Parallel Implementation of Finite Element Code using PETSc

by
Jennifer K. Houchins

A paper submitted to the
Department of Mathematical Sciences
Clemson University

In partial fulfillment of the
requirements for the degree of

Master of Science
Mathematical Sciences

Advisors: Dr. Christopher L. Cox and Dr. Eleanor W. Jenkins

Acknowledgments

I would like to thank my advisors, Dr. Cox and Dr. Jenkins, and also my committee member, Dr. Warner for their continued guidance and patience throughout this work. I would also like to thank J. Barr von Oehsen, Ed Duffy, Brad Settlemyer, and Haipeng Li from the Center for Advanced Engineering Fibers and Films for all of their help. Also, I would like to thank my parents and husband for all their support and encouragement in the completion of this work.

This work was supported by the ERC program of the National Science Foundation under award number EEC-9731680.

Contents

List of Tables

List of Figures

1 Introduction

In this work we are interested in incorporating the Portable Extensible Toolkit for Scientific Computation (PETSc) into the Center for Advanced Engineering Fibers and Films viscoelastic flow model to allow for a parallel implementation and added flexibility in choosing a method of solution. In order to reach this goal, two sample finite element problems are examined, a one-dimensional steady state problem and the two-dimensional Stokes problem. The Stokes equations for the steady flow of a viscous fluid represent slow flows (i.e., a low Reynolds' number) and therefore play a fundamental role in the numerical solution of the Navier-Stokes equations. In particular, the Stokes Equations represent the limiting case of zero Reynolds' number for the Navier-Stokes equations of the viscoelastic flow problem [2]. It is for this reason that the Stokes problem is extremely important to the modeling of incompressible viscous fluid flows.

The application motivating this work is viscoelastic flow associated with polymeric fiber and film processes. Methods for handling time-dependence and nonlinearities in these equations often involve an iterative procedure in which the Stokes equations, or a modified version, are solved at each step. See for example, Saramito's implementation of the θ-method in [3]. PETSc contains libraries of such numerical methods that can be applied directly to these problems. In this work, we will explore obtaining PETSc and utilizing it's features; in particular, the built-in parallel assembly functionality, which will give us more flexibility in moving to a parallel implementation of our application codes. This manuscript will include a discussion of PETSc and its features, the process of integrating PETSc into the existing application codes of the two sample problems, and converting to a parallel implementation of these application codes.

1.1 What is PETSc?

The Portable Extensible Toolkit for Scientific Computation (PETSc)[1] was developed in the Mathematics and Computer Science Division at Argonne National Laboratory. The version of PETSc used in this work is 2.3.0. The release date for this version was April 26, 2005. PETSc can be found at http://www.mcs.anl.gov/petsc as a free download. This website also includes full documentation of PETSc with installation instructions, troubleshooting guide, tutorials, and a list of frequently asked questions (FAQ). PETSc is a set of software tools for users writing large-scale application codes involving the solution of Partial Differential Equations (PDEs) and similar problems who wish to avoid hard coding their own numerical solvers. PETSc consists of a set of libraries that contain routines for creating vectors, matrices, and distributed arrays, both sequential and parallel, as well as libraries of linear and nonlinear numerical solvers. PETSc also incorporates time-stepping methods and graphics. Figure 1 describes the abstract components of PETSc. PETSc utilizes the Message Passing

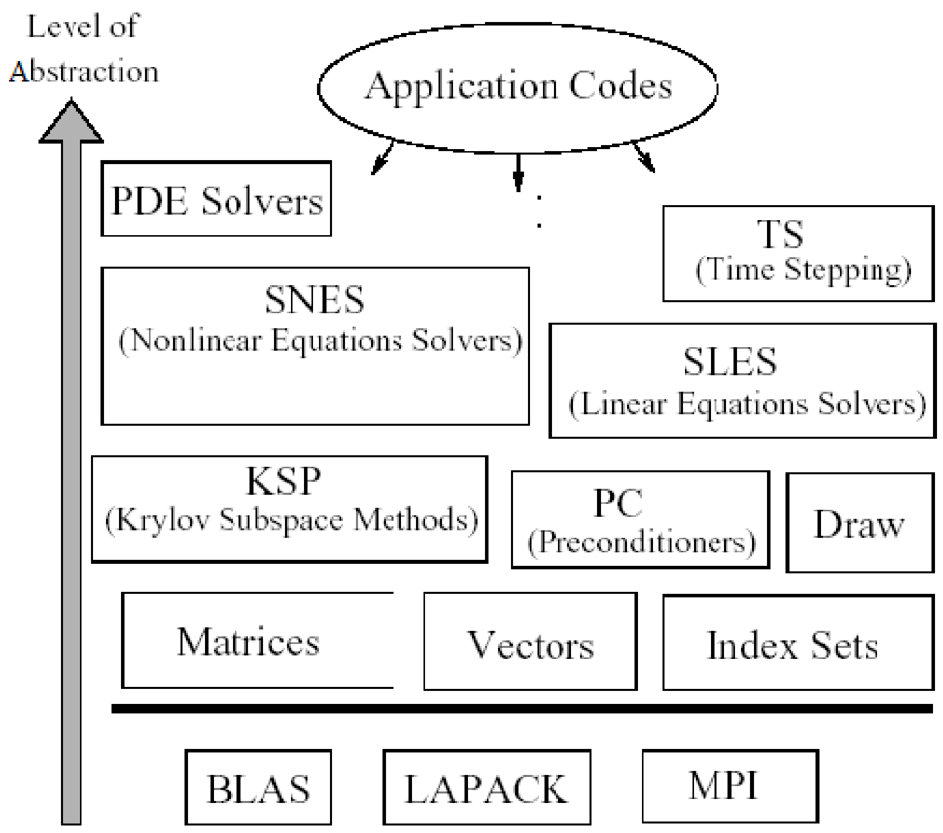

Figure 1: Organization of the PETSc libraries [1].

Interface (MPI) standard for all message passing communication. PETSc is well-documented and provides many example codes to help new users make use of PETSc components in their own application codes. PETSc can be used to simply solve a linear system of equations or it can be used to solve the more complex finite element problems upon which we will focus. Using PETSc will allow more flexibility in our application codes as well as simplifying the amount of coding required. For instance, PETSc contains a library of linear solvers called KSP, in which the user only has to change a run-time option for the KSP context in order to switch the type of solver to be used in the solution of their particular problem. This design makes it easy to compare the performance of many solvers for a problem and determine the optimal method of solution. PETSc also provides many runtime options which allow the user to log a program's performance and obtain information such as floating point operation (flop) rates, message passing activity and memory usage. Another attractive feature provided by PETSc is a runtime option that will start the debugger should the program encounter an error. PETSc also allows users to utilize external packages and solvers. This wide range of features combined with the flexibility of being able to use PETSc with code written in C, C++, or Fortran and its portability to multiple platforms make PETSc an attractive option for our application. In the following sections we discuss configuring PETSc, installing MPICH2, and using PETSc in existing application codes.

