Solving coupled nonlinear partial differential equations using PETSc

Matt Hennessy

August 25, 2009

Contents

1 Introduction 2

2 A toy problem 2
  2.1 Computational aspects 3

3 Concepts and functions from PETSc 4
  3.1 Distributed arrays 4
  3.2 Vectors 6
    3.2.1 Creating vectors 6
    3.2.2 Assigning/accessing vectors 7
  3.3 Matrices 9
    3.3.1 Assembling matrices 9

4 Preconditioning and solving equations 11
  4.1 Scalable Nonlinear Equation Solvers (SNES) 11
  4.2 Krylov Subspace Methods (KSP) 13
  4.3 Preconditioning (PC) 13
    4.3.1 Direct methods 14

5 Toy problem revisited 14
  5.1 The makefile 14
  5.2 Headers 15
  5.3 Parameters and utilities 17
  5.4 The main program 18
  5.5 Setting the initial guess 22
  5.6 Evaluating the equations 23
  5.7 Evaluating the Jacobian 26

6 Running the program 30
  6.1 Changing linear solvers and the preconditioners 31

7 Summary 32
1 Introduction

Obtaining a numerical solution to a system of nonlinear partial differential equations (PDEs) can be a daunting computational task. Fortunately, several steps in the numerical procedure are well suited to be performed using the multiple cores or processors of a large cluster or workstation. While this has the obvious advantage of reducing the wall time that is needed to obtain a solution, using a multiprocessor computer can also allow one to study problems that are much too large for a single processor computer.

The advantages that are gained in speed and problem size when using parallelized algorithms are balanced by a marked increase in code complexity. Even simple tasks that seem trivial to perform using a single processor can become significantly more complicated when multiple processors are to be used. For the ‘typical’ user, many of these challenges can be made transparent by making use of freely available software libraries that provide high level code which has already been parallelized. This allows the user to write their code as if it is serial code because all of the processes associated with the parallelization occur behind high level function calls. The main challenge when writing code using these libraries is no longer the parallelization, but learning how to use new software.

The goal of this report is to demonstrate the usage of the PETSc libraries (version 2.3.3), which are a set of parallelized libraries written primarily for obtaining the numerical solution of PDEs. In particular, this report will solve an example problem in parallel and describe the functions that are required to do so.

2 A toy problem

For the purposes of this report, we study the system of PDEs in two spatial dimensions

\[
\begin{align*}
0 &= \nabla^2 u + uv, \\
0 &= \nabla^2 v - uv,
\end{align*}
\]

where \(\nabla^2\) is the Laplacian operator in two dimensional Cartesian coordinates and \((x, y) \in [0,1]^2\). Solutions are assumed to be \(x\) periodic and the additional boundary conditions are

\[
\begin{align}
u(x, 0) &= [\sin(2\pi x) + 1]/2, \\
u(x, 1) &= 0,
\end{align}
\]

(1)

for the \(u\) component and

\[
\begin{align}
v(x, 0) &= 0, \\
v(x, 1) &= [\sin(2\pi x) + 1]/2.
\end{align}
\]

(2)

for \(v\).

To solve these equations numerically, a second order centered finite difference scheme is used. The corresponding stencil for this problem, which needs to be known as accurately as possible for using PETSc, is the five point star. The grid is assumed to be uniform and is divided into \(N_x\) and \(N_y\) points in the \(x\) and \(y\) directions, respectively. Furthermore, the \(N_x\) points that along the \(x\) direction are assumed to be distinct and the function values on the \(y\) boundary are solved as part of the problem. To accommodate the fact that the C programming language begins indexing arrays at zero, the grid points are indexed according to \(x_i = ih_x, i = 0, 1, \ldots, N_x - 1\) and \(y_j = jh_y, j = 0, 1, \ldots, N_y - 1\), where \(h_x = 1/N_x\) and \(N_y = 1/(N_y - 1)\). The grid in the \(x\) direction can be visualized by discretizing a circle (see Figure 1) into \(N_x\) points, \(x_i\), where the \(i = 0\) and \(i = N_x - 1\)
points are adjacent to each other. As we shall see, discretizing periodic spatial directions in this manner will be the most convenient for usage with PETSc.

Upon discretization, the partial differential equations become algebraic equations given by

\[
0 = u_{i+1,j} - 2u_{i,j} + u_{i-1,j} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_x^2} + u_{i,j}v_{i,j},
\]

\[
0 = v_{i+1,j} - 2v_{i,j} + v_{i-1,j} + \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{h_y^2} - u_{i,j}v_{i,j},
\]

where \(u_{i,j} \simeq u(x_i, y_j)\) and \(v_{i,j} \simeq v(x_i, y_j)\). These equations obviously hold for \(i = 1, 2, ..., N_x - 2, j = 1, 2, ..., N_y - 2\). However, by discretizing the \(x\) direction as mentioned above, these same equations are used for the \(i = 0\) and \(i = N_x - 1\) points because PETSc has the ability to automatically wrap the stencil around to the other side of the grid. Equations for the unknowns at the \(j = 0\) and \(j = N_y - 1\) points are found by discretizing the boundary conditions presented above in (1) and (2). The resulting system of nonlinear algebraic equations are solved using PETSc’s parallelized Newton’s method.

2.1 Computational aspects

One of the main reasons why numerical PDE solving algorithms are well suited for parallelization is because domain decomposition can be used. That is, the spatial domain of the problem can be divided among the processors so that each processor will be responsible for a different piece of the global grid. A processor’s subset of the global grid is called it’s local grid. Thus, each processor will be responsible for storing, and evaluating if necessary, the fraction of the solution vector, the function vector, and the Jacobian matrix that corresponds to that processor’s local grid. However, the correct evaluation of the equations and the Jacobian is slightly more complicated when multiple processors are used compared to when the same operations are performed sequentially.

To concretely illustrate these difficulties, consider solving the above problem using two processors so that the \(y\) domain is split in half. Thus, processor 0 will be responsible for the domain \((x, y) \in [0, 1] \times [0, 1/2)\) and processor 1 will be responsible for \((x, y) \in [0, 1] \times [1/2, 1]\). In terms of discrete grid points, the \(j\) index of processor 0 will be in the range \(0 \leq j \leq [(N_y - 1)/2] - 1\). For processor 1, the range is \([(N_y - 1)/2] \leq j \leq N_y - 1\). The difficulty arises when processor 0 evaluates its equations at the \(j = [(N_y - 1)/2] - 1\) grid points. By examining the stencil it can be seen that processor 0 requires solution values at the grid points \(j = [(N_y - 1)/2]\), which is outside
of its local range. Therefore, these values will not be in processor 0’s fraction of the solution vector. Similarly, processor 1 requires the solution at \( j = \lceil (N_y - 1)/2 \rceil - 1 \) when it evaluates the equations at \( j = \lfloor (N_y - 1)/2 \rfloor \). The points that a processor needs information at, but which do not belong to its local grid, are called \textit{ghost points}. Thus, when evaluating the equations and the Jacobian, a processor not only needs the solution values at its local grid points, but also the solution values at its ghost points.

Since dynamically allocating memory is an expensive operation, one of the most important steps in writing high performance code is to ensure that all of the matrices are correctly preallocated. In the case here, knowing the structure of the Jacobian (the number of nonzeros in each row and the locations of these nonzeros) is crucial.

### 3 Concepts and functions from PETSc

PETSc has been written specifically for obtaining numerical solutions to PDEs. Because of this, it has not only addressed all of the difficulties discussed above, but made them almost transparent to the user who is writing the code. In this section, several useful PETSc concepts and functions that deal with these issues will be introduced.

#### 3.1 Distributed arrays

When solving problems on logically regular rectangular grids, such as the case here, using PETSc’s distributed arrays (DAs) are extremely useful. Once created, the DA object will partition the grid among the processors and will keep track of which grid points (including ghost points) each processor will need access to. Furthermore, the DA object can be used to create the correct vectors and matrices (preallocation included) for distribution among the processors.

