ON THE PETSC CODE SSAFLOWLINE.C, AND ITS RESIDUAL, JACOBIAN MATRIX, AND PICARD PRECONDITIONER COMPONENTS

ED BUELER

These notes document the numerical choices made in the code `ssaflowline.c`. This code, and these choices, should be compared to those in the Matlab/Octave codes `flowline.m`, `ssaflowline.m`, and `testshelf.m` in directory `mfiles/`, which are described in detail in the lecture.

Discrete residual. The equation we solve is the no-basal-friction ($C = 0$) version of the flow-line SSA stated in the lectures:

$$\left(2BH|u_x|^{p-2}u_x\right)_x - \frac{1}{2}\rho g(1-r)\left(H^2\right)_x = 0. \tag{1}$$

The solution of this equation $u(x)$ is the velocity in the ice shelf. This PDE[1] applies on the interval $0 < x < L$ where $x = 0$ is the location of the grounding line and $x = L$ is the location of the calving front. It has these boundary conditions:

$$u\big|_{x=0} = u_g, \qquad \left[2BH|u_x|^{p-2}u_x - \frac{1}{2}\rho g(1-r)H^2\right]_{x=L} = 0, \tag{2}$$

where the thickness is a fixed function $H = H(x)$. Note that n, $p = 1+1/n$, A, $B = A^{-1/n}$, ρ, ρ_w, $r = \rho/\rho_w$, u_g, and L are all positive constants. The second boundary condition can be rewritten in the form we actually use,

$$u_x\big|_{x=L} = \gamma \qquad \text{where} \qquad \gamma = \left(\frac{\rho g(1-r)}{4B}H(L)\right)^n. \tag{2b}$$

We have clearly stated the continuum problem. In addition to choosing a discretization scheme, the first goal is to construct the *residual*, the amount by which a candidate solution to the discrete problem does not solve the discrete equations. That is, instead of immediately formulating a scheme for solving the discrete equations, we first address the simpler job of merely measuring how wrong is any candidate (guess) for the solution.

We will discretize the problem by finite differences, on an equally-spaced grid of M points $x_0, x_1, \ldots, x_{M-1}$. Let $\Delta x = L/(M-1)$ and $x_i = i\Delta x$. The unknowns U_i in the discrete problem (below) approximate the corresponding exact values on the grid $u(x_i)$, and form

Date: September 6, 2010.

[1]Actually the differential equation is a two-point boundary value problem for an ODE. In many ways it is "too simple"; see the last subsection. But we treat it as a PDE as we are considering methods that generalize easily to true PDEs.

a vector $\mathbf{U} = (U_0, \dots, U_{M-1})$. So we now write the discretized PDE (1) and its boundary conditions as M nonlinear equations in M real unknowns:

$$0 = f_0(\mathbf{U}) = U_0 - u_g \tag{3a}$$

$$0 = f_1(\mathbf{U}) = \eta(U_2 - U_1)H_{3/2}(U_2 - U_1) - \eta(U_1 - u_g)H_{1/2}(U_1 - u_g) - (\Delta x)K(H_{3/2}^2 - H_{1/2}^2) \tag{3b}$$

$$0 = f_i(\mathbf{U}) = \eta(U_{i+1} - U_i)H_{i+1/2}(U_{i+1} - U_i) - \eta(U_i - U_{i-1})H_{i-1/2}(U_i - U_{i-1}) - (\Delta x)K(H_{i+1/2}^2 - H_{i-1/2}^2), \qquad [i = 2, 3, \dots, M-2] \tag{3c}$$

$$0 = f_{M-1}(\mathbf{U}) = \eta(U_{M-2} + 2\Delta x\gamma - U_{M-1})H_{M-1/2}(U_{M-2} + 2\Delta x\gamma - U_{M-1}) - \eta(U_{M-1} - U_{M-2})H_{M-3/2}(U_{M-1} - U_{M-2}) - (\Delta x)K(H_{M-1/2}^2 - H_{M-3/2}^2) \tag{3d}$$

where $K = \rho g(1-r)/(4B)$ and

$$\eta(Z) = \left(\left(\frac{Z}{\Delta x}\right)^2 + \epsilon^2\right)^{(p-2)/2} \tag{4}$$

The functions f_i are called the *residuals*. They form a residual vector denoted $\mathbf{F}(\mathbf{U})$.