1.2 Configuring PETSc

Here we will discuss downloading and configuring PETSc on the Fedora Core 4 operating system under the Bourne Again SHell (bash). To get started, one must first go to http://www-unix.mcs.anl.gov/petsc/petsc-as/download/index.html and download either petsc.tar.gz, the full distribution of PETSc, or petsc-lite.tar.gz, the version of PETSc with no documentation. If storage space is a concern, then one can download the petsc-lite.tar.gz and access all of the documentation online at http://www-unix.mcs.anl.gov/petsc/petsc-as/documentation/index.html. One can also download previous versions of PETSc, but it is recommended that the most recent version of PETSc be obtained. Once the proper version of PETSc is downloaded and moved to the directory where PETSc is to be installed, the user will need to execute the following commands:

```
> gunzip -c petsc.tar.gz | tar -xof-
> cd petsc-2.3.0
> PETSC_DIR=‘pwd‘; export PETSC_DIR
> ./config/configure.py --download-f-blas-lapack=1 --download-mpich=1
> make
> make test
```

which would create the directory petsc-2.3.0, move into the directory, configure PETSc with the options to download BLAS, LAPACK, and MPICH assuming that they are not already installed, run make on PETSc and test to make sure everything works properly. When these commands were executed on the Fedora Core 4 operating system, a message saying "No such file or directory" was encountered after the configure command. When checking the installation page of the PETSc website [4], it was found that upon encountering such a message, the user should try changing the command

```
> ./config/configure.py --options
```

to the command

```
> python ./config/configure.py --options
```

which did remedy the problem and allowed the command to be executed. To obtain a full list of all options for configuring PETSc, the user can give the configure command the -help option. A sample of the output produced when configuring PETSc is shown in Figure 2.

The configuration of PETSc ran smoothly until it began running make on MPICH and then it gave the message "KILLED." Therefore, MPICH2 was downloaded and installed separately, and then the MPICH2 install directory was given to the configure command of PETSc as the

```
-with-mpi-dir={full path to mpich2 install directory}
```

option. The process of downloading and installing MPICH2 is discussed in the next section.

```
=========================================================================
             Configuring PETSc to compile on your system
=========================================================================
Compilers:
  C Compiler:         /usr/local/bin/mpich2-install/bin/mpicc  -Wall -g3
  Fortran Compiler:   /usr/local/bin/mpich2-install/bin/mpif90  -I. -Wall -g
PETSc:
  **
  ** Configure has determined that your PETSC_ARCH must be specified as:
  ** PETSC_ARCH: linux-gnu
  **
  PETSC_DIR: /usr/local/petsc-2.3.0
  ** Please make the above changes to your environment or on the command line for make.
  **
  Scalar type:real
  Clanguage: C
MPI:
  Includes: ['/usr/local/bin/mpich2-install/include', '-I.', '-I/usr/local/bin/
            mpich2-install/include', '-I/usr/local/bin/mpich2-install/include']
  Library: ['libnsl.a', 'librt.a']
X11:
  Includes: ['-I/usr/X11R6/include']
  Library: ['-L/usr/X11R6/lib -lX11']
BLAS/LAPACK: -Wl,-rpath,/usr/local/petsc-2.3.0/externalpackages/fblaslapack/linux-gnu
-L/usr/local/petsc-2.3.0/externalpackages/fblaslapack/linux-gnu -lflapack -Wl,-rpath,
/usr/local/petsc-2.3.0/externalpackages/fblaslapack/linux-gnu
-L/usr/local/petsc-2.3.0/externalpackages/fblaslapack/linux-gnu -lfblas
```

Figure 2: Output Produced When Configuring PETSc.

1.3 Installing MPICH2

MPICH2 is a freely available implementation of the Message-Passing Interface (MPI), the standard for message-passing communication[5]. The CH in MPICH2 stands for Chameleon, the layer used to provide portability to existing message-passing systems. The version of MPICH2 used in this work is 1.0.2p1, which was released on July 13, 2005 and is available for download at http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm#download. Once the current version of MPICH2 is downloaded, it can be installed with the following set of commands:

```
> gunzip -c mpich2-1.0.2p1.tar.gz | tar xf -
> mkdir /tmp/you/mpich2
> cd /tmp/you/mpich2 /usr/local/bin/mpich2-1.0.2p1/configure \
     -prefix=/usr/local/bin/mpich2-install 2>&1 | tee configure.log
> make 2>&1 | tee make.log
> make install 2>&1 | tee install.log
> export PATH=/usr/local/bin/mpich2-install/bin:$PATH
```

Execution of these commands will make a temporary directory for the build of MPICH2, configure MPICH2 in the default location of /usr/local/bin with the install directory of mpich2-install, build MPICH2, and then install the MPICH2 commands such as mpirun and mpiexec, and add the bin subdirectory of the installation directory to the PATH variable. Once all of this is completed, the user can execute the commands

```
> which mpd
> which mpicc
> which mpiexec
> which mpirun
```

to make sure everything is working correctly. The default process manager for MPICH2 is MPD, which is a ring of daemons on the machines that you intend to run your MPI programs. MPD looks in your home directory for a file named .mpd.conf which contains a single line

```
secretword=<secretword>
```

where <secretword> is known only to yourself and the file is readable and writable only by you. Now execution of the commands

```
> mpd &
> mpdtrace
> mpdallexit
```

will bring up a ring, test the mpdtrace command, which should return the host name of the machine you are running on, and bring down the ring. Once satisfied with MPICH2, the installation directory can be supplied to PETSc as a configuration option in order to utilize MPICH2 in conjunction with PETSc. Once this process is complete and PETSc is installed with use of MPICH2, the user is then ready to start adding PETSc to existing application codes.

1.4 Adding PETSc to Existing Code

Adding PETSc to an existing code is a relatively easy process. The first thing a user needs to do is set the PETSc environment variables, PETSC_DIR and PETSC_ARCH. PETSC_DIR specifies the full path of the home directory of PETSc while PETSC_ARCH indicates the type of architecture for which PETSc was compiled. These environment variables can be set in a number of ways. For instance, they can both be specified on the command line as options to make when compiling and linking code. Another way to set them is in the makefile for the code that will be using PETSc. All PETSc codes use makefiles which we discuss later in this section. However, for those who will be using PETSc more extensively, it is easier to set these variables as a part of the user's environment, such as the UNIX C shell or the Bourne Again Shell (bash). For the UNIX C shell, this can be done with adding the line

```
setenv PETSC_DIR [full path to petsc directory]
```

to the user's .cshrc file. The command to do this in the bash environment is

```
 export PETSC_DIR=full path to petsc directory
```

which should go in the user's .bash_profile file. Similar commands can be used to set the PETSC_ARCH variable. The user also needs to know the method to begin MPI jobs on their particular system. Once these steps have been taken, the user is then ready to start adding PETSc objects and function calls to their application code. To utilize PETSc libraries the user needs to include the appropriate header file. For example, to use the library for Krylov Subspace Methods, one needs to add the include statement

```
#include "petscksp.h"
```

