Preface

Why read this book?

This book is about numerically solving linear and nonlinear partial differential equations (PDEs) by writing C code [Kernighan and Ritchie, 1988] that directly calls PETSc. It tries to both explain the ideas and illustrate them through example codes. The example~~code~~s come with enough background information and context so that readers can easily use them as a basis for further developments. Demonstrated performance and scalability are goals, so runtime options are explained and compared, and explored in the exercises.

This book is written from the conviction that *better access to common knowledge among experts* advances scientific computing as a discipline. An expert in PETSc may say about this book that "I knew all that" *and* that "this book is a fast on-ramp to what I already know." That is precisely my hope.

So, let's suppose you have taken a mathematics course or two in partial differential equations (PDEs). You have written a few codes in C, and probably some in MATLAB or python or similar ~~prototyping~~ languages. You are interested in solving PDE or similar models numerically in parallel on big problems. Then this book is for you.

scripting

Note: Matlab + Python are far more than prototyping languages

What is PETSc?

The Portable, Extensible Toolkit for Scientific computing (PETSc)[1] is an open-source, mathematical software library built on top of the standard software layer for large-scale parallel computation, namely the Message Passing Interface (MPI) [Gropp et al., 1999]. Thus PETSc is a framework capable of solving problems ~~like~~ PDEs at "large scale," that is, at high resolution and on supercomputers with hundreds to millions of cores. PETSc also runs on your laptop, and that is where most examples from this book should be tried first.

such as

PETSc is not particularly new. Version 2.0, the first version to make an impact in the scientific computing world, was ~~built~~ in 1994. A well-known monograph Smith et al. [1996][2] uses PETSc

developed

[1] Say it "pets sea." The homepage for PETSc, including download and installation instructions, is www.mcs.anl.gov/petsc.

[2] B. Smith, P. Bjorstad, and W. Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 1996

2.0 for scalable solutions of linear PDEs. That book focusses on preconditioned iterative linear solvers and domain decomposition. For example, methods ~~like~~ such as additive Schwarz are shown to scalably-solve the Poisson equation on irregular domains.

But PETSc is now[3] at version 3.6. It has evolved into a more powerful toolbox with a much richer API (application program interface). Typical examples and applications are for nonlinear PDEs. Nonlinear, multigrid, and multiphysics[4] parts of the API are now highly-visible to users. The PETSc strategy is to compose Newton's method and mesh topology tools with a run-time choice of preconditioners and iterative linear solvers. Navigating this algorithmic "stack" requires more user knowledge than a generation ago.

In summary, PETSc may not be a silver bullet, but it presents users with many powerful tools for solving ~~hard~~ difficult problems, well beyond iterative linear algebra. As twenty years have passed since version 2.0 and Smith et al. [1996], a new book about PETSc is appropriate.

What I need from you, the reader

To make sense of this book, some of the mathematical theory of PDEs must be familiar. Evans [2010] is recommended for this theory, but it is not really a prerequisite.

I will also assume that the reader has a bit of *practical intuition* about PDE problems—should I use the common term "maturity"?—including exposure to nonlinear problems.[5] Of course, all applied mathematicians, distinctly including this author, are wanting when it comes to having the best intuition for nonlinear PDEs.

Multiple numerical discretization paradigms will arise here, and at least one numerical approach to PDEs should already be in the reader's toolbox. That might be the finite element method (FEM) [Braess, 2007, Elman et al., 2005, Karniadakis and Sherwin, 2013], finite differences [Morton and Mayers, 2005], finite volumes [LeVeque, 2002]. Spectral methods [Trefethen, 2000] are outside of our scope. Previous exposure to multigrid ideas [Briggs et al., 2000] would be helpful, but the concepts will be reviewed as we approach this key topic.

Many ideas from numerical linear algebra [Greenbaum, 1997, Trefethen and Bau, 1997] will appear, often with no or brief introduction. The definitions of vector norms and (induced) matrix norms, along with the LU and Cholesky decompositions, are assumed. The textbook by Trefethen and Bau [1997] is thus the closest of the above-mentioned texts to an actual prerequisite for the material in this book.

[3] Version 3.6.2 is current in November 2015.

[4] This buzzword refers to a diverse system of coupled PDEs with nontrivial scalings among the variables.

[5] [Ockendon et al., 2003] is recommended.

Starting in Chapter 8 I will assume that you are interested in utilizing unstructured grids, though not at the exclusion of structured grid approaches. Basics of the FEM method will be reviewed in Chapters 5, 8, and 10, but the reader with some background understanding will benefit most. Priority topics for a reader's FEM review include the weak form of a PDE and the idea of assembling the equations element-by-element.

There is much that this book does NOT do

I'll assume you want to solve PDEs, though there are many other uses of PETSc. Furthermore, this book

- does not replace either the PETSc *User's Manual*, or online searches of the PETSc manual pages, for understanding the API,
- does not help you install PETSc,* ~~footnote~~
- does not use Fortran or C++,[6]
- does not help with most of the many packages PETSc links to,
- does not do a complete job of teaching the FEM or any other discretization paradigm for PDEs,
- does not seriously address whether its numerical solutions are good models of physical problems,
- does not consider spatial dimensions ~~except 1, 2, and~~ beyond 3,
- does not prove anything,[7] and
- does not adequately cover what is known about PDEs, much less what is not known.

At the command line

Before we really get started, what "computer skills" do I assume? In summary, something more than what you need to get started with MATLAB, but certainly less than professional programmer abilities. The numerical programming here is stereotyped C coding using a modest language subset.

You need to have written and compiled C programs before. Running and modifying the examples will inevitably expose some subtleties of the C language, but no more than would appear in a first college course in computer programming using C or a similar language. Concepts of compiling and linking, of including header files, of passing arguments by value and reference, and, perhaps most importantly, the two concepts of pointer variables and arrays-as-pointers, should all be familiar.

* The PETSc team will help with failed installs see www.mcs.anl.gov/petsc/documentation/bugreporting.html

[6] All examples are in C, ~~but we use~~ utilizing ANSI C99 features.

[7] Theorems are stated precisely when appropriate. In examples we give evidence for convergence and scalability when possible.

1

Getting started with PETSc

Get the example codes

Before starting into our first example, please use `git` to get the example C codes:

```
$ git clone https://github.com/bueler/p4pdes.git
```

Codes from subdirectory `p4pdes/c/ch`*N*`/` are for Chapter *N* of this book.

A code that does almost nothing, but in parallel

The purpose of the PETSc library is to help you solve scientific and engineering problems on multi-processor computers. As PETSc is built on top of the Message Passing Interface (MPI; [Gropp et al., 1999]) library, some of its flavor comes through. Therefore we start with an introductory PETSc code which calls MPI for some basic tasks.

Our program `e.c`, shown in its entirety in Code 1.1, approximates Euler's constant by its Maclaurin series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.718281828 \tag{1.1}$$

It does the computation in a distributed manner by computing one term of the infinite series on each process, giving a better estimate of e when run on more MPI processes. This is a silly use of PETSc, but it is ~~also~~ an easy-to-understand parallel computation.

As with most C programs, Code 1.1 starts by including needed headers. However, as with most codes in this book, only `petsc.h` is needed because it includes MPI too.

Like any C program, `e.c` has a function called `main()` which takes inputs from the command line, namely `argc` and `argv`. The former is an `int` holding the argument count and the latter is an array of www.mcs.anl.gov/petsc/documentation/installation.html

to install. You will do steps that look like

```
export PETSC_DIR=/home/bueler/petsc
export PETSC_ARCH=linux-c-dbg
./configure --download-mpich --download-triangle --with-debugging=1
make all
```

though the specific configure options may be quite different in your environment. Here we download the MPICH package, because on this machine there no existing MPI installation, and the triangle package, because it is used in Chapter 8. We configure a version of PETSc with debugging symbols because this is essential for understanding the trace-back which happens at run-time errors. After above steps work successfully, run "make test" and see if it passes the tests.

is

When PETSc is correctly installed the environment variables PETSC_DIR and PETSC_ARCH point to a valid installation, and the MPI command mpiexec is from the same MPI installation as was used in configuring PETSc.[1]

Do the following to compile e.c:

```
$ cd p4pdes/c/
$ cd ch1/
$ make e
```

Calling "make" uses makefile in the ch1/ directory; an extract is shown in Code 1.2. All the makefiles in this book have this recommended [Balay et al., 2014] form.

extract from ch1/makefile

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules

e: e.o chkopts
	-${CLINKER} -o e e.o  ${PETSC_LIB}
	${RM} e.o
```

Code 1.2: All our makefiles look like this.

Run the code like this:

```
$ ./e
e is about 1.000000000000000
rank 0 did 1 flops
```

The value 1.0 is a very poor estimate of e, ~~but it~~ does better with more MPI processes:

the code

[1] Type "which mpiexec" to find which one you are running. You may need to modify your PATH environment variable to get the right mpiexec.

```
$ mpiexec -n 5 ./e
e is about 2.708333333333333
rank 0 did 0 flops
rank 4 did 3 flops
rank 2 did 1 flops
rank 3 did 2 flops
rank 1 did 0 flops
```

With $N = 20$ processes, and thus $N = 20$ terms in series (1.1), we get a good double-precision estimate:

```
$ mpiexec -n 20 ./e
e is about 2.718281828459045
rank 0 did 0 flops
...
```

"Double precision," of course, refers to the 64-bit floating-point representation of real numbers. This type, which is normally aliased to `PetscReal` inside PETSc codes, corresponds to about 15 decimal digit accuracy [Trefethen and Bau, 1997].

Perhaps the reader is now worried that this book was written using a cluster with 20 physical processors whereas the reader has a little laptop with only a couple of cores. Not so. In fact these 5 and 20 process runs work just fine on the author's 4-core laptop. MPI processes are created as needed, using an old feature of operating systems: multitasking. Actual speedup from parallelism is another matter entirely, to which we will return.

Returning to `e.c` in Code 1.1, each MPI process computes term $1/n!$ where n is returned by `MPI_Comm_rank()`.[2] Note `PETSC_COMM_WORLD` is an MPI communicator [Gropp et al., 1999] containing all processes generated by using "`mpiexec -n N`" at the command line. A call to `MPI_Allreduce()` computes a partial sum of (1.1) and sends the result back to each process. These direct uses of the MPI ~~library~~ API are a part of using PETSc, because ~~it~~ PETSc generally avoids duplicating MPI functionality.

We print the computed estimate of e once, but each process also prints its rank and the work it did. The formatted print command `PetscPrintf()`, similar to `fprintf()` from the C standard library, is called twice in the code. The first time it uses MPI communicator `PETSC_COMM_WORLD` and the second time `PETSC_COMM_SELF`. The first of these printing jobs is thus *collective* over all processes, and only one line of output is produced, while the second is individual to each rank[3] and we get `N` printed lines. The `PETSC_COMM_SELF` output lines can appear in apparently random order because the print occurs as soon as that rank reaches the `PetscPrintf()` command in the code.

[2] A call to `MPI_Comm_size()` would return `N`, the communicator's *size*.

[3] A process is often just called a *rank* in MPI language.

Every PETSc *program*

Every PETSc program should start and end with the commands `PetscInitialize()` and `PetscFinalize()`:

```
PetscInitialize(&argc,&args,NULL,help);
... everything else goes here ...
PetscFinalize();
```

As the last argument to `PetscInitialize()` we supply a `help` string. This string is a good place to say what is the purpose of the code,and thus it is often declared in the very first line of a PETSc code. To see the help string, and a longer list of possible PETSc options, do:

```
$ ./e -help
```

or

```
$ ./e -help | less
```

Through option `-help`, PETSc programs have a built-in help system for runtime options that is both light-weight and surprisingly-effective. For example, to see options related to logging performance, do

```
$ ./e -help | grep log_
```

See Exercise 1.2 for an example of how to add a new option to your own program.

Unfortunately with respect to source-code aesthetics, all our PETSc example codes have error-checking clutter. While languages other than C might help with decluttering, we are stuck with ugly lines that look like

```
ierr = PetscCommand(...); CHKERRQ(ierr);
```

The explanation is that almost all PETSc methods, and most user-written methods in PETSc programs, return an `int`* for error checking, with value 0 if successful. In the line above, `ierr` is passed to the `CHKERRQ()` macro. It does nothing if `ierr == 0` but it stops the program with a "traceback" otherwise. A traceback is a list of the nested methods, in reverse order, showing the line numbers and method names of the location where the error occurred. Traceback is most effective if PETSc is configured with debugging symbols as we did above, i.e. with configure option `---with-debugging=1`.

* The special typedef of PETSc Error Code is used to prevent confusing error codes with regular integers used in the programs.

This traceback is a first line of defense when debugging run-time errors, so in this book we always capture-and-check the returned error code using the `CHKERRQ()` macro. However, after Chapter 1 we will ~~strip~~ remove the "`ierr =`" and "`CHKERRQ(ierr);`" clutter from each code when it is displayed in the text.

Parallel reductions are generally not bit-wise deterministic

The code `e.c` does a finite sum in parallel, namely an N-term truncation of (1.1) on N processors, by a call to `MPI_Allreduce()`. Such parallel reduction operations are also key steps in linear algebra algorithms like GMRES (Chapter 2).

On one hand, floating-point arithmetic is not, ~~however,~~ commutative or associative, though nearly so.[4] On the other hand, when done in parallel a total order on the sum is not pre-determined, ~~and~~ this is similarly true for other parallel reductions ~~like~~ such as products. For example, one can suppose when using `MPI_Allreduce()` for summation that the terms are collected on the rank 0 processor and that they are added to a running sum as they arrive; this is the stage at which the non-determinacy applies. After the sum is done on rank 0 the result is sent ("scattered") to all other ranks, without further floating-point consequences. See [Gropp et al., 1999].

It follows that parallel reductions generally exhibit non-determinacy coming from the order of operations. Such non-determinacy is well-known; see for example the entry on "determinacy", and on "associative non-determinacy" in particular, in [Padua, 2011]. Because of this basic fact a seemingly-deterministic program like `e.c` could, at least in theory, have a different outcome at the level of rounding error when run a second time in parallel.

Though the ~~fact of~~ non-determinism of parallel reductions may never bite the reader, it is a numerical fact of life. *Stable* algorithms, in particular backward stable ones [Trefethen and Bau, 1997], do not suffer large output changes between runs as a result of this ~~fact.~~ non-determinism

Demonstrating the non-determinacy on a simple sum like (1.1) is difficult because it helps to have many processors[5] *and* significant noise in the interconnect between processors, so that terms are collected in a different order on each run. These conditions are hard to achieve using ~~software~~ MPI processes on a small (low core count) machine, so Exercise 1.2 simply demonstrates the non-commutativity of the addition operation in a serial run.

Looking ahead, Chapter 2 introduces more important "facts of life," of which we mention two here. The first relates to floating-point arithmetic, namely that some linear systems are intrinsically ill-conditioned. Conditioning is related to the stability of algorithms [Trefethen and Bau, 1997], but it is not (fundamentally) related to parallel computations. The need for, and parallel non-determinism

[4] See [Higham, 2002, Trefethen and Bau, 1997] for theory. See Exercise 1.2 for an example of non-commutativity.

[5] The example on page 563 of [Padua, 2011] proposes 10000 processes.

process but $N = 1000$ terms, show that that the last couple of decimal digits, in a 16 significant digit display, vary from run to run. If we did permuted partial sums of series (1.1) would we see the same variation?

1.3 Program `e.c` does redundant work, and a terrible job of load-balancing, because the computation of the factorial $n!$ on the rank n process requires $n - 1$ flops. Modify `e.c` to a new code `balance.c` which balances the load almost perfectly, giving one divide operation on each `rank` $>$ 0 process. Use blocking send and receive operations (`MPI_Send()`,`MPI_Recv()`) to pass the result of the last factorial to the next rank. (Now the code does ~~lots~~ a great deal of unnecessary communication and waiting, so neither `e.c` nor `balance.c` are good examples for future use!)

Perhaps ~~labe~~ label with I, II, ... IV (see page 41) to prevent confusion with i, ii, ... you use in other places?

2

Finite-dimensional linear systems

Some facts of (numerical) life

~~Our goal is to compute more interesting quantities than Euler's constant e, and~~ At the core of many numerical PDE solutions is a finite-dimensional linear system. ~~So~~ In this section we both recall basic ideas of numerical linear algebra and apply PETSc to solve linear systems.

Suppose $\mathbf{b} \in \mathbb{R}^N$ is a column vector and $A \in \mathbb{R}^{N \times N}$ is a square matrix.[1] The linear system

$$A\mathbf{u} = \mathbf{b} \tag{2.1}$$

has a unique solution $\mathbf{u} \in \mathbb{R}^N$ if A is invertible, namely

$$\mathbf{u} = A^{-1}\mathbf{b}. \tag{2.2}$$

This is simple in theory.

It is not so simple in practice, however, to solve linear systems on a computer. Here are two facts to keep in mind while working numerically with linear systems [Trefethen and Bau, 1997]:

i) *limit to accuracy*: If real numbers are represented with machine precision ϵ then the solution of (2.1) can only be computed within an error $\kappa(A)\epsilon$ where $\kappa(A) = \|A\|\|A^{-1}\|$ is the *condition number* of A (for the induced matrix norm $\|\cdot\|$).

ii) *cost of direct solutions*: If A is a generic $N \times N$ matrix then computation of solution (2.2) by a direct method ~~like~~ such as Gauss elimination, whether actually forming A^{-1} or not, is an $O(N^3)$ operation.[2]

Fact i) is about conditioning not methods. Informally speaking, there are matrices A and $\tilde{A}$ that are the same to within machine precision ϵ but for which the infinite-precision solutions to (2.2) differ by an amount $\kappa(A)\epsilon$. Rounding errors, which act somewhat like random

[1] PETSc can handle complex matrices, but all matrices are real in this book.

[2] See Lecture 32 in Trefethen and Bau [1997] for caveats with respect to "$O(N^3)$." Nonetheless it is the right power to state when making this point.

Because PETSc objects are generally distributed across, and accessible from, multiple MPI processes, the first argument of an `ObjectCreate()` method is an MPI communicator ("`COMM`"). We will usually use `PETSC_COMM_WORLD`; it is the communicator formed from all `N` processes when we start a run with "`mpiexec -n N`." Because they are "collective" operations [Gropp et al., 1999], all processes in `COMM` must call the `ObjectCreate()` and `ObjectDestroy()` methods.

Assembly and parallel layout of `Vecs` *and* `Mats`

A `Vec` or `Mat` stores its entries in parallel across all the processes in the MPI communicator used when creating it. For example, the create-assemble sequence of a `Vec` with four entries might look like this:

```
Vec x;
PetscInt   i[4] = {0, 1, 2, 3};
PetscReal  v[4] = {11.0, 7.0, 5.0, 3.0};

VecCreate(COMM,&x);
VecSetSizes(x,PETSC_DECIDE,4);
VecSetFromOptions(x);
VecSetValues(x,4,i,v,INSERT_VALUES);
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

* In the Sameway that PETSc Real is used to normally indicate double precision values PetscInt is used to normally represent ~~ints~~ 32 bit int values. See the discussin in chapter 7 on PetscInt + PetscReal for large and very illconditioned Problems

The four entries of `Vec x` are set by the call to `VecSetValues()`, putting values from array `v` at the indices given by `i`. The operation of setting values in `x` may require communication between processes, however, because entries which are to be stored on one process could be set by another process. Such communication occurs between the `VecAssemblyBegin()` and `VecAssemblyEnd()` commands.

The reader is allowed to think of a PETSc `Vec` as a one-dimensional C array with its contents split across the processes in the MPI communicator used in the `VecCreate()` command. For example, if the above code appears in `mycode.c`, and if it is run sequentially on one process, i.e. as

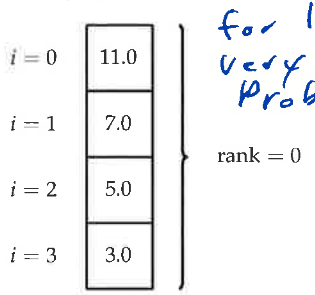


Figure 2.1: A sequential `Vec` layout, all on rank = 0 process.

```
$ ./mycode.c
```

then, at the end of the above create-set-assemble sequence, the storage of `x` looks like Figure 2.1. However, if run as

```
$ mpiexec -n 2 ./mycode.c
```

then the layout looks like Figure 2.2. In this case the argument PETSC_DECIDE in VecSetSizes() is active, and PETSc's decision will be to put the first two entries of x on the rank 0 process and the other two on the rank 1 process.

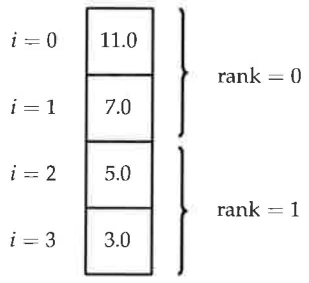


Figure 2.2: A parallel Vec layout on two processes. Because we call "VecSetSizes(x,PETSC_DECIDE,4)", PETSc decides to split the storage in the middle.

PETSc Mat objects are comparable to Vecs, but they are not merely 2D C arrays even in serial (i.e. in one-process runs). Compared to Vecs they require additional choices regarding parallel distribution and storage formats. Though this is hidden inside the implementation of Mat, the most common storage format is *parallel compressed sparse row storage*, what PETSc calls the MATMPIAIJ type. In this type a range of rows is owned by each process (parallel row storage), within each owned range of rows only the specifically-allocated entries are stored (sparse), and these nonzero entries are stored contiguously in memory using an additional index array (compressed). The "specifically-allocated" entries are generally the nonzero entries, but they are always referred-to as "nonzero entries" in sparse representations even though they occasionally have zero values.

Mat objects are linear operators and their major "purpose" is to multiply Vecs. Of course, the result Vec from a Mat-Vec product is a linear combination of the columns of the Mat. Thus, in practice, parallel row storage of the Mat means these things:

- PETSc internally distributes the rows of the Mat A the same way as the entries of the intended *output* (i.e. column) Vec. Thus if $Ax = b$ for some x then row i of A is on the rank m processor if and if entry i of b is on the rank m processor.[4]
- Before PETSc computes a Mat-Vec product, PETSc communicates ("scatters") ~~the whole~~ Vec to each process.

  the needed portions of Vec
- After the scatter the Mat-Vec product is a local operation, requiring no further communication.