For a two dimensional problem, one would create a two dimensional distributed array with the command

\[
\text{DACreate2d(MPI_Comm comm, DAPeriodicType wrap, DAS stencilType stencil_type, PetscInt M, PetscInt N, PetscInt m, PetscInt n, PetscInt dof, PetscInt s, PetscInt *lx, PetscInt *ly, DA *da)}
\]

where the important arguments are

- \textit{comm} is an MPI communicator which tells the DA which processors to use.
- \textit{wrap} tells the DA if there are any periodic spatial dimensions.
- \textit{stencil_type} relates to the type of finite difference stencil (box or star). The PETSc manual says this is not the case though.
- \( \text{M, N} \) are the number of grid points in the \( x \) and \( y \) direction, respectively.
- \textit{dof} are the number of degrees of freedom (how many unknowns/components) at each grid point.
- \textit{s} is the stencil width. From a practical standpoint, the width of the finite difference stencil. Again, the PETSc manual says this is not the case.
- da is the DA object to be created.

Once the DA object has been created, it is possible for a processor to obtain the information about
the global grid and its local grid by calling the function

\[\text{DAGetLocalInfo}(\text{Da}, \text{DALocalInfo *info})\]

This function takes the DA object, da, as input and returns a DALocalInfo structure that contains
the local and global grid information. The form of DALocalInfo is

\[
\text{typedef struct} \{ \\
\text{PetscInt } \text{dim,dof,sw;} \\
\text{PetscInt } \text{mx,my,mz;} \\
\text{PetscInt } \text{xs,ys,zs;} \\
\text{PetscInt } \text{xm,ym,zm;} \\
\text{PetscInt } \text{gxs,gys,gzs;} \\
\text{PetscInt } \text{gxm,gym,gzm;} \\
\text{DAPeriodicType } \text{pt;} \\
\text{DAStencilType } \text{st;} \\
\text{DA } \text{da;} \\
\} \text{ DALocalInfo;}
\]

To understand the important entries of this structure, consider discretizing a nonperiodic \(x\) domain
into 20 points using two processors and assume the stencil width is one. Then

- \(mx,my,mz\) contain the global number of grid points in the \(x\), \(y\), and \(y\) directions, respectively.
  In terms of the example, \(mx = 20\) for both processors.

- \(xs,ys,zs\) are the indices corresponding to the start of this processor’s grid, excluding ghost
  points. These indices use global indexing. In the example, \(xs = 0\) for processor 0 and \(xs = 10\)
  for processor 1.

- \(xm,ym,zm\) are the number of grid points (excluding ghost points) this processor is responsible
  for. In this case, \(xm = 10\) for both processors

- \(gxs,gys,gzs\) are the indices corresponding to the start of this processor’s grid, if ghost points
  are included. Again, global indexing is used. In terms of the example, \(gxs = 0\) for processor
  0 and \(gxs = 9\) for processor 1.

- \(gxm,gym,gzm\) are the number of grid points (including ghost points) this processor is respon-
  sible for. In this example, \(gxm = 11\) for both processors.

Since each processor in the DA can have different values for these entries, the phrase ‘this processor’
refers to whichever processor is accessing those entries. Using the above functions, it is possible to
loop over a three dimensional grid in parallel using the commands

\[\text{DA da;} \\
\text{PetscInt } i,j,k; \\
\text{DALocalInfo info;}
\]
DACreate3d(...);
DAGetLocalInfo(da, &info);

for (i = info.xs; i < info.xs + info.xm; i++) {
    for (j = info.ys; j < info.ys + info.ym; j++) {
        for (k = info.zs; k < info.zs + info.zm; k++) {

            /* evaluate equations or Jacobian */

        }
    }
}

In this loop, each processor will only go over its own portion of the grid, excluding ghost points.

3.2 Vectors

PETSc has created its own object, called Vec, for storing and manipulating vectors. While there are an abundance of PETSc functions for creating, accessing, and manipulating vectors, this report will only focus on those which are most relevant for the solving the example problem.

There are two main types of vectors that a user will encounter when distributed arrays are used to manage the partitioning of the grid. The first type is the global vector. This vector is used to store all of the unknowns or equations associated with a grid. Therefore, a global vector will have \(dof \times mx \times my \times mz\) elements, where we have made reference to the entries in a DALocalInfo structure (see above). Global vectors are distributed among the processors and ordered in such a way that each processor’s fraction of the global vector corresponds directly with it’s local grid. Therefore, each processor will store \(dof \times xm \times ym \times zm\) (number of components times number of local grid points) elements of the global vector. It can be seen that a processor’s portion of the global vector will not contain any ghost point information. This is where local vectors are helpful.

A local vector is really a vector designed for one purpose; to act as a temporary work vector that will contain ghost point information in addition to the local grid information that is already stored in a processor’s piece of a global vector. Thus, a processor’s local vector will contain \(dof \times gxm \times gym \times gzm\) elements, where \(dof \times xm \times ym \times zm\) of these elements are the same as those in it’s piece of the global vector. The remaining entries will be used to store ghost point information.

When using Newton’s method with DAs, three vectors will be required. Two of these will be the global solution vector and the global function vector that are distributed across all of the processors. Since solution values at the ghost points are needed to evaluate the function and the Jacobian, the third vector will be a local vector that will contain the ghost point information.

3.2.1 Creating vectors

Creating vectors and assigning/accessing their values are, for the most part, two difference processes. To create a vector that is based on a distributed array, one can use PETSc’s DA-vector specific functions. For example, calling the function

\[
\text{DACreateGlobalVector(DA da, Vec *global_vec)}
\]

will create (allocate memory for) a global vector called \text{{global_vec}}. Similarly, calling
DACreateLocalVector(DA da, Vec *local_vec)

will create the local vector, but not actually assign values to it. Placing the ghost point values in
a local vector will be described below.

In most instances, when PETSc vectors are created all of their entries are assigned the value of
zero. It could be possible, however, that some elements will contain garbage. To ensure this is not
the case, the entries of a vector can be set to zero using the command

VecZeroEntries(Vec vec)

This function can only be called after a vector has been created, i.e., has been allocated memory.

3.2.2 Assigning/accessing vectors

Manipulating vectors in PETSc is a relatively straightforward multistep process. The first step
involves getting a pointer to an array that contains the vector’s data. For a vector that has been
distributed over multiple processors, a given processor’s array will only be able to access elements
which belong to that processor. Using this pointer, elements can then be accessed or assigned.
When the user is finished assigning values or accessing the vector, they must ‘restore’ the vector.
That is, they invoke a function that updates the vector based on what was done to the array.

One of PETSc’s most convenient features allows a user to turn a DA vector (global or local) into
a multicomponent structure of multidimensional arrays for problems that have multiple components
and multiple spatial dimensions. For the toy problem above, one could define a structure called a
Field (the structure can actually be called anything) that contains two components, one for u and
one for v,

typedef struct {
    PetscScalar u, v;
} Field;

where u and v will be arrays that are indexed using the global coordinates of the grid. The variable
type PetscScalar can be equivalent to a real double or a complex double, depending on how PETSc
is configured. It is possible to turn the DA vector into a Field, for example, using the function

DAVecGetArray(DA da, Vec vec, void* array)

where vec is the DA vector and array is a multidimensional array of type Field. This function
essentially converts the vector into the more convenient Field form and as such, a Field array will
contain the same elements as the vector that was used to create it. This means that the Field
created by a DA global vector will not contain ghost points, since the original vector did not have
ghost points. Ghost points will be included if vec is a DA local vector. The elements of the vector
can now be assigned or accessed through the Field structure. When the user is done with the Field,
the vector is restored using the function

DAVecRestoreArray(DA da, Vec vec, void* array)
As a concrete example, consider the above toy problem (Section 2). We will set the initial guess of a solution vector to be \( u(x, y) = 0, \ v(x, y) = 1 \). Assuming the Field structure has already been defined, we need to create the DA and a global vector, get the local grid information, convert the vector into a Field structure, set the values, and the restore the vector. This can be done using the following commands

```c
DA da;
DALocalInfo info;
PetscInt i,j;
Vec guess;
Field **g;

DACreate2d(...);
DACreateGlobalVector(da, &guess);
DAGetLocalInfo(da, &info);
DAVecGetArray(da, guess, &g);

for (i = info.xs; i < info.xs + info.xm; i++) {
    for (j = info.ys; j < info.ys + info.ym; j++) {
        g[j][i].u = 0;
        g[j][i].v = 1;
    }
}

DAVecRestoreArray(da, guess, &g);
```

The details of `DACreate2d` will be discussed below. It should be noted that PETSc manages the memory associated with the Field array \( g \), hence the declaration `Field **g` instead of `Field g[ym][xm]`. A similar process can be used to access elements of a DA vector and to evaluate the function (which will be stored as a DA global vector).