Some comments about equations (3) and (4) are appropriate:

- All equations, except the $i = 0$ case (3a), are scaled by "clearing denominators", that is, by multiplying the discretized equation by Δx^2.
- The $i = 1$ case in equation (3b) is not quite in the general form (3c) because, for greater symmetry of the Jacobian matrix (below), the known value $u(x_0) = u_g$ at the left neighbor is used directly.
- The $i = M - 1$ case (3d) comes from introducing a "virtual" unknown U_M and then approximating the derivative of the solution at the calving front by a centered difference formula:
$$u_x(L) = u_x(x_{M-1}) \approx \frac{U_M - U_{M-2}}{2\Delta x}.$$
Both the calving front boundary condition (2b) and the PDE itself (1) are enforced at $x = x_{M-1}$, which allows elimination (by-hand) of the virtual U_M [5].
- The function $\eta(Z)$ is a viscosity, up to a scalar factor. This viscosity is regularized in equation (4) by a positive constant $\epsilon > 0$, with value equal to the strain rate of a 1 m/a velocity change over the full length L of the ice shelf (200 km), so $\epsilon \approx 1.6 \times 10^{-13}\,\mathrm{s}^{-1}$. No division by zero can, therefore, occur in evaluating the viscosity, which is always positive and finite.

Newton's method and the Jacobian matrix. In equations (3) we have written the discrete problem in the abstract form

$$\mathbf{F}(\mathbf{U}) = 0.$$

Newton's method ([6], subsection 9.6) is a well-known iterative technique for solving such systems of equations. It works if the residuals are differentiable functions and the initial iterate is sufficiently close to the desired solution.

The method is usually written in "update" form,[2] as follows. First we solve a linear system for the Newton step $\mathbf{w}$, and then we actually update the approximate solution $\mathbf{U}_n \to \mathbf{U}_{n+1}$:

$$J(\mathbf{U}_n)\mathbf{w} = -\mathbf{F}(\mathbf{U}_n), \tag{5}$$
$$\mathbf{U}_{n+1} = \mathbf{U}_n + \mathbf{w}$$

For each candidate solution $\mathbf{U}$, the *Jacobian* $J(\mathbf{U})$ is a matrix with entries

$$J(\mathbf{U})_{ij} = \frac{\partial f_i}{\partial U_j}. \tag{6}$$

The Jacobian matrix (6) is tedious to write down in detail, but we do it anyway. The matrix is tridiagonal, and we only give the nonzero entries, as follows:

$$\begin{aligned}
J_{00} &= 1 \\
J_{11} &= -H_{3/2}\,\omega(U_2 - U_1) - H_{1/2}\,\omega(U_1 - u_g) \\
J_{12} &= H_{3/2}\,\omega(U_2 - U_1) \\
J_{i,i-1} &= H_{i-1/2}\,\omega(U_i - U_{i-1}) && [i = 2, 3, \dots, M-2] \\
J_{i,i} &= -H_{i+1/2}\,\omega(U_{i+1} - U_i) - H_{i-1/2}\,\omega(U_i - U_{i-1}) && [i = 2, 3, \dots, M-2] \\
J_{i,i+1} &= H_{i+1/2}\,\omega(U_{i+1} - U_i) && [i = 2, 3, \dots, M-2] \\
J_{M-1,M-2} &= H_{M-1/2}\,\omega(U_{M-2} + 2\Delta x\gamma - U_{M-1}) + H_{M-3/2}\,\omega(U_{M-1} - U_{M-2}) \\
J_{M-1,M-1} &= -H_{M-1/2}\,\omega(U_{M-2} + 2\Delta x\gamma - U_{M-1}) - H_{M-3/2}\,\omega(U_{M-1} - U_{M-2})
\end{aligned}$$

where

$$\omega(Z) = Z\,\eta'(Z) + \eta(Z)$$

is the derivative of the function $Z \mapsto Z\,\eta(Z)$ and

$$\eta'(Z) = \frac{p-2}{\Delta x^2}\,Z\left(\left(\frac{Z}{\Delta x}\right)^2 + \epsilon^2\right)^{(p-4)/2}.$$

The nonzero pattern of the Jacobian matrix is symmetric (Figure 1), but the actual matrix is not because of the way the Neumann boundary condition is imposed. Specifically,

$$J_{M-1,M-2} = H_{M-1/2}\,\omega(U_{M-2} + 2\Delta x\gamma - U_{M-1}) + H_{M-3/2}\,\omega(U_{M-1} - U_{M-2})$$

is not equal to

$$J_{M-2,M-1} = H_{M-3/2}\,\omega(U_{M-1} - U_{M-2}).$$

For $i = 2, 3, \dots, M-3$ we do have $J_{i,i-1} = J_{i-1,i}$, however. It is an advantage of the finite element method (FEM), relative to our finite difference methods here, that the Jacobian is symmetric if the FEM is applied properly.