One doesn't really need to know all this to assemble a matrix. For example, here is one way to create and assemble a 4 × 4 Mat object one row at a time:

```
Mat A;
PetscInt  i, j[4] = {0, 1, 2, 3};
PetscReal v[4];

MatCreate(PETSC_COMM_WORLD,&A);
MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
MatSetFromOptions(A);
MatSetUp(A);

i = 0;  v[0] = 1.0;  v[1] = 2.0;  v[2] = 3.0;
MatSetValues(A,1,&i,3,j,v,INSERT_VALUES);
```

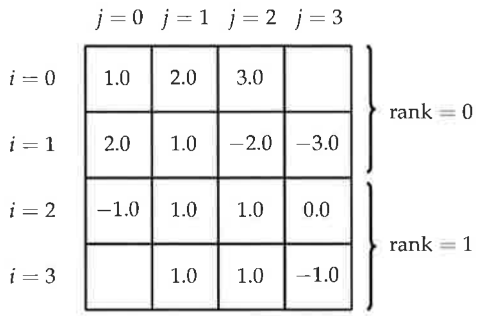
[4] This is the outcome when PETSC_DECIDE is used in setting both the Vec and Mat sizes and they are of the same dimension.

```
i = 1;  v[0] = 2.0;  v[1] = 1.0;  v[2] = -2.0;  v[3] = -3.0;
MatSetValues(A,1,&i,4,j,v,INSERT_VALUES);
i = 2;  v[0] = -1.0;  v[1] = 1.0;  v[2] = 1.0;  v[3] = 0.0;
MatSetValues(A,1,&i,4,j,v,INSERT_VALUES);
j[0] = 1;  j[1] = 2;  j[2] = 3;
i = 3;  v[0] = 1.0;  v[1] = 1.0;  v[2] = -1.0;
MatSetValues(A,1,&i,3,j,v,INSERT_VALUES);

MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

indicate

The method `MatSetValues()` sets multiple values, in this case a row. The "`1,&i`" arguments say that we are setting one row with global index `i`. The "`3,j`" or "`4,j`" arguments say that integer array `j` has the 3 or 4 global column indices.



	$j=0$	$j=1$	$j=2$	$j=3$	
$i=0$	1.0	2.0	3.0		rank = 0
$i=1$	2.0	1.0	−2.0	−3.0	
$i=2$	−1.0	1.0	1.0	0.0	rank = 1
$i=3$		1.0	1.0	−1.0	

Figure 2.3: A parallel `Mat` layout on two processes. Blank entries are not allocated.

If the above lines appeared in `mycode.c`, and if it were run

```
$ mpiexec -n 2 ./mycode
```

then the layout would be as in Figure 2.3.

PETSc can show us the entries in the `Mat` in different formats at runtime:

```
$ ./mycode -mat_view
Mat Object: 1 MPI processes
  type: seqaij
row 0: (0, 1)  (1, 2)  (2, 3)
row 1: (0, 2)  (1, 1)  (2, -2)  (3, -3)
row 2: (0, -1)  (1, 1)  (2, 1)  (3, 0)
row 3: (1, 1)  (2, 1)  (3, -1)
$ ./mycode -mat_view ::ascii_dense
Mat Object: 1 MPI processes
  type: seqaij
 1.00000e+00  2.00000e+00  3.00000e+00  0.00000e+00
 2.00000e+00  1.00000e+00  -2.00000e+00  -3.00000e+00
 -1.00000e+00  1.00000e+00  1.00000e+00  0.00000e+00
 0.00000e+00  1.00000e+00  1.00000e+00  -1.00000e+00
```

binary:a.dat

The first view shows the compressed sparse storage, with values as pairs with column index and value. The second view is a traditional ("dense") display where all zero values are shown, whether allocated or not. Other possibilities for outputting a `Mat`, not shown, include "`-mat_view ::ascii_matlab`," which dumps in Matlab's text format, and "`-mat_view ~~binary -viewer_binary_filename a.dat~~`" which saves to file `a.dat` in PETSc's scalable binary format. In any case, `-mat_view` output happens at the completion of the `MatAssemblyBegin/End()` calls.

In the last two cases shown above, the matrix was stored in *serial* compressed sparse row format, the `MATSEQAIJ` type, because of the one-process run. If the code is run in parallel, i.e. by `mpiexec -n N ./mycode`, then `-mat_view` reports `type: mpiaij` corresponding to `Mat` type `MATMPIAIJ`, the "parallel compressed sparse row storage" described above.

Numerical linear algebra 1: residual and iterations

Direct methods like Gauss elimination [Trefethen and Bau, 1997] are one way to solve a linear system. The most powerful methods for ~~our later PDE-solving uses~~ PDEs, however, are iterative. ~~They use the *residual* of some approximation of the solution to the linear system~~.

By definition, the residual of $\mathbf{u}_0$ in linear system (2.1) is the vector

$$\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0. \tag{2.3}$$

Evaluating the residual for a known vector $\mathbf{u}_0$ requires only applying A to it, an $O(N^2)$ operation at worst. Because most discretization schemes for PDEs generate matrices A that are *sparse*, with many more zero entries than nonzeros, and because often the number of nonzeros per row is typically small and independent of N, the operation $A\mathbf{u}_0$ can often be done in $O(N)$ operations.

The *Richardson iteration*,[5] simply adds a multiple ω of the last residual at each step,

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \omega(\mathbf{b} - A\mathbf{u}_k). \tag{2.4}$$

If significantly fewer than $O(N^2)$ steps were needed to make $\mathbf{u}_k$ an adequate approximation of the exact solution $\mathbf{u}$, then the Richardson iteration could improve on Gauss elimination. ~~On the other hand,~~ In most cases the Richardson iteration may not converge, as in the next Example.

Example. Consider the linear system

$$A\mathbf{u} = \begin{bmatrix} 10 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \end{bmatrix} = \mathbf{b} \tag{2.5}$$

which has solution $\mathbf{u} = [1\ 2]^\top$. If we start with estimate $\mathbf{u}_0 = [0\ 0]^\top$ then the $\omega = 1$ Richardson iteration (2.4) gives a sequence of vectors

$$\mathbf{u}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{u}_1 = \begin{bmatrix} 8 \\ 1 \end{bmatrix}, \mathbf{u}_2 = \begin{bmatrix} -63 \\ 9 \end{bmatrix}, \mathbf{u}_3 = \begin{bmatrix} 584 \\ -62 \end{bmatrix}, \dots \tag{2.6}$$

This sequence is not heading toward the solution.

---

[5] Also called *simple iteration* [Greenbaum, 1997].

If we rewrite (2.4) as

$$\mathbf{u}_{k+1} = (I - \omega A)\mathbf{u}_k + \omega \mathbf{b} \tag{2.7}$$

then it is easy to believe that the "size" of the matrix $I - \omega A$ will determine whether $\lim_{k\to\infty} \mathbf{u}_k$ exists. To make this precise we recall important definitions.

A complex number[6] $\lambda \in \mathbb{C}$ is an *eigenvalue* of a square matrix $B \in \mathbb{R}^{N\times N}$ if there is a nonzero vector $\mathbf{v} \in \mathbb{C}^N$ so that $B\mathbf{v} = \lambda\mathbf{v}$. The set of all eigenvalues of B is the *spectrum* $\sigma(B)$ of B. The *spectral radius* $\rho(B)$ is the maximum magnitude of the eigenvalues of B. The *singular values* are the square roots of the eigenvalues of the matrix $B^T B$, a symmetric and positive-definite matrix with nonnegative eigenvalues. Singular values are geometrically-defined as the lengths of semi-axes of the (hyper-)ellipsoid in $\mathbb{R}^N$ that results from applying B to all vectors in the unit (hyper-)sphere of $\mathbb{R}^N$ [Trefethen and Bau, 1997].

Properties of a matrix described in terms of its eigenvalues or singular values are generically called "spectral properties." For example, $\|B\|_2$ is the largest singular value of B and $\|B^{-1}\|_2$ is the inverse of the smallest singular value of B, so these are spectral properties. The 2-norm condition number $\kappa(B) = \|B\|_2/\|B^{-1}\|_2$ is thus also a spectral property; it is visualized as the eccentricity of the ellipsoid above.

Exercise 2.2 asks you to show that the Richardson iteration (2.4) or (2.7) will converge if and only if all the eigenvalues of $B = I - \omega A$ are inside the unit circle:

$$(2.4) \text{ converges if and only if } \rho(I - \omega A) < 1. \tag{2.8}$$

One can also show that $\rho(B) \le \|B\|$ in any induced matrix norm, so that (2.4) converges if $\|I - \omega A\| < 1$.

Numerical linear algebra 2: preconditioning

Considering spectral properties brings us to an important general observation about linear systems. Namely, that there are many systems which are equivalent to (2.1). In fact, if $M \in \mathbb{R}^{N\times N}$ is an invertible matrix then the systems

$$(M^{-1}A)\mathbf{u} = M^{-1}\mathbf{b} \tag{2.9}$$

and

$$(AM^{-1})(M\mathbf{u}) = \mathbf{b} \tag{2.10}$$

have exactly the same set of solutions as (2.1). However, matrices $M^{-1}A$ and AM^{-1} generally have different eigenvalues, condition numbers, and so on—different spectral properties—from A.

[6] Though B is real, λ may be complex. Also, if λ is an eigenvalue of a real matrix B then so is its complex conjugate $\bar{\lambda}$.

Note since B is always real in the book you can just use B^T and don't need B conjugate transpose

While the accuracy of the approximate solution to (2.1) cannot be improved beyond the $\kappa(A)\epsilon$ level by switching to systems (2.9) or (2.10), as fact i) on page 17 cannot be bypassed, ~~methods~~ algorithms may take advantage of better spectral properties of $M^{-1}A$ or AM^{-1}, compared to those of A, to generate approximations to the solution more quickly. This idea is effective if M^{-1} is easy to apply in the sense that (substantially) fewer operations are needed to solve the system $M\mathbf{v} = \mathbf{c}$ than to solve the original system.

Systems (2.9) and (2.10) are referred to as *preconditioned* versions of (2.1), with (2.9) called *left-preconditioning* and (2.10) *right-preconditioning*. The next example shows how preconditioning can make the Richardson iteration converge. In this case the diagonal of A is used as M, an example of *Jacobi preconditioning*.

Example, continued. Suppose we extract the diagonal of A from (2.5):

$$M = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}. \qquad (2.11)$$

Being diagonal, M is easy to invert and apply. The preconditioned Richardson iteration using M, namely

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \omega(M^{-1}\mathbf{b} - M^{-1}A\mathbf{u}_k), \qquad (2.12)$$

is better behaved. With $\mathbf{u}_0 = [0\ 0]^*$ and $\omega = 1$ again we ~~get this~~ obtain a sequence from (2.12):

$$\mathbf{u}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{u}_1 = \begin{bmatrix} 0.8 \\ 1.0 \end{bmatrix}, \mathbf{u}_2 = \begin{bmatrix} 0.9 \\ 1.8 \end{bmatrix}, \mathbf{u}_3 = \begin{bmatrix} 0.98 \\ 1.90 \end{bmatrix}, \dots \qquad (2.13)$$

This sequence is apparently ~~going~~ converging to $\mathbf{u} = [1\ 2]^*$. The explanation is not hard to see; compare

$$\rho(I - A) = -9.1 \quad \text{and} \quad \rho(I - M^{-1}A) = 0.32, \qquad (2.14)$$

and recall (2.8).

Intuitively speaking, the vector norm of a residual $\|\mathbf{r}_0\| = \|\mathbf{b} - A\mathbf{u}_0\|$ measures how "wrong" is $\mathbf{u}_0$ as an approximation to $\mathbf{u} = A^{-1}\mathbf{b}$. This idea is incomplete in two senses, as follows.

First, we surely would want the norm of the *error* itself, i.e.

$$\mathbf{e}_0 = \mathbf{u}_0 - \mathbf{u} \qquad (2.15)$$

as the direct measure of how wrong $\mathbf{u}_0$ is. However, exact knowledge of the error $\mathbf{e}_0$ is equivalent to exact knowledge of the solution $\mathbf{u}$

itself~~, if we already have the approximation $\mathbf{u}$~~$_0$. Thus only bounds on $\|\mathbf{e}_0\|$ might be reasonably be expected, even though we can compute the full residual vector $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$ at will.

Secondly, errors are most meaningful if they are relative. For instance, knowing "$\|\mathbf{e}_0\| \le 10^{-6}$" does not tell us that $\mathbf{u}_0$ is an accurate solution to the system $A\mathbf{u} = \mathbf{b}$ if $\|A^{-1}\| = 1$ and $\|\mathbf{b}\| = 10^{-6}$, in which case we would know in advance that $\|\mathbf{u}\| \le 10^{-6}$.

These ideas both relate to the conditioning of A. Indeed, the connection between the relative norm we *can* compute, namely $\|\mathbf{r}_0\| / \|\mathbf{b}\|$, and the relative norm we *want* is well-known:

$$\kappa(A)^{-1} \frac{\|\mathbf{r}_0\|}{\|\mathbf{b}\|} \le \frac{\|\mathbf{e}_0\|}{\|\mathbf{u}\|} \le \kappa(A) \frac{\|\mathbf{r}_0\|}{\|\mathbf{b}\|}, \tag{2.16}$$

where $\kappa(A) = \|A\| \|A^{-1}\|$ is the condition number in the induced norm. Proving (2.16) is Exercise 2.3; it requires only straightforward manipulations with norms.

Numerical linear algebra 3: Krylov space methods

The most powerful iterative methods for solving system (2.1) generate optimal[7] estimates $\mathbf{u}_k$ which are linear combinations of vectors $\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \dots, A^{k-1}\mathbf{v}$. Here $\mathbf{v}$ is a fixed vector; often $\mathbf{v} = \mathbf{b}$ or is the initial residual.

These methods are collectively called *Krylov space methods* because the span of such vectors is a *Krylov space*. Examples include the Richardson iteration, conjugate gradient (CG), and minimum residual methods (MINRES or GMRES).[8] The effectiveness of a given Krylov method on a given system depends on the eigenvalues or singular values of the matrix A, i.e. on its spectral properties. Many Krylov space methods are ~~built into and~~ fully-supported by PETSc, and we will use them in ~~all~~ later Chapters.

To be concrete, for a square matrix $A \in \mathbb{R}^{N \times N}$ and a vector $\mathbf{v} \in \mathbb{R}^N$, a Krylov space is defined as

$$\mathcal{K}_n(A, \mathbf{v}) = \text{span}\{\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \dots, A^{n-1}\mathbf{v}\}. \tag{2.17}$$

We will use just "$\mathcal{K}_n$" when the context is clear. Suppose $\tilde{\mathbf{u}} \in \mathcal{K}_n$. Then

$$\tilde{\mathbf{u}} = c_0\mathbf{v} + c_1 A\mathbf{v} + c_2 A^2\mathbf{v} + \dots + c_{n-1}A^{n-1}\mathbf{v}$$

for some coefficients c_j, or equivalently

$$\tilde{\mathbf{u}} = p_{n-1}(A)\mathbf{v}$$

for the $n-1$ degree polynomial $p_{n-1}(x) = c_0 + c_1 x + \dots + c_{n-1}x^{n-1}$ applied to A.

[7] In various senses. See Table 2.1 below for two such.

[8] We will address these methods in this and the next chapter. See Greenbaum [1997] or Saad [2003] for details, algorithms, and many more methods.

Thus if we want $\tilde{\mathbf{u}}$ to approximate $\mathbf{u}$, the solution to (2.1), then we want to find a polynomial p_{n-1} so that

$$p_{n-1}(A) \approx A^{-1}, \tag{2.18}$$

at least in the action of applying $p_{n-1}(A)$ to $\mathbf{v}$. Krylov space methods do this.

One can prove that if $p_{n-1}(z)$ is close to $1/z$ on the finite set of eigenvalues of A, i.e. on the spectrum $\sigma(A)$ in the complex plane, then (2.18) follows. Thus, whether p_{n-1} is a "good" polynomial for approximately inverting A is a spectral question about A, and $p_{n-1}(z) \approx 1/z$ is by no means required on large domains (open sets) of the complex plane.

The construction of a "good" p_{n-1} would be a question of approximation theory on finite subsets of the complex plane if only the spectrum $\sigma(A)$ were known, but in fact that is asking too much. We typically know A either through its entries or action on vectors, and we might have general spectral information about A through the context in which it was generated.[9] Accurately computing the full spectrum $\sigma(A)$, however, is at least as difficult as solving a linear system $A\mathbf{u} = \mathbf{b}$.

For an example in which a polynomial is generated during a Krylov iteration, consider again the Richardson iteration (2.4) with $\omega = 1$, namely $\mathbf{u}_{k+1} = \mathbf{u}_k + (\mathbf{b} - A\mathbf{u}_k)$. A straightforward calculation starting with $\mathbf{u}_0 = \mathbf{b}$ shows

$$\mathbf{u}_k = q_k(A)\mathbf{b} \quad \text{with} \quad q_{k+1}(x) = 1 + (1-x)q_k(x) \tag{2.19}$$

and $q_0(x) = 1$. Figure 2.4 shows these polynomials. It is easy to ~~check~~ that $q_k(x) \to 1/x$ for $x \in (0,2)$, and the Figure suggests it ~~too.~~ On the other hand, (2.8) says $q_k(A)\mathbf{b} \to A^{-1}\mathbf{b}$ if the spectral condition $\rho(I - A) < 1$ holds. The latter condition says exactly that all eigenvalues of A must be within distance one of $1 + 0i$ in the complex plane, and thus that all real eigenvalues of A must be in $(0,2)$.

verify
as well

Though the Richardson iteration generates $\mathbf{u}_k$ which are from a Krylov space, and thus $\mathbf{u}_k$ are polynomials in A applied to $\mathbf{b}$, they are by no means the best such approximations. We now give a "gloss" of two other well-known Krylov methods. They generate the same kind of polynomial approximations, but for polynomials p_n which minimize vector norms over a Krylov space.

Define $\mathcal{P}_n^1$ to be the set of all real polynomials p of degree n such that $p(0) = 1$. For a real square matrix $A \in \mathbb{R}^{N \times N}$ and $\mathbf{v} \in \mathbb{R}^N$, define the affine space (set)

$$\mathcal{K}_n^1(A, \mathbf{v}) = \mathbf{v} + \operatorname{span}\{A\mathbf{v}, A^2\mathbf{v}, \dots A^{n-1}\mathbf{v}\}. \tag{2.20}$$

[9] As the discretization of a PDE with known spectral properties, for example.

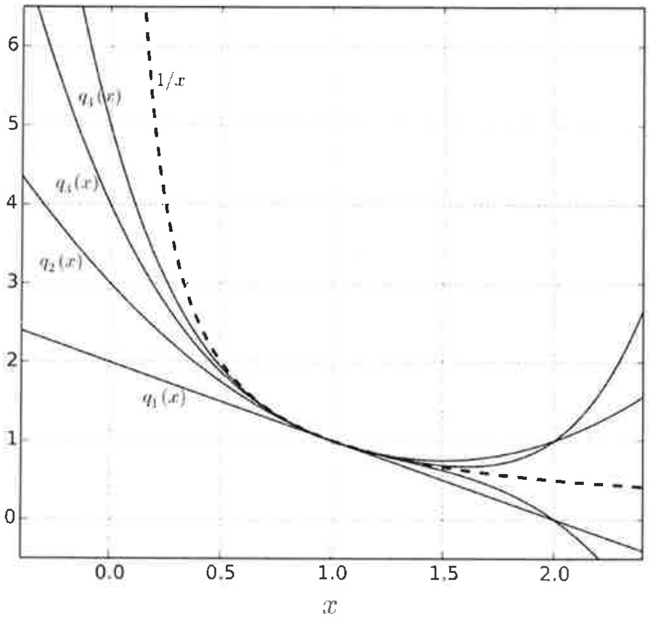


Figure 2.4: The $\omega = 1$ case of the Richardson iteration (2.4) approximates $\mathbf{u} = A^{-1}\mathbf{b}$ by $\mathbf{u}_k = q_k(A)\mathbf{b}$ for polynomials $q_k(x)$ which approximate $1/x$ on the interval $(0, 2)$.

~~In these terms,~~ Table 2.1 compares three Krylov methods, namely the Richardson iteration (2.4), conjugate gradients (CG), and generalized minimum residuals (GMRES). For CG and GMRES as algorithms, see Greenbaum [1997] or Saad [2003].

In practice, GMRES uses "restarts" to avoid filling memory with its internal representation of the Krylov space. (This mechanism stops practical GMRES from being a "pure" Krylov method.) The PETSc default is to restart after 30 iterations, so GMRES uses $\sim 30N$ memory locations on an N-dimensional problem, compared to $3N$ for CG and $2N$ for Richardson.

The classical *Jacobi* and *Gauss-Seidel iterative methods* [Greenbaum, 1997] are not Krylov space methods as they stand. They involve extracting parts of (entries of) A, as a matrix, which cannot be done by operations $A^k\mathbf{v}$. These classical iterations can, however, be regarded as preconditioned forms of the Richardson iteration [Greenbaum, 1997]; see Exercise 2.1.

We get access to the many Krylov methods implemented in PETSc any time we create a `KSP` object. Which method is used can be controlled at runtime, as long as we call the `KSPSetFromOptions()` method (below)

Runtime experimentation with Krylov methods is always appropriate. For the Poisson problem in Chapter 3, for example, preconditioned CG is effective, though we will see that the most emphasis should be put on the preconditioning stage.

NAME	*SYMMETRY*	*OPTIMALITY*	*SPECTRAL CONDITION*
Richardson	any	none	$\rho(I-\omega A) \ll 1$
CG	symmetric positive-definite	$\mathbf{e}_k$ minimizes $f(\mathbf{v}) = \langle \mathbf{v}, A\mathbf{v}\rangle$ over $\mathbf{v} \in \mathcal{K}_k^1(A, \mathbf{e}_0)$	exist $p_n \in \mathcal{P}_n^1$ small on $\sigma(A) \subset (0,\infty)$
GMRES	any	$\mathbf{r}_k$ minimizes $f(\mathbf{v}) = \|\mathbf{v}\|_2$ over $\mathbf{v} \in \mathcal{K}_k^1(A, \mathbf{b})$	exist $p_n \in \mathcal{P}_n^1$ small on $\sigma(A) \subset \mathbb{C}$

Table 2.1: Comparison of three well-known Krylov methods for the problem $A\mathbf{u} = \mathbf{b}$. "Symmetry" means required properties of A to apply the method. "Spectral condition" informally states spectral properties of A which lead to rapid convergence. Recall $\mathbf{r}_k = \mathbf{b} - A\mathbf{u}_k$ and $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$.

Our brief introduction of iterative linear algebra is now ~~done~~, though it is hardly adequate, and we return to PETSc codes and concrete linear systems. In the upcoming material the reader can initially treat PETSc's KSP linear solver object as a black box, but then explore how it works through runtime options, recalling the above material as needed.

complete

A small linear system

We already know how to create, fill, and destroy Vec and Mat objects. Code 2.1 shows vecmatksp.c which does these steps for the linear system

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 3 \end{bmatrix}. \tag{2.21}$$

A KSP object solves the linear system, with the specific method chosen at runtime. It has the expected Create/SetFromOptions/Destroy sequence. In addition, at the set-up stage for the linear system we tell the KSP about the matrix via the command

```
KSPSetOperators(ksp,A,A);
```

Why do we list A twice in calling KSPSetOperators()? The reason is that at runtime we generally choose a preconditioning method which builds M^{-1} in equation (2.9) or (2.10) from A or from an approximation of A. For example, *incomplete LU* factorization of A ("ILU(0)") can be used in generating M^{-1}, or M could be the diagonal of A as in the example on page 25. The second matrix argument to KSPSetOperators() is this Mat from which M^{-1} is built.

- The default `PC` is `bjacobi`, i.e. application of approximate diagonal-block inverses as M^{-1}.

- Inside the `PC` is a `sub_pc` object, also a `PC`, which is ILU(0),[11] and this generates the approximate diagonal-block inverses.

The reader can confirm the situation by running