As mentioned above, sometimes it is necessary to have access to values at ghost points. Above we showed how to create a local vector that will contain the ghost point information, but we have yet show how to assign values to it. This is done by first calling the function

```c
DAGlobalToLocalBegin(DA da,Vec global,InsertMode mode,Vec local)
```

followed by

```c
DAGlobalToLocalEnd(DA da,Vec global,InsertMode mode,Vec local)
```

The argument `mode` can be `INSERT_VALUES` or `ADD_VALUES`, and it determines whether the entries from the global vector are inserted into the local vector or added to the entries already existing in the local vector. These functions carry out the message passing that is required to place the ghost point values in the local vector of each processor. Therefore, a significant amount of processor communication might be occurring between these two functions, and since this can be a slow process, PETSc allows the user to execute code between these functions if desired. Using the commands `DAVecGetArray` and `DAVecRestoreArray` with an assigned local vector will produce a structure of arrays (a Field structure, for instance) that contain the ghost point values, in addition to the values at the local grid points.
3.3 Matrices

Matrices, like vectors, have their own variable type, called Mat, for passing matrices to various functions. Again, there are many different ways to create and manipulate matrices, but this report will only focus on a small subset of those.

Creating a matrix that is based on a DA is incredibly easy. After the DA has been created, the user can call the function

\[
\text{DAGetMatrix(DA da, MatType mtype, Mat *J)}
\]

where \textit{mtype} is the type of matrix to be created (different types for sequential and parallel programs, different sparse storage formats, etc.) and \textit{J} is the matrix to be created. Using the DA object, which contains information about the stencil and the number of components of the problem, PETSc can calculate how many nonzero elements to allocate and it can determine where in the matrix these elements are located. Therefore, if the stencil is accurately known, the Jacobian can be correctly preallocated and built extremely fast.

3.3.1 Assembling matrices

The global indexing and the arrangement of the matrix is automatically handled by PETSc. This is done to ensure that the Jacobian will match the arrangement of DA global vectors. Thus, PETSc provides routines for building the Jacobian that do not require the user to know how the matrix is globally indexed. By providing PETSc with the grid indices and the component indices of the rows and columns to be assigned, it can determine where in the matrix to place the corresponding values. Multiple rows and columns can be inserted at once using the command

\[
\text{MatSetValuesStencil(Mat mat, PetscInt m, const MatStencil row[], PetscInt n, const MatStencil col[], const PetscScalar val[], InsertMode mode)}
\]

where

- \textit{mat} is the matrix that is being built.
- \textit{m}, \textit{n} are the number of rows and columns that are being inserted during this call.
- \textit{row}, \textit{col} are MatStencil structures that contain the grid indices and component indices of the rows and columns being inserted. For example, \textit{row.i} = \textit{i}+1, \textit{row.j} = \textit{j}, \textit{row.k} = \textit{k}-1, \textit{row.c} = 1, means the row is associated with the \textit{i}+1,\textit{j},\textit{k}-1-th grid point and the second component of the problem (indexing starts at zero).
- \textit{val} is an array of values to insert.
- \textit{mode} tells the program to either insert values or add values to the old ones.

A common way to build a matrix is row by row. In this procedure, the nonzero elements in a row, which correspond to column entries, are inserted at once. Thus, the \textit{col} MatStructure will be an array with as many entries as there are nonzeros in the row being inserted. To declare \textit{col}, one needs to know the number of nonzero elements in the row with the most nonzero entries. If this number cannot be determined, overestimation must be used.

Once the matrix values have been assigned, the matrix actually needs to be constructed, much like how the vectors needed to be restored after the arrays were used. Matrices are constructed using the commands
where type tells the program whether the matrix build is final or not. Because a significant amount of communication can also occur during the building of the matrix, the user can execute commands between these two functions. As a very simple and quick example of how these functions work, consider the matrix associated with

\[ 0 = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - u_i \]