[2] *Remark.* This form of Newton's method may not be familiar from calculus, so let's clarify. If we were solving a scalar problem $f(u) = 0$ then we would first write the linearization $\ell(u) = f(u_n) + f'(u_n)(u - u_n)$ and then the Newton step as $\ell(u_{n+1}) = f(u_n) + f'(u_n)(u_{n+1} - u_n) = 0$. Solving for u_{n+1} gives the calculus form of the iteration. The form $f'(u_n)(u_{n+1} - u_n) = -f(u_n)$ is the direct analog of (5). When the scalar derivative $f'(u_n)$ is replaced by a square matrix $J(\mathbf{U}_n)$, which may not have an inverse at some step $\mathbf{U}_n$, and when we need to choose a numerical method to solve the linear system in (5), it is desireable to *not* write the name of the (generally dense) inverse matrix $J(\mathbf{U}_n)^{-1}$.

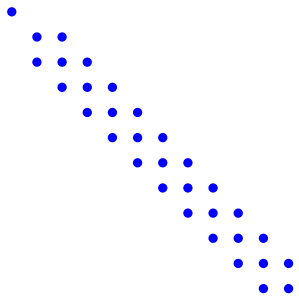

FIGURE 1. Symmetric sparsity pattern of Jacobian matrix $J(\mathbf{U})$ for $M = 12$. The matrix itself is not symmetric, with $J_{M-2,M-1} \neq J_{M-1,M-2}$, as a consequence of the chosen finite difference implementation of the Neumann boundary condition at the calving front.

Newton's method comes with few guarantees. It is generally not globally convergent. This means that for some starting points the method may not converge to anything, or it may converge to a point which is not a solution of the equations. There are, however, robust "globalization" techniques which greatly improve this aspect and make Newton's method very practical (e.g. [6], subsection 9.7 sketches the Dennis & Schnabel choice which we apply for equations (3)), and these techniques are available in PETSc (below).

The very important advantage of Newton's method is that it is quadratically convergent—*very fast* convergent—when the residual functions are twice-differentiable [3]. Properly-globalized Newton's method is known to work very fast on many nonlinear PDE problems, as long as good information is available to inform a choice of initial guess, and as long as attention is paid to the way the Jacobian is extracted and to the way the linear problem for the Newton step is solved. That is our goal here.

Picard iteration, and its relationship to the Jacobian matrix. By considering the Newton method and the Jacobian matrix above, we have an apparently quite different way of solving our finite difference equations, compared to the Picard iteration used in the Matlab/Octave codes. We show first that the difference is not as large as one might think, and then we show how to exploit the simpler matrix which appears in the Picard formulation, as part of the Newton step.

For the Picard iteration in the Matlab/Octave codes we could have written

$$A(\mathbf{U}_n)\mathbf{U}_{n+1} = \mathbf{b} \tag{7}$$

from the discrete SSA equations in the form $A(\mathbf{U})\mathbf{U} = \mathbf{b}$. In this case

$$\mathbf{F}(\mathbf{U}) = A(\mathbf{U})\mathbf{U} - \mathbf{b}$$

is the residual vector which we seek to make zero.

The Picard iteration can easily be written in the same form as the Newton step. First we subtract $A(\mathbf{U}_n)\mathbf{U}_n$ from each side of (7):

$$A(\mathbf{U}_n)\mathbf{U}_{n+1} - A(\mathbf{U}_n)\mathbf{U}_n = \mathbf{b} - A(\mathbf{U}_n)\mathbf{U}_n.$$

Now we recognize $-\mathbf{F}(\mathbf{U}_n)$ on the right side, factor the left side, and write the whole thing in update form:

$$A(\mathbf{U}_n)\mathbf{w} = -\mathbf{F}(\mathbf{U}_n), \tag{8}$$
$$\mathbf{U}_{n+1} = \mathbf{U}_n + \mathbf{w}$$

Form (8) simply replaces $J(\mathbf{U}_n) \to A(\mathbf{U}_n)$ in (5).