to their include statements. Adding the header file for the library of solvers that one wishes to use will automatically include the other header files PETSc needs, such as those for base routines, system routines, viewers, vectors, and matrices. Next, programs that will use PETSc must begin by calling

```
PetscInitialize(int *argc,char **argv,char *file,char *help);
```

which is the function that initializes PETSc and also MPI if it has not been previously initialized. Respectively, programs using PETSc must also finalize PETSc by calling

```
PetscFinalize();
```

as one of their final statements.

```
ALL : example
CFLAGS =
FFLAGS =
CPPFLAGS =
FPPFLAGS =
include ${PETSC_DIR}/bmake/common/base

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

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

Figure 3: Sample Makefile for Maintaining a Single Program Using PETSc.

1.4.1 Makefiles

Once the user has finished adding the needed PETSc calls, the code needs to be compiled. PETSc programs use makefiles to maintain portability to multiple platforms. A sample makefile for a PETSc program can be found in Figure 3.

In Figure 3, the line

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

is very important and should not be altered within the makefile. This line automatically includes the needed definitions and other makefiles for a particular installation of PETSc. Also, the variable $ {PETSC_LIB} that appears on the linking line in Figure 3 specifies the PETSc libraries in correct linking order. If the user wishes to use only a specific library, then this variable can be replaced by others such as $ {PETSC_KSP_LIB} or $ {PETSC_SNES_LIB} . For more detailed information on makefiles, the user should consult [1].

1.4.2 Matrices and Vectors in PETSc

Matrices and vectors are extremely important to the finite element method. For this reason, we are particularly interested in the maintenance of these structures within the PETSc framework. First, we will discuss matrices in the PETSc environment. To begin using PETSc's matrices, the user must add the statement

```
#include "petscmat.h"
```

if they have not already included the header file for a group of solvers, such as

```
#include "petscksp.h"
```

which will automatically include that needed header files for the PETSc matrices and vectors. Once the appropriate header files have been included, the user can declare a PETSc matrix in the following way,

```
 Mat A;
```

which is the declaration for an object of the type matrix in PETSc.

After declaring a matrix object, the user must then use PETSc calls to create the matrix. There are several ways to create a matrix with PETSc. If one wishes to create a sequential matrix with PETSc and control the preallocation of memory, then the PETSc call one would use is

```
 MatCreateSeqAIJ(PETSC_COMM_SELF,int m,int n,int nz,int *nnz,Mat *A);
```

which would create a sequential matrix where PETSC_COMM_SELF is the communicator for a single processor, m is the number of rows, n is the number of columns, nz is the expected number of nonzeros per row, and nnz is an optional array of nonzeros per row if the number of nonzeros varies greatly for different rows. To have PETSc control all the memory allocation nz can be set to 0 and nnz can be specified as PETSC_NULL. Note that it is less costly to preallocate the memory for the matrix and if the number of nonzeros in any row is underestimated PETSc will automatically obtain the additional space needed with a small cost to the efficiency at the final assembly of the matrix. The call to create a parallel matrix in PETSc is very similar to the above. It is

```
 MatCreateMPIAIJ(PETSC_COMM_WORLD,int m,int n,int M,int N,
              int d_nz,int *d_nnz,int o_nz,int *o_nnz,Mat *A);
```

where m, M, and N specify the number of local rows and number of global rows and columns, n is the number of columns corresponding to a local parallel vector, d_nz and o_nz are the number of diagonal and off-diagonal nonzeros per rows, and d_nnz and o_nnz are optional arrays similar to nnz for the sequential creation command. As with the sequential command, PETSc can control all allocation of memory by having d_nz and o_nz set to zero and d_nnz and o_nnz set to PETSC_NULL. Either the local or global parameters (but not both) can be set to PETSC_DECIDE to have PETSc determine how the matrix is split. If the user does not use PETSC_DECIDE then the parameters must be chosen in such a way as to be compatible with the way that the corresponding vectors are split. Another way to a create a matrix with PETSc is to use the commands,

```
MatCreate(PETSC_COMM_WORLD,Mat *A);
MatSetSizes(Mat A,int m,int n,int M,int N);
```

which create a sequential matrix when using one processor and a parallel matrix when using two or more processors. The user specifies either the global dimensions M and N or the local dimensions m and n and PETSc controls all memory allocation. This method allows the user to switch between matrix types without having to change the PETSc calls and automatically uses the sparse AIJ format, or compressed sparse row format (CSR), for the matrix. The user should consult [1] for information on additional matrix format options.

Now that the matrix has been created, it is time to insert values. This can be done in two ways with PETSc, by either inserting a single value or by inserting groups of values. Also, there are similar procedures for creating vectors and inserting values into those vectors. The calls pertaining to the creation and maintenance of PETSc vectors can be found in [1]. We will examine the process of creating PETSc matrices and vectors as well as using PETSc to execute the solve and provide parallel implementation within two sample finite element codes in the following sections.

1.5 The Platform

The code development and testing for this work was performed on the CAEFF cluster dagobah. Dagobah is a sixteen node cluster. Each node has two 2.4 GHz Opteron 64 bit processors with 8 GB RAM, 160 GB hard drive, and 1 Gbps ethernet and infiniband. The cluster also has a 24 port Gig-E switch and 24 port infiniband switch. To perform a test run, a pbs script was used to submit a job to the cluster's queue. When the job is run through the queue, two output files are produced, job_name.o[jobid#] and job_name.e[jobid#], where the job name is given in the script file, the o signifies the output file, the e signifies the error file (should your job produce any errors), and the job id number is the number assigned to the job when it enters the queue. Execution of the command qstat will give a brief description of the status of each job currently in the queue. The script used to submit the job, the hosts file used for parallel implementation, and a sample output file can all be found in Appendix B.

2 A 1D problem

The first sample problem is a one-dimensional steady-state problem. The differential equation is given in [6] as

$$(-pu')' + qu = f \tag{1}$$

where f, u, p, and q are all functions of x. The boundary conditions are given by

$$0 \leq x \leq 1 \tag{2}$$

$$u(0) = u_0 \tag{3}$$

$$u(1) = u_1 \tag{4}$$

where $u(0) = u_0$ and $u(1) = u_1$ are imposed explicitly on the solution. That is, the code for the finite element solution uses Dirichlet boundary conditions at each end of the interval. Therefore, for n intervals, we will have $n - 1$ unknowns. The weak formulation of the differential equation is given by

$$\int_0^1 (pu'v' + quv)dx = \int_0^1 fvdx \tag{5}$$

where the test space functions satisfy

$$v(0) = v(1) = 0 \tag{6}$$

and we have continuous piecewise linear trial functions for the Ritz-Galerkin approximation. The numerical integration performed uses Simpson's Rule,

$$\int_{x_{i+1}}^{x_i} w(x)dx \approx \frac{x_{i+1} - x_i}{6} \left[w(x_i) + 4w\left(\frac{x_i + x_{i+1}}{2}\right) + w(x_{i+1}) \right]. \tag{7}$$

In the following sections, we will discuss the original framework of the code, the process of adding PETSc to the code, and transitioning to a parallel implementation.

2.1 The Original Code

The original code consisted of eleven C files, a makefile, main.c, fem.h, fem.c, bspl.c, p.c, q.c, trislv.c, f.c, exact.c, and errors.c [7]. The makefile controlled the linking of the files and would create a code listing after execution of the command

```
> make list
```

called listing. An executable could be obtained by executing the

```
> make main.exe
```

command. Running the executable would produce the output found in Figure 4.