where \( i = 0, 1, \ldots, N - 1, h = 1/(N - 1), u_0 = 0, u_{N-1} = 1 \). The code below will create this matrix row by row, where it has been noted that each row will have at most three entries.

```c
DA da;
DALocalInfo info;
Mat M;
PetscInt i, N = 25, num_cols;
MatStructure row, col[3];
PetscScalar val[3], h = 1.00 / (N - 1);

DACreate1d(...);
DAGetLocalInfo(da, &info);
DAGetMatrix(da, MATMPIAIJ, &M);

for (i = info.xs; i < info.xs + info.xm; i++) {
    row.i = i;
    row.c = 0;

    if (i == 0 || i == N - 1) {
        col[0].i = i;
        col[0].c = 0;
        val[0] = 1.00;
        num_cols = 1;
    } else {
        col[0].i = i-1;
        col[0].c = 0;
        val[0] = 1 / h / h;

        col[1].i = i;
        col[1].c = 0;
        val[1] = -2 / h / h - 1;

        col[2].i = i+1;
    }
}
col[2].c = 0;
val[2] = 1 / h / h;

num_cols = 3;
}
MatSetValuesStencil(M, 1, &row, num_cols, col, val, INSERT_VALUES);
}

MatAssemblyBegin(M,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(M,MAT_FINAL_ASSEMBLY);

While this example is unrealistically simple, it illustrates how the matrix is built using the global
indexing of the grid and not the global indexing of the matrix.

4 Preconditioning and solving equations

The above concepts and functions from PETSc illustrate how to overcome several difficulties that
are encountered when evaluating the nonlinear equations and their Jacobian matrix in parallel.
Before using these tools on the toy problem presented above in Section 2, it will be useful to
discuss how to solve linear and nonlinear equations with PETSc.

We have seen how PETSc has created objects for managing distributed arrays, vectors, and
matrices. The way PETSc manages preconditioning and solutions of equations is very similar.
Options associated with preconditioning, linear equation solvers, and nonlinear equation solvers
are managed by the objects PC, KSP, and SNES, respectively. There is a hierarchy between these
objects, in fact. Solving nonlinear equations via Newton’s method requires the solution of linear
systems, and the linear systems are often preconditioned. Thus, each SNES object is linked to a
KSP object, and each KSP object is linked to a PC object.

4.1 Scalable Nonlinear Equation Solvers (SNES)

PETSc provides a high level interface to several advanced Newton’s method algorithms through
the SNES object. Setting up an SNES object is a straightforward process. The user must first
create the SNES object using the command

SNESCreate(MPI_Comm comm,SNES *snes)

where snes is the new SNES object. The next step involves associating the SNES with the nonlinear
equations. This is done with the command

SNESSetFunction(SNES snes,Vec soln,
PetscErrorCode (*func)(SNES,Vec,Vec,void*),void *ctx)

where soln is the solution vector (usually a DA global vector), func is the C function that evaluates
the nonlinear equations, and ctx is a user defined data structure that can be passed to the equation
evaluation function. This structure could contain parameter values and the DA object. The C function that evaluates the equations must have the form

\[ \text{func (SNES snes, Vec soln, Vec fun, void *ctx)} \]

where \text{func} is the name of the function, \text{soln} is the solution vector, \text{fun} is the function vector, and \text{ctx} is a data structure.

There are several options for evaluating the Jacobian. To manually set the Jacobian and its preconditioning matrix, one can call the function

\[ \text{SNESjacobian(SNES snes, Mat J, Mat B,}\]  
\[ \text{PetscErrorCode (*func)(SNES,Vec, Mat*, Mat*, MatStructure*, void*), void *ctx)} \]

where \text{J} is the Jacobian matrix, \text{B} is the preconditioner matrix, \text{func} is the function which evaluates these matrices, and \text{ctx} is a structure of data that can be passed to this function. The function which evaluates the matrices must be of the form

\[ \text{func (SNES snes, Vec soln, Mat *J, Mat *B, int *flag, void *ctx)} \]

The \text{Mat} objects \text{J} and \text{B} are the same as those described above. Also passed to this function is the solution vector \text{soln}, the \text{SNES} object \text{snes}, a flag that is used to tell the \text{SNES} object if the nonzero structure of the matrices change between successive Newton iterations, and a user defined structure \text{ctx}.

Alternatively, PETSc has an extensive suite of functions for creating the Jacobian via finite differencing. For example, when calling \text{SNESJacobian}, the user can pass \text{PETSC_NULL} as the function for evaluation the Jacobian and then use command line arguments that tell PETSc to construct a finite difference Jacobian. These flags will be shown below.

There are, in fact, a number of options that can be passed to the \text{SNES} object from the command line. These include setting tolerances, displaying convergence of the solution, etc. These command line options are set by calling the function

\[ \text{SNESfromOptions(SNES snes)} \]

Calling this function is actually mandatory when one is using PETSc to solve nonlinear equations. Furthermore, the options that are passed through the command line overwrite any options which may be explicitly set in the source code.

When all the \text{SNES} options have been set (and the \text{KSP/PC} options, see below) the system of equations can be solved using the command

\[ \text{SNESsolve(SNES snes, Vec b, Vec soln)} \]

where \text{b} is a vector for the constant part of the equations and \text{soln} is the solution vector. If there is no constant part, one can pass \text{PETSC_NULL} for \text{b}. The initial values of \text{soln} that have been assigned prior to calling \text{SNESsolve} will form the initial guess of the solution.
4.2 Krylov Subspace Methods (KSP)

The solution of the linear systems that occur when applying Newton’s method are managed by the KSP objects in PETSc. These objects are also used when one wishes to solve a single linear system as well. However, since solving single linear systems with PETSc is not the topic of this report, this topic won’t be discussed further. However, many of the concepts and functions that have been introduced here for solving nonlinear equations will be useful for solving linear systems. As mentioned above, the SNES object is linked to a KSP object. Sometimes it is necessary to access this KSP object so that the user can set certain options. An SNES’s KSP object can be accessed using the command

\texttt{SNESGetKSP(SNES snes, KSP *ksp)}

The type of iterative method that is to be used can be set with the function

\texttt{KSPSetType(KSP ksp, KSPType type)}

where \texttt{type} is essentially a string that tells PETSc which type of method to use. There is a large number of methods that can be used, for example, GMRES(\(k\)), FGMRES(\(k\)), and BiConjugate Gradient. Depending on which method is used to solve the linear systems, additional options can be set. For example, to set the GMRES(\(k\)) restart parameter, one calls

\texttt{KSPGMRESSetRestart(KSP ksp, PetscInt restart)}

Calling the function

\texttt{KSPSetFromOptions(KSP ksp)}

allows the user to set the KSP options from the command line.

4.3 Preconditioning (PC)

The PC object controls the preconditioning options in a manner that is nearly identical to how the KSP controls the linear equation solver options. Each KSP object has a PC object, which can be accessed via

\texttt{KSPGetPC(KSP ksp, PC *pc)}

Setting the type of preconditioning done through the function

\texttt{PCSetType(PC pc, PCType type)}

Several preconditioning types exist, for example, Jacobi, block Jacobi, and additive Schwarz. Each type of preconditioning has options specific to it that can be set by finding the appropriate function. The type of preconditioning and its options can can set through the command line by using the function

\texttt{PCSetFromOptions(PC pc)}

Having the ability to modify the preconditioning through the command line is a very handy feature when one attempting to find the optimal preconditioner.
4.3.1 Direct methods

While PETSc focuses on using iterative methods to solve linear systems in parallel, it does have the ability to use direct methods. By default, these methods can only be performed sequentially. External packages must be installed and configured if parallel direct methods are to be used. The flags to use direct methods are passed to the function \texttt{PCSetType}. For example, LU decomposition can be used with the follow commands

\begin{verbatim}
KSP ksp;
PC pc;
KSPCreate(...);
KSPGetPC(ksp, &pc);
PCSetType(pc, PCLU);
\end{verbatim}

Further details associated with direct methods can be viewed in the PETSc documentation.

5 Toy problem revisited

With a knowledge of PETSc’s inner workings, it is now possible to write a C program that will solve the toy problem that has been presented above in Section 2. While this problem is unrealistically simple, we will write the code as if it is part of a larger problem that is spread over multiple C source files. In particular, our program will contain eight files:

1. makefile: This will be used to manage compilation of source files
2. headers.h: Contain necessary headers and declares the auxiliary functions that appear in parameters.c and utilities.c
3. parameters.c: Contains parameter values.
4. utilities.c: Contains commonly used functions that carry out many different tasks
5. example.c: This is the main program which solves the toy problem
6. guess.c: Forms an initial guess of the solution
7. equations.c: Evaluates the nonlinear equations
8. jacobian.c: Evaluates the Jacobian

Each file will be described in the proceeding subsections.

5.1 The makefile

Large programs are generally divided among several source files and during the building of a program, it may not be necessary to recompile all of these files. The make program, and its associated makefile, manage the compilation and recompilation of source files.

The first line in the makefile will list the programs that can be compiled by make. In this case, there is only one program, which is called example, so the first line of the makefile reads
all: example

If there were additional programs, their name would be placed after example. Since the PETSc libraries are required during compilation, the user must specify the directory of these libraries. This is achieved by entering the line

```
include ${PETSC_DIR}/bmake/common/base
```

where `PETSC_DIR` is the PETSc directory environment variable which is determined and set when PETSc is configured. The next steps are to explicitly state the source files that are to be compiled and to list their object files. Thus, the next two lines in `makefile` will read

```
SRC=example.c utilities.c parameters.c guess.c equations.c jacobian.c
OBJ=example.o utilities.o parameters.o guess.o equations.o jacobian.o
```

The make program will scan the source files for changes since the last compile and recompile if necessary. The final steps are compiling the source files and linking the object files. This is done with the commands

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

The last flag in the second line links the object files with all of the PETSc libraries. If only a subset of PETSc's libraries are to be used, then `PETSC_LIB` can be changed accordingly. When the `makefile` is complete, typing `make` at the command line will attempt to compile the program.

### 5.2 Headers

Every C program begins with including the required header files, and programs written with PETSc are no different. However, most of the conventional C headers need not be explicitly included because they are already included in one of PETSc's header files. The PETSc header files that are to be included depend on which parts of PETSc are being used. In this example, we use the SNES functions and the DA functions, so these headers will need to be included. Furthermore, we wish to time our program, so the PETSc timing headers will also be included. Thus, the beginning of the `header.h` file will contain