It is easy to extract the Picard matrix $A(\mathbf{U})$ from the residual-evaluation function which computes $\mathbf{F}(\mathbf{U})$. We identify the coefficients, supposing that the viscosity is fixed at its values from $\mathbf{U}$ at a previous step. Again the matrix is tridiagonal:

$$\begin{aligned}
A_{00} &= 1 \\
A_{11} &= -H_{3/2}\,\eta(U_2 - U_1) - H_{1/2}\,\eta(U_1 - u_g) \\
A_{12} &= H_{3/2}\,\eta(U_2 - U_1) \\
A_{i,i-1} &= H_{i-1/2}\,\eta(U_i - U_{i-1}) && [i = 2, 3, \ldots, M-2] \\
A_{i,i} &= -H_{i+1/2}\,\eta(U_{i+1} - U_i) - H_{i-1/2}\,\eta(U_i - U_{i-1}) && [i = 2, 3, \ldots, M-2] \\
A_{i,i+1} &= H_{i+1/2}\,\eta(U_{i+1} - U_i) && [i = 2, 3, \ldots, M-2] \\
A_{M-1,M-2} &= H_{M-1/2}\,\eta(U_{M-2} + 2\Delta x\gamma - U_{M-1}) + H_{M-3/2}\,\eta(U_{M-1} - U_{M-2}) \\
A_{M-1,M-1} &= -H_{M-1/2}\,\eta(U_{M-2} + 2\Delta x\gamma - U_{M-1}) - H_{M-3/2}\,\eta(U_{M-1} - U_{M-2})
\end{aligned}$$

The careful reader will notice that the Picard matrix $A(\mathbf{U})$ is "the same" as the Jacobian matrix $J(\mathbf{U})$, with the replacement $\omega \to \eta$. This is true, and indeed the only real simplification gained by considering the Picard matrix, compared to the Jacobian, is that the *derivative* of the visosity $\eta(Z)$ is not needed. This simplicity advantage of the Picard matrix becomes more significant with two or three spatial dimensions, as in more realistic ice shelf modeling.

Implementation in PETSc. As pointed out in the lecture, the goal here is to demonstrate the use of the PETSc library [1]. PETSc contains a rich set of linear and nonlinear solver tools which, unlike Matlab/Octave, "scale up" to handle the largest computations done in science and engineering. On our problem, and problems like it, we are able to do build practical parallel simulations by writing code which works on sub-domains of our problem. This subdomain code is then "handed" to the PETSc Newton solver object. For PDE problems, PETSc codes for linear and nonlinear PDEs always demonstrate a parallel "domain decomposition" of the most basic sort. Codes for nonlinear PDES always a residual evaluation routine. One may also write routines which evaluate and assemble the Jacobian matrix or approximations to it (i.e. Picard matrix in our case).

The C code `ssaflowline.c` contains these routines. To build and run `ssaflowline.c` see Figure 2. The code is intended for inspection by the reader. It is hoped that this PETSc application is simultaneously simple enough and nontrivial enough so that the mathematical details do not hide the structure of the scientific computational tools.

The major components of the code `ssaflowline.c` are:

- `AppCtx`: This structure is passed into the residual and Jacobian calculation routines so that these routines "know about" the constants set by default and by the

```
$ cd karthaus/petsc/
$ source set_my_petsc
$ make
$ ./ssaflowline               # run it
$ ./ssaflowline -help |less   # see options
```

FIGURE 2. Build and run `ssaflowline.c`. You will need to edit the file `set_my_petsc` to set your `PETSC_DIR` and `PETSC_ARCH` to the correct values for your PETSc.

user's options. Also, for example, the gridded thickness values are here so that they can be used in the calculations.