```
                finite element solution

          n     l-2 error       max error
          8     0.000417727     0.000581229
          16    0.000105368     0.000146215
          32    2.64006e-05     3.65662e-05
```

Figure 4: Output produced from execution of original serial 1D code.

Main.c was the driver code which set up the grid, called fem.c, and displayed the error results. The header file fem.h contains all the function prototypes for the functions contained in the following files. The function fem contained in fem.c implemented the initialization of the arrays which would make up the system matrix and right hand side vector. It also set up the quadrature nodes and weights, built the system matrix and right hand side vector, performed the solve of the system, and called the function errors from errors.c. The file bspl.c performed the function evaluation of the linear basis function and their derivatives. The files p.c and q.c computed the coefficient values of the 0^{th} and 1^{st} order terms of the original differential equations. The file trislv.c performed the solve for the tri-diagonal system created and overwrote the right hand side vector with the solution vector. The file f.c computed the right hand side vector for the original differential equation, exact.c returned the value of the exact solution of the differential equation to used to set the boundary conditions for error analysis, and errors.c calculated the l_2-norm and l_∞-norm errors between the finite element solution and the exact solution. This code did not involve any dynamic memory allocation.

2.2 Transitioning to a PETSc Solve

To start the transition to a PETSc solve, the makefile was edited to link in the PETSc libraries and the driver program was edited to initialize and finalize PETSc. Performing these changes allowed for checking that things were working correctly. With the changes to the makefile, the PETSc function calls could now be utilized. The original makefile can be found in Appendix A. We will now discuss the changes that were made.

The first change that was made was the addition of the lines

```
PETSC_DIR=/home/software/petsc-2.3.1-p2
PETSC_ARCH=linux-gnu-c-real-debug
MPI_PATH=/usr/local/mpich2
CXX=mpicxx
```

to the makefile to set the PETSc environment variables PETSC_DIR and PETSC_ARCH. They also set the path for the version of MPICH2 with which PETSc was configured. Also the compiler being used was changed to the MPICH2 compiler mpicxx.

The next change to the makefile was that the line

```
INCLUDE = -I. -I/usr/include/ -I/usr/local/include -I/usr/local/lib/glib/include
```

was replaced with the lines

```
STUFF = -DMPICH_IGNORE_CXX_SEEK -I. -I/usr/include/ -I/usr/local/include\
   -I${PETSC_DIR}/include\
   -I${PETSC_DIR}/bmake/${PETSC_ARCH}\
   -I${MPI_PATH}/include
```

and the variable INCLUDE was replaced with STUFF in all of the compilation lines because the keyword **include** is important to PETSc. In particular the lines

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

and

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

were added to the makefile just before the lines that link and compile the files for the code and at the end of the makefile respectively. Also, the LDFLAGS variable was changed to

```
LDFLAGS =\
   -L$(PETSC_DIR)/lib/$(PETSC_ARCH)\
   -L${MPI_PATH}/lib\
   -L${MPI_PATH}/lib64\
   -lpetscdm\
   -lpetscksp\
   -lpetscmat\
   -lpetscvec\
   -llapack\
   -lpthread\
   -lpetsc
```

to ensure all of the necessary PETSc and MPICH2 libraries could be located and linked into the code. The variable CXX which was set to mpicxx in the lines above was used as the compiler in the updated version of the makefile and replaced the **cc** in all of the compilation lines. With these changes in place, the code can be edited to use any and all of the PETSc features that are useful to our application. The final version of the makefile used for the one-dimensional application code can be found in Appendix A. And for ease of typing, the makefile was edited so that typing

```
> make driver
```

now compiles and links all of the code to make the executable called driver. Now that we have full use of PETSc and all of its features, we can discuss the changes made to the main parts of the one-dimensional code.

First, the functions contained in the files bspl.c, errors.c, exact.c, f.c, p.c, q.c, and trislv.c remained unchanged. The first file to be modified after the makefile was main.c. For our application, a linear system was of interest. Therefore, a struct was created to contain a PETSC linear system matrix, right hand side vector, solution vector, Krylov Subspace (KSP) solver context, and preconditioner (PC) context. Then all of the objects that we are interested in are stored in the struct and can be passed through the code together. The code forming the struct is just

```
#include "petscmat.h"
#include "petscksp.h"
typedef Mat PETSC_MAT;
typedef Vec PETSC_VEC;

typedef struct
{
    PETSC_MAT     Amat; /*linear system matrix*/
    KSP           ksp; /*linear solver context*/
    PC            pc;   /*preconditioner context*/
    PETSC_VEC     rhs; /*petsc rhs vector*/
    PETSC_VEC     sol; /*petsc solution vector*/

}PETSC_STRUCT;
```

which is part of the share.h header file that was added to the existing code and can be found in Appendix A. Then to utilize this linear system in the code, the declaration

```
 PETSC_STRUCT sys;
```

was added to the main function. The linear system matrix, right hand side vector, and solution vector were created with the commands

```
ierr = MatCreate(PETSC_COMM_WORLD, &obj->Amat);
ierr = MatSetSizes(obj->Amat,PETSC_DECIDE,PETSC_DECIDE,m,n);
ierr = MatSetFromOptions(obj->Amat);
```

and

```
ierr = VecCreate(PETSC_COMM_WORLD, &obj->rhs);
ierr = VecSetSizes(obj->rhs, PETSC_DECIDE, m);
ierr = VecSetFromOptions(obj->rhs);
ierr = VecDuplicate(obj->rhs, &obj->sol);
```

within the functions

```
Mat_Create(PETSC_STRUCT *obj, PetscInt m, PetscInt n)
```

and

```
Vec_Create(PETSC_STRUCT *obj, PetscInt m)
```

which were included in the file mycalls.c and where the PetscInt variables m and n are the sizes of the matrix and vectors. For our application, m and n are equal.

Mycalls.c was added to the original code to keep all PETSc maintenance behind the scenes for a more succinct main function and in keeping with the idea of object oriented programming. Mycalls.c and its header file, mycalls.h, can be found in Appendix A with the one-dimensional code in its entirety. The were also functions added to the mycalls.c file to do the final assembly of the matrix and vectors and to create the KSP context and preconditioner and perform the solve. A function

```
Petsc_View(PETSC_STRUCT obj, PetscViewer viewer)
```

was also added to create a MatLab file called "results.m" which contains the linear system matrix, right hand side, and solution if the user wishes to see the individual entries of each. A sample "results.m" file is included in Appendix A.

Once the linear system was created, the code in fem.c was edited to insert the values for the entries of the PETSc matrix and right hand side vector. To do this a variable, called add_term, was created to store the value to update the entry of the matrix or right hand side. Then the original code to update the matrix and right hand side vector