```c
#include <stdlib.h>
#include "petscsnes.h"
#include "petscda.h"
#include "petsctime.h"
```

Note that the SNES routines also use the KSP (linear equation solving) routines. However, the headers for these functions are automatically included in `petscsnes.h`.

It is convenient to define in the headers file, the data structures that will be used in various functions. For example, converting the DA global vector into a Field will require the definition of the Field structure. In fact, we will also define two more structures that will be convenient. In total, the following three structures will be defined

- **Field**: Contains the components of the problem. In this case, $u$ and $v$.  

15
- **Params**: Contains the parameter values.

- **Data**: Will be the structure passed to the equation and Jacobian evaluation functions. This structure will contain the parameter structure and the DA object (so these functions will have access to grid information without creating a new DA).

The Field structure will be defined using the code

```c
typedef struct {
    PetscScalar u, v;
} Field;
```

which is identical to the Field definition above. The parameters in the problem are the number of grid points in the \(x\) and \(y\) direction, \(N_x\) and \(N_y\), and the distance between two \(x\) grid points, \(h_x\), and the distance between two \(y\) grid points, \(h_y\). Thus, the Params structure is defined as

```c
typedef struct {
    PetscInt Nx, Ny;
    PetscScalar hx, hy;
} Params;
```

Values for these parameters will be assigned in the file `parameters.c`. The last structure will be the structure that is passed to the functions which evaluate the equations and the Jacobian (the `ctx` in `SNESSetFunction` and `SNESSetJacobian`). This structure will be called Data and will contain a Params structure and a DA object,

```c
typedef struct {
    Params P;
    DA da;
} Data;
```

The DA object will be the one created using `DACreate2d`.

As mentioned above, the parameter values will be set by a function located in the file `parameters.c`. This function will be declared in the headers file to allow easy access to it by other external functions that may be in other source files. Furthermore, all of the handy utility functions that appear in `utilities.c` should be declared in the headers file for the same reason. Thus, the last lines of the header file will be

```c
/* functions in parameters.c */
Params parameters();

/* functions in utilities.c */
void printvec(Vec X, const char file[]);
```

where the details of these functions will be described below.
5.3 Parameters and utilities

For easy access to parameters, it is convenient to store them in a separate file. With the Params structure already being defined, this file only has to access this structure and assign values to it. The assigned structure can then be passed to the other functions as the user sees fit. The first thing in the parameters.c file, and in all the C files, should be a line that includes the headers file headers.h. We will assume that the parameters file and the headers file are located in the same directory. Once this line has been added, the function that assigns the parameter values can be written. One possible way to write the parameters file is

```c
#include "headers.h"

Params parameters() {
    Params P;
    P.Nx = 100;
    P.Ny = 102;
    P.hx = 1.00 / (P.Nx);
    P.hy = 1.00 / (P.Ny - 1);
    return P;
}
```

This allows the user to declare and assign a Param structure, called P, using the command

```c
Params P = parameters();
```

The entries in this structure can be easily accessed by using simple commands. For example, to access \( N_x \), one would type \( P.Nx \).

The utilities file is meant to contain various functions for doing useful tasks, such as printing vectors or matrices, or computing inner products of functions. In this problem, the only function in the utilities file will print a parallel vector to a user specified file. Thus, the utilities.c file can be written as

```c
#include "headers.h"

void printvec(Vec X, const char file[]) {
    PetscViewer me;
    PetscViewerCreate(PETSC_COMM_WORLD, &me);
    PetscViewerSetType(me, PETSC_VIEWER_ASCII);
    PetscViewerFileSetName(me, file);
    PetscViewerSetFormat(me, PETSC_VIEWER_ASCII_SYMMODU);
    VecView(X, me);
}
```
Notice the headers file has been included. This function takes a vector and a string as input. The string will be used to name the file. The functions PetscViewerSetType and PetscViewerSetFormat allow the user to control how the vector is printed. In this case, the vector will be printed as a list of numbers. Other options can print the vector with indices for its elements. When printing a sparse matrix to a file, these options can output the matrix as a MATLAB .m file with the sparsity preserved. The command printvec(soln_vec, ‘solution.dat’); will print a vector called soln_vec to the file solution.dat.

### 5.4 The main program

The main program, which is written in the file example.c, contains, for the most part, all of the functions described above for setting up an SNES and then solving associated system of nonlinear equations. This file will also contain call some additional functions which are used in start the MPI environment and to create the DA.

The file example.c begins by calling the headers file and then declaring any functions which are used in this C file but are contained in other source files. Recall that some functions have been already declared in the headers file and will not have to be declared again. For this program, the functions that make the initial guess of the solution, and that build the equations and the Jacobian have to be declared. The functions for building the equations and the Jacobian must have the form above (see SNESetFunction and SNESetJacobian). Thus, example.c begins with

```c
#include "headers.h"
```

The type PetscErrorCode is for debugging purposes and won’t be discussed here. More details can be found in the PETSc manual. The next lines of the contain the skeleton of the function,

```c
int main(int argc,char **argv)
{
}
```

The lines that follow in most C functions are the declarations of the objects and the variables that are to be used in that function. However, instead of explicitly writing the variable declarations all at once, they will be declared as the functions which use them are called. Although this will make the code appear as if there are declarations scattered throughout it, all declarations will be made before the main body of the function.

The first function that will be called in example.c is
PetscInitialize(&argc, &argv, PETSC_NULL, PETSC_NULL);

which initiates the MPI and PETSc environments. This is exactly analogous to first calling
MPI_Init in MPI programs. In fact, MPI_Init is called in PetscInitialize. In PETSc pro-
goams, the user can specify a help message that can be printed when the program is run. The last
PETSC_NULL indicates that no help message has been specified.

With the PETSc environment having been started, it is now possible to begin calling PETSc
functions. The first thing that needs to be done is to get a Params structure with parameter values.
With these parameter values, the grid can be created using DACreate2d and the Data structure
can be assigned.

DA da;
Params P;
Data data;

P = parameters();
DACreate2d(PETSC_COMM_WORLD, DA_XPERIODIC, DA_STENCIL_STAR, P.Nx, P.Ny,
PETSC_DECIDE, PETSC_DECIDE, 2, 1, PETSC_NULL, PETSC_NULL, &da);

data.P = P;
data.da = da;

In the DACreate2d function, we have passed it the MPI communicator PETSC_COMM_WORLD. This
communicator contains all of the processors that are specified at run time. The $x$ direction in the
toy problem is periodic. PETSc can automatically handle periodic boundary conditions by telling
the DA which directions are periodic. Thus, we pass DA_XPERIODIC as the second argument. The
problem was discretized using the five point star stencil (DA_STENCIL_STAR), which has stencil width
one (1). The number of grid points, which are defined in the Params structure $P$, are specified by
passing $P.Nx$ and $P.Ny$ to DACreate2d. We let PETSc decide (PETSC_DECIDE) how to partition $x$
grid and the $y$ grid among the processors. Each grid point has two unknowns, $u$ and $v$, so the
degrees of freedom (the number of components of the problem) are 2. The third last and second
last arguments are PETSC_NULL because PETSc is managing the partitioning of the grid among the
processors. Finally, the created DA object will be $da$. The last two lines assign the Data structure
that will be passed to the equation/Jacobian evaluation functions.

With the DA object created, it is possible to preallocate space for the solution vector, soln_vec,
the function vector, fun_vec, and the Jacobian matrix, $J$. Recall these vectors are DA global vectors.

Vec fun_vec, soln_vec;
Mat J;

DACreateGlobalVector(da, &fun_vec);
DACreateGlobalVector(da, &soln_vec);
DAGetMatrix(da, MATMPIAIJ, &J);

The Jacobian will have matrix type MATMPIAIJ, which is PETSc's default way of storing sparse
parallel matrices. As mentioned above, the function DAGetMatrix will determine the location of
the nonzero values in the Jacobian. This allows the Jacobian to be evaluated extremely quickly.
The next lines of code will first create the SNES object for solving the nonlinear equations and then tell this object which functions to use for equation/Jacobian evaluation and where to store these objects.