- `FillThicknessAndExactSoln()`: As described in the lectures, we are solving for velocity in a steady-state case where both the exact ice thickness and velocity can be computed by-hand. This routine computes both of these fields. The exact velocity is only used for evaluating the numerical error.
- `FormInitialGuess()`: As described in the lectures, we can use a linear function as a guess for the velocity. This initializes the Newton iteration. (Option `-ssa_guess 2` allows use of the exact solution as an initial guess. This is useful for debugging.)
- `FormFunctionLocal()`: Routine which calculates the residual $\mathbf{F}(\mathbf{U})$ from the current value of the solution (in the Newton iteration). The goal of the Newton method object (`SNES snes` in `main()`) is to make this residual zero.
- `FormTrueJacobianMatrixLocal()`: Calculates the Jacobian matrix $J(\mathbf{U})$ from the current value of the solution.
- `FormPicardMatrixLocal()`: Calculates the Picard matrix $A(\mathbf{U})$ from the current value of the solution.
- `main()`: Reads options and sets up `AppCtx user`. Based on options, sets up `SNES snes`, `DA user.da`, and related `Vec`s. Calls `SNES` routines to set the method for computing the Jacobian, based on user options. Ask the SNES object to solve the equations, by Newton's method, and gets/displays feedback on iterations and convergence.

The "`Local`" which appears in the names of the calculation routines (above) relates to the PETSc `DA`=(distributed array object) which does the domain decomposition across the grid. The solution, the residual, and the thickness functions are all PETSc `Vec` objects, which are (conceptually) vectors in $\mathbb{R}^M$, but which are distributed across processors according to the information in the `DA`.

For additional understanding of `ssaflowline.c`, the user should see the source code itself and read the *PETSc User's Manual* [1], especially the descriptions of `Vec`, `Mat`, `DA`, and `SNES` objects.

"Under the hood" of the example done here is a PETSc `KSP` object which runs a Krylov subspace iterative linear solver. We use the PETSc default `KSP` choice of GMRES because, in part, our Jacobian and approximate Jacobians are not symmetric. The `KSP` is called

when the SNES object managing the Newton method needs to solve a linear system for the step $\mathbf{w}$. Thus using SNES *automatically* produces an outer/inner iteration structure.

To write a first-draft PETSc code on a problem like the one here, one writes a residual calculation routine plus minimal PETSc structure. I have kept this first draft, namely ssaflowline_v0.c. There are nonlinear equation solution techniques which do not need the bug-prone assembly of Jacobian (or Picard, or any other) matrices. In fact, with this most-minimal code we can try two methods:

i) Option -snes_mf: A matrix-free form of the Newton method known as unpreconditioned "Jacobian-free Newton-Krylov" for domain decomposition problems [4]. It is a significant technique because, when appropriately used, it may be the the key to petaflops-level scientific simulations. Actually making it scale requires building something matrix-like as a preconditioner, however, and we do that below.

ii) Option -snes_fd: Naive finite differencing of the residual evaluation routine to construct an approximate Jacobian. An un-necessarily large matrix is assembled, for a PDE problem like the one here.

In using the first-draft with the above options it became clear that the residual evaluation gave an improved approximate Jacobian if the Dirichlet boundary condition at the grounding line was applied symmetrically, so that the $i = 1$ residual equation looked like (3b) instead of the form it had in ssaflowline_v0.c, namely

$$0 = f_1(\mathbf{U}) = \eta(U_2 - U_1)H_{3/2}(U_2 - U_1) - \eta(U_1 - u_0)H_{1/2}(U_1 - U_0).$$

The first draft was improved to the final code ssaflowline.c in several other ways. I added user options setting most parameters at runtime. Then, because method *ii)* above for computing the finite difference Jacobian can be improved greatly by giving PETSc the information that unknown U_i depends only on unknowns U_{i-1} and U_{i+1}, I supplied a *coloring* for the finite-difference method; see [1]. (The DA object knows the details and I did not write anything subtle, but there is a bit of set up.)

Then I added a Picard evaluation routine. Even though this is not the true Jacobian, it can be used with option -snes_mf_operator to get much of the effect of the true Jacobian; see below. Finally, I added a true Jacobian evaluation routine. The default for ssaflowline is to use this true Jacobian.