```
b[i]=b[i]+test*wght*f(xx); /* construct RHS */
```

and

```
atri[i][jj]=atri[i][jj]+aij*wght;
```

becomes

```
add_term=test*wght*f(xx); /* construct RHS */
ierr = VecSetValue(obj->rhs,(PetscInt)i,(PetscScalar)add_term,ADD_VALUES);
```

and

```
add_term=aij*wght;
ierr = MatSetValue(obj->Amat,(PetscInt)i,(PetscInt)j,(PetscScalar)add_term,ADD_VALUES);
```

respectively. A similar change is made to update for the boundary conditions. After this assembly process is completed in the fem function, the function

```
Petsc_Assem(PETSC_STRUCT *obj)
```

from mycalls.c is used to call the final assembly routines that PETSc requires before the solve can be performed.

After the final assembly, the system is now ready to be solved. The code to create the KSP context and perform the solve is located in the mycalls.c function

```
Petsc_Solve(PETSC_STRUCT *obj)
```

which consists of the following PETSc calls:

```
PetscErrorCode ierr;
ierr = KSPCreate(PETSC_COMM_WORLD,&obj->ksp);
ierr = KSPSetOperators(obj->ksp,obj->Amat,obj->Amat,DIFFERENT_NONZERO_PATTERN);

ierr = KSPGetPC(obj->ksp,&obj->pc);
ierr = PCSetType(obj->pc,PCNONE);
ierr = KSPSetTolerances(obj->ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);

ierr = KSPSetFromOptions(obj->ksp);

ierr = KSPSolve(obj->ksp,obj->rhs,obj->sol);

ierr = VecAssemblyBegin(obj->sol);
ierr = VecAssemblyEnd(obj->sol);
```

in which the solve is performed by the PETSc function

```
ierr = KSPSolve(obj->ksp,obj->rhs,obj->sol);
```

and the solution vector goes through final assembly in order to be viewed. In this portion of code, the KSP method being used is the default Generalized Minimal Residual (GMRES) with Classical Gram-Schmidt for the orthogonalization. The convergence tolerance we have set, rtol, is the decrease of the residual norm relative to the norm of the right hand side. This has been set to 10^{-7}. PETSc has three convergence tolerances (rtol,atol,and dtol - all based on the l_2 norm) and we have left atol and dtol as the default values which can be found in the user's manual [1]. And according to the user's manual, convergence is detected at the k^{th} iteration if

$$\|r_k\|_2 < max(rtol * \|b\|_2 \, , atol) \tag{8}$$

where $r_k = b - Ax_k$ and divergence is detected if

$$\|r_k\|_2 > dtol * \|b\|_2 \, . \tag{9}$$

In order to make sure the PETSc solution was correct, the original solution vector obtained from the tri-diagonal solver trislv was compared to the PETSc solution vector. When it was found that the two solutions matched, the PETSc solution was entered into a C double vector and supplied to the errors function in place of the original solution vector. To do this, the PETSc functions

```
ierr = VecGetArray(sys.sol,&get);
```

and

```
ierr = VecRestoreArray(sys.sol,&get);
```

were used to obtain a pointer (the variable get) to access the individual entries of the PETSc solution vector and restore the PETSc solution vector. Using this pointer, the PETSc solution was stored in the vector used by the errors function to calculate the l_2 and l_∞ norm errors which were output to a file called output.txt. At this point, the original solve was removed and the code in the fem function was split into two new functions, assem and bookkeep, to further the object oriented technique. The function bookkeep now performs all initialization and the set-up of the quadrature nodes and weights while the function assem builds the linear system. These functions are called from the main function along with the functions implemented in mycalls.c and the fem function is no longer used. The full one-dimensional code transitioned to a PETSc solve can be found Appendix A. And now that the code is transitioned to using PETSc for the solve, we are ready to edit the code to be run in parallel which we discuss in the following section.

2.3 Parallelization of the 1D Code

The way in which PETSc features were implemented in the functions provided in mycalls.c allows a parallel matrix and parallel vectors to be created automatically when the program is run on two or more processors. Similarly, if only one processor is specified the code becomes nothing more than the sequential code. PETSc also controls the way in which the system matrix, right hand side, and solution vectors were split across the processors being used on the cluster. This can be specified by changing the local sizes in the functions that create the matrix and vectors. However, for computational flexibility we have chosen to simply let PETSc decide the local sizes.

Since PETSc knows which portions of the matrix and vectors are locally owned by each processor, the solve is also completed in parallel without any extra coding required. However, the error calculations require more work. PETSc provides functions for calculation the l_2 and l_∞ norms, but they are separate calls. Since the original code includes a function for calculating both errors at the same time, we simply need to put the solution vector back together on each processor. Then each processor can do its own error calculation and write the results to the output file. This is done using MPI function calls to broadcast the locally owned part of the solution vector to all other processors in order to build a complete solution vector on each processor. This code to do this is shown in the Figure 5.

Each processor then executes the code that computes the errors and the errors are output with the rank of the corresponding processor. The output is shown in Figure 6. The rank

```
int sendcount,recvcount;
ierr = VecGetArray(sys.sol,&get);

for(i=0;i<n-1;i++)
     b[i] = get[i];  // store result in b for error calculation

PetscInt low,high,otherlow,otherhigh;
MPI_Status status;
PetscInt count;
int tag = 9999;
VecGetOwnershipRange(sys.sol,&low,&high);
VecGetLocalSize(sys.sol,&count);
// first set correct order on local node according to low&high
for(j=0;j<count;j++)
     bbb[low+j] = b[j];
for(i=0;i<size;i++)
{
    if(i != rank)
    {
     MPI_Send( &count, 1, MPI_INT, i,tag, MPI_COMM_WORLD );
     MPI_Send( &low, 1, MPI_INT, i,tag, MPI_COMM_WORLD );
     MPI_Send( &high, 1, MPI_INT, i,tag, MPI_COMM_WORLD );
     MPI_Send( b, count, MPI_DOUBLE, i,tag, MPI_COMM_WORLD );
    }
}
// each processor will receive #size-1 message
for(i=0;i<size;i++)
{
    if(i!=rank)
    {
     MPI_Recv (&recvcount,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
     MPI_Recv (&otherlow,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
     MPI_Recv (&otherhigh,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
     MPI_Recv (&bb[0],recvcount,MPI_DOUBLE,i,tag,MPI_COMM_WORLD,&status);

     for(j=0;j<recvcount;j++)
     {
       bbb[otherlow+j] = bb[j];
     }

    }

}
//synchronization
MPI_Barrier(MPI_COMM_WORLD);

for(i=0;i<n-1;i++)
{
  b[i] = bbb[i];  // store result in b for error calculation
}
```

Figure 5: Broadcasting locally owned portion of solution to other processors.

