma = min(1.2*dt*max(max(max(abs(U)))/hx,max(max(abs(V)))/hy),1);
   Ue = [uW;U;uE]; Ue = [2*uS'-Ue(:,1) Ue 2*uN'-Ue(:,end)];
   Ve = [vS' V vN']; Ve = [2*vW-Ve(1,:);Ve;2*vE-Ve(end,:)];
   Ua = avg(Ue')'; Ud = diff(Ue')'/2;
   Va = avg(Ve);   Vd = diff(Ve)/2;
   UVx = diff(Ua.*Va-gamma*abs(Ua).*Vd)/hx;
   UVy = diff((Ua.*Va-gamma*Ud.*abs(Va))')'/hy;
   Ua = avg(Ue(:,2:end-1));   Ud = diff(Ue(:,2:end-1))/2;
   Va = avg(Ve(2:end-1,:)')'; Vd = diff(Ve(2:end-1,:)')'/2;
   U2x = diff(Ua.^2-gamma*abs(Ua).*Ud)/hx;
   V2y = diff((Va.^2-gamma*abs(Va).*Vd)')'/hy;
   U = U-dt*(UVy(2:end-1,:)+U2x);
   V = V-dt*(UVx(:,2:end-1)+V2y);

   % implicit viscosity
   rhs = reshape(U+Ubc,[],1);
   u(peru) = Ru\(Rut\rhs(peru));
   U = reshape(u,nx-1,ny);
   rhs = reshape(V+Vbc,[],1);
   v(perv) = Rv\(Rvt\rhs(perv));
   V = reshape(v,nx,ny-1);

   % pressure correction
   rhs = reshape(diff([uW;U;uE])/hx+diff([vS' V vN']')'/hy,[],1);
   p(perp) = -Rp\(Rpt\rhs(perp));
   P = reshape(p,nx,ny);
   U = U-diff(P)/hx;
   V = V-diff(P')'/hy;

   % visualization
   if floor(25*k/nt)>floor(25*(k-1)/nt), fprintf('.'), end
   if k==1|floor(nsteps*k/nt)>floor(nsteps*(k-1)/nt)
      % stream function
      rhs = reshape(diff(U')'/hy-diff(V)/hx,[],1);
      q(perq) = Rq\(Rqt\rhs(perq));
      Q = zeros(nx+1,ny+1);
      Q(2:end-1,2:end-1) = reshape(q,nx-1,ny-1);
      clf, contourf(avg(x),avg(y),P',20,'w-'), hold on
      contour(x,y,Q',20,'k-');
      Ue = [uS' avg([uW;U;uE]')' uN'];
      Ve = [vW;avg([vS' V vN']);vE];
      Len = sqrt(Ue.^2+Ve.^2+eps);
      quiver(x,y,(Ue./Len)',(Ve./Len)',.4,'k-')
      hold off, axis equal, axis([0 lx 0 ly])
      p = sort(p); caxis(p([8 end-7]))
```

```
      title(sprintf('Re = %0.1g   t = %0.2g',Re,k*dt))
      drawnow
   end
end
fprintf('\n')

%=======================================================================

function B = avg(A,k)
if nargin<2, k = 1; end
if size(A,1)==1, A = A'; end
if k<2, B = (A(2:end,:)+A(1:end-1,:))/2; else, B = avg(A,k-1); end
if size(A,2)==1, B = B'; end

function A = K1(n,h,a11)
% a11: Neumann=1, Dirichlet=2, Dirichlet mid=3;
A = spdiags([-1 a11 0;ones(n-2,1)*[-1 2 -1];0 a11 -1],-1:1,n,n)'/h^2;
```

Acknowledgments

The author is thankful to Per-Olof Persson and Jean-Christophe Nave for their useful comments, which helped make the code more efficient, and to Gilbert Strang for his helpful remarks on notation and presentation.

References

[1] A.J. Chorin, *Numerical solution of the Navier-Stokes equations*. Math. Comput. 22, 745–762, 1968.

[2] A.J. Chorin, J.E. Marsden, *A mathematical introduction to fluid mechanics*. Third edition, Springer, 2000.

[3] M. Griebel, T. Dornseifer, T. Neunhoeffer, *Numerical simulation in fluid dynamics: A practical introduction*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[4] G. Strang, *Computational Science and Engineering*, First Edition. Wellesley-Cambridge Press, 2007.