```
SNES snes;
SNESCreate(PETSC_COMM_WORLD, &snes);
SNESSetFunction(snes, fun_vec, equation, &data);
SNESSetJacobian(snes, J, J, jacobian, &data);
```

Thus, all processors in `PETSC_COMM_WORLD` will participate in solving the equations. The function vector will be stored in `fun_vec` and the function `equation` will be used to evaluate them. The structure `data` will be passed to this function. The Jacobian matrix will be stored in the matrix `J` (the first J), and this matrix will also be used as the preconditioning matrix (the second J). The function `jacobian` will be used to construct the Jacobian and the preconditioning matrix, and the structure `data` will also be passed to this function. Since only one structure can be passed to the functions which evaluate the equations and the Jacobian, it is important to ensure that this structure contains all of the information that is required for equation/Jacobian evaluation.

The default options for the SNES and its associated linear equation solver, which are FGMRES(30) with Jacobi preconditioning, might not be adequate for complex problems. The next lines will access the KSP and PC objects associated with the SNES and change them to use GMRES(100) with block Jacobi preconditioning. Furthermore, the functions will be called that allow the user to overwrite these options from the command line.

```
KSP ksp;
PC pc;
SNESGetKSP(snes, &ksp);
KSPSetType(ksp, KSPGMRES);
KSPGMRESSetRestart(ksp, 100);
KSPGetPC(ksp, &pc);
PCSetType(pc, PCBJACOBI);
PCSetFromOptions(pc);
KSPSetFromOptions(ksp);
SNESetFromOptions(snes);
```

If desired, the solution tolerances can be set here as well.

Before actually solving the system, the initial guess is set using the `guess` function and some information about the grid is printed to the screen.

```
guess(soln_vec, data);
PetscPrintf(PETSC_COMM_WORLD, "grid size is %d x %d \n", P.Nx, P.Ny);
```

PETSc has its own functions for printing information when multiple processors are used. In this case, `PetscPrintf` will get the root process to print the grid size to stdout. If `printf` was used instead, each process would print the information unless an `if` statement was used.
With all of the options and the initial guess set, it is now possible to solve the system. We will also time how long this process takes.

```c
PetscLogDouble v1, v2, elapsed_time;
PetscGetTime(&v1);
SNESolve(snes, PETSC_NULL, soln_vec);
PetscGetTime(&v2);
elapsed_time = v2 - v1;
```

It might be of interest to the user to determine whether the SNES was able to converge to a solution or not. In the case where the SNES is unable to find a solution, it will be useful to know what went wrong. We will also get the total number of GMRES iterations and the number Newton iterations that are needed to find a solution. From this information, it will be possible to determine the average number of GMRES iterations that are required to solve each linear system in Newton iterations. This information will be printed to the screen, along with the time needed to solve the nonlinear equations.

```c
SNESConvergedReason snes_reas;
PetscInt lin_its, nonlin_its;
SNESGetConvergedReason(snes, &snes_reas);
if (snes_reas < 0) {
    PetscPrintf(PETSC_COMM_WORLD, "SNES Failed! Reason %d\n", snes_reas);
}
SNESGetLinearSolveIterations(snes, &lin_its);
SNESGetIterationNumber(snes, &nonlin_its);
PetscPrintf(PETSC_COMM_WORLD, "average number of GMRES iterations =
(total GMRES its) / (Newton its) = %d / %d = %.4e\n",
lin_its, nonlin_its, ((double) lin_its) / nonlin_its);
PetscPrintf(PETSC_COMM_WORLD, "nonlinear eqns solved in %.2f s \n", elapsed_time);
```

The variable `snes_reas` will be an integer value that corresponds to a reason why the SNES was able or unable to converge to a solution. These reasons can then be looked up in the PETSc documentation.

The penultimate lines in the code will be related to file output. In particular the solution vector and the grid information will be printed to a file.

```c
FILE *fid;
printvec(soln_vec, "soln.dat");
PetscFOpen(PETSC_COMM_WORLD, "soln_info.dat", "w", &fid);
PetscFFprintf(PETSC_COMM_WORLD, fid, "%d\n%d", P.Nx, P.Ny);
PetscFClose(PETSC_COMM_WORLD, fid);
```
The function `printvec` is defined in the `utilities.c` file. The three PETSc commands above tell
the root process to open the file `soln_info.dat`, print the number of grid points, and then close
the file.

Finally, the program will free the memory associated with the vectors and the Jacobian since
they are no longer needed. This is done by using PETSc’s destroy commands.

```c
VecDestroy(soln_vec);
VecDestroy(fun_vec);
MatDestroy(J);
```

The last lines in the program will be a call to

```c
PetscFinalize();
```

which is analogous to calling `MPI_Finalize` in MPI programs and will return a value of zero to
indicate successful completion

```c
return 0;
```

5.5 Setting the initial guess

The convergence of Newton’s method can, at times, be strongly dependent on the initial guess. The
more accurate an initial guess is to the true solution, the more likely it is that Newton’s method
will converge. Instead of using the trivial solution as the initial guess for the toy problem, we will
use the functions

\[
\begin{align*}
  u(x, y) &= \frac{1}{2} [\sin(2\pi x) + 1](1 - y)^2, \\
  v(x, y) &= \frac{1}{2} [\sin(2\pi x) + 1]y^2
\end{align*}
\]

as the initial guess since they satisfy all of the boundary conditions. We will see that the DA
commands introduced in Section 3.2.2 will make setting up this initial guess a very straightforward
task.

The function begins with including the headers file and its definition. This function will require
only two inputs; the vector to hold the guess and a Data structure.

```c
#include "headers.h"

PetscErrorCode guess(Vec guess, Data data) {
}
```

In real code, the first lines would be declarations, but as mentioned above, declarations of variables
will be made when a function uses them. For auxiliary functions that do not call `PetscInitialize`,
the first line after the declarations should call

```c
PetscFunctionBegin;
```

This function is used for error handing (see PETSc documentation for more details). Once this
function has been called, the PETSc functions described above can be used. The first step in this
function will be to access the local grid information using `DAGetLocalInfo` using the DA object
that is contained in the Data structure. Then, the `guess` vector, which is assumed to be a global
vector, is converted into a Field, where it should be noted that no ghost point information will be
required for setting the initial guess.
DA da = data.da;
DALocalInfo info;
Field **g;

DAGetLocalInfo(da, &info);

DAVecGetArray(da, guess, &g);

We can now set the elements of the guess vector by setting elements in the Field array. To do this,
we loop over the grid and set the components of g equal to the functions used for the initial guess.

PetscInt i,j;
PetcScalar x,y;

for (i = info.xs; i < info.xs + info.xm; i++) {
    for (j = info.ys; j < info.ys + info.ym; j++) {
        x = i * hx;
        y = j * hy;

        g[j][i].u = 0.5 * (sin(2*PETSC_PI*x) + 1) * pow(1 - y, 2);
        g[j][i].v = 0.5 * (sin(2*PETSC_PI*x) + 1) * pow(y, 2);
    }
}

With new values in the Field array, we restore the vector by calling

DAVecRestoreArray(da, guess, &g);

To end the function, we use PETSc's return function instead of the C's return function. Again,
this is used for error handling.

PetscFunctionReturn(0);

5.6 Evaluating the equations

The function which evaluates the equations is very similar to the function that creates the initial
guess. It is only slightly more complicated because of the need to create a DA local vector which
has information about the solution at ghost points. We start by including the headers file and
defining the function according to SNESSetFunction

#include "headers.h"

PetscErrorCode equation(SNES snes, Vec soln_vec, Vec fun_vec, void* ptr) {

}
Since evaluating the equations involves evaluating the finite difference approximation of the Laplacian operator, this operator can be written as a macro to simplify the code:

```c
#define lap(m) ((X[j][i+1].m - 2*X[j][i].m + X[j][i-1].m) / hx / hx + (X[j+1][i].m - 2*X[j][i].m + X[j-1][i].m) / hy / hy)
```

Thus, when the code is compiled, all instances of `lap(u)` and `lap(v)` will be replaced with this macro definition. In this macro, it has been assumed that the solution is represented as a Field called `X`. The Data structure will be contained in the variable `ptr` that is passed to this function. The first thing that needs to be done is to recover the Data structure from `ptr`, which can then be used to get the Params structure and the DA object:

```c
Data *data = (Data*)ptr;
Params P = data->P;
DA da = data->da;
```

With these structures being assigned in their declaration, it is possible to begin the function and get the local grid information, which will be required for function evaluation:

```c
DALocalInfo info;
```

```c
PetscFunctionBegin;
DAGetLocalInfo(da, &info);
```

To evaluate the function, the global solution vector will need to be converted into a local vector containing ghost point information. This is done by creating a local vector and using the LocalToGlobal commands:

```c
Vec sv_local;
DACreateLocalVector(da, &sv_local);
DAGlobalToLocalBegin(da, soln_vec, INSERT_VALUES, sv_local);
DAGlobalToLocalEnd(da, soln_vec, INSERT_VALUES, sv_local);
```

The vector `sv_local` is the local solution vector. The command `INSERT_VALUES` has been used to indicate that we want the values from the global solution vector, `soln_vec` to be inserted into the local vector. Before the function can be evaluated, the local solution vector and the global function vector need to be converted into Field arrays:

```c
Field **X;
Field **F;
DAVecGetArray(da, sv_local, &X);
DAVecGetArray(da, fun_vec, &F);
```

Recall that because `sv_local` is a local vector, the Field array `X` will have access to ghost points.

The evaluation of the function is a straightforward process. The grid is looped over and if statements are used to separate the equations for the $y$ boundary conditions from the equations in the interior. It is important to note that no special equations are needed to evaluate the function at the ends of the $x$ domain because PETSc knows this domain is periodic (from the `DA_XPERIODIC` usage in `DACreate2d`), so the stencil is automatically wrapped to the other edge of the domain.
PetscInt i, j, Ny = P.Ny;
PetscScalar hx = P.hx, hy = P.hy, x, y;

for (i = info.xs; i < info.xs + info.xm; i++) {
    for (j = info.ys; j < info.ys + info.ym; j++) {
        x = i * hx;
        y = j * hy;

        if (j == 0) {
            F[j][i].u = X[j][i].u - 0.5 * (sin(2*PETSC_PI*x) + 1);
            F[j][i].v = X[j][i].v;
        }

        else if (j == Ny - 1) {
            F[j][i].u = X[j][i].u;
            F[j][i].v = X[j][i].v - 0.5 * (sin(2*PETSC_PI*x) + 1);
        }

        else {
            F[j][i].u = lap(u) + X[j][i].u * X[j][i].v;
            F[j][i].v = lap(v) - X[j][i].u * X[j][i].v;
        }
    }
}

The last calls in this function restore the function vector and the local vector, destroy the local vector to free memory since this vector is no longer required, and return a value of zero.

DAVecRestoreArray(da, fun_vec, &F);
DAVecRestoreArray(da, sv_local, &X);

VecDestroy(sv_local);

PetscFunctionReturn(0);

If the local solution vector, sv_local, is not destroyed by the end of this function, the memory that is allocated to it during each call to DACreateLocalVector will not be freed until the program terminates. This can have severely limit performance if memory resources are scarce.
5.7 Evaluating the Jacobian

The function which evaluates the Jacobian (and the preconditioning matrix) begins very similar to the function which evaluates the equations. The structures are recovered from the `ptr` structure which is passed to the function. Furthermore, the function gets the grid information and the solution vector is converted into a Field. In this particular example, the ghost points are not required and a local solution vector does not have to be created. In practise, this will not be the case.

```c
#include "headers.h"

PetscErrorCode jacobian(SNES snes, Vec soln_vec, Mat *jac, Mat *B, MatStructure *flag, void* ptr)
{
    Data *data = (Data*)ptr;
    Params P = data->P;
    DA da = data->da;
    DALocalInfo info;

    Field **X;

    /* begin function */
    PetscFunctionBegin;

    DAGetLocalInfo(da, &info);
    DAVecGetArray(da, soln_vec, &X);

    }
```

The next steps involve the actual construction of the Jacobian and the preconditioner. Since, in this example, the Jacobian matrix is the preconditioning matrix, there is no need to build a separate matrix. This was indicated in the call to `SNESSetJacobian`. We will evaluate the Jacobian one row at a time. We declare two MatStencils, one for the row and one for the column. These allow us to use global grid indexing instead of global matrix indexing when building the Jacobian. Since any given row will have at most six columns, the MatStencil variable for the columns will actually be an array with six elements. Variables for storing the number of entries in a row and the entries themselves will also be declared.

```c
MatStencil row, col[6];
PetscInt ctr;
PetscScalar val[6];
```

As mentioned above, overestimation should be used if it is not possible to determine the maximum number of entries in a row. Underestimation will likely lead to errors and an incomplete Jacobian.

The Jacobian is built by looping over the grid points, which, when using PETSc, is equivalent to looping over all of the rows in a matrix. In the loop, the `row` and `col` structures must be assigned. Recall that each structure has four components; three of these indicate the i, j, and k index of the three dimensional grid and the fourth indicates the corresponding component of that row. In this
case, the k component of the structure can be ignored because this is a two dimensional problem. Setting the i and j components of the row structure (row.i and row.j) are trivial, they are just the index of the x and y grid, respectively. The component index of the row structure (row.c) depends on the component index of the equation being differentiated. For example, rows relating to the u equation have row.c = 0 since u has been assumed to be the first component of the system. The components of the col structures depend on the grid indices and the component indices of the nonzero derivatives that appear in each row. After the row structure, the col structure, and values for the entries have been assigned, that row is built using the MatSetValuesStencil function.

PetscInt i, j;

for (i = info.xs; i < info.xs + info.xm; i++) {
    for (j = info.ys; j < info.ys + info.ym; j++) {
        row.i = i;
        row.j = j;

        if (j == 0 || j == Ny - 1) {
            col[0].i = i;
            col[0].j = j;
            ctr = 1;

            /* the u part */
            row.c = 0;
            col[0].c = 0;
            val[0] = 1;

            MatSetValuesStencil(*jac, 1, &row, ctr, col, val, INSERT_VALUES);

            /* the v part */
            row.c = 1;
            col[0].c = 1;
            val[0] = 1;

            MatSetValuesStencil(*jac, 1, &row, ctr, col, val, INSERT_VALUES);
        }
        else {
            /* u part */
            row.c = 0;
            col[0].i = i-1;

            /* v part */
            row.c = 1;
            col[0].i = i;
            col[0].j = j;
            val[0] = 1;

            MatSetValuesStencil(*jac, 1, &row, ctr, col, val, INSERT_VALUES);
        }
    }
}

else {
    /* u part */
    row.c = 0;
    col[0].i = i-1;

    /* v part */
    row.c = 1;
    col[0].i = i;
    col[0].j = j;
    val[0] = 1;

    MatSetValuesStencil(*jac, 1, &row, ctr, col, val, INSERT_VALUES);
}
col[0].j = j;
col[0].c = 0;
val[0] = 1.0 / hx / hx;

col[1].i = i+1;
col[1].j = j;
col[1].c = 0;
val[1] = 1.0 / hx / hx;

col[2].i = i;
col[2].j = j - 1;
col[2].c = 0;
val[2] = 1.0 / hy / hy;

col[3].i = i;
col[3].j = j + 1;
col[3].c = 0;
val[3] = 1.0 / hy / hy;

col[4].i = i;
col[4].j = j;
col[4].c = 0;

col[5].i = i;
col[5].j = j;
col[5].c = 1;
val[5] = X[j][i].u;

ctr = 6;
MatSetValuesStencil(*jac, 1, &row, ctr, col, val, INSERT_VALUES);

/* v part */
row.c = 1;

col[0].i = i-1;
col[0].j = j;
col[0].c = 1;
val[0] = 1.0 / hx / hx;

col[1].i = i+1;
col[1].j = j;
col[1].c = 1;
val[1] = 1.0 / hx / hx;

col[2].i = i;
col[2].j = j - 1;
col[2].c = 1;
val[2] = 1.0 / hy / hy;

col[3].i = i;
col[3].j = j + 1;
col[3].c = 1;
val[3] = 1.0 / hy / hy;

col[4].i = i;
col[4].j = j;
col[4].c = 1;

col[5].i = i;
col[5].j = j;
col[5].c = 0;

ctr = 6;
MatSetValuesStencil(*jac, 1, &row, ctr, col, val, INSERT_VALUES);

If the preconditioner matrix B was to be built, a similar loop could be used. Once the loop which builds the Jacobian has finished, the build must be finalized using the MatAssembly commands. Furthermore, the flag variable which is passed to this function is set to SAME_NONZERO_PATTERN, which indicates that the nonzero structure of the Jacobian does not change between consecutive Newton iterations.

MatAssemblyBegin(*jac, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(*jac, MAT_FINAL_ASSEMBLY);

*flag = SAME_NONZERO_PATTERN;

The last two lines in the function restore the solution vector and return zero.
6 Running the program

Running the make program will compile the code and produce an executable file called example that can be run just like any other MPI program. For example, typing