```
mpiexec -n 2 ./vecmatksp -ksp_view
```

Surely that's enough runtime options for now. Of course there will be more, especially in later Chapters when we solve PDEs.

A sparse system of arbitrary size

We have seen how PETSc code sets up and solves linear systems, but there is more to say. The next example `tri.c`, split into Codes 2.2 and 2.3, introduces the following additional concepts and associated function calls:

switch to bullets, see Note on page 40. these i, ii conflict with other use in this chapter

i) Creating an integer option by `PetscOptionsXXX()` calls, so that the size of the linear system can be controlled at runtime.

ii) Using `VecDuplicate()` for allocation.

iii) Assembling a system of arbitrary size across an arbitrary number of processes, using `MatGetOwnershipRange()` to only set locally-owned rows.

iv) Using `VecAXPY()`, `VecNorm()`, and `PetscPrintf()` to compute and display the numerical error in a case where the exact solution is known.

First we build and run the code:

```
$ make tri
$ ./tri -ksp_monitor -a_mat_view ::ascii_dense
Mat Object:(a_) 1 MPI processes
  type: seqaij
  3.00000e+00  -1.00000e+00   0.00000e+00   0.00000e+00
 -1.00000e+00   3.00000e+00  -1.00000e+00   0.00000e+00
  0.00000e+00  -1.00000e+00   3.00000e+00  -1.00000e+00
  0.00000e+00   0.00000e+00  -1.00000e+00   3.00000e+00
  0 KSP Residual norm 3.302822756884e+00
  1 KSP Residual norm 5.519370044893e-16
error for m = 4 system is |x-xexact|_2 = 5.1e-16
```

[11] There is even a `sub_ksp` object, but it is `preonly`.

Next we assemble the matrix A. This is a boring tridiagonal matrix with 3 on the diagonal and -1 in the super- and sub-diagonals. Though boring, we want to assemble it efficiently in parallel, something that will be important when solving 2D and 3D PDEs in later chapters. However, only when `tri.c` is run do we know how many processes are in use. The method `MatGetOwnershipRange()` tells our program, running on a particular process (rank), what rows it owns locally. In the case of a many structured matrices like this one, we can avoid all interprocess communication by assembling exactly the rows we own. As seen at the top of Code 2.3, we call

```
MatGetOwnershipRange(A,&Istart,&Iend)
```

to ~~get~~ obtain the starting and ending row indices for the local process. These are used as limits in a `for` loop over the locally own rows. We use `MatSetValues()` to actually set the entries of A and `MatAssemblyBegin/End()` to complete the assembly of A.

We need to assemble the right-hand side $\mathbf{b}$ of the linear system and also the exact solution $\mathbf{x}_{\text{exact}}$ to the linear system ($A\mathbf{x}_{\text{exact}} = \mathbf{b}$). The easiest way, for demonstration purposes here, is to *choose* $\mathbf{x}_{\text{exact}}$ and then compute $\mathbf{b}$ by multiplying by A. Thus, we set (unimportant) values for `xexact`, and call `VecAssemblyBegin/End()` on it. Then we compute `b` by calling `MatMult(A,xexact,b)`.

As in `vecmatksp.c` ~~before~~ we set up the `KSP` and then call `KSPSolve()` to approximately solve $A\mathbf{x} = \mathbf{b}$. Option `-ksp_monitor` prints the residual norm $\|\mathbf{b} - A\mathbf{x}_k\|_2$ at runtime. In this case we also want to see that the actual error

$$\|\mathbf{x} - \mathbf{x}_{\text{exact}}\|_2$$

is small when the `KSP` completes its work. So, after getting `x` from `KSPSolve()` we compute the error with two commands,

$$\begin{array}{lcrcl} \texttt{VecAXPY(x,-1.0,xexact)} & : & \mathbf{x} & \leftarrow & -1\,\mathbf{x}_{\text{exact}} + \mathbf{x} \\ \texttt{VecNorm(x,NORM\_2,\&errnorm)} & : & \texttt{errnorm} & \leftarrow & \|\mathbf{x}\|_2. \end{array}$$

and then print `errnorm` by calling `PetscPrintf()`.

A first look at performance

The linear system assembled by `tri.c` is about as easy to solve as they get. It is tridiagonal, symmetric, diagonally-dominant, and positive definite.[12] So PETSc ought to be able to solve it quickly, almost all Krylov methods should apply, and parallelization ought to be effective.

We can time a one-process solution:

```
$ time ./tri -tri_m 10000
error for m = 10000 system is |x-xexact|_2 = 8.0e-13
```

[12] See Exercise 2.8.

KSP	PC	N=1 time (s)	N=4 time (s)
`preonly`	`lu`	`10.74`	
	`cholesky`	`5.84`	
`richardson`	`jacobi`	`13.48`	`5.45`
`gmres`	`none`	`9.99`	`5.30`
	`jacobi`	`10.23`	`4.49`
	`ilu`	`4.77`	
	`bjacobi+ilu`		`2.99`
`cg`	`none`	`7.22`	`3.18`
	`jacobi`	`7.49`	`3.31`
	`icc`	`4.81`	
	`bjacobi+icc`		`2.87`

Table 2.2: Times for `tri.c` to solve systems of dimension $m = 2 \times 10^7$. In this case the matrix is *tridiagonal*, *symmetric*, *diagonally-dominant*, and *positive definite*. All runs were on WORKSTATION (see page 38).

suggest use ICC(0) and IC(0) never since introduces unneeded notation.

`-sub_pc_type icc`, respectively. As already noted, the first of these is the parallel default, and the Table suggests why this is.

- Counting floating point operations shows the leading-order work for Cholesky is $m^3/3$ while for LU it is $2m^3/3$ [Trefethen and Bau, 1997]. This is reflected in the direct-solve results using `-ksp_type preonly`.

- A diagonally-dominant tridiagonal problem is very well-behaved for direct methods because no fill-in or pivoting occurs when they are applied, but the reader should not assume that generic $N = 10^7$ dimension systems are good candidates for direct solves.

- Cholesky and CG methods, namely KSP method `cg` and PC methods `cholesky` and `icc` (incomplete-Cholesky, also denoted "IC(0)"), respectively, apply to symmetric positive-definite matrices only. We see some benefit to using CG on one-process runs, compared to the general (non-symmetric) method GMRES, but the matrix here is so well-behaved that exploiting symmetry gives no noticable benefit.

swap to match what follows

- There is some speed-up from $N = 1$ to $N = 4$ processes on this single node 4-core WORKSTATION. The two (or slightly more) times speed-up is relatively uniform across methods. But apparently using four cores does not guarantee anywhere near four-times speed-up.

In future Chapters, "real" linear and non-linear systems will be generated by discretizing PDEs. Then we will create timing tables like Table 2.2, and re-consider the results.

It is worth noting at this point that PETSc can generate graphics showing convergence of the above iterative (Krylov) methods, at least if X11 windows are installed. The line graph in Figure 2.5, from

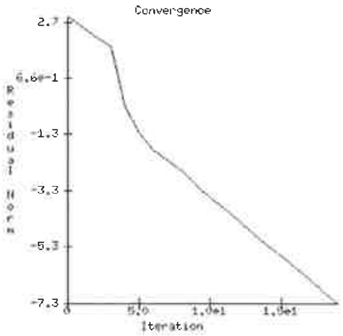


Figure 2.5: PETSc can use X windows to produce line graphs at run time. (This is not to say they are pretty.)

```
$ ./tri -tri_m 1000000 -ksp_rtol 1.0e-10 -pc_type jacobi \
    -ksp_monitor_lg_residualnorm -draw_pause 1
```

shows the residual norm logarithm versus the iteration number.

most?

Parallel preconditioning

There is one more "fact of life" to point out, again relating to numerical linear algebra. We can demonstrate it using `tri.c` runs:

```
$ mpiexec -n N ./tri -tri_m 100 -ksp_type cg -pc_type bjacobi \
    -sub_pc_type icc -ksp_converged_reason
```

Compare $N = 2$ and $N = 20$. We get convergence to the default tolerance in 3 and 5 iterations, respectively, and the final reported numerical error is quite different in the two cases.

In general, we observe there is a

iv) *processor-count dependence of preconditioning*: Many preconditioning schemes, including block-Jacobi and Schwarz methods (Chapter 7), act differently depending on the number of processors.

This seems to match the iii) on page 19 but note on page 34 you have a i) ... iv)

Note that this processor-dependence has a stronger influence on final answers than the other parallel-only "fact of life", namely that parallel reductions are not deterministic (Chapter 1).

The difference does not arise in the Krylov iteration implementation. Rather, the preconditioning matrices M^{-1} applied at each Krylov iteration are actually different matrices in the $N = 2$ and $N = 20$ cases. In particular, on N processes the block-Jacobi matrix is

$$M^{-1} = \begin{bmatrix} M_0^{-1} & & & \\ & M_1^{-1} & & \\ & & \ddots & \\ & & & M_{N-1}^{-1} \end{bmatrix} \qquad (2.22)$$

where each block M_i is the product of the incomplete ~~Chebys~~hev factors acting on those rows which are owned on the rank i process. For any nontrivial preconditioner, even one based on diagonal blocks of a tridiagonal matrix as here, different sets of rows "communicate" at the preconditioning stage depending on the processor count.

icc

What if you do not want this dependence? On the one hand, you can make the reported numerical error nearly processor-count-independent by asking that it be small (e.g. `-ksp_rtol 1.0e-14`), but then the difference in iteration count is even stronger (3 and 13 iterations on $N = 2$ and $N = 20$ processes, respectively). On the other hand, if you replace "`-pc_type bjacobi -sub_pc_type icc`" with "`-pc_type none`" or "`-pc_type jacobi`" then runs on $N = 2$ and

$N = 20$ processes all get identical numerical error results after 12 iterations.[14] These preconditioners do not communicate any information between rows at all, and thus they exhibit no dependence. However, *good* preconditioners, as we will see, cannot be that trivial.

Fundamentally, preconditioning is useful if can both be applied quickly and if it has a significant spectral effect. Quick parallel application requires avoiding inter-processor communication, but significant spectral effect requires using multi-row information from those rows owned by a processor. We will not fight against this situation, but we must be careful to evaluate parallel preconditioner effectiveness by measuring execution time, or execution time per degree of freedom, for example, and not only counting Krylov iterations.

Exercises

2.1 Suppose a square matrix A with nonzero diagonal entries is decomposed into diagonal and lower/upper triangular parts as $A = D + L + U$. Show that the Jacobi iteration $\mathbf{u}_{k+1} = D^{-1}(\mathbf{b} - L\mathbf{u}_k - U\mathbf{u}_k)$ is the same as the $\omega = 1$ Richardson iteration (2.4) applied to the left-preconditioned system (2.9) with $M = D$. Formulate and prove a corresponding statement about the Gauss-Seidel iteration $\mathbf{u}_{k+1} = (D + L)^{-1}(\mathbf{b} - U\mathbf{u}_k)$.

2.2 Show (2.8).

2.3 Show (2.16).

2.4 On page 27 we note that a Krylov space approximation $p_{n-1}(A)\mathbf{b}$ to the solution $A^{-1}\mathbf{b}$ is close if $p_{n-1}(z)$ is close to $1/z$ on the spectrum of A. Prove this for invertible diagonalizable matrices $A = S\Lambda S^{-1}$, with Λ diagonal, by showing

$$\|p_{n-1}(A) - A^{-1}\| \leq \kappa(S) \max_{\lambda \in \sigma(A)} |p_{n-1}(\lambda) - \lambda^{-1}|$$

where $\kappa(S) = \|S\|\|S^{-1}\|$ is the condition number.[15]

2.5 For the $\omega = 1$ Richardson iteration (2.4) and $\mathbf{u}_0 = \mathbf{b}$ we ~~get~~ $\mathbf{u}_k = q_k(A)\mathbf{b}$ for polynomials q_k given by (2.19). Show that if $\lim_{k\to\infty} q_k = q_\infty$ exists then $q_\infty(x) = 1/x$. On the other hand, by setting $y = 1 - x$ and defining $Q_k(y) = q_k(1 - y)$, show $Q_k(y)$ is the partial sum of a well-known series with a well-known radius of convergence.

2.6 Consider Richardson iteration in the example

```
./tri -tri_m 100 -ksp_monitor -ksp_type richardson
```

obtain

[14] I.e. all four runs show the same numerical error and iteration count.

[15] Here $\|\cdot\|$ is any induced matrix norm [Trefethen and Bau, 1997]. However, if A is *normal* then S can be chosen to be unitary, in which case $\kappa(S) = 1$ in the 2-norm.

NO CAP

~~U~~n-preconditioned Richardson iteration fails (i.e. add `-pc_type none`); explain. The default preconditioner succeeds (i.e. `-pc_type ilu`), but ILU(0) is cheating because it becomes a complete LU factorization on this tridiagonal and diagonally-dominant A; we are really seeing a direct solve. The same can be said for ICC(0). Confirm that, as in the example on page 25, Richardson iteration succeeds with `-pc_type jacobi`. Explain.

2.7 The accuracy of direct solves (e.g. `-ksp_type preonly -pc_type cholesky`) in `tri.c`, as measured by the reported error norm $\|\mathbf{x} - \mathbf{x}_{\text{exact}}\|_2$, decreases with increasing dimension. Confirm and explain.

2.8 Un-preconditioned GMRES solves the linear system in `tri.c` reasonably efficiently. We can explain this by asking PETSc to compute the eigenvalues of A by using option[16]

```
-ksp_compute_eigenvalues
```

Because otherwise it computes the eigenvalues of the preconditioned operator $M^{-1}A$, add `-pc_type none`. Try dimensions $N = 10, 100, 1000$. Why does the run

```
./tri -tri_n 1000 -pc_type none -ksp_compute_eigenvalues
```

only show 11 eigenvalues of this 1000×1000 matrix? How do these eigenvalues explain the good behavior of unpreconditioned GMRES?[17]

2.9 Table 2.2 includes a number of blanks. For each one, explain why it is blank, experimenting if needed.

2.10 Table 2.2 gives execution times not iteration count. Generate the corresponding table of `KSP` iteration count by adding option `-ksp_converged_reason` to the run commands. Note the large number of "coincidences" in iteration count, i.e. cases where iteration counts are identical; explain. Which preconditioners have a strong or weak effect on iteration count?

2.11 As the reader will undoubtedly experience, segmentation faults and memory leaks are inevitable when one develops PETSc codes. A standard tool for detecting/diagnosing these is `valgrind`.[18] We recommend using it. As an exercise, run

```
valgrind ./vecmatksp
```

to see what `valgrind` shows for a leak-free program. Then comment-out a `VecDestroy()` call in `vecmatksp.c` and rerun to see a common type of memory leak.

[16] The relevant PETSc manual page says this option is "intended only for assistance in understanding the convergence of iterative methods, not for eigenanalysis. For accurate computation of eigenvalues we recommend using the excellent package SLEPc." See `slepc.upv.es`.

[17] See [Trefethen and Bau, 1997] for help with both of these questions.

[18] See `valgrind.org`.

`da` output: the resulting distributed array object

In Code 3.1, the second and third arguments are `DM_BOUNDARY_NONE` because our Dirichlet boundary conditions do not need communication to the next process' domain, nor periodic wrapping. In the fourth argument we use `DMDA_STENCIL_STAR` because only cardinal neighbors of a grid point will be used when forming the matrix.[2] The two `PETSC_DECIDE` arguments which follow tell PETSc to distribute the grid over processes according to the size of (number of processes in) the MPI communicator using PETSc internal logic as illustrated above. The next two arguments, in the ninth and tenth positions, say that our PDE is scalar (`dof`= 1) and that the FD method will only need one neighbor in each direction (`s`= 1). The next two arguments after that are `NULL` because we are *not* telling PETSc any details about how to distribute processes over the grid; it `DECIDEs` for itself. Finally, the `DMDA` object is created as an output (i.e. pass-by-reference).

~~#~~ for scalar arguments one use ~~PETSC_DE~~ PETSC_DECIDE to have PETSc select the value for array/pointer argument one uses NULL.

The call to `DMDASetUniformCoordinates()` in Code 3.1 sets the domain to be $[0, 1] \times [0, 1]$ in the sense that the `DM` object knows the spacing and locations of the grid points. The last two arguments are ignored in this case; they would set limits on the third dimension if `da` were created with `DMDACreate3d()`.

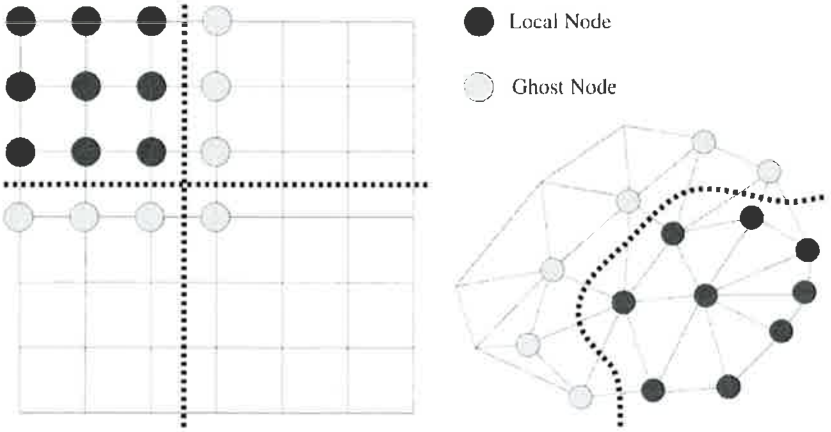


Figure 3.6: PETSc's parallel decomposition of structured (left) and unstructured (right) grids, showing owned ("local") and accessible ("ghost") nodes for one process.

The standard PETSc view of what `DM` s "look like" is in Figure 3.6. The code in Code 3.1 generates something like the left figure, namely a structured-grid `DM`, except that the one shown in Figure 3.6 has `DMDA_STENCIL_BOX` stencil type, unlike ours. On the right is an unstructured grid, of the type created in Chapter 8 for the finite element method. In both cases the Figure shows the nodes owned by a given process ("local" nodes) and those other nodes that are accessible by the local process ("ghost" nodes).

[2] Figure 3.7 below shows the "stencil" of our FD method.

Finite difference method

We were ~~trying~~ attempting to approximate PDE problem (3.1) and (3.2), not just build a grid. The following FD method leads to the next step~~,~~ of creating and assembling a `Mat` and `Vecs`, for the linear system corresponding to the PDE.

By a well-known Taylor's theorem argument [Morton and Mayers, 2005], if $F(x)$ is sufficiently smooth then

$$F''(x) = \frac{F(x+h) - 2F(x) + F(x-h)}{h^2} + O(h^2) \qquad (3.4)$$

as h goes to zero. This formula, applied to partial derivatives, will approximate the Laplacian in equation (3.1). In fact, if $u_{i,j}$ is the gridded approximation to the value $u(x_i, y_j)$ of the exact solution $u(x,y)$ at a grid point,[3] and if $f_{i,j} = f(x_i, y_j)$, then from (3.4) we have this FD approximation to equation (3.1):

$$-\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} - \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} = f_{i,j}. \qquad (3.5)$$

Equation (3.5) applies at all interior points, where $1 \le i \le M-2$ and $1 \le j \le N-2$. The boundary conditions (3.2) become

$$u_{0,j} = 0, \quad u_{M-1,j} = 0, \quad u_{i,0} = 0, \quad u_{i,N-1} = 0, \qquad (3.6)$$

for all i, j.

At grid location (x_i, y_j), equation (3.5) relates the unknown $u_{i,j}$ to its four cardinal neighbors $u_{i+1,j}$, $u_{i-1,j}$, $u_{i,j+1}$, and $u_{i,j-1}$. This pattern (Figure 3.7) is a *stencil*, in particular a "star" stencil. By contrast, a "box" stencil would additionally involve the four diagonal neighbors. In 2D, a star stencil relates five unknowns, while a box stencil relates nine.

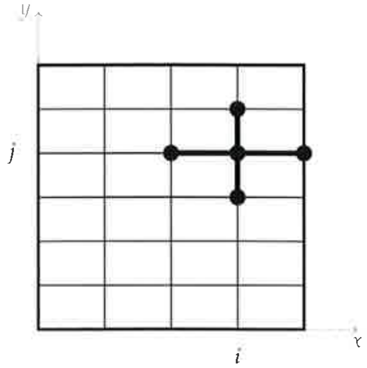


Figure 3.7: This "star" stencil simply illustrates FD scheme (3.5).

We will treat all values $u_{i,j}$ as unknowns, whether on the boundary or in the interior, so we have $L = MN$ unknowns. Equations (3.5) and (3.6) form a linear system of L equations,

$$A\mathbf{u} = \mathbf{b}, \qquad (3.7)$$

where A is a $L \times L$ matrix and $\mathbf{u}, \mathbf{b}$ are $L \times 1$ column vectors.

However, to show entries of A and $\mathbf{b}$ in linear system (3.7) we must globally-order the unknowns. Such an ordering is implemented inside a PETSc `DMDA`, and indeed our code (`poisson.c` below) will use only the grid-wise coordinates (i,j).[4] Here we expose the ordering for the purpose of displaying the system in matrix-vector form.

The ordering used in a one-process (serial) run by a 2D `DMDA` is shown in Figure 3.8. On an M by N grid one could write it as

$$U_k = u_{i,j} \quad \text{where} \quad k = jM + i \qquad (3.8)$$

[3] This is an important phrase! We compute values $u_{i,j}$ from the finite difference equations. We generally do not know the values $u(x_i, y_j)$.

[4] The ability to assemble `Mats` and `Vecs` with (i,j)-type indexing is one reason structured-grid codes using `DMDA` can be quite short.

for $i = 0, 1, \dots, M-1$ and $j = 0, 1, \dots, N-1$, so $k = 0, 1, \dots, MN-1$. PETSc does such index transformations inside the DMDA implementation.

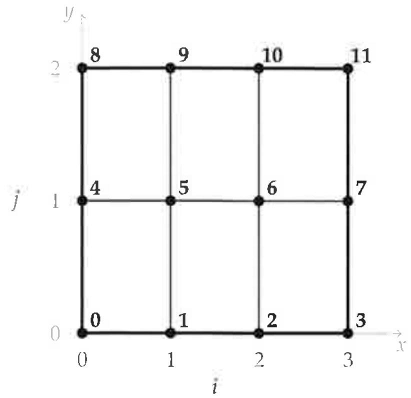


Figure 3.8: Ordering of unknowns (3.8) on a $M = 4$ and $N = 3$ grid. Index k from (3.8) is shown in **bold**.

Example. In the $M = 4$ and $N = 3$ case (Figure 3.8) we have grid spacing $h_x = 1/3$ and $h_y = 1/2$. Only the $k = 5$ and $k = 6$ equations are not boundary conditions (3.6). The linear system (3.7) is

$$\begin{bmatrix}
1 & & & & & & & & & & & \\
 & 1 & & & & & & & & & & \\
 & & 1 & & & & & & & & & \\
 & & & 1 & & & & & & & & \\
 & & & & 1 & & & & & & & \\
 & c & & & b & a & b & & & c & & \\
 & & c & & & b & a & b & & & c & \\
 & & & & & & & 1 & & & & \\
 & & & & & & & & 1 & & & \\
 & & & & & & & & & 1 & & \\
 & & & & & & & & & & 1 & \\
 & & & & & & & & & & & 1
\end{bmatrix}
\begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix}
=
\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ f_{1,1} \\ f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $a = 2/h_x^2 + 2/h_y^2 = 26$, $b = -1/h_x^2 = -9$ and $c = -1/h_y^2 = -4$.

The matrix A is not symmetric. Furthermore it is not well-scaled, for such a small example, because the 2-norm condition number is $\kappa(A) = \|A\|_2 \|A^{-1}\|_2 = 43.16$.

Before assembling system (3.5) and (3.6) by writing PETSc code, there are two nontrivial observations about it. These observations lead to an equivalent linear system that is easier to solve both in the sense that we have more options for solving the system,[5] and in the sense that the condition number is smaller.

First, equations (3.5) have very different "scaling" from equations (3.6). For example, if $M = N = 1001$, so that $h_x = h_y = 0.001$, then the coefficient of $u_{i,j}$ in (3.5) is $4/(.001)^2 = 4 \times 10^6$, while the coefficients from (3.6) are equal to 1. To make the equations better scaled, we multiply (3.5) by the grid cell area $h_x h_y$ to ~~get~~

obtain

$$2(a+b)u_{i,j} - a\left(u_{i+1,j} + u_{i-1,j}\right) - b\left(u_{i,j+1} + u_{i,j-1}\right) = h_x h_y f_{i,j} \qquad (3.9)$$

where $a = h_y/h_x$ and $b = h_x/h_y$. Using (3.9), all the equations in the system will have coefficients of comparable size, unless the cell aspect ratio h_y/h_x is very large or small. If $h_x = h_y$ then diagonal entries are 4 and off-diagonal entries are -1.

[5] More KSP and PC choices, and more that converge.

the rank 1 and 3 processes have `info.xs` = 3 and `info.xm` = 2. There are similar y ranges.

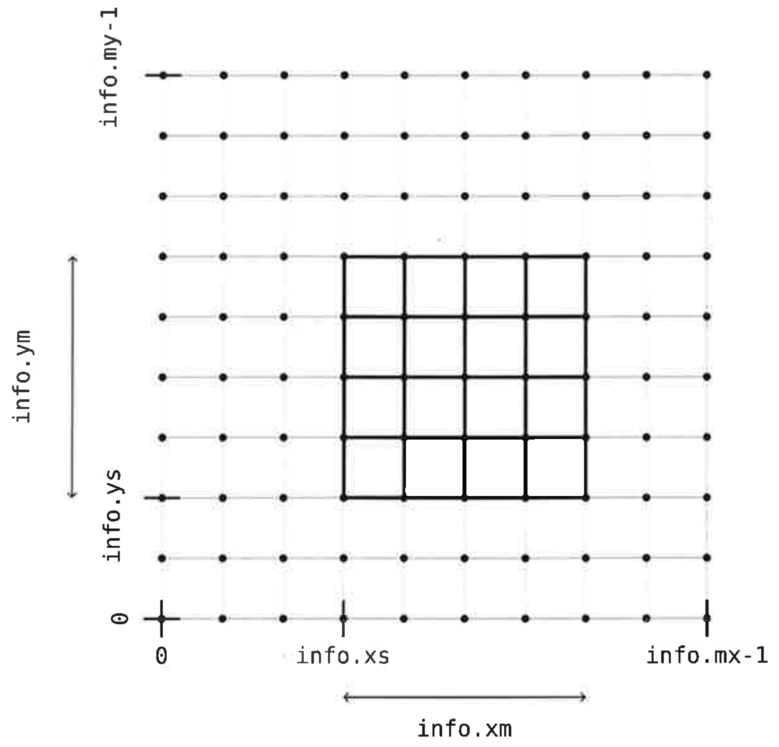


Figure 3.9: A `DMDALocalInfo` struct describes the indices for a local process' part of a 2D grid, plus the global grid size.

The local-ownership index ranges from `info` are used in the `for` loops which re-appear every time we do operations on a structured 2D grid:

```
for (j=info.ys; j<info.ys+info.ym; j++) {
  for (i=info.xs; i<info.xs+info.xm; i++) {
    DO SOMETHING AT GRID POINT (i,j)
  }
}
```

Still considering the code in Code 3.2, the `Mat` object A assembled by `formdirichletlaplacian()` has ranges of rows owned by each process, the standard parallel layout `MATMPIAIJ` of `Mat` objects in PETSc (Chapter 1). However, because we work with the locally-owned subgrid using (i, j) indices, we can forget the actual layout of a `Mat` and only focus on the part of the grid owned by the process, instead of worrying about the matrix itself.

ignore

In particular, local indices (i, j) can be used when inserting entries into `Mat A`, which is really a dynamical data structure for matrix assembly. Thus in Code 3.2 we see one use of `MatSetValuesStencil()`

for each locally-owned grid point. For a generic interior point this command inserts five coefficients into the matrix. The key data structure is of type `MatStencil`, an apparently-trivial struct

```
typedef struct {
  PetscInt k,j,i,c;
} MatStencil;
```

In our 2D case, with a single degree of freedom at each node,[7] we only use the `i` and `j` members of `MatStencil`. From (3.9), the actual matrix entries are $a_{i,i} = 2\left(h_y/h_x + h_x/h_y\right)$ on the diagonal, and $a_{i,j} = -h_y/h_x$ or $a_{i,j} = -h_x/h_y$ for off-diagonals.

A particular problem, an exact solution

At this point we need a specific Poisson problem so our example code can solve it. To do this we *choose*[8] an exact solution, taking care that it satisfies homogeneous Dirichlet boundary conditions ($u = 0$ along ∂S):

$$u(x,y) = (x^2 - x^4)(y^4 - y^2). \tag{3.10}$$

Then we merely differentiate to get $f = -\nabla^2 u$. Thus (3.10) solves (3.1) with right side

$$f(x,y) = 2(1-6x^2)y^2(1-y^2) + 2(1-6y^2)x^2(1-x^2). \tag{3.11}$$

From now on we will refer to u in (3.10) as "u_{ex}", the exact solution. This same problem and solution appears in Chapter 4 of [Briggs et al., 2000], so these formulas are not original.

Observe that the truncation error term $O(h^2)$ in equation (3.4) has a coefficient proportional to fourth derivatives [Morton and Mayers, 2005] so our FD method will not be exact on this problem. That is, u_{ex} has nonzero fourth derivatives. This is *good* for testing, and we would not want to use the simpler form $u(x,y) = (x - x^2)(y^2 - y)$, for example, to test convergence rate of the code, because we want the ~~decay~~ of numerical error with refining grids to be generic.

(which can be exactly represented with the finite difference discretization)

decrease

see ~~#~~ exercise 3.8

Code 3.3 shows how (3.10) is implemented as `formExact()` and how (3.11) is implemented as `formRHS()`. We will reuse these parts in Chapters 6 and 7.

[7] The Poisson equation (3.1) is a scalar PDE so the unknown at each grid point is the scalar $u_{i,j}$. A system of equations like Navier-Stokes would have `dof`> 1 when we call `DMDACreateXd()`, and the "c" member of `MatStencil` would get used.

[8] For the code `tri.c` in Chapter 2 we did something similar, i.e. choosing exact solution $\mathbf{u}$ before computing $\mathbf{b} = A\mathbf{u}$ by matrix multiplication.

DMDAVecRestoreArray(), and we explicitly ask for the Vec objects to be assembled by calling VecAssemblyBegin/End(), ~~just~~ as in Chapter 1.

ch3/poisson.c **part I**

```
static char help[] = "A structured-grid Poisson problem with DMDA+KSP.\n\n";

#include <petsc.h>
#include "structuredpoisson.h"

int main(int argc,char **args) {
  DM              da;
  Mat             A;
  Vec             b,u,uexact;
  KSP             ksp;
  PetscReal       errnorm;
  DMDALocalInfo   info;

  PetscInitialize(&argc,&args,(char*)0,help);

  // default size (9 x 9) can be changed using -da_grid_x M -da_grid_y N
  DMDACreate2d(PETSC_COMM_WORLD,
               DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
               -9,-9,PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,
               &da);
  DMDASetUniformCoordinates(da,0.0,1.0,0.0,1.0,-1.0,-1.0);

  // create linear system matrix A
  DMCreateMatrix(da,&A);
  MatSetOptionsPrefix(A,"a_");
  MatSetFromOptions(A);

  // create right-hand-side (RHS) b, approx solution u, exact solution uexact
  DMCreateGlobalVector(da,&b);
  VecDuplicate(b,&u);
  VecDuplicate(b,&uexact);

  // fill known vectors
  formExact(da,uexact);
  formRHS(da,b);

  // assemble linear system
  formdirichletlaplacian(da,1.0,A);
```

Code 3.4: Set up DM, Mat, and Vec objects, and assemble the linear system.

Solving the PDE

Code 3.4 shows how poisson.c creates the various objects needed to solve the Poisson problem, namely one DM, one Mat, and three Vecs. A DM object can compute matrix and vector sizes from the grid dimensions, so we call DMCreateMatrix() and DMCreateGlobalVector()

```
  1 KSP Residual norm 2.656923348626e-02
  2 KSP Residual norm 8.679141000397e-03
  3 KSP Residual norm 1.557150861763e-03
  4 KSP Residual norm 2.239919982542e-04
  5 KSP Residual norm 2.519822315367e-05
  6 KSP Residual norm 2.152764600588e-06
  7 KSP Residual norm 2.650467236964e-07
on 9 x 9 grid:  error |u-uexact|_inf = 0.000763959
```

Recall that a default 9 × 9 grid was chosen in calling DMDACreate2d(). We see seven iterations of the KSP method, a small final residual norm, and an apparently small numerical error. It is reasonable to think that we have solved the problem, but further inspection is definitely in order.

Options -da_grid_x M -da_grid_y N set the grid at runtime, and we can examine both the grid (i.e. the DM object) and our assembled Mat graphically to check these objects. If X11 ~~or other windowing~~ is correctly linked in your PETSc installation then

```
$ ./poisson -da_grid_x 5 -da_grid_y 7 -dm_view draw -draw_pause 5
```

gives Figure 3.10, same as the $M = 5, N = 7$ grid in Figure 3.2, including the global node ordering by formula (3.8). Options

```
$ ./poisson -da_grid_x 5 -da_grid_y 7 -a_mat_view draw -draw_pause 5
```

show a graphic similar to Figure 3.11. These views suggest that we have the right kind of matrix structure for the Poisson problem, namely a symmetric sparse matrix with tridiagonal blocks along the diagonal and a banded structure.

As a specific check on our matrix assembly, this sparse matrix view is easily checked to be identical to the matrix in the linear system on page 50:

```
$ ./poisson -da_grid_x 4 -da_grid_y 3 -a_mat_view
Mat Object:(a_) 1 MPI processes
  type: seqaij
row 0: (0, 1)  (1, 0)  (4, 0)
row 1: (0, 0)  (1, 1)  (2, 0)  (5, 0)
row 2: (1, 0)  (2, 1)  (3, 0)  (6, 0)
row 3: (2, 0)  (3, 1)  (7, 0)
row 4: (0, 0)  (4, 1)  (5, 0)  (8, 0)
row 5: (1, 0)  (4, 0)  (5, 4.33333)  (6, -1.5)  (9, 0)
row 6: (2, 0)  (5, -1.5)  (6, 4.33333)  (7, 0)  (10, 0)
row 7: (3, 0)  (6, 0)  (7, 1)  (11, 0)
row 8: (4, 0)  (8, 1)  (9, 0)
row 9: (5, 0)  (8, 0)  (9, 1)  (10, 0)
row 10: (6, 0)  (9, 0)  (10, 1)  (11, 0)
row 11: (7, 0)  (10, 0)  (11, 1)
on 4 x 3 grid:  error |u-uexact|_inf = 0.0085927
```

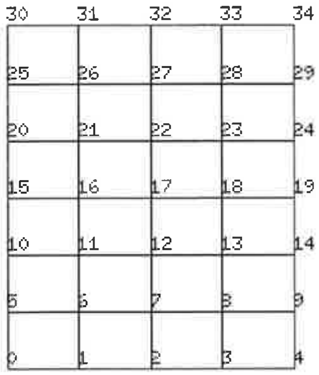


Figure 3.10: PETSc can show the structured-grid DMDA at runtime, here for a single-process run.

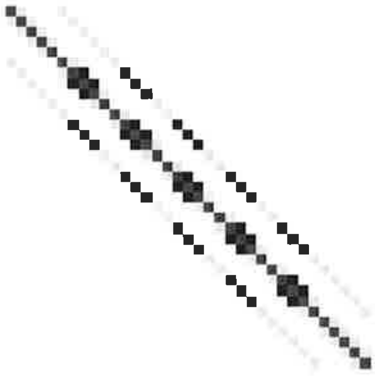


Figure 3.11: PETSc can show the matrix structure too. The actual graphic is in color, but here the two dark shades show positive and negative entries while the light shade shows allocated locations which are zero.

Additionally, a "movie" of the KSP iterates $\{\mathbf{u}_k\}$ comes from running

```
$ ./poisson -da_grid_x 100 -da_grid_y 100 -ksp_monitor_solution
```

(not shown). Alternatively, as in Chapter 2, a line graph of the (preconditioned) residual norm is from `-ksp_monitor_lg_residualnorm` (also not shown). These various viewing options should be used on under-development PDE-solving code on a structured grid.

There are two ways to specify grid refinement. One is to specify the grid dimensions directly as above, but the other is to have the DM refine the grid by factors of two. More precisely, the number of *subintervals* is increased by a factor of two. For example, this option replaces our default grid of 8 subintervals (i.e. 9 × 9 grid *points*) by 16 subintervals (i.e. 17 × 17 points) in each direction:

```
$ ./poisson -da_refine 1
on 17 x 17 grid:  error |u-uexact|_inf = 0.000196764
```

Choices of solver and parameters can be made at runtime, once we have adequately-described the problem in terms of PETSc objects. We have delayed questions of convergence and efficiency until runtime, but here is what we might want to know now:

- is our numerical method correctly implemented? (*convergence*)
- what is going on inside PETSc? (*exposure*)
- how to get high performance? (*efficiency*)

In the next three sections we address these in turn, with only a superficial stab at efficiency. Attaining efficiency requires a better understanding of this goal, and better choices for preconditioners, than we have now.

Convergence

We want to see the ~~numerical errors~~ error between our solution and the ~~Continuum~~ ~~Continuum~~ ~~problem~~ exact solution decrease as we refine the grid. Such decrease is expected because the finite differences become better approximations of the corresponding derivatives, but we want to *see* it, so we know our actual implementation is correct. The rate at which the error decreases should match what we expect from theory.

Recall that `poisson.c` already generates the max norm error (Code 3.5), and we know how to refine the grid by factors of two, so here is a bash loop refinement study:

```
$ for K in 0 1 2 3 4 5 6; do ./poisson -da_refine $K; done
on 9 x 9 grid:  error |u-uexact|_inf = 0.000763959
on 17 x 17 grid:  error |u-uexact|_inf = 0.000196764
```

ILU(0) as the preconditioner, while in parallel the defaults are GMRES and block Jacobi as a preconditioner, but where each diagonal block—there are four in the above parallel run—is preconditioned with ILU(0). In Chapter 2 we pointed out that this parallel preconditioner, like most of them, gives processor-count dependent results.

The serial PETSc defaults for KSP and PC give a certain timing and a certain number of iterations on a given grid:

```
$ timer ./poisson -da_refine 5 -ksp_converged_reason
Linear solve converged due to CONVERGED_RTOL iterations 506
on 257 x 257 grid:  error |u-uexact|_inf = 1.73086e-06
real 1.39
```

That is a lot of iterations, and it is not clear if this timing is fast or slow. Without thinking too hard, experimentation should show us better options. Table 3.1 below was generated by running

```
$ timer ./poisson -da_refine 5 -ksp_converged_reason -ksp_type KSP -pc_type PC
```

The iteration count is also worth noting when it can help explain the timing, though at this stage we are identifying the un-defined term "efficiency" as just "execution time."

KSP	PC	time (s)	iterations
gmres	none	8.44	4705
	ilu	1.35	506
	ilu + restart=200	1.46	174
cg	none	0.52	606
	jacobi	0.57	606
	icc	0.33	177
	icc + rtol=10^{-14}	0.38	261
preonly	cholesky	8.66	1
minres	none	0.76	579

Table 3.1: Times and number of KSP iterations for serial runs of poisson.c on 257 × 257 grids. The assembled matrix is *symmetric, diagonally-dominant,* and *positive definite.* All runs were on WORKSTATION (see page 38).

From Table 3.1 we see that for GMRES, having a preconditioner is essential. That is, ILU(0) substantially reduces both iteration count and time compared to no preconditioner.

The number of iterations suggests that GMRES went through several restarts, which it does by default every 30 iterations. For the current problem memory overflow is no issue, so we avoid restart using option -ksp_gmres_restart 200. This reduces iteration count but not execution time, and we do not recommend such a procedure for ~~bigger~~ problems.

the

larger

The system matrix A is symmetric and positive definite, so we recall from Chapter 2 that the CG (conjugate gradients) Krylov method should apply, and that Cholesky and incomplete Cholesky ~~(IC(0))~~ are available as preconditioners. In Table 3.1 we compare these methods also. Because -ksp_preonly -pc_type cholesky is a direct solver, fair comparison to iterative methods suggests we should solve the equations accurately, so we add the tighter tolerance -ksp_rtol 1.0e-10 to all iterative runs in Table 3.1, except for the run where this was further tightened to -ksp_rtol 1.0e-10.

We see that Jacobi preconditioning has little benefit over *un*-preconditioned CG, because the diagonal is effectively constant,[10] but that IC(0) for CG is the best so far. The direct Cholesky solver is slower, presumably because of fill-in of the band seen in Figure 3.11.

We have so far not mentioned the minimum residual method [Greenbaum, 1997, MINRES] which applies to indefinite symmetric matrices, but it is implemented in PETSc too. We see that it seems to do no better than CG, at least in their un-preconditioned form. Elman et al. [2005, p. 88], however, states that "when solving discrete Poisson problems the convergence of MINRES is almost identical to that of CG."

If we were to stop now we might conclude that we have found a good KSP and PC combination for the Poisson equation, namely CG + ~~IC(0)~~, and proceed to harder problems. This conclusion would be premature.

ICC

The scaling flaw in CG iterations

Unfortunately, even with the preconditioners tried above, CG has a distinct flaw.[11] Namely, the iteration count grows with refined grids, so that [Elman et al., 2005, p. 76]

> for uniformly refined grids, the number of CG iterations required to meet a fixed tolerance will approximately double with each grid refinement.

This is exactly what we see from the following bash loop:

```
$ for NN in 1 2 3 4 5; do
> ./poisson -da_refine $NN -ksp_type cg -pc_type none -ksp_converged_reason; done
Linear solve converged due to CONVERGED_RTOL iterations 36
on 17 x 17 grid:  error |u-uexact|_inf = 0.000196729
Linear solve converged due to CONVERGED_RTOL iterations 73
on 33 x 33 grid:  error |u-uexact|_inf = 4.91819e-05
Linear solve converged due to CONVERGED_RTOL iterations 148
on 65 x 65 grid:  error |u-uexact|_inf = 1.22921e-05
Linear solve converged due to CONVERGED_RTOL iterations 299
on 129 x 129 grid:  error |u-uexact|_inf = 3.07512e-06
Linear solve converged due to CONVERGED_RTOL iterations 606
on 257 x 257 grid:  error |u-uexact|_inf = 7.69971e-07
```

Of course, this is un-preconditioned CG. While it is not surprising that -pc_type jacobi does no better (Exercise 3.3), it is easy to check

[10] See Exercise 3.3.

[11] At least from a 21st-century point of view. Table 3.1 might have satisfied in 1970.

that our favorite method so far has the same poor scaling. Figure 3.13 shows that though ~~IC(0)~~ gives somewhat faster solves and lower iteration counts, the *scaling* of those counts is the same as `-pc_type none`: the number of iterations doubles with each factor-of-two grid refinement.

ICC

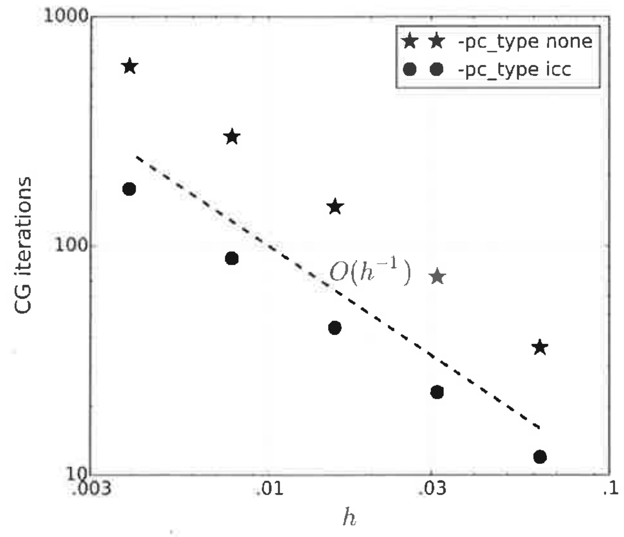


Figure 3.13: As the grid is refined ($h \to 0$), un-preconditioned CG scales poorly on our Poisson problem, and ~~IC(0)~~ preconditioning does not improve the scaling. ICC

A theoretical bound on the number of CG iterations would help to understand these results, but stating such a theorem requires preparation. Suppose that A is an $N \times N$, symmetric, and positive-definite matrix. We can define a new norm, the *A-norm*, on $\mathbf{v} \in \mathbb{R}^N$,

$$\|\mathbf{v}\|_A = \langle \mathbf{v}, A\mathbf{v}\rangle^{1/2} = (\mathbf{v}^\top A\mathbf{v})^{1/2}. \tag{3.12}$$

Recall that the eigenvalues $\lambda_j(A)$ of such a matrix A are positive and identical to the singular values, so that the 2-norm condition number of A is the ratio of the largest and smallest eigenvalues,

$$\kappa_2(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} \geq 1.$$

Finally, recall that $\mathbf{e}_k = \mathbf{u}_k - \mathbf{u}$ denotes the error of $\mathbf{u}_k$ as a solution to $A\mathbf{u} = \mathbf{b}$. In these terms we can state the following conceptually-useful theorem giving the sense in which CG is optimal[12] and supplying an error bound.

Theorem. *Suppose $\mathbf{u}_j$ are the iterates from un-preconditioned CG. Then:*

(i) Let $\mathcal{P}_j^1$ be the space of all real polynomials $p(x)$ of degree at most j such that $p(0) = 1$. Then

$$\|\mathbf{e}_j\|_A = \min_{p \in \mathcal{P}_j^1} \|p(A)\mathbf{e}_0\|_A.$$

[12] In fact this sense was already stated in Table 2.1.

(ii) The relative A-norm of the error at the jth iteration is bounded by an amount which is computable from the 2-norm condition number $\kappa = \kappa_2(A)$, namely

$$\frac{\|\mathbf{e}_j\|_A}{\|\mathbf{e}_0\|_A} \le 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^j.$$

(iii) The relative 2-norm of the residual is likewise bounded:

$$\frac{\|\mathbf{r}_j\|_2}{\|\mathbf{r}_0\|_2} \le 2\sqrt{\kappa}\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^j.$$

Part (*i*) of the Theorem is proved by Greenbaum [1997, p. 50]. Part (*ii*) follows from (*i*) by using Chebyshev polynomials to construct a polynomial which is small on the interval $[\lambda_{\min}(A), \lambda_{\max}(A)]$ [Greenbaum, 1997, p. 51]. Part (*iii*) follows from (*ii*) by understanding the norm $\|\cdot\|_A$ in terms of the 2-norm and the matrix $\sqrt{A}$; see Exercise 3.5. We state part (*iii*) for the simple reason that $\|\mathbf{r}_k\|_2$ is a quantity computed by PETSc while $\|\mathbf{e}_k\|_A$ is not. Parts (*ii*) and (*iii*) are not optimal bounds, though such can be found [Greenbaum, 1997, p. 51].

Consider the number of iterations j needed to reduce $\|\mathbf{e}_j\|_A$ by a factor of $\epsilon < 0$. Note $(x-1)/(x+1) = 1 - 2/(x+1)$, and recall that $\ln(1-x) \approx -x$ for small x. If $\sqrt{\kappa}$ is large then part (*ii*) of the Theorem says that $\|\mathbf{e}_j\|_A$ is reduced from its initial value $\|\mathbf{e}_0\|_A$ by a factor of ϵ if

$\epsilon < 0$?! Seems wrong.

$$j \le \frac{\ln(\epsilon/2)}{\ln\left(1 - \frac{2}{\sqrt{\kappa}+1}\right)} \approx -\frac{\ln(\epsilon/2)}{2}(\sqrt{\kappa}+1).$$

In summary,

$$\frac{\|\mathbf{e}_j\|_A}{\|\mathbf{e}_0\|_A} \le \epsilon \quad \text{if} \quad j = O\left(\sqrt{\kappa_2(A)}\right). \tag{3.13}$$

The same conclusion applies to reducing $\|\mathbf{r}_j\|_2$ by a factor of ϵ, though the analysis is not so simple; see Exercise 3.5.

Though the theorem states a key connection between condition number and iteration count, the precise condition number of a discretized-PDE matrix A_h, coming from a mesh with spacing h, is not typically available. On the other hand we can ask PETSc to approximate eigenvalues of A_h as the Krylov iteration proceeds.[13] Doing this will largely explain the poor scaling of CG iteration count for our symmetric-matrix, structured-grid, Poisson problem case:

```
$ ./poisson -ksp_type cg -pc_type none -ksp_compute_eigenvalues
Iteratively computed eigenvalues
0.304482 + 0i
...
7.69543 + 0i
on 9 x 9 grid:  error |u-uexact|_inf = 0.00076388
```

[13] We need the iteration to use the *un*-preconditioned operator when computing eigenvalues, so use -pc_type none or ~~right-side preconditioning, -ksp_pc_side RIGHT.~~

I don't think using right will work!

In fact 17 eigenvalues were printed, because there were 17 KSP iterations and the method computes one additional eigenvalue per iteration. We suppressed all but the first (smallest) and last (largest), as they suffice for approximating the condition number. Thus, on the above grid with spacing $h = 1/8$:

$$\kappa_2(A_{h=1/8}) \approx \frac{7.69543}{0.304482} = 25.274.$$

Re-running with option `-da_refine N` for `N`$= 1, 2$ we ~~get~~: obtain

$$\kappa_2(A_{h=1/16}) \approx \frac{7.92311}{0.0768589} = 103.09,$$
$$\kappa_2(A_{h=1/32}) \approx \frac{7.98073}{0.0192611} = 414.34.$$

We now see the issue clearly. Each time the grid is refined by a factor of two (i.e. $h \to h/2$), the condition number $\kappa_2(A_h)$ increases by a factor of four. Then (3.13) says we should expect an increase in iteration count by a factor of two.

In fact, as an *a priori* statement, the 2-norm condition number of the discrete Poisson matrix A_h on a uniform rectangular grid with spacing h is known to be

$$\kappa_2(A_h) = O(h^{-2}), \tag{3.14}$$

exactly as seen experimentally above. This can be shown by exact analysis using the eigenvectors of the discrete Poisson problem in the uniform rectangular case [Briggs et al., 2000]. A more robust and general FEM error analysis which applies to unstructured grids having a mesh-uniformity bound [Elman et al., 2005] also applies. In either case, combining (3.13) and (3.14) shows that we should expect to need

$$j = O(h^{-1}), \tag{3.15}$$

iterations to solve our discrete Poisson problem to a given relative tolerance using un-preconditioned CG.

We have also seen that ~~IC(0)~~ preconditioning reduces the time and iteration count, but that it does not improve on scaling (3.15); recall Figure 3.13. Detailed analysis of the effects of incomplete Cholesky precondition ~~must be~~ difficult, and it will certainly not be pursued here, but we do now have the correct context to understand this claim about the result Elman et al. [2005, p. 82]:

ICC

are

> One known result [*for CG for the Poisson equation*] is that the asymptotic behavior of the condition number using ~~IC~~(0) preconditioning is unchanged: $\kappa(M^{-1}A) = O(h^{-2})$.

ICC

So ~~IC~~(0) is not as promising a preconditioner as we had first hoped.

ICC

Damn this IC ☺

Krylov is not enough . . . good *preconditioning is needed*

When solving the sparse linear systems from discretized PDEs, Krylov iterations ~~like~~ CG and preconditioning methods like IC(0) are useful tools because they can improve upon naively-applied direct linear algebra methods. However, besides the bad news above that CG iterations scale badly with refining grids, there is more "bad news" for our Krylov approach. Namely, that smartly-applied direct linear algebra techniques can also beat CG+IC(0), at least in the two-dimensional case of this Chapter.

such as

Specifically, suppose we run `poisson` with option `-da_refine 7`, to give a 1025 × 1025 grid and about 10^6 degrees of freedom. Suppose we compare IC(0)-preconditioned CG iterations with two other methods:

```
$ ./poisson -da_refine 7 -ksp_type cg -pc_type icc
$ ./poisson -da_refine 7 -ksp_type preonly -pc_type cholesky -pc_factor_mat_ordering_type nd
$ ./poisson -da_refine 7 -ksp_type preonly -pc_type lu -pc_factor_mat_ordering_type nd
```

(To be precise, we added options `-ksp_converged_reason` and `-ksp_rtol 1.0e-10` to the CG run.) We ~~get~~ the first three rows of Table 3.2, the first of which requires no further explanation because it is the best-so-far way of using CG. The second and third runs, which are faster, are direct methods.

obtain

Code	KSP	PC	time (s)	iterations
`poisson`	`cg`	`icc`	27.10	1056
	`preonly`	`cholesky + nd`	24.96	1
		`lu + nd`	16.33	1
`fish2`	`cg`	`mg`	2.60	5

Table 3.2: Time and iteration count for `-da_refine 7` runs, a 1025 × 1025 grid on a single MPI process on WORKSTATION. The `fish2` code appears in Chapter 6.

indicates

Option `-pc_factor_mat_ordering_type nd` ~~asks~~ that the direct method be applied using the *nested dissection* ordering [George, 1973]. While we give no details, in outline the unknowns and equations are re-ordered to reduce the cost of the factorization. For LU factorization the nested dissection ordering is actually the PETSc default for single-processor runs.[14] In any case, the two direct solvers in Table 3.2 are clearly competitive with IC(0)-preconditioned CG.

At this point the impulsive reader might give up on Krylov methods, but that also would be premature. We do need to find a better preconditioner if we are going to have scalable solutions of this Poisson PDE problem. Finding such methods will eventually show that properly-preconditioned Krylov iterations give both efficient and parallel-scalable solutions of large nonlinear and variable-coefficient elliptic PDE problems in 2D and 3D, something direct methods of

[14] And for this reason, the result of the `-ksp_type preonly -pc_type lu` was not in Table 3.1.

for N ∈ {1,...,6} and X ∈ {none,jacobi}. Explain. Now confirm that IC(0) preconditioning gives lower iteration counts but the same bad scaling.

3.4 Add `-log_summary` to an un-preconditioned CG run, e.g.

```
$ ./poisson -ksp_converged_reason -ksp_type cg -pc_type none -log_summary
```

By looking at the "Count" column, and noting that the iteration count comes from `-ksp_converged_reason` output, confirm that the computational work of one CG iteration consists of two inner products (`VecTDot`), three vector updates (`VecAXPY` and `VecAYPX`), and one matrix-vector product (`MatMult`). Find a pseudo-code for CG in some textbook, and confirm this work pattern.

3.5 Let A be a symmetric positive-definite $N \times N$ matrix. Note A is diagonalizable and has positive eigenvalues.

(*i*) Show that (3.12) defines a norm on $\mathbb{R}^N$.

(*ii*) Define the matrix $\sqrt{A}$ as the unique symmetric and positive-definite matrix such that $(\sqrt{A})^2 = A$. Show that $\|v\|_A = \|\sqrt{A}\, v\|_2$ and that $\kappa_2\left(\sqrt{A}\right) = \sqrt{\kappa_2(A)}$.

(*iii*) Recalling that $A\mathbf{e} = -\mathbf{r}$, show that

$$\frac{1}{\|\sqrt{A}\|_2}\|\mathbf{r}\|_2 \le \|\mathbf{e}\|_A \le \|\sqrt{A}\|_2\|\mathbf{r}\|_2.$$

(*iv*) Prove part (*iii*) of the Theorem on page 63.

3.6 If you have difficulty reproducing timings like those in Table 3.2, note that they come from a PETSc configuration with "optimized" configuration option[15] `-with-debugging=0`. If you have not already done so, see the PETSc installation page

`www.mcs.anl.gov/petsc/documentation/installation.html`

and generate a new configuration, with a new `PETSC_ARCH` value, with this "optimized" configuration option. Generate your own version of Table 3.2.

3.7 Use `DMDACreate3d()` and etc. in a code `poisson3D.c` which solves a 3D Poisson problem on the unit cube $\mathcal{C} = [0,1]^3$ using the same `DMDA` and `KSP` methods as in `poisson.c`. Looking forward, how is the approach taken in Chapter 6 different from your code?

3.8 Modify structuredpoisson to use a right hand side function of $u(x,y) = (x-x^2)(y^2-y)$ and compute the errors as you refine the mesh. why do you get this behavior?

[15] *Not* runtime option!

4
Nonlinear equations

The simplest thing to say about nonlinear equations is that they change the functional form of the residual, compared to the linear case. For a linear system the residual $\mathbf{r}$ is a certain function of the unknowns $\mathbf{u}$, namely $\mathbf{r} = \mathbf{F}(\mathbf{u}) = \mathbf{b} - A\mathbf{u}$. Now we consider cases in which F is a higher-order polynomial, a transcendental function, or some more general function.

~~In fact, let us simply~~ Suppose for now that $\mathbf{F} : \mathbb{R}^N \to \mathbb{R}^N$ is differentiable. The input $\mathbf{x}$ and output $\mathbf{F}(\mathbf{x})$ are column vectors,[1] so in that sense F acts like square-matrix multiplication $\mathbf{x} \mapsto A\mathbf{x}$. Just as a linear solver like GMRES (Chapter 2) reduces the linear residual $\mathbf{r}_k = \mathbf{b} - A\mathbf{u}_k$ to zero by generating a sequence $\mathbf{u}_k$, for nonlinear F we want to solve

$$\mathbf{F}(\mathbf{x}) = 0 \qquad (4.1)$$

by iteration, generating approximations $\mathbf{x}_k$ so that $\mathbf{F}(\mathbf{x}_k)$ goes to zero.

Newton's method linearizes (4.1) around the most recent iterate $\mathbf{x}_k$ and then "moves" to the location $\mathbf{x}_{k+1}$ which solves the linear problem. The new location is, we hope, closer to the solution. Each iteration requires solving a linear system, and we already have PETSc technology for that. However, the cost of performing the linearization must be taken into account, choosing a smart distance to move will require additional choices, and all existing choices regarding the linear solver—especially preconditioning—remain active. Solving a nonlinear problem generally requires all the tools for linear systems considered in Chapter 2, and more.

Large systems of nonlinear equations often arise in applications from PDE-type physical models with nonlinearities. However, the current Chapter is largely about finite-dimensional systems of nonlinear equations (4.1) as problems of their own. Late in this Chapter we do give an example of a discretized one-dimensional nonlinear PDE.

[1] Our name change $\mathbf{u} \to \mathbf{x}$ comes from now thinking more geometrically about the location of $\mathbf{x}$.

Example. Given parameter $b > 1$, the nonlinear equations

$$y = \frac{1}{b} e^{bx}, \qquad x^2 + y^2 = 1,$$

form intersecting curves in the plane. The curves intersect twice, as shown in Figure 4.1. These equations are put in standard form (4.1) by writing

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} \frac{1}{b} e^{bx_0} - x_1 \\ x_0^2 + x_1^2 - 1 \end{bmatrix} \tag{4.6}$$

for $\mathbf{x} \in \mathbb{R}^2$ a column vector: $\mathbf{x} = [x_0 \;\; x_1]^\top$. Thus

$$J_{\mathbf{F}}(\mathbf{x}) = \begin{bmatrix} e^{bx_0} & -1 \\ 2x_0 & 2x_1 \end{bmatrix} \tag{4.7}$$

As also shown in Figure 4.1, for $b = 2$, if we start the Newton iteration with $\mathbf{x}_0 = [1 \; 1]^\top$ then the sequence of iterates from (4.5) is

$$\mathbf{x}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{bmatrix} 0.619203 \\ 0.880797 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 0.394157 \\ 0.948623 \end{bmatrix}, \quad \dots$$

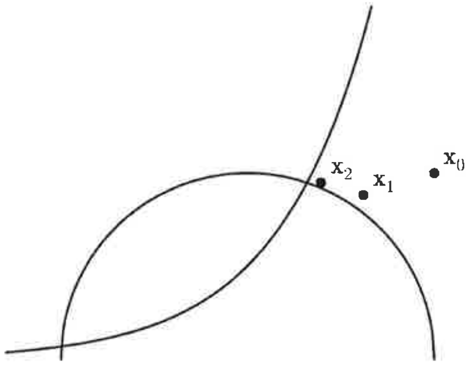


Figure 4.1: Newton iterates $\mathbf{x}_k$ approach a solution of $\mathbf{F}(\mathbf{x}) = 0$ for $\mathbf{F}$ in (4.6) and $b = 2$.

Using SNES with call-backs

We will compute the Newton iterates in the above example by using a nonlinear solver object of type SNES[2] from PETSc. Note SNES has the usual `Create/SetFromOptions/Destroy` sequence. Our code provides a function $\mathbf{F}$ which is a "call-back" in the sense that we supply it to the SNES, which then calls it with argument $\mathbf{x}$ when it needs $\mathbf{F}(\mathbf{x})$ during the Newton iteration.

Later we will also provide the SNES with a function which computes the Jacobian function $J_{\mathbf{F}}$. However, the Jacobian can be approximated by repeated F evaluations because a derivative can also be approximated by finite differences. Thus our first code avoids such a Jacobian "call-back" ~~for now~~.

The whole of `expcircle.c`, to solve the above Example, is in Code 4.1. The `main()` method starts by allocating `Vec x` of fixed dimension 2 which will hold ~~both~~ the initial iterate $\mathbf{x}_0$. (Once the Newton iteration is ended it holds the converged estimate of the solution.) Because both components of $\mathbf{x}_0$ are 1, it is initialized using `VecSet()`. Next a duplicate `Vec r` is created because the SNES needs it as space to store ~~for~~ the (nonlinear) residual.

Then the SNES is created and configured. Formula (4.6) is in method `FormFunction()`, which is supplied to the SNES object using `SNESSetFunction()`.

[2] SNES stands for "scalable nonlinear equation solver."

We call `SNESSetFromOptions()` because it ~~both~~ gives us run-time control on how the Jacobian is calculated[3] and on how the length of the step $\mathbf{s}$ is ~~actually~~ determined.[4] Then (4.1) is solved by a call to `SNESSolve()`. Finally the new values in `x`, presumably the converged solution, are printed at the command line using `VecView()` with a `STDOUT` viewer.

ch4/expcircle.c

```
static char help[] = "Newton’s method for a two-variable system.\n"
    "No analytical Jacobian.  Run with -snes_fd or -snes_mf.\n\n";

#include <petsc.h>

PetscErrorCode FormFunction(SNES snes, Vec x, Vec F, void *ctx) {
    const PetscReal  b = 2.0, *ax;
    PetscReal        *aF;

    VecGetArrayRead(x,&ax);
    VecGetArray(F,&aF);
    aF[0] = (1.0 / b) * PetscExpReal(b * ax[0]) - ax[1];
    aF[1] = ax[0] * ax[0] + ax[1] * ax[1] - 1.0;
    VecRestoreArrayRead(x,&ax);
    VecRestoreArray(F,&aF);
    return 0;
}

int main(int argc,char **argv) {
    SNES   snes;          // nonlinear solver context
    Vec    x, r;          // solution, residual vectors

    PetscInitialize(&argc,&argv,NULL,help);
    VecCreate(PETSC_COMM_WORLD,&x);
    VecSetSizes(x,PETSC_DECIDE,2);
    VecSetFromOptions(x);
    VecSet(x,1.0);
    VecDuplicate(x,&r);

    SNESCreate(PETSC_COMM_WORLD,&snes);
    SNESSetFunction(snes,r,FormFunction,NULL);
    SNESSetFromOptions(snes);
    SNESSolve(snes,NULL,x);
    VecView(x,PETSC_VIEWER_STDOUT_WORLD);

    VecDestroy(&x);  VecDestroy(&r);  SNESDestroy(&snes);
    PetscFinalize();
    return 0;
}
```

Code 4.1: A first `SNES`-using code. Solves nonlinear system (4.1) with $\mathbf{F}$ given in (4.6).

In order to match the calling sequence of `SNESSetFunction()`, `FormFunction()` must have a particular "signature" as a C function:

[3] Options `-snes_fd` and `-snes_mf` are allowed; see Table 4.2 below.

[4] Through `-snes_linesearch_type` and related options; see page 98.

```
0.319632
0.947542
```

Thus after 5 iterations the Newton method has reduced the residual norm by a factor of 10^9 and stopped with solution $x_0 = 0.319632$ and $x_1 = 0.947542$. Compare Figure 4.1.

The above run ~~also~~ uses option `-snes_fd`, the purpose of which the reader may already see. Clearly the Newton iteration (4.5) requires the Jacobian, but we have only supplied the `SNES` with an implementation of function $\mathbf{F}(\mathbf{x})$, not with $J_{\mathbf{F}}(\mathbf{x})$. As mentioned, the entries of the latter matrix are derivatives which can be approximated by finite differences. Specifically, let $\delta \neq 0$ and let $\mathbf{e}_j \in \mathbb{R}^N$ denote the standard unit vector with entry one in the jth position and zeros otherwise. An entry in matrix $J = J_{\mathbf{F}}(\mathbf{x})$ is approximated

$$J_{ij} = \frac{\partial F_i}{\partial x_j} \approx \frac{F_i(\mathbf{x} + \delta \mathbf{e}_j) - F_i(\mathbf{x})}{\delta}. \tag{4.8}$$

When using `-snes_fd`, PETSc chooses δ internally and applies (4.8). For example, $\delta = \sqrt{\epsilon}$, where ϵ is machine precision, gives a reasonably-accurate approximation if the inputs to $\mathbf{F}$ are all of order approximately one and the function $\mathbf{F}$ can be accurately-evaluated [Kelley, 2003].

Inside SNES

It is helpful to describe Newton iteration from the point of view of the actions taken by the `SNES` solver object. In outline, it does these steps:

(*i*) from the current iterate $\mathbf{x}_k$, $\mathbf{F}(\mathbf{x}_k)$ is evaluated using a call-back function as set in `SNESSetFunction()`, e.g. `FormFunction()` above,

(*ii*) the Jacobian $J = J_{\mathbf{F}}(\mathbf{x}_k)$ is

a. computed and assembled by a call-back to user-supplied code, if it is available, as set using `SNESSetJacobian()`, e.g. `FormJacobian()` in code `ecjacobian.c` below, *or*

b. computed and assembled by evaluating $\mathbf{F}(\mathbf{x}_k + \delta \mathbf{e}_j)$ for $j = 0, \dots, N-1$, thus calling `FormFunction()` N times, and then using formula (4.8) N^2 times to compute all entries of J, *or*

c. computed and assembled by calling `FormFunction()` substantially fewer than N times to compute $\mathbf{F}(\mathbf{x}_k + \delta \mathbf{v})$ for special vectors $\mathbf{v}$, by using a graph-coloring algorithm based on the sparsity pattern of J to construct the vectors $\mathbf{v}$, and using formula (4.19) below, *or*

d. not assembled, but, in a Krylov iterative method for solving system (4.5a), the action $J\mathbf{y}$, of the Jacobian on vectors $\mathbf{y}$, is computed by finite-differences, (*iii*) linear system (4.5a) is solved for **s** by some KSP object, using whatever additional preconditioning matrix has been chosen,

(*iv*) vector update (4.5b) is done, with possible reduction or expansion in the length of **s** according to the line-search object, as addressed starting on page 98, *and*

(*v*) a convergence test is made, and we repeat at (*i*) if not converged.

Our single run above of `expcircle.c` used option (*ii*)b for the Jacobian, but option (*ii*d) also works. Our next code will allow option (*ii*)a as well. The graph-coloring technique (*ii*)c will be addressed starting on page 88.

If you run `expcircle.c` without option `-snes_fd` or `-snes_mf` then you get an error message about an un-assembled matrix:

```
$ ./expcircle
[0]PETSC ERROR: --------------------- Error Message ------------------------
[0]PETSC ERROR: Object is in wrong state
[0]PETSC ERROR: Matrix must be assembled by calls to MatAssemblyBegin/End();
...
```

This message is somewhat opaque unless you are conscious of the need to form the Jacobian matrix at each Newton iteration. That is, *something* must supply a Jacobian at step (*ii*), and the supplied-Jacobian-code case (*ii*)a does not work for `expcircle.c`.

The benefit of using options (*ii*)b–(*ii*)d above is that we do not need to write any error-prone code based on taking derivatives of our function **F**. Avoiding writing and debugging Jacobian implementations may speed implementation by reducing the time *you* spend on the task. One possible disadvantage is likely to be apparent to the reader, namely that formula (4.8) is only an *approximation* of a Jacobian entry. ~~However,~~ In most cases using a finite-difference-approximated Jacobian in the Newton step is not problematical [Kelley, 2003].

On the other hand, there is a significant performance problem in using (4.8) naively for PDE-type applications of Newton's method. In steps (*i*) and (*ii*)b together we do $N+1$ calls to `FormFunction()` per Newton iteration. This is a worrying amount of work if N is large, as it would be for a system of nonlinear equations coming from discretizing a PDE. We will therefore return to this issue later in the current Chapter, and provide more detail on (*ii*)c and (*ii*)d.

Evaluating **F** can dominate the work in the Newton iteration, and there are systems where it is an intrinsically-expensive function to evaluate. The work done in solving linear system (4.5a) is the other main concern. A good PETSc habit, to start right now, is to use `-log_summary` to see which kind of work actually dominates.

which we want to send to zero. In particular, we want to stop the Newton iteration when $\mathbf{x}_k$ is a good approximation of $\mathbf{x}^*$. Of course, $\mathbf{e}_k$ is just as hard to compute as $\mathbf{x}^*$; we do not generally have exact access to either one. Instead we have computable quantities $\mathbf{x}_k$ and $\mathbf{r}_k = \mathbf{F}(\mathbf{x}_k)$ available for inspection. Thus both parts of the following theorem are important.

Theorem. *[Kelley, 2003, Theorems 1.1 and inequalities (1.13)] Suppose that $\mathbf{F} : \mathbb{R}^N \to \mathbb{R}^N$ is differentiable, $J_\mathbf{F}$ is Lipschitz near $\mathbf{x}^*$, and $J_\mathbf{F}(\mathbf{x}^*)$ is a nonsingular matrix. Let $\|\cdot\|$ denote a vector norm and its induced matrix norm, and let $\kappa(A) = \|A^{-1}\|\|A\|$ denote the condition number of an invertible matrix A. If $\mathbf{x}_0$ is sufficiently close to $\mathbf{x}^*$ then, in exact arithmetic,*

(i) there is $K \geq 0$ such that for all k sufficiently large,

$$\|\mathbf{e}_{k+1}\| \leq K\|\mathbf{e}_k\|^2, \qquad (4.10)$$

(ii) and if $\kappa = \kappa\left(J_\mathbf{F}(\mathbf{x}^*)\right)$ then

$$\frac{\|\mathbf{e}_k\|}{4\kappa\|\mathbf{e}_0\|} \leq \frac{\|\mathbf{F}(\mathbf{x}_k)\|}{\|\mathbf{F}(\mathbf{x}_0)\|} \leq \frac{4\kappa\|\mathbf{e}_k\|}{\|\mathbf{e}_0\|}. \qquad (4.11)$$

By definition, a sequence $\{\mathbf{x}_k\}$ in $\mathbb{R}^N$ *converges quadratically to* $\mathbf{x}^*$ if the sequence of errors $\{\mathbf{e}_k\}$ satisfies (4.10) for some $K \geq 0$. Thus the Theorem says that, under strong assumptions about the regularity and nonsingularity of the Jacobian, the iterates converge quadratically to a solution of (4.1). Heuristically, once $\|\mathbf{e}_k\|$ gets reasonably small then, ~~from then on,~~ the number of correct digits in $\mathbf{x}_k$ *doubles* with each additional iteration.

We seem to see quadratic convergence in Figure 4.2, but actually it shows the residual norm $\|\mathbf{F}(\mathbf{x}_k)\|_2$ and not the error norm $\|\mathbf{e}_k\|_2$. However, the second part of the Theorem says that the residual decrease at the kth iteration (i.e. $\|\mathbf{F}(\mathbf{x}_k)\|/\|\mathbf{F}(\mathbf{x}_0)\|$) is within a factor, determined by the conditioning of the Jacobian at the solution, of the error decrease ($\|\mathbf{e}_k\|/\|\mathbf{e}_0\|$).

The Theorem confirms that residual norm decay like that shown in Figure 4.2 corresponds to quadratic convergence of $\mathbf{x}_k$ to a solution $\mathbf{x}^*$. If we want to reduce the (generally-unknowable) error $\mathbf{e}_k$ by a given amount then it can suffice to reduce the residual norm by a comparable amount. The factor 4κ by which the two relative norms differ in (4.11) is large if the conditioning of the Jacobian at the solution is poor, but, just as in the linear case, a large Jacobian condition number would also mean lost precision in solving (4.1) by *any* numerical means.[8]

[8] Recall the numerical facts-of-life in Chapter 2.

Input `Vec x` and pointer `void *ctx` have the same meaning as in `FormFunction()`.

The difference is that we must set a `Mat` as output, based on formula (4.7) in this case. In ~~setting-up~~ the output `Mat` the roles of `MatSetValues()`, real array `v[4]` for the entries themselves, and integer arrays `row[2]` and `col[2]` as global indices, are all the same as in Chapter 2. Also ~~just~~ as before, when reading `x` we can use `VecGetArrayRead()` and `VecRestoreArrayRead()`.

setting up

ch4/ecjacobian.c **part II**

```
int main(int argc,char **argv)
{
  SNES   snes;         // nonlinear solver context
  Vec    x,r;          // solution, residual vectors
  Mat    J;
  AppCtx user;

  PetscInitialize(&argc,&argv,NULL,help);
  user.b = 2.0;

  VecCreate(PETSC_COMM_WORLD,&x);
  VecSetSizes(x,PETSC_DECIDE,2);
  VecSetFromOptions(x);
  VecDuplicate(x,&r);

  MatCreate(PETSC_COMM_WORLD,&J);
  MatSetSizes(J,PETSC_DECIDE,PETSC_DECIDE,2,2);
  MatSetFromOptions(J);
  MatSetUp(J);

  SNESCreate(PETSC_COMM_WORLD,&snes);
  SNESSetFunction(snes,r,FormFunction,&user);
  SNESSetJacobian(snes,J,J,FormJacobian,&user);
  SNESSetFromOptions(snes);

  VecSet(x,1.0);
  SNESSolve(snes,NULL,x);
  VecView(x,PETSC_VIEWER_STDOUT_WORLD);

  VecDestroy(&x);  VecDestroy(&r);  SNESDestroy(&snes);  MatDestroy(&J);
  PetscFinalize();
  return 0;
}
```

Code 4.3: This `main()` method allocates a `Mat` to hold the Jacobian.

An interesting detail appears here, however. There are actually *two* output `Mat`s for `FormJacobian()` to set. The first, called `J` here, corresponds to the Jacobian matrix itself, which, in this simple case, we want to supply. The second, called `P` here, is the "material" we supply to build a preconditioner. It might, at least in other then the ability of the diffusion term to "damp-out" maxima may be exceeded by the increasing production from large values of u, so that these terms cannot balance, which is (4.12). If R is negative and increasing then the analogous concern applies to minima of u. These concerns are demonstrated by the example $R(u) = \lambda e^u$ with $\lambda > 0$ in Exercise 4.8; that problem is not solvable for sufficiently-large λ.

Equation (4.12) is a nonlinear elliptic PDE,[12] but in one-dimension. Elliptic PDE techniques show the problem is well-posed if R is a *non*-increasing function [Kinderlehrer and Stampacchia, 1980, pages 93-94]. To be concrete, consider Dirichlet boundary conditions

$$u(0) = \alpha \quad \text{and} \quad u(1) = \beta. \tag{4.14}$$

Here page numbers

Here no page numbers

(We have chosen a convenient interval $x \in [0, 1]$, but for other intervals we can shift and scale x as needed.) If R is continuous and non-increasing then the nonlinear operator in (4.12) is strictly-monotone [Kinderlehrer and Stampacchia, 1980]. Because it is also coercive on the appropriate function space,[13] which says intuitively that the highest-order (diffusion) term is effective at damping out large variations in u which correspond to a large norm $\|u'\|_2$, abstract arguments show unique existence of a solution.

Now we consider the example

$$-u'' + \rho\sqrt{u} = 0. \tag{4.15}$$

This is of form (4.12) with $R(u) = -\rho\sqrt{u}$ and $f(x) = 0$. Because R is non-increasing and continuous if $\rho > 0$, the corresponding Dirichlet problem is well-posed.

We ~~are actually using~~ will use (4.15) as a first example, ~~however,~~ because we want to verify our numerical solution using an exact solution, and its form makes this easy. The solution is found [Ockendon et al., 2003] by noting that both second-derivative and square-root operations convert certain 4th degree polynomials into quadratic polynomials. We also ~~get~~ obtain the boundary conditions from the exact solution. ~~Therefore we~~ Substitute $u(x) = M(x+1)^4$ into (4.15) to find $M = (\rho/12)^2$, and ~~we~~ set $\alpha = M$ and $\beta = 16M$.

On an N point grid, the finite-difference scheme ~~we propose~~ for (4.12) is

$$-\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} - R(u_j) = f(x_j) \tag{4.16}$$

where $h = 1/(N-1) > 0$ is the grid spacing, $x_j = jh$ for $j = 0, 1, \dots, N-1$, and $u_j \approx u(x_j)$.

Code `reaction.c` shown in Codes 4.4 and 4.5 solves this problem. We use scheme (4.16), a `SNES` object for the Newton iteration, and a `DMDA` object for the grid.

[12] By mild abuse of the letter "P".

[13] Namely the Sobolev space $H_0^1[0, 1]$, after a linear change of variables to set $\alpha = \beta = 0$.

These are "Local" methods in the sense that their inputs are C pointers for arrays instead of Vecs, and our implementation of InitialAndExactLocal() explains how the Local methods work. As seen in Code 4.5, before this method is called we use DMDAVecGetArray() on the Vecs u and uexact. This gives PetscReal* pointers which are then handed to InitialAndExactLocal(). Also note that a DMDALocalInfo struct is passed in so that the local part of the grid (i.e. info.xs and etc.) and the global grid size (i.e. info.mx) can be accessed through this struct without needing the DMDA object itself.[14] The call-back done by SNES on FormFunctionLocal() and FormJacobianLocal() works the same way.

In main() we use DMDASNESSetFunctionLocal() instead of SNESSetFunction()[15] because our local call-back functions have a different signature based on DMDA-derived pointers. The Jacobian is set by a similar method DMDASNESSetJacobianLocal(). Also, though we implement a Jacobian, we do not allocate a Mat to hold it because the DMDA object has enough information about the grid and stencil so as to ~~pre~~-allocate ~~a~~ the Mat internally.

Convergence under grid refinement

On a modestly-refined grid we can compare the number of reaction.c Newton iterations from both analytical and finite-difference Jacobians. Recalling that the resolution of a structured grid can be set with either -da_grid_x M or -da_refine N, we do:

```
$ ./reaction -snes_converged_reason -da_refine 6
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 513 point grid:  |u-u_exact|_inf/|u|_inf = 4.62255e-08
$ ./reaction -snes_converged_reason -da_refine 6 -snes_fd
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 513 point grid:  |u-u_exact|_inf/|u|_inf = 4.62255e-08
```

The results are identical. One can also see the Newton iterates graphically by

```
$ ./reaction -da_refine 6 -snes_monitor_solution -draw_pause 1
```

One sees that the first Newton step moves us close to the solution (not shown).

Noting that our finite difference method has local truncation error $O(h^2)$, and because an exact solution allows computation of the numerical error, we should generate convergence data to check the implementation. The result of this bash loop

```
$ for N in 0 2 4 6 8 10 12 14 16; do
>   ./reaction -da_refine $N -snes_rtol 1.0e-10; done
on 9 point grid:  |u-u_exact|_inf/|u|_inf = 0.000188753
```

[14] Recall Figure 3.9.

[15] Compare ecjacobian.c.

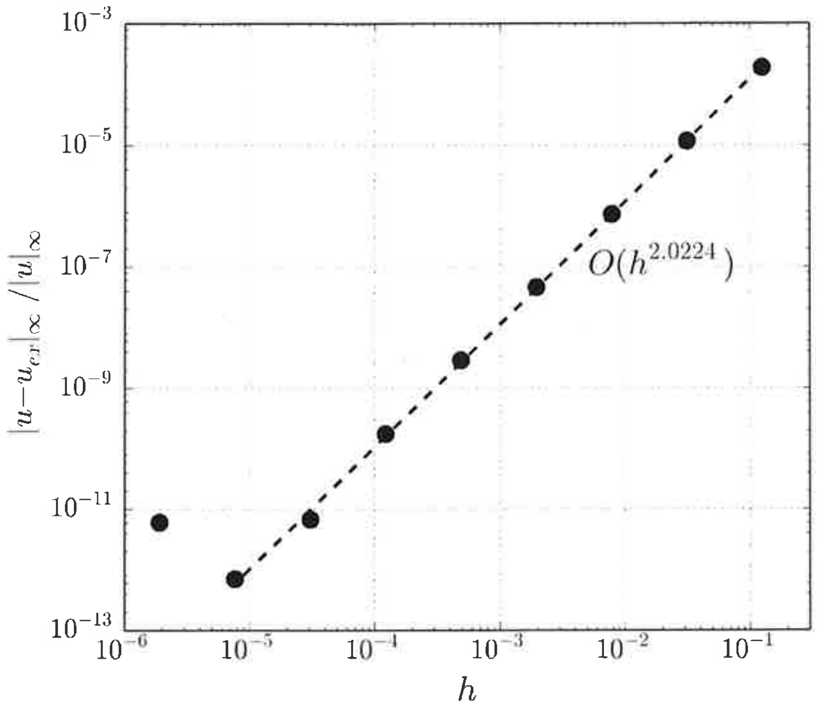


Figure 4.4: This convergence evidence suggests `reaction.c` is correctly-implemented.

```
on 33 point grid:  |u-u_exact|_inf/|u|_inf = 1.1825e-05
...
on 131073 point grid:  |u-u_exact|_inf/|u|_inf = 7.05476e-13
on 524289 point grid:  |u-u_exact|_inf/|u|_inf = 6.04273e-12
```

is shown in Figure 4.4. If we ignore the result on the finest grid then the convergence rate is $O(h^2)$.[16] We also see consistent evidence of quadratic convergence by adding `-snes_monitor` to the above runs. Thus we conclude our implementation is correct.

Finite-difference Jacobians by "coloring"

~~However,~~ When we look closer at the `-snes_fd` results from finite-difference Jacobians we see that it is not really working. There are far too many function evaluations per Newton step to be practical. This is because, in contrast to the earlier fixed-dimension examples, discretizing a PDE generates an arbitrarily-large number of unknowns.

An analytical Jacobian run on a grid of about 8000 nodes requires three Newton iterations and 0.04 seconds:

```
$ timer ./reaction -snes_converged_reason -da_refine 10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 8193 point grid:  |u-u_exact|_inf/|u|_inf = 1.75603e-10
real 0.04
```

[16] Error stagnation always occurs at some level of refinement because of accumulation of round-off error.

```
$ ./reaction -da_grid_x 7
```

Figure 4.5 shows this grid and the three-point stencil for the numerical discretization scheme (4.16). The stencil (equation) at x_j generically involves three unknowns, namely u_{j-1}, u_j, and u_{j+1}. Thus the jth row of the Jacobian J has three nonzero entries. Exceptions occur at the boundary nodes, where only two unknowns are involved. In any case, option `-mat_view ::ascii_dense` allows one to see the Jacobian matrices themselves during the Newton iteration.

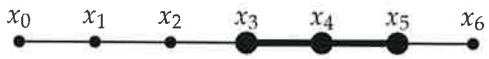


Figure 4.5: `reaction.c` uses discretization (4.16) at each interior node of the grid. This corresponds to a three-point stencil, as shown.

We take the unknowns, or the corresponding grid nodes, to be the vertices of a graph $G = G(J)$, as shown in the top of Figure 4.6. Graph G has an edge between two vertices if the corresponding unknowns both appear in at least one equation, i.e. in one instance of equation (4.16) in this case.

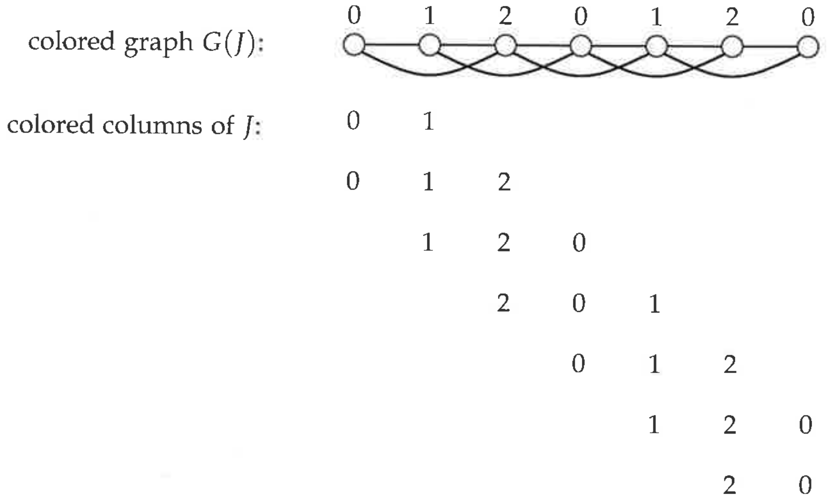


Figure 4.6: Graph $G(J)$ is built from the stencil and the grid, with an edge for every pair of unknowns that appears in an equation in system (4.1). Coloring this graph—$c = 3$ colors suffice in this case—also assigns colors to the columns of J.

Now suppose we color the vertices so that any two vertices which share an edge have distinct colors. The coloring in the Figure uses $c = 3$ colors. In the case of this simple graph, $c = 3$ is the minimum number needed, so that in this case $\chi(G) = c$ where $\chi(G)$ is the *chromatic number* of G [Chartrand et al., 2011].

see note

~~insert text here.~~

~~However,~~ optimally-coloring a graph is a hard problem which PETSc does not attempt to solve. The PETSc-implemented "incidence-degree ordering" [*default*] and "smallest-last ordering" graph-coloring algorithms are relatively-near to optimal in the provable sense that we get c-colorings for which $c \leq \alpha N^{1/2} \chi(G(J))$ [Coleman and Moré,

(both borrowed from the MINPACK software)

1983], for some constant $\alpha > 0$, where N is the number of equations in (4.1). More importantly, these algorithms do very well in the informal sense that c is quite close to $\chi(G(J))$ for a large selection of test matrices [Coleman and Moré, 1983].

Spelling

Furhtermore, these algorithms run in a time proportional to the sum of squares of the number of nonzeros in each rows of J. Thus we get a reasonably-good coloring not just in polynomial time in N, but in substantially-less time than $O(N^2)$ and often in $O(N)$ time for problems coming from discretized PDEs. (Recall we are trying to improve on the existing $O(N^2)$ cost of the `-snes_fd` method.)

We now also have a coloring of the columns of J, as shown in the bottom of Figure 4.6. Each row has distinct colors, which is one way to describe our purpose here. Next we generate vectors $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{c-1} \in \mathbb{R}^N$ by the rule that $\mathbf{v}_k$ has a 1 in entry j if k is the color of unknown j, and is otherwise zero. In this $c = 3$ case,

$$\mathbf{v}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathbf{v}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \qquad \mathbf{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}. \tag{4.17}$$

In fact, a function maps from node/unknown index j to color k,

$$k = k(j), \tag{4.18}$$

and $(\mathbf{v}_k)_j = \delta_{k,k(j)}$. Note $k(j) = j \mod 3$ in this case.

Finally we replace (4.8) with

$$J_{ij} = \frac{\partial F_i}{\partial x_j} \approx \frac{F_i(\mathbf{x} + \delta \mathbf{v}_{k(j)}) - F_i(\mathbf{x})}{\delta}. \tag{4.19}$$

The right sides of (4.8) and (4.19) compute exactly the same entries J_{ij}, but the latter requires far fewer evaluations of $\mathbf{F}$. In particular, all columns of J with color k are computed by (4.19) just using the *smallest* j for which $k(j) = k$. Thus, given a c-coloring of $G(J)$ there are exactly c evaluations $\mathbf{F}(\mathbf{x} + \delta \mathbf{v}_{k(j)})$, plus one more for $\mathbf{F}(\mathbf{x})$, to fill J using (4.19).

The news is good when we actually try it. The result is almost as fast in this 1D PDE case as using the analytical Jacobian:

```
$ timer ./reaction -snes_converged_reason -da_refine 10 -snes_fd_color
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 3
on 8193 point grid:  |u-u_exact|_inf/|u|_inf = 1.75633e-10
real 0.06
```

Compare the run on page 88.

Actually counting the number of function evaluations in the various cases is a good idea, and there is no need to alter `reaction.c` to do so.[19] Use `-log_summary` and look at the output. Because we want the "`SNESFunctionEval`" lines, `grep` extracts the count for the runs above:

```
$ ./reaction -da_refine 10 -snes_fd -snes_max_funcs 100000 -log_summary | grep SNESFunctionEval
SNESFunctionEval   24586 1.0 3.3170e+00 1.0 ...
$ ./reaction -da_refine 10 -snes_fd_color -log_summary | grep SNESFunctionEval
SNESFunctionEval      13 1.0 4.7762e-03 1.0 ...
$./reaction -da_refine 10 -log_summary | grep SNESFunctionEval
SNESFunctionEval       4 1.0 9.3341e-04 1.0 ...
```

The exact number of F evaluations was 24586, 13, and 4, respectively.

Before proceeding, consider the structured 2D grid used in Chapter 3, and in particular finite-difference scheme (3.9) for the Poisson equation (3.1). How would the corresponding graph-coloring approach work? The stencil of the scheme involves five unknowns $u_{i,j+1}, u_{i-1,j}, u_{i,j}, u_{i+1,j}, u_{i,j-1}$ (Figure 3.7). Therefore the graph $G(J)$ in this case includes a complete graph on five vertices, a K_5, as a subgraph at each (generic interior) node in the structured grid, as shown in Figure 4.7. Clearly $\chi(G(J)) \geq 5$. However, when PETSc does the default incidence-degree-ordering coloring algorithm then it finds that indeed five colors suffice to color $G(J)$, at least in refined-grid cases. We will be able to test verify this behavior when we have a SNES-based 2D PDE example to work with, as in the next Chapter.

See note for page 90.

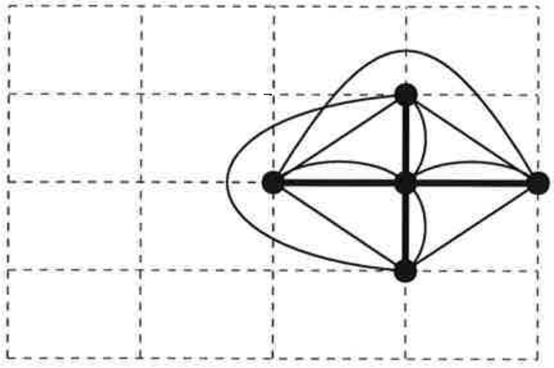


Figure 4.7: For the 2D finite-difference scheme used in Chapter 3, the graph $G(J)$ has a K_5 at every node because the stencil (thick) involves five unknowns.

Jacobian-free Newton-Krylov (JFNK)

Besides the finite-difference Jacobian approach above, using equation (4.8) or (4.19) to compute the entries of the matrix, there is a ~~different~~ another approach which requires no assembled Jacobian matrix at all. It approximates matrix-vector products by with the Jacobian a finite difference formula,

[19] Do not add print statements!

and uses a Krylov-type method to solve (4.5a) from a space

$$\mathcal{K} = \operatorname{span}\{\mathbf{r}, J\mathbf{r}, J^2\mathbf{r}, \dots, J^{m-1}\mathbf{r}\}. \tag{4.20}$$

Here $J = J_{\mathbf{F}}(\mathbf{x}_k)$ is the Jacobian at iterate k and $\mathbf{r}$ is a (linear) residual in equation (4.5a).

It goes by the name "Jacobian-free Newton-Krylov" [Knoll and Keyes, 2004] or just "JFNK". To give more detail, suppose we use initial estimate 0 for the solution to (4.5a). The residual is $\mathbf{r} = -\mathbf{F}(\mathbf{x}) - J0 = -\mathbf{F}(\mathbf{x})$. To compute $J\mathbf{r}$, $J^2\mathbf{r} = J(J\mathbf{r}), \dots$ in (4.20), recall definition (4.2) of the derivative of $\mathbf{F}$. It implies that if $\mathbf{v}$ is any vector then

$$J\mathbf{v} \approx \frac{\mathbf{F}(\mathbf{x} + \delta\mathbf{v}) - \mathbf{F}(\mathbf{x})}{\delta} \tag{4.21}$$

for small $\delta \neq 0$. Approximating the Jacobian-vector product by (4.21) avoids evaluation of any entries of J.

Computing $\mathbf{r} = -\mathbf{F}(\mathbf{x})$, to get the Krylov method started, requires evaluating $\mathbf{F}$. Then each successive vector in basis (4.20) requires an additional evaluation of $\mathbf{F}$, namely

$$J^{\ell}\mathbf{r} \approx \frac{\mathbf{F}(\mathbf{x} + \delta J^{\ell-1}\mathbf{r}) - \mathbf{F}(\mathbf{x})}{\delta} \tag{4.22}$$

for $\ell = 1, \dots, m$. Equation (4.22), the central calculation in JFNK, uses m evaluations of $\mathbf{F}$ to compute the whole Krylov basis (4.20). By contrast, use of finite difference formula (4.8) to compute a generic $N \times N$ Jacobian J requires N evaluations of $\mathbf{F}$. Thus JFNK ~~is~~ efficient if (4.5a) is solved to desired tolerance in $m \ll N$ Krylov iterations.

can be

However, there are a few basic points to consider. First, once you have a matrix you can do many things, such as incomplete matrix factorizations, other than computing a matrix-vector product or a Krylov basis, so giving up on matrices may be premature. Second, and related, we may need to precondition linear equation (4.5a), in which case we want the Krylov space for the linear operator $M^{-1}J$, not for J itself. Also, in cases such as structured-grid PDE schemes where it can be applied, we have seen that coloring greatly reduces the cost of using (4.8) to assemble J. JFNK is a good strategy to the extent that it actually *works*, and to the extent it beats the competition.

* Note JFNK is implemented in PETSc and invoked by option `-snes_mf`, where "`mf`" in stands for "matrix-free." We shall see below, in the important and even essential preconditioned case, that there may be a matrix involved in JFNK anyway. The method never involves fully-assembling the Jacobian matrix itself, however, and thus the "Jacobian-free" label is justified.

The issues above can be pursued in more detail on a concrete example, so let us give it a try on a very coarse grid in the 1D diffusion-reaction PDE example `reaction.c`:

* Note that -snes_fd -snes_fd_color and -snes_~~fd~~ mf ~~can also be~~ ~~i[illegible]~~ ~~have~~ also have PETSc functional interfaces which provide more generality and ~~flexibilty~~ flexibility.

Also recall that preconditioning matrices M and M^{-1} are usually never assembled. Even if an assembled preconditioner-material matrix P is present, M is generally constructed nontrivially from P. For example, if we supply some assembled matrix P as the preconditioner material, but we ask for ILU(0) preconditioning, then M is actually the product of the ILU(0) factors of P, not P itself.

Jacobian cases

At this point we risk overwhelming the reader with options, so we pause to review the possibilities before testing them on the `reaction.c` problem.

Table 4.2 summarizes options relating to residual and Jacobian call-backs in `SNES`-using codes. If no Jacobian or approximate Jacobian routine is provided in the user-written code then only finite-difference evaluation of an assembled Jacobian matrix (i.e. `-snes_fd` or `-snes_fd_color`) or finite-difference evaluation of the Jacobian-vector product inside the Krylov method (`-snes_mf`, i.e. JFNK) are available. If a Jacobian routine *is* provided then the Newton iteration itself occurs when no option is given. However, the provided Jacobian may be used only to precondition the JFNK Jacobian-vector product, which is option `-snes_mf_operator`. This last option may give quadratic convergence even if the provided Jacobian is inexact, that is, even if P is a somewhat-poor approximation of the Jacobian.

You forgot -snes_fd_color ✓

-snes_fd_color
✓
✓
not needed

	no option	`-snes_fd`	`-snes_mf`	`-snes_mf_operator`
only **F**	error	✓	✓	error
F and P	✓	✓	✓	✓
F and J	✓	✓	✓	✓

Table 4.2: Jacobian options when using `SNES`, compared by need for user-implemented functions. Symbol "P" denotes an easy-to-invert approximate Jacobian while "J" denotes the actual Jacobian. A big check mark shows recommended usage.

From the code side, at a minimum a method for **F** must be implemented in all cases, as there is no other way for PETSc to know what equations you are solving. There are two ways of providing **F**: If the problem is based on a structured grid, as in `reaction.c` above, use `DMDASNESSetFunctionLocal()`. In general, use `SNESSetFunction()`. If only **F** is provided, do not create or preallocate a `Mat` for the Jacobian, as this is done internally by the `SNES` for `-snes_fd` runs, while no assembled Jacobian `Mat` exists for `-snes_mf` runs.

If an exact (J) or approximate (P) Jacobian function is implemented then there are two cases:

(i) On a structured grid, provide the function (e.g. "`FormJacobianLocal()`") using `DMDASNESSetJacobianLocal()`. In this case the `Mat` holding the

Testing JFNK with preconditioning

We now do refined-grid runs of `reaction.c` to show the `-snes_mf_operator` option in action. We have already seen that the `-snes_fd_color` option is effective for reducing evaluations of **F** if a Jacobian is not implemented, but in the same case the un-preconditioned JFNK method `-snes_mf` has serious difficulties as it requires unreasonable numbers of Krylov iterations and function evaluations. Now we can show that `-snes_mf_operator` using only a rough approximation of the Jacobian as a preconditioner gives good performance.

Specifically, suppose we modify this line in `reaction.c` (Code 4.4),