With these additions there are three more computational paradigms to try; we continue the enumerated list above. Note that options with prefix "ssa_" are new in ssaflowline.c, and are not in PETSc itself:

iii) No option: Use the true, assembled Jacobian. It can be used with various built-in preconditioning methods controlled by option -pc_type. The default preconditioner on one process is ILU. The default for parallel runs is block Jacobi with ILU on the blocks.

iv) Option -ssa_fd: Use a DA-based coloring routine to compute the finite difference approximation to the Jacobian much more efficiently.

v) Option combination -snes_mf_operator -ssa_picard: Assemble the Picard matrix, but then use it only as a preconditioning matrix in applying the Krylov method, while still computing the effect of the Jacobian by the matrix-free method. For

large-scale ice flow problems we should believe that this is a key technique, because the Jacobian evaluation means complex, bug-prone code, which may not even be possible to write when many physical processes are interacting, yet a rough, approximate Jacobian used as a preconditioner suffices to accelerate the matrix-free application of the Jacobian to a vector [4].

Irritating detail. On one of my first attempts using the unpreconditioned matrix-free method yielded a divergence error, despite coming close to the exact solution by reported errors. This is resolved with an obscure option which modifies the way the matrix-free Jacobian is applied (to use a different step-size algorithm):

```
$ ./ssaflowline -snes_mf
...   DIVERGED_LS_FAILURE Number of Newton iterations = 6
  numerical errors in velocity: ...   2.5481e-02 relative maximum
$ ./ssaflowline -snes_mf -mat_mffd_type ds
...   CONVERGED_FNORM_RELATIVE Number of Newton iterations = 4
  numerical errors in velocity: ...   1.3373e-02 relative maximum
```

Results. We want to know that our method, which is to say our actual code, is correct. There are two important aspects to this. We claim we are solving equation (1), so one aspect is *verification*, to measure difference between the final numerical result of our code and an exact solution, in a case where the latter is available. The other aspect is the quadratic convergence mentioned above: if we have constructed a good initial quess and we have chosen a reasonable preconditioning routine then we will see that the Newton steps cause rapid convergence even for large problems.

So I did runs that looked like this:

```
./ssaflowline -snes_monitor -snes_rtol 1e-12 -ksp_rtol 1e-12 -da_grid_x M
```

for values $M = 10, 100, 10^3, 10^4, 10^5$ and in all runs the `SNES` object reported convergence. These are single process runs; parallelization is not yet the issue. The $M = 10^5$ run only takes 1.2 seconds on my laptop, so raw performance on this 1D problem is also not an issue!

The verification results from these runs in Figure 3 show that once we reach $M = 10^4$ we are getting results correct to about 6 digits, relative to the exact solution. The 2 m grid with $M = 10^5$ gives no more accuracy in this sense than the 20 m grid with $M = 10^4$.

Figure 4 gives the evidence for quadratic—more properly superlinear—convergence of the "outer" nonlinear iteration managed by the `SNES` object. That is, Newton is working. Note that if the residual norm was decaying exponentially in iteration number (= "linear convergence" [3]), $\|\mathbf{F}(\mathbf{U}_n)\| \sim \alpha^n$ for $\alpha < 1$, then Figure 4 would show straight lines for each M. But in all cases the residual norm decays at an increasing rate, until we reach the level controlled by linear iteration tolerance, rounding error, and condition number of the discrete nonlinear problem. Thus we have probably written the correct Jacobian!

Our code runs just fine on many processors. For example,

```
$ mpiexec -n 10 ./ssaflowline -da_grid_x 10000
```

on ten processes computes the a solution with the same accuracy as on one (2.2284×10^{-6} versus 2.2284×10^{-6} relative maximum error). But are we getting a benefit from this parallelism?

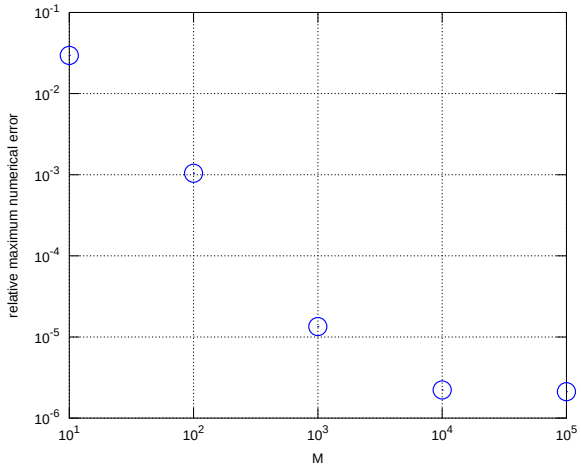

FIGURE 3. Verification: Grid refinement ($M = 10, 100, 10^3, 10^4, 10^5$) produces reduction in numerical error, relative to exact solution, and measured by comparing the maximum velocity error anywhere on the grid to the largest velocity.