```
mpirun -np 2 example
```

will run the program with two processors, yielding the following output (where additional line breaks have manually been added)

```
grid size is 100 x 102
```
```
average number of GMRES iterations = (total GMRES its) / (Newton its) = 269 / 3 = 8.9667e+01
```
```
nonlinear eqns solved in 1.21 s
```

The solution that is obtained can be viewed in Figure 2. Additional information can be printed to the screen by passing certain flags at run time. For example, to monitor the convergence of the solution, the user can type

```
mpirun -np 2 example -snes_monitor
```

The output will be identical to above, but the solution norm at each Newton iteration will also be printed,

```
0 SNES Function norm 8.162817026456e+02
1 SNES Function norm 2.654346250560e+00
2 SNES Function norm 3.379326045044e-04
3 SNES Function norm 6.401449953644e-09
```

Similarly, passing the flag `-ksp_monitor` will print the residual norm of the linear system after each GMRES iteration. This can be useful for monitoring how effective a preconditioner is.

Monitoring memory usage might also be required. This can be done by passing the flags `-memory_info -malloc`, which will append the maximum memory (in bytes) that each processor used, along with some additional information, to the regular program output. For instance, adding these flags to the example program add the following output

```
Summary of Memory Usage in PETSc
[0]Current space PetscMalloc()ed 1.28479e+07, max space PetscMalloced() 1.32604e+07
[0]Current process memory 2.06807e+07 max process memory 2.06807e+07
[1]Current space PetscMalloc()ed 1.28479e+07, max space PetscMalloced() 1.32604e+07
[1]Current process memory 2.06316e+07 max process memory 2.06316e+07
```

The performance of a program depends largely on efficient memory management. If memory has to constantly be allocated to build a matrix, for example, the performance of the program will suffer greatly. The `-info` flag will output a large amount of information, including the number of mallocs that are required when building a matrix. To display only the malloc information, one can type

```
mpirun -np 2 example -info | grep malloc
```

It should be noted that all other output will be suppressed.
6.1 Changing linear solvers and the preconditioners

PETSc has the ability to change the type of linear equation solver and preconditioner dynamically. That is, these options can be changed at the command line without the need to recompile the program. This is an especially convenient feature when trying to determine which combination of linear equation solver and preconditioner to use.

The type of linear equation solver can be set by passing the flag \texttt{-ksp.type method}, where \texttt{method} is the method to be used. For example, one could set \texttt{method} to be \texttt{cg}, so that the conjugate gradient method is used. Setting the preconditioning method is done in a similar manner. For instance, the flag \texttt{-pc.type jacobi} will use Jacobi preconditioning. The specific options associated with the linear equation solver and the preconditioning can easily be set as well. The PETSc documentation can be consulted for a full (and extensive) list of the linear solvers, preconditioners, and their options. To illustrate how easy it is to change the solution method, the toy problem can be solved with the Stabilized BiConjugate Gradient Squared linear equation solver with additive Schwarz preconditioning using two processors via the commands

\begin{verbatim}
mpirun -np 2 example -ksp_type bcgs -pc_type asm
\end{verbatim}

The resulting output is

\begin{verbatim}
grid size is 100 x 102
average number of GMRES iterations = (total GMRES its) / (Newton its) = 183 / 3 = 6.1000e+01
nonlinear eqns solved in 0.81 s
\end{verbatim}

Therefore, this combination of linear equation solver and preconditioning method is superior to the default method that is hard coded into the program, which is GMRES(100) with block Jacobi preconditioning.
To check which method PETSc is using to solve a set of equations, the user can pass the flags
\texttt{-snes\_view} and \texttt{-ksp\_view}. These will display, among other things, the type of Newton’s method
that is being used and the tolerances, along with all of the information about the linear equation
solver and its associated preconditioner.

7 Summary

PETSc is an easy to use and robust set of parallelized libraries that can be used to solve nonlinear
partial differential equations on a multiprocessor computer. Because it was written specifically for
solving PDEs, many of the difficulties that are encountered when working in parallel, such as the
need to gather information at ghost points, become almost transparent to the user. Furthermore,
PETSc has a large suite of linear equation solvers and preconditioning methods that can easily be
changed without the need to recompile the program. To illustrate the tools that PETSc provides,
a simple nonlinear PDE is solved using PETSc functions.
## Index

Accessing vectors, 7  
Assembling matrices, 9  
Assigning vectors, 7  

Convergence monitoring, 30  
Creating matrices, 9  
Creating vectors, 6  

DA, see Distributed arrays  
DACreate2d, 4  
DACreateGlobalVector, 6  
DACreateLocalVector, 7  
DAGetLocalInfo, 5  
DAGetMatrix, 9  
DAGlobalToLocalBegin, 8  
DAGlobalToLocalEnd, 8  
DALocalInfo, 5  
DAVecGetArray, 7  
DAVecRestoreArray, 8  
Direct methods, 14  
Distributed arrays, 4  

File output, 22  

Getting grid information, 5  
Ghost points, 4  
Global vector, 6  
Grid information, see getting grid information  

Headers, 15  

Initial guess, 12, 22  
Iteration number, 21  

Jacobian, 12  
Jacobian via finite differences, 12  

Krylov methods, 13  
KSP, see linear equations, 13  
KSPGetPC, 13  
KSPSetType, 13  

Linear equations, see solving equations  
local grid, 3  
Local vector, 6  

Loop over grid, 6  
makefile, 14  
MatAssemblyBegin, 10  
MatAssemblyEnd, 10  
MatDestroy, 22  
Matrices, 9  
MatSetValuesStencil, 9  
Memory, freeing, 22  
Monitoring memory, 30  

Nonlinear equations, see solving equations  

PC, see preconditioning, 13  
PCSetFromOptions, 13  
PCSetType, 13  
PetscFinalize, 22  
Preconditioning, 11, 13  
Printing to file, see file output  
Printing to stdout, 20  

SNES, 11, see nonlinear equations  
snes_monitor, 30  
SNESCreate, 11  
SNESGetKSP, 13  
SNESGetFromOptions, 12  
SNESSetFunction, 11  
SNESSetJacobian, 12  
SNESolve, 12  
Solving equations, 11  

Timing, 21  

VecDestroy, 22  
Vectors, 6  
VecZeroEntries, 7