```
Number of processors = 2, rank = 0
Number of processors = 2, rank = 1
Solves a tri-diagonal system using KSP.

      finite element solution

rank     n       l-2 error        max error
[0]      8       0.000417727      0.000581229
[1]      8       0.000417727      0.000581229
[0]      16      0.000105368      0.000146215
[1]      16      0.000105368      0.000146215
[0]      32      2.62932e-05      3.6437e-05
[1]      32      2.62932e-05      3.6437e-05
```

Figure 6: Output produced from execution of parallel 1D code.

in Figure 6 refers to the processor that is responsible for the output. For the output shown in Figure 6, the code was run with the command

```
> mpiexec -n 2 ./driver
```

where mpiexec is the appropriate run command for MPICH2, -n is the option to specify the number of processors (two in this case), and driver is the name of the executable file produced from compiling and linking the one-dimensional code. So to make the one-dimensional code run completely in parallel, these were the only necessary changes to the main function. All other parts of the code remain the same and the updated main.c file can be found in Appendix A.

2.4 1D Results

There are several advantages to using PETSc and its features for this code. The first thing to note is that, while the original code overwrites the right hand side vector with the solution vector, the PETSc solve does not overwrite the right hand side vector but rather stores the solution in a separate vector. This allows for the preservation of the linear system which can then be output to the MatLab file "results.m." The solutions from the original code and PETSc were compared, as shown in Table 1, and it was found that PETSc produced the same solution as the original solver.

The reader must note that this problem can not use more than seven processors. The reason for this is that first loop through the code produces a system of size $(n-1)$x$(n-1)$

Original Solution	**PETSc Solution**
0.999028	9.9902823719975131e-01
0.996103	9.9610305140256372e-01
0.991225	9.9122446427192334e-01
0.984393	9.8439240042891485e-01
0.975607	9.7560685317576734e-01
⋮	⋮
0.234389	2.3438847398201978e-01
0.178721	1.7872125627672725e-01
0.121101	1.2110076603986547e-01
0.061527	6.1527011505653227e-02

Table 1: Comparison of Original and PETSc solutions.

where $n = 8$. If the problem were spread across more processors, then that would result in PETSc running out of entries of the matrix or vector to put on the processors of rank greater than 7 and the code would crash. It was also found that the PETSc solution produced a reduced l_2 and l_∞ error for systems of $n = 16$ and greater as can be seen in Figures 4 and 6. Therefore, PETSc allows us to write a parallel application code without having to split the components of the system up across the processors ourselves or produce a parallel solver. This gives us the flexibility to move to more complex problems, and in particular, problems of higher dimensions. In the following section, we will discuss just such a problem.

3 2D Stokes Problem

Our second sample problem is the two-dimensional Stokes problem. In [8] we find that this problem is governed by the conservation of mass

$$\nabla \cdot u = 0 \tag{10}$$

and the conservation of momentum

$$-\nabla \cdot \tau + \nabla p = 0 \tag{11}$$

along with the extra-stress tensor

$$\tau = 2\eta D(u) \tag{12}$$

which is proportional to $D(u)$ where

$$D(u) = \frac{1}{2}(\nabla u + \nabla u^T) \tag{13}$$

is the deformation tensor, u is the fluid's velocity, p is the fluid's pressure, and the viscosity, η, is assumed constant. The domain is

$$0 \leq x, y \leq 1 \tag{14}$$

with u specified on the entire boundary and p specified at one boundary point. The finite element trial functions used to approximate u are continuous piecewise quadratic, those to approximate p are continuous piecewise linear, and those to approximate τ are discontinuous piecewise linear (to set up the structure for the viscoelastic constitutive model).

3.1 The Original Code

The original two-dimensional Stokes code is quite a bit more complex than that of the one-dimensional code discussed earlier in this work. The Stokes code consisted of twenty-two files including the makefile [9]. The framework is similar to that of the original one-dimensional code. There is the main function provided in the main.c file, which opens the input file and reads in the input parameters, such as domain boundaries, that will be supplied to the stokes function, which is the driver subroutine for the finite element solution. The heart of the Stokes code lies in the stokes function contained in the stokes.c file. It is the stokes function that sets up the quadrature nodes and weights by calling the quadsetup function, calls the function lbasis to evaluate the linear basis functions at the quadrature nodes, and calls the function qbasis to evaluate the quadratic basis functions at the quadrature nodes. The stokes function also calls the bookkeeping function which sets up the coordinates, unknown, and element numbering. For the stokes code there are four assembly routines, asm_v_q_u_p, asm_v_tau, asm_sigma_u, and asm_sigma_tau, which build the linear system from within the stokes function. After the system is built by the four assembly routines, the stokes function calls the bande subroutine to perform the banded direct solve of the linear system. Once the solve is complete, the function avgp is used to obtain a zero-mean pressure and update the solution vector accordingly. Finally, the functions errors and strerrors are called with the solution vector as one of their parameters in order to obtain the l_2 and H_1 norm errors. To compile and link the code, the user need only type the command

```
> make main.exe
```

and then the command

```
> main.exe
```

to run the code. When running the code, an output file called "op.txt" is produced that contains all of the error calculations along with the solution. So, now that the original framework of the two-dimensional code has shown itself to be more complex and yet similar to that of the one-dimensional code, we will use the same approach of adding PETSc to the two-dimensional code that was discussed earlier for the one-dimensional problem. This process is detailed in the following section.

3.2 Transitioning to a PETSc Solve

Once again, to start transitioning to a PETSc solve, the makefile was the first file to be edited. The same changes that were made to the one-dimensional problem's makefile were necessary for the makefile of the Stokes code. The next change to the Stokes code was the addition of the files share.h, mycalls.h, and mycalls.c which were added to the one-dimensional code. By adding these files, the PETSc struct that was created to hold the PETSc linear system and its components and all of the functions written to have all PETSc maintenance behind the scenes could be utilized in the Stokes code. Once all of the PETSc functionality could be utilized, the code was edited to create the linear system and the four assembly routines were edited to update the values of the entries in the system matrix and right hand side vectors in the same manner that was used for the one-dimensional code. That is, a variable called add_term was introduced within each assembly routine that was set to the value meant to update the corresponding entry in either the system matrix or the right hand side vector and then supplied to the MatSetValue or VecSetValue PETSc functions to perform the updating. To accomplish this, the PETSc struct declared was supplied as a parameter to each of the assembly routines and their function definitions were changed accordingly in the stokes.h header file. The Petsc_Solve function included in the mycalls.c file was then used to perform the PETSc solve on the Stokes system that had been built. Then the avgp function was modified to update the PETSc solution vector for the zero-mean pressure by supplying the struct to avgp as a parameter in the same manner as it was supplied to the assembly routines. Once the PETSc solution vector had been updated for the zero-mean pressure, the lines of code

```
errors(mesh,quad,linbasis, quadbasis, solnflag, err, b);
strerrors(mesh,quad,linbasis,quadbasis,solnflag,strerr,b,alpha);
```

that calculated the l_2 and H_1 norm errors for the original solution were commented out and replaced with the lines