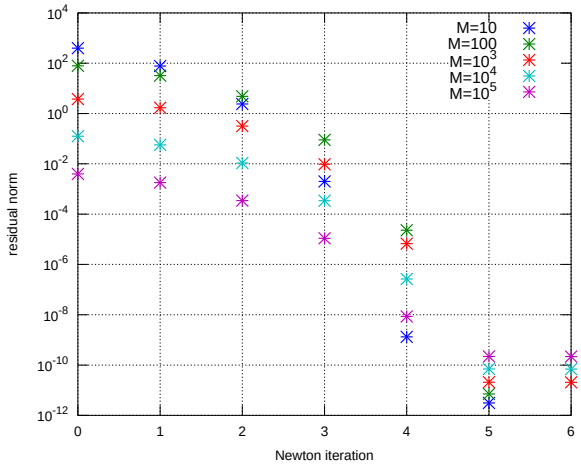

FIGURE 4. Evidence of quadratic, or at least super-linear, convergence: The residual norm decays faster than linearly with iteration number (i.e. faster than (residual norm) $\sim \alpha^n$ where n is the iteration count). Note that on the coarsest grid ($M = 10$) we see the most rapid convergence to level 10^{-14} smaller than the initial residual norm.

For a large-scale problem there are several measures of parallel computational performance which we could now be interested in. One might measure the number of linear iterations, the number of floating-point operations, or the actual ("wall clock") time needed to solve a large problem in parallel. For real problems we want to see performance which will scale up to apply to models covering large parts of realistic ice sheets at high spatial

resolution. This surely reflects the main goal of using a toolkit like PETSc for ice sheet modeling. An example for ice sheet modeling is in [2]; see figure 24.

The time has come to admit, however, that the 1D ice shelf problem addressed here in these notes is *too simple*. Actually measuring computational performance in this example does not illustrate what happens to 2D and 3D problems, for these reasons:

i) The matrices here are tridiagonal, which means that a default preconditioner (ILU) is both exact *and* fast on one process. The default block-Jacobi-plus-ILU-on-blocks choice in parallel is inexact, and thus much less fast because actual iterations must occur, on more than one process. Having more than one process is therefore disadvantageous on this simple problem because the preconditioned-KSP no longer solves in just one iteration. (In other words, PETSc is "wise" and reverts to the obvious best choice for a diagonally-dominant tridiagonal problem!)

ii) The domain is one dimensional but the modeled physical process, namely the resolution of stresses in an ice shelf, is long range. Dividing a roughly-square, 2D ice shelf region among 100 processes requires effects to be communicated across (through) 10 processes. But here in 1D the effect must "domino" through all 100 processes if we choose to use that many.

Finally we point out a basic, positive benefit of our efforts here: compiled C code runs a lot faster than interpreted Octave, as expected. This run

```
$ ./ssaflowline -da_grid_x 200000
```

on a 1 m grid ($M = 2 \times 10^5$ points with length $L = 200$ km) returned a numerical velocity field that was correct to a relative error of 2×10^6. It completed in 6 Newton iterations and about 2 wall clock seconds. By contrast, the $M = 4000$ case for the Octave code `testshelf.m` took 16 seconds and returned an answer correct to 5×10^6. The fact that our code uses PETSc is probably not very important in this regard.

Acknowledgements. Help from Jed Brown was central to building this example.

References

[1] S. Balay et al., *PETSc Users Manual*, Tech. Rep. ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010. `http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manual.pdf`.

[2] E. Bueler and J. Brown, *Shallow shelf approximation as a "sliding law" in a thermodynamically coupled ice sheet model*, J. Geophys. Res., 114 (2009). F03008, doi:10.1029/2008JF001179.

[3] R. L. Burden and J. D. Faires, *Numerical Analysis*, Brooks/Cole, Pacific Grove, CA, seventh ed., 2001.

[4] D. A. Knoll and D. E. Keyes, *Jacobian-free Newton-Krylov methods: a survey of approaches and applications*, J. Comp. Phys., 193 (2004), pp. 357–397.

[5] K. W. Morton and D. F. Mayers, *Numerical Solutions of Partial Differential Equations: An Introduction*, Cambridge University Press, second ed., 2005.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 2nd ed., 1992.