```
col[1] = i;    v[1] = 2.0 - h*h * dRdu;
```

to remove "`- h*h * dRdu`", corresponding to the nonlinear term $\rho\sqrt{u}$ in (4.15), to ~~get~~

obtain

```
col[1] = i;    v[1] = 2.0;
```

This change, which we call a "`J->P`" below, keeps the tridiagonal sparsity pattern of the Jacobian. Furthermore it preserves many characteristics of the spectrum of the linearizations of the operator in (4.15) by keeping the highest-order term u''.

With this change, convergence is slowed for no option, that is, when we try to use the new approximate Jacobian as though it were exact. Specifically, on a coarse grid the number of iterations goes from 4 to 15, and the residual norms suggest that convergence is no longer quadratic:

```
$ ./reaction -da_refine 4 -snes_monitor    # before change
  0 SNES Function norm 1.671129624018e-02
  1 SNES Function norm 3.609252641302e-04
  2 SNES Function norm 4.167490508951e-07
  3 SNES Function norm 4.935230509260e-13
on 129 point grid:  |u-u_exact|_inf/|u|_inf = 7.39662e-07
$ ./reaction -da_refine 4 -snes_monitor    # with J->P change
  0 SNES Function norm 1.671129624018e-02
  1 SNES Function norm 3.822032062916e-03
...
 14 SNES Function norm 3.879363487638e-10
 15 SNES Function norm 1.119521798815e-10
on 129 point grid:  |u-u_exact|_inf/|u|_inf = 7.38159e-07
```

This loss of Newton method performance from a significantly incorrect Jacobian is expected in theory [Kelley, 2003].

However, for this modified Jacobian case, `-snes_mf_operator` is now a fast option for higher-resolution grids, fully competitive with the exact Jacobian and finite-difference-by-coloring cases already seen. Again on a 8000 point grid, using the approximate Jacobian "as is" causes too many Newton iterations and less-than-quadratic convergence:

```
$ timer ./reaction -snes_converged_reason -da_refine 10    # with J->P change
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 15
on 8193 point grid:  |u-u_exact|_inf/|u|_inf = 1.36236e-09
real 0.13
```

Now we try -snes_mf_operator, which is designed for this approximate-Jacobian situation:

```
$ timer ./reaction -snes_converged_reason -da_refine 10 -snes_mf_operator  # with J->P change
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 4
on 8193 point grid:  |u-u_exact|_inf/|u|_inf = 1.80588e-10
real 0.05
```

The number of iterations and the time are significantly reduced, and both are comparable to the exact Jacobian case (page 88).

One might give the following summary advice on SNES usage:

> Before implementing a Jacobian, try finite-difference evaluation -snes_fd first, including a look at whether coloring (option -snes_fd_color) applies to your case. As a general rule, it can be applied and it is often effective on a structured grid, but on an unstructured grid (Chapters 8 and 10) the coloring method requires additional work. Note JFNK with no preconditioning (option -snes_mf) is rarely effective, though easy to try. Now consider implementing the exact Jacobian J. If that seems like too much work or is too error-prone, consider a simpler approximate Jacobian P used with option -snes_mf_operator. In PDE cases, in particular, P might only capture the highest-order derivatives in J. To test convergence with P, compare "no option" runs which use P as though it were J, but wherein less-than-quadratic convergence is expected, and -snes_mf_operator runs where quadratic convergence should be recovered.

Add section on lagging preconditioner and lagging recalculation of Jacobian using -snes_mf_operator

Line search Newton methods

Once the Newton step $\mathbf{s}_k$ solving equation (4.5a) is computed, the new iterate $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$ from (4.5b) may not be what we want. For example, the residual norm $\|\mathbf{F}(\mathbf{x}_k + \mathbf{s}_k)\|$ may exceed $\|\mathbf{F}(\mathbf{x}_k)\|$, suggesting that no progress is being made in solving (4.1). Improving this situation is the problem of *globalizing* the convergence of the Newton iteration, so that the iterates $\mathbf{x}_k$ are more likely to head toward the sometimes-small region of quadratic convergence around the solution to (4.1).

The *line search* globalization technique [Dennis and Schnabel, 1983] is to replace (4.5b) with

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k \mathbf{s}_k. \qquad (4.26)$$

The coefficient v_ℓ is equal to $v_{r,s} = v(x_r, y_s)$ for $(x_r, y_s) \in \square_{i,j}$ corresponding under the element map to $(\xi_\ell, \eta_\ell) \in \square_*$. Also, on the reference element the (x,y) gradient has formula:

$$(\nabla_{x,y} v)(\xi,\eta) = \left\langle \frac{2}{h_x} \sum_{\ell=0}^{3} v_\ell \frac{\partial \chi_\ell}{\partial \xi}, \frac{2}{h_y} \sum_{\ell=0}^{3} v_\ell \frac{\partial \chi_\ell}{\partial \eta} \right\rangle. \tag{5.24}$$

Quadrature

We do not plan to exactly-compute the integrals in (5.11), or in the corresponding sum for (5.8). Instead we will use numerical integration (quadrature). One reason is that for general p it would be quite challenging to exactly-integrate the term "$|\nabla u|^p$."

First, change-of-variables transfers the integral to the reference element. Suppose $v(x,y)$ is any integrable function on element $\square_{i,j}$. Using the element map and (5.18) we have

$$\int_{\square_{ij}} v(x,y)\, dx\, dy = \frac{h_x h_y}{4} \int_{\square_*} v(\xi,\eta)\, d\xi\, d\eta \tag{5.25}$$

where $v(\xi,\eta) = v(x(\xi,\eta), y(\xi,\eta))$.

Next, recall Gauss-Legendre quadrature on integrals in one dimension [Greenbaum and Chartier, 2012]:

$$\int_{-1}^{1} f(z)\, dz \approx \sum_{q=0}^{n-1} w_q f(z_q). \tag{5.26}$$

The degree n rule (5.26) is exact for polynomials of degree $2n-1$ and less. For degrees $n = 1, 2, 3$, the quadrature *nodes* z_q and *weights* w_q for integration over the interval $[-1, 1]$ are given in 5.1. Such a one-dimensional rule extends to tensor product formulas for integrals ~~like~~ (5.25) over $\square_*$, namely

such as

$$\int_{\square_*} v(\xi,\eta)\, d\xi\, d\eta \approx \sum_{r=0}^{n-1} \sum_{s=0}^{n-1} w_r w_s v(z_r, z_s). \tag{5.27}$$

n	nodes z_q	weights w_q
1	0	2
2	$-\frac{1}{\sqrt{3}}, +\frac{1}{\sqrt{3}}$	1, 1
3	$-\sqrt{\frac{3}{5}}, 0, +\sqrt{\frac{3}{5}}$	$\frac{5}{9}, \frac{8}{9}, \frac{5}{9}$

Table 5.1: Nodes and weights for low-degree Gauss-Legendre quadrature rules, for integrals (5.26).

Formulas (5.23), (5.24), and (5.25) are used to evaluate the integrand in (5.11) on the reference element using the nodal values of u and f. Informally, we define

$$G(\xi,\eta) = \left[\frac{1}{p} |\nabla u|^p - f u\right]_{\square_*} \tag{5.28}$$

The details are in Exercise 5.5.

In these terms, using quadrature (5.27), formula (5.11) becomes computable:

$$\begin{aligned} I^h[u] &= \frac{h_x h_y}{4} \sum_{i=0}^{m_x} \sum_{j=0}^{m_y} \int_{\square_*} G(\xi,\eta)\, d\xi\, d\eta \\ &\approx \frac{h_x h_y}{4} \sum_{i=0}^{m_x} \sum_{j=0}^{m_y} \sum_{r=0}^{n-1} \sum_{s=0}^{n-1} w_r w_s G(z_r, z_s) \end{aligned} \tag{5.29}$$

6

This is enough for a prototype implementation.

Implementation with objective only

Our code is displayed in full in six parts, Codes 5.1–5.6. However, the last part, which implements (5.8), is shown after we get the code running initially using only an implementation of the objective (5.1).

The first part (Code 5.1) configures the problem. Options choose the exponent p and whether we want to use a manufactured solution for verification. The corresponding variables are stored in a context `struct` called "`PLapCtx`" so they can be passed among our functions as needed. Code 5.1 also includes a method to print basic information available at the end of the solve.

The exact solution is constructed in Code 5.2. It is about as simple as possible for the p-laplacian, and we only construct it in the $p = 2$ and $p = 4$ cases.[6] In brief summary, the exact solution $u(x,y)$ is chosen exactly as in Chapter 3, namely equation (3.10):

$$u(x,y) = (x^2 - x^4)(y^4 - y^2), \tag{5.30}$$

and f is set accordingly by (5.9). In cases where we do not test against a manufactured solution, we simply set $f = 1$. For now, only zero Dirichlet boundary values $g(x,y) = 0$ are implemented.

The by-hand derivative calculation which generates $f(x,y)$ from the exact solution $u(x,y)$ is acceptably simple in the restricted cases $p = 2, 4$, but it is still error-prone. However, because by-hand calculation errors are likely to be un-correlated to implementation errors in other parts of the code, agreement is likely to reflect correctness.

[6] If $p = 2$ then the strong form (5.9) is the linear Poisson equation addressed in Chapter 3.

7
Parallel scaling and performance

Percent completed: 5%.

Discuss petscInt of 64 bit integer for large problems
and
petscReal of __float for problems with large condition numbers where 64 bit double is not enough,

8

An unstructured finite element method

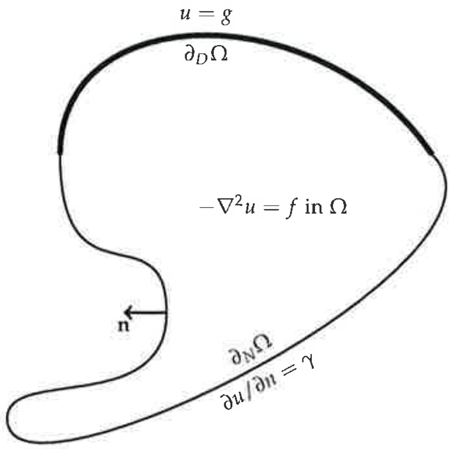
The cliched Poisson problem ~~can~~ will be exploited some more in this chapter. It gives us the opportunity to use PETSc for important tasks we have not yet seen, including reading an unstructured mesh into PETSc `Vecs`, symmetric implementation of boundary conditions, and explicit preallocation of a `Mat` in parallel.

bullets?

Example: The Poisson problem

Let $\Omega \subset \mathbb{R}^d$ be a bounded (open) region. Suppose its boundary $\partial\Omega$ is well-behaved, for instance that it is Lipschitz-continuous [Ciarlet, 2002, section 1.2] or even polygonal. Suppose $\partial\Omega$ is decomposed into (measurable) disjoint subsets $\partial_D\Omega$ and $\partial_N\Omega$ whose union is the entire boundary $\partial\Omega$. The *Poisson problem*, in strong form and including nonhomogeneous Dirichlet and Neumann boundary conditions, is

$$\begin{aligned} -\nabla^2 u &= f \quad \text{on } \Omega, \\ u &= g \quad \text{on } \partial_D\Omega, \\ \frac{\partial u}{\partial n} &= \gamma \quad \text{on } \partial_N\Omega \end{aligned} \tag{8.1}$$


$u = g$
$\partial_D\Omega$
$-\nabla^2 u = f$ in Ω
$\mathbf{n}$
$\partial_N\Omega$
$\partial u/\partial n = \gamma$

where $\mathbf{n}$ is the outward unit normal on $\partial\Omega$ and $\partial u/\partial n = \mathbf{n} \cdot \nabla u$. The data of problem (8.1), besides the region Ω and its boundary, includes a *source term* $f \in L^2(\Omega)$, *Dirichlet data* $g \in L^2(\partial_N\Omega)$, and *Neumann data* $\gamma \in L^2(\partial_N\Omega)$.

As (8.1) is stated there may be no solution where "$\nabla^2 u$" makes sense as a continuous function, even for polygonal regions, continuous boundary values, and continuous source functions. In particular, there may be no $u \in C^2(\Omega)$ which is continuous up to the boundary (i.e. $u \in C(\bar{\Omega})$) and so that $\nabla^2 u = f$. There is, however, a solution if we change to a *weak formulation*.[1] Furthermore, if $\partial_D\Omega$ has positive size, in an appropriate sense [Ciarlet, 2002, Theorem 1.2.1], then the solution of the weak formulation is unique. We will state the weak

[1] A proof of this well-posedness claim is in Ciarlet [2002] and in Evans [2010]. These are technicalities for us, however, as our goal is computational performance in cases where the Poisson problem is mathematically well-behaved and easily approximated.

formulation after ~~glossing the definitions of the needed function~~ spaces.

Recalling $L^2(\Omega)$ is the space of all square-integrable real functions on Ω, define

$$H^1(\Omega) = \{u \in L^2(\Omega) \,|\, \nabla u \text{ exists a.e. and } \nabla u \in L^2(\Omega)\},$$

Chapter 5 already introduced the more general W spaces. Rewrite this section taking that into account.

which is a Sobolev space [Evans, 2010]. This space has two subsets we use, namely functions with value g on $\partial_D\Omega$ and those with value 0 on $\partial_D\Omega$, respectively, which we denote $H^1_g(\Omega)$ and $H^1_0(\Omega)$. Note that $H^1_0(\Omega)$ is a linear subspace of $H^1(\Omega)$. While $H^1_g(\Omega)$ is generally not a subspace (e.g. because the zero function is not in it), it is an affine subspace, and we refer to both $H^1_g(\Omega)$ and $H^1_0(\Omega)$ as "subspaces".

To get to the weak formulation of the Poisson problem we suppose we already have a classical solution u of (8.1). Then we choose any $v \in H^1_0(\Omega)$, multiply the first equation in (8.1) by v, and integrate by parts:

$$\int_\Omega \nabla u \cdot \nabla v - \int_{\partial\Omega} \frac{\partial u}{\partial n} v = \int_\Omega f v.$$

Next we use the other data, namely that $v = 0$ on $\partial_D\Omega$ and that there is Neumann data γ on $\partial_N\Omega$:

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v + \int_{\partial_N\Omega} \gamma v \quad \text{for any } v \in H^1_0(\Omega). \tag{8.2}$$

Equation (8.2) is the weak formulation of the Poisson problem. Any $u \in H^1_g(\Omega)$ satisfying (8.2) is called a *weak solution*. A key observation is that u itself incorporates the Dirichlet boundary condition, because it lives in $H^1_g(\Omega)$, while both the Neumann boundary data γ and the source function f appear in equation (8.2).

Main ideas of strong and weak formulations:

- If $u \in H^1_g(\Omega)$ solves the strong form (8.1) then it solves (8.2) also.
- If $u \in H^1_g(\Omega)$ solves the weak form (8.2) then we accept it, by definition, as a solution of the Poisson problem.

A finite element method (FEM) for the Poisson problem in the plane

Again, rewrite slightly taking into account material written already in chapter 5.

An FEM for the Poisson problem comes from requiring the weak formulation (8.2) to be true for u in a much smaller, indeed finite-dimensional, subspace of $H^1_g(\Omega)$, and for test functions v ranging over a finite-dimensional subspace of $H^1_0(\Omega)$. In the "Galerkin" method here, these subspaces will be essentially the same. We will build these subspaces, in the current example, using an unstructured triangulation on $\Omega \subset \mathbb{R}^2$; from now on in this Chapter we restrict to $d = 2$ dimensions.

Furthermore, to make our finite-dimensional spaces true subspaces of $H^1(\Omega)$—to make our FEM *conforming*—we require that Ω be polygonal, with $\partial\Omega$ a closed polygon. Segments of $\partial\Omega$ must have positive length, and be either entirely in $\partial_D\Omega$ or entirely in $\partial_N\Omega$.

We also assume $\partial_D\Omega$ is a closed set so that, at vertices of $\partial\Omega$ where the Dirichlet boundary and Neumann boundary meet, the vertex is Dirichlet.

By definition, a *triangulation* is a finite set of non-overlapping, non-empty open triangles $\triangle_k \subset \mathbb{R}^2$ which tile Ω:

$$\mathcal{T}_h = \left\{ \triangle_k \;\middle|\; \cup_k \overline{\triangle}_k = \overline{\Omega} \quad \text{and} \quad \Omega_k \cap \Omega_l = \emptyset \text{ if } k \neq l \right\}.$$

We index the K triangles in $\mathcal{T}_h$ by $k = 0, \dots, K-1$. The N vertices (nodes) in $\mathcal{T}_h$ are indexed by $j = 0, 1, \dots, N-1$, with locations

$$\mathbf{x}_j = (x_j, y_j).$$

An example triangulation is shown in Figure 8.1.

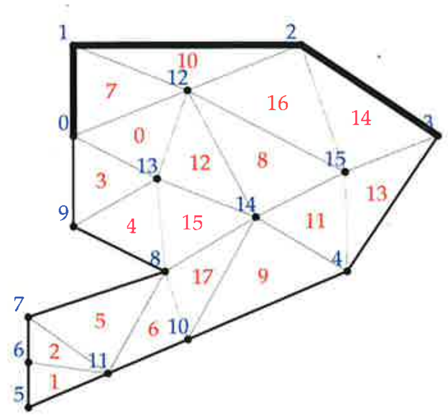


Figure 8.1: A triangulation $\mathcal{T}_h$ with $K = 22$ triangles (elements) numbered $k = 0, 1, \dots, K-1$ (red) and $N = 16$ nodes numbered $j = 0, 1, \dots, N-1$ (blue). Nodes $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ are in the Dirichlet boundary $\partial_D\Omega$.

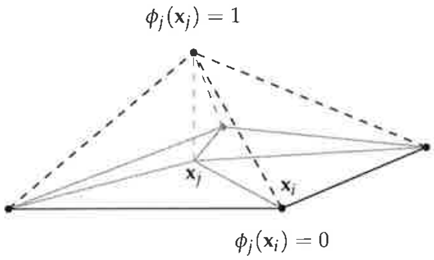
For now the reader can regard the subscript "h" in "$\mathcal{T}_h$" as merely-traditional notation. It denotes the typical or maximum size h (e.g. diameter) of the triangles, and it serves as a reminder that we want to approximate the solution in the limit $h \to 0$. Also note that, in contrast to Elman et al. [2005], which we generally follow, and other references on the FEM or its implementation in languages like MATLAB, our indexing is zero-based. This is so because we implement in C and we want to avoid any confusion when comparing text and codes. Breaking long mathematical traditions, rows and columns of vectors and matrices will also have numbering starting with zero in this book.

Maybe move this to chapter 5

We informally call the triangles *elements*, though there is more to the definition of "element." We are going to approximate the Poisson problem with $\mathbf{P}_1$ finite elements, which means that our finite-dimensional subspace contains only piecewise-linear functions which are linear on each triangle $\triangle_k$.

For each node j there is a $\mathbf{P}_1$ basis function, or "hat" function, $\phi_j(x, y)$ which is linear on each triangle, continuous on all of $\overline{\Omega}$, and equal to one on only one node j:

$$\phi_j(\mathbf{x}_i) = \delta_{ij}.$$



The functions ϕ_j are in $H^1(\Omega)$, with piecewise-constant partial derivatives $\partial\phi_j/\partial x$ and $\partial\phi_j/\partial y$. Also, the set $\{\phi_j\}_{j=0,\dots,N-1}$ is linearly-independent. On each triangle, ϕ_j has three degrees of freedom, because on $\triangle_k$ there exist coefficients $A_k, B_k, C_k \in \mathbb{R}$ so that

$$\phi_j(\mathbf{x}) = A_k + B_k x + C_k y \quad \text{on } \triangle_k,$$

where $\mathbf{x} = (x, y)$.

We can immediately use these basis functions to approximate the Dirichlet data g and extend it to the region Ω. We will assume from now on that the Dirichlet boundary $\partial_D\Omega$ is closed, and index the L nodes which are in the Dirichlet boundary by $\mathbf{x}_{j_l} \in \partial_D\Omega$ for $l = 0, \dots, L-1$. (Figure 8.1 shows an example with $L = 4$ and $j_l = l$ for $l = 0, 1, 2, 3$, but any subset of boundary points can be Dirichlet as long as the index values j_l are well-defined.) Now we can define an extended interpolant $\hat{g}$ of g as the function which has the correct value on the Dirichlet boundary nodes and which extends to all of Ω in a continuous and piecewise-linear way:

$$\hat{g}(\mathbf{x}) = \sum_{l=0,\dots,L} g(\mathbf{x}_{j_l})\phi_{j_l}(\mathbf{x}). \tag{8.3}$$

By using $\hat{g}$ and the basis functions ϕ_j, we can now describe ~~three~~ two finite-dimensional subspaces of $H^1(\Omega)$:[2]

$$\begin{aligned} S^h &= \operatorname{span}\{\phi_j \,|\, \text{all } j\}, \\ S^h_0 &= \operatorname{span}\{\phi_j \,|\, \mathbf{x}_j \notin \partial_D\Omega\} \subset S^h, \\ S^h_g &= S^h_0 + \hat{g} \subset S^h. \end{aligned}$$

~~both~~ we then introduce which is not technically a subspace of S^h.

Then $\dim(S^h) = N$ while $\dim(S^h_0) = \dim(S^h_g) = N - L$, with S^h_g only an affine subspace of S^h.

Our FEM requires that the weak formulation (8.2) be true of $u_h \in S^h_g$ for all $v \in S^h_0$. Thus we first write u_h in the basis for S^h_g using $N - L$ unknown coefficients u_j:

$$u_h(\mathbf{x}) = \hat{g}(\mathbf{x}) + \sum_{\mathbf{x}_j \notin \partial_D\Omega} u_j\,\phi_j(\mathbf{x}). \tag{8.4}$$

Then we require that the weak formulation hold for all ϕ_i in the basis of S^h_0. That is, using definition (8.3) and expansion (8.4), we require

$$\begin{aligned} \sum_{\mathbf{x}_j \notin \partial_D\Omega} u_j \int_\Omega \nabla\phi_j \cdot \nabla\phi_i &= \int_\Omega f\phi_i + \int_{\partial_N\Omega} \gamma\phi_i \\ &\quad - \sum_{l=0,\dots,L} g(\mathbf{x}_{j_l}) \int_\Omega \nabla\phi_{j_l} \cdot \nabla\phi_i \end{aligned} \tag{8.5}$$

for all i such that $\mathbf{x}_i \notin \partial_D\Omega$. The coefficients u_j, for all j such that $\mathbf{x}_j \notin \partial_D\Omega$, are the unknowns in this equation.

Note that the support (i.e. nonzero set) of ϕ_j includes only the node $\mathbf{x}_j$ and all triangles (elements) $\triangle_k$ for which $\mathbf{x}_j$ is a node of $\triangle_k$. Thus the integral "$\int_\Omega \nabla\phi_j \cdot \nabla\phi_i$" in (8.5) is ~~usually zero. Specifically, it is~~ zero if $\mathbf{x}_i$ and $\mathbf{x}_j$ are not both nodes of at least one triangle in the triangulation. This is the property that produces very sparse matrices.

Triangular meshes from TRIANGLE

PETSc itself does not include any tools for triangulating regions of the plane, sowe use the widely-available and easy-to-use TRIANGLE[3]

Some

[2] Traditionally, basis functions for S^h_g are called the *trial* functions, and basis functions for S^h_0 are called *test* functions. We will generally just use the labels "S^h_g" and "S^h_0."

[3] See www.cs.cmu.edu/~quake/ triangle.html for documentation and source code. TRIANGLE may be available as a package in your operating system.

software [Shewchuk, 1996] for this task. TRIANGLE is both limited to planar regions and only capable of writing ASCII files. Thus it is not a choice for performance, but of convenience.

TRIANGLE uses a simply-formatted ASCII file (extension .poly) as input to describe a polygonal region Ω, and to indicate Dirichlet and Neumann portions of the boundary $\partial\Omega$. For example, consider the input file bump.poly shown in Code 8.1. This example polygon, a rectangle with a triangular bump in the base, is shown in Figure 8.2. It will reappear several times in this book as we solve more interesting PDEs ~~on~~ it. The two apparently-unnecessary vertices introduced along the bottom help identify the Neumann part of the boundary, but note that bump.poly includes a Dirichlet/Neumann flag along each boundary segment.

with

ch8/bump.poly