```
errors(mesh,quad,linbasis, quadbasis, solnflag, err, c);
strerrors(mesh,quad,linbasis,quadbasis,solnflag,strerr,c,alpha);
```

so that the errors could be calculated using the PETSc solution obtained. However, to accomplish this the PETSc solution was obtained by using the VecGetArray function supplied by PETSc in the same way that was used in the sequential solve for the one-dimensional code. Now that the PETSc solve has been achieved for the Stokes code sequentially, the method of moving to a parallel PETSc solve can be discussed.

3.3 Parallelization of the Stokes Code

The parallelization of the two-dimensional Stokes code was also similar to that of the one-dimensional code. The code in Figure 7 shows what is necessary to broadcast the portion of the solution that each processor owns to all the other processors. Once this is added to the stokes function, the two-dimensional code will run in parallel. Again, the parallel assembly of the system matrix and vector is taken care of by PETSc. The final product is like that of the one-dimensional code. If only one processor is specified to the mpiexec command, then we have essentially the sequential case, whereas if two or more processors are specified, PETSc automatically creates a parallel system.

3.4 Stokes Results

As with the one-dimensional code, the original solution was compared to the PETSc solution (both sequential and parallel) and it was verified that PETSc produced a similar solution vector. To do this the avgp function was removed when it was discovered that pressure needed to be set at a point in order to achieve the same solution with PETSc as the original solution. The results are shown in Table 2.

Original Sequential Solution	**PETSc Sequential**	**PETSc Parallel**
0.25	0.249997	0.25
0	-2.79823e-09	5.57771e-08
-0.360482	-0.360482	-0.360483
0.720965	0.720965	0.720965
-8.88178e-16	-2.79812e-09	5.57771e-08
⋮	⋮	⋮
-0.253426	-0.253427	-0.253426
-0.25	-0.250003	-0.25
2.23493	2.23493	2.23493
0.0695929	-0.0695908	0.0695931
-0.841596	-0.841599	-0.841596

Table 2: 2D Stokes Comparison of Original and PETSc solutions.

The l_2 and H_1 error norms were also calculated based on the PETSc solution and compared to those calculated based on the original solution. Again, PETSc produced the same results to those of the original solution. Those results are shown in Table 3.

4 Summary and Conclusions

So, we have seen that PETSc allows for added flexibility and more efficient coding time for application codes for problems of various sizes. More time can be spent in assembling a more

```
 PetscScalar *get;
 PetscInt  low,high,otherlow,otherhigh,count;
 MPI_Status  status;
 int test,test2,sendcount, recvcount,tag = 9999;
 double  c[nunknown],d[nunknown],e[nunknown];
 for(test2=0;test2<nunknown+1;test2++){
     c[test2]=0.0;
     d[test2]=0.0;
     e[test2]=0.0;
 }

 VecGetArray(sys.sol,&get);

for(test=1;test<nunknown+1;test++){

   e[test] = get[test];
}

VecGetOwnershipRange(sys.sol, &low, &high);
VecGetLocalSize(sys.sol, &count);
//first set the correct order on local node according to ownership range
for(test2=0;test2<count;test2++)
    d[low+j]=e[j];
for(test=0;test<size;test++)
{
    if(test != rank){
       MPI_Send(&count, 1, MPI_INT, test, tag, MPI_COMM_WORLD);
       MPI_Send(&low,1,MPI_INT,test, tag, MPI_COMM_WORLD);
       MPI_Send(&high,1,MPI_INT,test,tag, MPI_COMM_WORLD);
       MPI_Send(e, count, MPI_DOUBLE,  test, tag, MPI_COMM_WORLD);
    }
}
//each processor will receive size-1 messages
for(test=0;test<size;test++)
{
    if(test !=rank){
       MPI_Recv(&recvcount,1,MPI_INT, test, tag,MPI_COMM_WORLD, &status);
       MPI_Recv(&otherlow,1,MPI_INT,test,tag,MPI_COMM_WORLD, &status);
       MPI_Recv(&otherhigh,1,MPI_INT,test,tag,MPI_COMM_WORLD,&status);
       MPI_Recv(&c[0],recvcount,MPI_DOUBLE,test,tag,MPI_COMM_WORLD,&status);
       for(j=0;j<recvcount;j++) {
          d[otherlow+j] = c[j];
       }
   }
}
//synchronization
MPI_Barrier(MPI_COMM_WORLD);

for(test=1;test<nunknown+1;test++){

   c[test] = d[test];
}
VecRestoreArray(sys.sol,&get);
```

Figure 7: Code to make 2D Stokes code fully parallel.

	Original Solution	**PETSc Sequential**	**PETSc Parallel**
u1 l-2 error	0.043873	0.043873	0.043873
u2 l-2 error	0.043873	0.043873	0.043873
p l-2 error	0.538038	0.538064	0.538064
u1 h-1 error	0.672666	0.672666	0.672666
u2 h-1 error	0.672666	0.672666	0.672666
u l-2 error	0.062046	0.062046	0.062046
u h-1 error	0.951293	0.951293	0.951293
str1 l-2 error	0.433029	0.433029	0.433029
str2 l-2 error	0.389751	0.389751	0.389751
str3 l-2 error	0.433029	0.433029	0.433029
str l-2 error	0.823917	0.823917	0.823917

Table 3: 2D Stokes Comparison of norm errors for nx=ny=4.

complex problem instead of spending that time writing a solver specific to that problem. In this way, perhaps we can solve more problems with the time that will be saved not writing solvers built for specific problems but allowing PETSc to do the solve for us. PETSc also gives the added flexibility of moving to a parallel implementation without having to split up the problem for ourselves. All parallel assembly of matrices and vectors can be left to PETSc. This will allow for moving to larger and more complex problems such as the CAEFF viscoelastic flow problem.

4.1 Future Work

In the future, timing comparisons of the serial and parallel implementations should be considered for larger problems when communication time will not play a larger role than that of time spent computing. Timing comparisons of different platform configurations may also be considered (e.g.- Infiniband vs Myrinet network hardware) in order to determine if there is a correlation between communication time and means of communication. Also, these methods will be used to incorporate PETSc solvers and parallel implementation into the CAEFF integrated viscoelastic flow model. Finally, the reader should note that the parallelization methods used in this work are not necessarily the most efficient, since the automatic parallelization features of PETSc were used. Future plans call for timing comparisons between

PETSc's default parallelization and a more hands-on partitioning approach.

References

[1] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.1, Argonne National Laboratory, 2006.

[2] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer, second edition, 2002.

[3] P. Saramito. A new θ-scheme algorithm and incompressible FEM for viscoelastic fluid flow. *Math. Mod. and Num. Anal.*, 28:1–34, 1994.

[4] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[5] W. Gropp and E. Lusk. MPICH2 Web page, 2002. http://www-unix.mcs.anl.gov/mpi/mpich/index.htm.

[6] Christopher L. Cox. MthSc 983: Selected Topics in Computational Mathematics: Finite Element Methods Course Notes, 2005.

[7] Tamra Payne. One-dimensional Finite Element Code in C for an Elliptic Problem, 1998.

[8] Patti A. Sylvia. On viscoelastic flow modeling using Newton's Method. Master's thesis, Clemson University, 2003.

[9] Christopher L. Cox. Two-dimensional Finite Element Code in C for Stokes Equations, 2003.

A 1D Finite Element Code

A.1 Original Makefile