```
# A polygon with nine vertices, in 2D, no attributes, and no markers for
# these vertices.
9 2 0 0
# Outside rectangle has these vertices:
 1   -3.0  0.0
 2   -3.0  3.0
 3    3.0  3.0
 4    3.0  0.0
# Triangular bump has these vertices:
 5    1.0  0.0
 6    1.0  1.0
 7    0.0  0.0
# These added vertices help mark Neumann boundary segments
 8    2.0  0.0
 9   -1.5  0.0
# There are nine segments on the boundary, each with one marker.
# Marker values: 2 = Dirichlet, 3 = Neumann.
9 1
 1   1 2   2
 2   2 3   2
 3   3 4   2
 4   4 8   3
 5   8 5   3
 6   5 6   2
 7   6 7   2
 8   7 9   3
 9   9 1   3
# Zero holes.  It is a simply-connected region.
0
```

Code 8.1: A description of the boundary polygon in Figure 8.3, suitable for reading by TRIANGLE.

The triangulation shown in Figure 8.3 came from a single command which asks TRIANGLE to take bump.poly and generate a triangulation which has a polygon output file (option -p), relatively-

```
$ c3convert -f bump.1
```

This reads ASCII files `bump.1.{node,ele,poly}` and writes a PETSc-formatted binary file `bump.1.petsc`.

We will not show `c3convert.c`, but we summarize the high points. First PETSc is initialized and we ~~get~~ the rank of the current (MPI) process. We only ask the first process ("rank zero") in the MPI communicator to do any work.[5] This part of the code first reads the header information in the `.node` file, and allocates PETSc `Vecs` according. We use `VecCreateSeq` to allocate the a sequential `Vec vx`, which contains the x-coordinate of the nodes, only on the rank zero process. Then `VecDuplicate` is used to allocate two more `Vecs` with the same layout, `vy` and `vBT`. This last `Vec` will contain a flag $\{0,2,3\}$ for each node, where 0 is an interior node, 2 is a Dirichlet boundary node, and 3 is a Neumann boundary node.

obtain

Then we read the node locations from the `.node` file. The reading itself is done with the standard C library call `fscanf`. Then `VecSetValues` is used to set one entry at a time. After setting these values, which stores a list of entrys into an internal PETSc dynamic data structure, we ask PETSc to assemble the `Vecs`.

The next part of `c3convert.c` reads boundary polygon information from the `.poly` file. Each segment of the boundary polygon corresponds to two node indices. We store the segments in a `Vec` with blocksize 2. Then we read the header information in the `.ele` file and allocates a `Vec` called `vE` for the elements. This part of the code is an important transformation of the data structures. In fact, `vE` has block size 15,[6] and, in contrast to the format from TRIANGLE, it contains all the information about each element that we need to do assemble the matrix equation. Each of its blocks is the C `struct` shown in Code 8.2.

extract from ch8/readmesh.h

```
typedef struct {
  PetscScalar j[3],  // global indices of vertices (nodes) j[0], j[1], j[2]
              bN[3], // boundary type of node:  bN[0], bN[1], bN[2] in {0,1,2}
              bE[3], // isboundary for edge:  bE[0], bE[1], bE[2] in {0,1},
                     //   where bE[0] = <0,1>, bE[1] = <1,2>, bE[2] = <2,0>
              x[3],  // node x-coordinates x[0], x[1], x[2]
              y[3];  // node y-coordinates y[0], y[1], y[2]
} elementtype;
```

Code 8.2: The `elementtype` struct.

In the next part of `c3convert.c` we fill the `Vec` for elements with all of the information read so far, including the node indices for each

[5] This code can be invoked "`mpiexec -n NN c3convert`", but it behaves as a serial code.

[6] A PETSc `Vec` is designed to hold `PetscScalar` data types, i.e. `double`. So we are being quite wasteful for integer indices and boolean flags.

element which we read from the `.ele` file. This stage is fundamentally serial, because we must look at the entire mesh to find the node coordinates and node/segment boundary type for each node and edge of each element. In this part there is an important detail about triangulations, which affects the data structure for elements. Namely, we cannot tell if an edge of an element is in the boundary just by whether both endpoints are in the boundary. For example, the element (triangle) labeled "5" in Figure 8.3 has an edge from node 5 (on the Dirichlet boundary) to node 7 (on the Neumann boundary). But triangle 5 is *not* a boundary element. Thus we need to have list of flags for the boundary segments themselves. Thus the `elementtype` structure above has both a boundary type for each node of each element and a boundary type for each edge of each element.

At this point we have the ~~whole~~ triangulation in~~to~~ PETSc `Vecs`. The almost-last part of `c3convert.c` simply creates a PETSc "viewer" and "view"* all of the `Vecs` which contain the mesh. We will be able to reread these `Vecs` in parallel~~, as long as we re-read them in the same order~~. The final bit of `c3convert.c` checks if option `-check` is given, and if so we read back the binary file in parallel. This part of `c3convert.c` calls two methods from a separate code (and re-used component) `readmesh.c` (Codes 8.3 and 8.4).

entire

* save to disk in PETSc's binary format

The first method `getmeshfile()` finds a PETSc binary file from the `-f` option. The other major method `readmesh()`, in Code 8.4, creates and reads three `Vecs` in parallel from it, using utility methods `createloadname()` and `getcheckmeshsizes()`. Note that prefixes are set on each `Vec` so that the block size is correctly read.

ch8/readmesh.c **part I**

```
PetscErrorCode getmeshfile(MPI_Comm comm, const char suffix[],
                           char filename[], PetscViewer *viewer) {
  PetscBool      fset;
  PetscOptionsBegin(comm, "", "options for readmesh", "");
  PetscOptionsString("-f", "filename root with PETSc binary, for reading", "", "",
                     filename, PETSC_MAX_PATH_LEN, &fset);
  PetscOptionsEnd();
  strcat(filename,suffix);
  PetscPrintf(comm,"  opening mesh file %s ...\n",filename);
  PetscViewerBinaryOpen(comm,filename,FILE_MODE_READ,viewer);
  return 0;
}
```

Code 8.3: Determine the filename of a PETSc binary file that has a mesh.

ch8/readmesh.c **part II**

```
PetscErrorCode createloadname(MPI_Comm comm, PetscViewer viewer, const char prefix[],
                              const char name[], Vec *v) {
  VecCreate(comm,v);
  VecSetOptionsPrefix(*v,prefix);
  VecLoad(*v,viewer);
  PetscObjectSetName((PetscObject)(*v),name);
  return 0;
}

PetscErrorCode getcheckmeshsizes(MPI_Comm comm, Vec E, Vec x, Vec y,
                                 PetscInt *N, PetscInt *K, PetscInt *bs) {
  PetscInt Ny;
  if (N) {
    VecGetSize(x,N);
    VecGetSize(y,&Ny);
    if (Ny != *N) {  SETERRQ(comm,3,"x,y arrays invalid: must have equal length"); }
  }
  if (K) {
    VecGetSize(E,K);
    if (*K % 15 != 0) {  SETERRQ(comm,3,"element array E invalid (!= 15 K entries)"); }
    *K /= 15;
  }
  if (bs) {
    VecGetBlockSize(E,bs);
    if (*bs != 15) {  SETERRQ(comm,3,"element array E has invalid block size (!= 15)"); }
  }
  return 0;
}

PetscErrorCode readmesh(MPI_Comm comm, PetscViewer viewer, Vec *E, Vec *x, Vec *y) {
  PetscInt bs,N,K;
  PetscPrintf(comm,"  reading mesh Vec E,x,y from file ...\n");
  createloadname(comm, viewer, "E_", "E-element-full-info", E);
  createloadname(comm, viewer, "x_", "x-coordinate", x);
  createloadname(comm, viewer, "y_", "y-coordinate", y);
  getcheckmeshsizes(comm,*E,*x,*y,&N,&K,&bs);
  PetscPrintf(comm,"    block size for E is %d\n",bs);
  PetscPrintf(comm,"    N=%d nodes, K=%d elements\n",N,K);
  return 0;
}
```

Code 8.4: Read the mesh in parallel from the file.

Constructing the FEM linear system

Now that we can get a triangulation into PETSc, we can return to the finite-dimensional weak formulation (8.5). This linear system

$$A\mathbf{u} = \mathbf{b}, \qquad (8.6)$$

has $A \in \mathbb{R}^{N\times N}$ and $\mathbf{u}, \mathbf{b} \in \mathbb{R}^N$, where N is the number of nodes. We will write a code which assembles A and $\mathbf{b}$ and solves for $\mathbf{u}$. PETSc

(8.6) in this Neumann case we will have to inform PETSc about the null space of constant functions.

In general, the next step is to edit $\tilde{A}$ in each Dirichlet row, that is, for each i where $\mathbf{x}_i \in \partial_D\Omega$. For each such row we replace the whole row with the corresponding row of the identity, and also we replace $\tilde{b}_i$ with $b_i = g(\mathbf{x}_i)$. Furthermore, in each column j for which $\mathbf{x}_j \in \partial_D\Omega$, we move all entries $\tilde{a}_{ij}$ where i is *not* a Dirichlet row index over to the right-hand side, multiplied by the negative of the boundary value $g(\mathbf{x}_j)$. We can write

$$\tilde{b}_i \to \tilde{b}_i - g(\mathbf{x}_j)\tilde{a}_{ij} \tag{8.9}$$

for these transformations. After the completion of this "editing" stage we get A and $\mathbf{b}$. Since this part of the matrix assembly is a key stage in our FEM codes, we now give a concrete example.

Example. Figure 8.4 shows a triangulation of the unit square with five nodes. The matrix $\tilde{A}$ has the following nonzero pattern; the zero entries are shown as spaces:

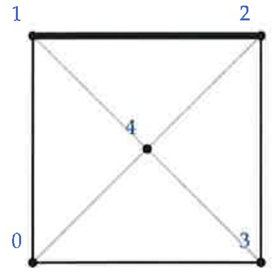


Figure 8.4: A triangulation of a square with five nodes. The top segment is the Dirichlet boundary.

$$\tilde{A} = \begin{bmatrix} \times & \times & & \times & \times \\ \times & \times & \times & & \times \\ & \times & \times & \times & \times \\ \times & & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}.$$

Note $\tilde{a}_{ij} = 0$ only where the integral $\int_\Omega \nabla\phi_j \cdot \nabla\phi_i$ is zero, a rare event in this small (coarse)-mesh case. Now, because the $i = 1, 2$ nodes live in the (closed) Dirichlet boundary $\partial_D\Omega$, we highlight the $i = 1, 2$ rows and $j = 1, 2$ columns which will be edited:

$$\tilde{A} = \begin{bmatrix} \times & \underline{\times} & & \times & \times \\ \underline{\times} & \underline{\times} & \underline{\times} & & \underline{\times} \\ & \underline{\times} & \underline{\times} & \underline{\times} & \underline{\times} \\ \times & & \underline{\times} & \times & \times \\ \times & \underline{\times} & \underline{\times} & \times & \times \end{bmatrix}.$$

Caution: The first row or column of any matrix in this book is numbered "0."

The underlined blue entries are changed to 0 or 1 so that these rows become rows of the identity; the old computed values $\tilde{a}_{ij}$ are tossed out. The underlined red entries, specifically the four entries $\tilde{a}_{01}$, $\tilde{a}_{32}$, $\tilde{a}_{41}$, and $\tilde{a}_{42}$, are moved over to the right side using transformation (8.9); these $\tilde{a}_{ij}$ values get used. The final linear system $A\mathbf{u} = \mathbf{b}$, is

$$\begin{bmatrix} \times & & & \times & \times \\ & 1 & & & \\ & & 1 & & \\ \times & & & \times & \times \\ \times & & & \times & \times \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} \tilde{b}_0 - g(\mathbf{x}_1)\tilde{a}_{01} \\ g(\mathbf{x}_1) \\ g(\mathbf{x}_2) \\ \tilde{b}_3 - g(\mathbf{x}_2)\tilde{a}_{32} \\ \tilde{b}_4 - g(\mathbf{x}_1)\tilde{a}_{41} - g(\mathbf{x}_2)\tilde{a}_{42} \end{bmatrix}$$

Note this
was discussed
in a
previous
chapter

A standard approach to computing the element-wise integrals `a(k,q,r)` is to refer triangle $\triangle_k$ to a reference triangle $\triangle_*$ with vertices $(0,0)$, $(1,0)$, $(0,1)$, as shown in Figure 8.5. The linear map from $\triangle_*$ to $\triangle_k$ with vertices (x_0,y_0), (x_1,y_1), (x_2,y_2), as shown in the Figure, is

$$\begin{aligned} x(\xi,\eta) &= x_0 + (x_1 - x_0)\xi + (x_2 - x_0)\eta, \\ y(\xi,\eta) &= y_0 + (y_1 - y_0)\xi + (y_2 - y_0)\eta. \end{aligned} \tag{8.12}$$

Furthermore, on $\triangle_*$ any linear function is a linear combination of these three local basis functions:

$$\chi_0(\xi,\eta) = 1 - \xi - \eta, \qquad \chi_1(\xi,\eta) = \xi, \qquad \chi_2(\xi,\eta) = \eta. \tag{8.13}$$

On $\triangle_k$, each of the basis functions ϕ_q is the mapped version of the corresponding χ_q:

$$\phi_q(x(\xi,\eta), y(\xi,\eta)) = \chi_q(\xi,\eta). \tag{8.14}$$

From (8.12) and (8.14) one can confirm that $\phi_q(x_r,y_r) = \delta_{qr}$.

Now is a good point at which to observe that the set of ϕ_j, over all nodes $j = 0,1,\dots,N$, is a partition of unity. We can see this by switching to local node coordinate q and using $\chi_0 + \chi_1 + \chi_2 = 1$, which is obvious from (8.13). That is, if $\mathbf{x} \in \triangle_k$ then

$$\sum_j \phi_j(\mathbf{x}) = \sum_{q=0,1,2} \phi_q(\mathbf{x}) = \sum_{q=0,1,2} \chi_q = 1. \tag{8.15}$$

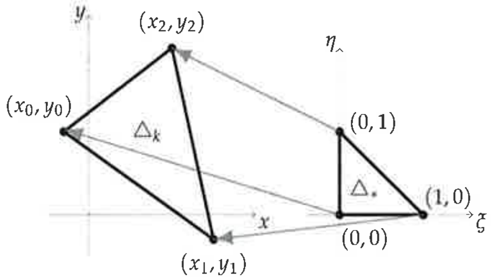


Figure 8.5: Mapping of a triangle $\triangle_k$ from the reference triangle $\triangle_*$.

discussed in previous chapter so rewrite taking that into account.

The Jacobian[8] of map (8.12) is

$$J = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial x}{\partial \eta} \\ \dfrac{\partial y}{\partial \xi} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} = \begin{bmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{bmatrix}. \tag{8.16}$$

Recalling both the change-of-variables formula for integrals[9] and the chain rule, by (8.14) we can write

$$\begin{aligned} \texttt{a(k,q,r)} &= \int_{\triangle_k} \nabla\phi_q \cdot \nabla\phi_r \\ &= \int_{\triangle_k} \frac{\partial \phi_q}{\partial x}\frac{\partial \phi_r}{\partial x} + \frac{\partial \phi_q}{\partial y}\frac{\partial \phi_r}{\partial y}\, dx\, dy \\ &= \int_{\triangle_*} \left(\frac{\partial \chi_q}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \chi_q}{\partial \eta}\frac{\partial \eta}{\partial x}\right)\left(\frac{\partial \chi_r}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \chi_r}{\partial \eta}\frac{\partial \eta}{\partial x}\right) \\ &\qquad + \left(\frac{\partial \chi_q}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial \chi_q}{\partial \eta}\frac{\partial \eta}{\partial y}\right)\left(\frac{\partial \chi_r}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial \chi_r}{\partial \eta}\frac{\partial \eta}{\partial y}\right)\, |\det(J)|\, d\xi\, d\eta. \end{aligned} \tag{8.17}$$

The last expression is the low point of this calculation. From now on the formulas simplify because the integrand is *constant* for the

[8] By definition, the *Jacobian* $J = J(\mathbf{x})$ of a smooth map F at a point $\mathbf{x}$ is its linearization at $\mathbf{x}$. That is, if $\mathbf{y} = F(\mathbf{x})$ and $\mathbf{y} + \Delta\mathbf{y} = F(\mathbf{x} + \Delta\mathbf{x})$ then $J(\mathbf{x})$ satisfies $\Delta\mathbf{y} = J(\mathbf{x})\Delta\mathbf{x} + o(|\Delta\mathbf{x}|)$.

[9] If F is a smooth map from $\mathbf{x} \in U$ to $\mathbf{y} \in F(U)$, J is the Jacobian of F, and g is integrable on $F(U)$, then

$$\int_{F(U)} g(\mathbf{y})\, d\mathbf{y} = \int_U g(F(\mathbf{x}))\, |\det(J)|\, d\mathbf{x}.$$

Bibliography

Note
SIAM wants
City for each
book.

R. Adams and J. Fournier. *Sobolev Spaces*. Academic Press, 2nd edition, 2003.

S. Balay et al. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Elasticity Theory*. Cambridge University Press, 3rd edition, 2007.

W. Briggs, V. E. Henson, and S. McCormick. *A Multigrid Tutorial*. SIAM Press, 2nd edition, 2000.

G. Chartrand, L. Lesniak, and P. Zhang. *Graphs & Digraphs*. CRC Press, 5th edition, 2011.

P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. SIAM Press, 2002. Reprint of the 1978 original.

T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring blems. *SIAM J. Numer. Anal.*, 20(1):187–209, 1983.

?6

J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, 1983.

E. Doedel, H. B. Keller, and J. P. Kernevez. Numerical analysis and control of bifurcation problems (I): bifurcation in finite dimensions. *Int. J. Bifurcation and Chaos*, 1(03):493–520, 1991.

H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*. Oxford University Press, 2005.

L. C. Evans. *Partial Differential Equations*. Graduate Studies in Mathematics. American Mathematical Society, 2nd edition, 2010.

A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.