```
OBJS = fem.o bspl.o p.o q.o f.o trislv.o exact.o errors.o

#If there are any .h files
HS = fem.h

INCLUDE = -I. -I/usr/include/ -I/usr/local/include -I/usr/local/lib/glib/include

LDFLAGS =  -L. -L/usr/lib -L/usr/local/lib

LDLIBS = -lglib -lm

all:  $(OBJS)

fem.o: fem.h fem.c
     cc $(INCLUDE) $(LDFLAGS) -g -c fem.c

bspl.o: fem.h bspl.c
     cc $(INCLUDE) $(LDFLAGS) -g -c bspl.c

p.o: p.c
     cc $(INCLUDE) $(LDFLAGS) -g -c p.c

q.o: q.c
     cc $(INCLUDE) $(LDFLAGS) -g -c q.c

trislv.o: trislv.c
     cc $(INCLUDE) $(LDFLAGS) -g -c trislv.c

f.o: f.c
     cc $(INCLUDE) $(LDFLAGS) -g -c f.c

exact.o: exact.c
     cc $(INCLUDE) $(LDFLAGS) -g -c exact.c

errors.o: fem.h errors.c
     cc $(INCLUDE) $(LDFLAGS) -g -c errors.c

main.o: main.c $(HS)
     cc $(INCLUDE) $(LDFLAGS) -g -c main.c

main.exe: main.o $(OBJS)
     cc $(INCLUDE) $(LDFLAGS) $(LDLIBS) -g -o main.exe main.o $(OBJS)

list:
     cat main.c >> listing
     cat fem.h >> listing
     cat fem.c >> listing
     cat bspl.c >> listing
     cat p.c >> listing
     cat q.c >> listing
     cat trislv.c >> listing
     cat f.c >> listing
     cat exact.c >> listing
     cat errors.c >> listing
```

A.2 Makefile After Adding PETSC

```
PETSC_DIR=/home/software/petsc-2.3.1-p2
PETSC_ARCH=linux-gnu-c-real-debug
MPI_PATH=/usr/local/mpich2
CXX=mpicxx
#If there are any .h files
HS = fem.h share.h
STUFF = -DMPICH_IGNORE_CXX_SEEK -I. -I/usr/include/ -I/usr/local/include\
     -I${PETSC_DIR}/include\
     -I${PETSC_DIR}/bmake/${PETSC_ARCH}\
     -I${MPI_PATH}/include
OBJECTS = mycalls.o bspl.o p.o q.o f.o exact.o errors.o assem.o bookkeep.o

LDFLAGS =\
     -L$(PETSC_DIR)/lib/$(PETSC_ARCH)\
     -L${MPI_PATH}/lib\
     -L${MPI_PATH}/lib64\
     -lpetscdm\
     -lpetscksp\
     -lpetscmat\
     -lpetscvec\
     -llapack\
     -lpthread\
     -lpetsc

LDLIBS = -lglib -lm
```

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

all:  ${OBJECTS}

mycalls.o: share.h mycalls.h mycalls.c
     ${CXX} ${STUFF} -g -c mycalls.c

bspl.o: fem.h bspl.c
     ${CXX} ${STUFF} -g -c bspl.c

p.o: p.c
     ${CXX} ${STUFF} -g -c p.c

q.o: q.c
     ${CXX} ${STUFF} -g -c q.c

f.o: f.c
     ${CXX} ${STUFF} -g -c f.c

exact.o: exact.c
     ${CXX} ${STUFF} -g -c exact.c

errors.o: fem.h errors.c
     ${CXX} ${STUFF} -g -c errors.c

assem.o: fem.h assem.c
     ${CXX} ${STUFF} -g -c assem.c

bookkeep.o: fem.h bookkeep.c
     ${CXX} ${STUFF} -g -c bookkeep.c

main.o: main.c ${HS}
     ${CXX} ${STUFF} -g -c main.c

driver: main.o ${OBJECTS}
     ${CXX} -o driver main.o ${OBJECTS} ${LDFLAGS}

list:
     cat *.c >> listing
     cat *.h >> listing

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

A.3 1D Sequential PETSc Solve

```
#include "fem.h"

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% assem - assembly function for finite element solution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Arguments - PETSC_STRUCT *obj : pointer to a PETSC_STRUCT
%%     int n : number of nodes - 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void assem(PETSC_STRUCT *obj,int n){

PetscErrorCode ierr;

/*   Loop over the intervals for assembly                                     */
        for(intrvl=0;intrvl<n;intrvl++)
       {
/*   Loop for Simpson's rule                                                  */
            for(iquad=0;iquad<3;iquad++)
          {
               wght=dlen[intrvl]*qdwt[iquad]; /* compute quadrature weight */
               xx=xquadpt[intrvl][iquad];     /* get quadrature point      */

                for(ibasis=0;ibasis<2;ibasis++) /* get basis function i */
             {
              ileft=ni_indx[intrvl][ibasis];
                basisi=bspl(xx,intrvl,ibasis,1); /* basis function */
                  drvi=bspl(xx,intrvl,ibasis,2);  /* derivative of basis
                                                      function        */
                  i=indx[ileft];
                  if(i>=0)
              {
                testprime=drvi;
                test=basisi;
                b[i]=b[i]+test*wght*f(xx); /* constuct RHS */
            add_term=test*wght*f(xx); /* constuct RHS */
                  ierr = VecSetValue(obj->rhs,(PetscInt)i,(PetscScalar)add_term,ADD_VALUES);

            for(jbasis=0; jbasis<2; jbasis++)
            {                 /* get basis function j */
                       jendpt=ni_indx[intrvl][jbasis];
                       basisj=bspl(xx,intrvl,jbasis,1);
                       drvj=bspl(xx,intrvl,jbasis,2);
                       j=indx[jendpt];
```

```
                        aij=p(xx)*testprime*drvj+q(xx)*test*basisj;
               if(j>=0)
      {  /* store tri-diagonal matrix in efficient form */
                       jj=2+j-i-1;
                        atri[i][jj]=atri[i][jj]+aij*wght;
                        add_term=aij*wght;
                    ierr = MatSetValue(obj->Amat,(PetscInt)i,(PetscInt)j,(PetscScalar)add_term,ADD_VALUES);
                   }
                         /*    Apply boundary conditions                                              */
                         if((intrvl==0)&&(j==-1))
                   {
                 b[i]=b[i]-aij*wght*exact(x[0]);
                 add_term=(-1.0)*aij*wght*exact(x[0]);
                 ierr = VecSetValue(obj->rhs,(PetscInt)i,(PetscScalar)add_term,ADD_VALUES);
                    }
                          if((intrvl==n-1)&&(j==-1))
                    {
                 b[i]=b[i]-aij*wght*exact(x[n]);
                 add_term=(-1.0)*aij*wght*exact(x[n]);
                 ierr = VecSetValue(obj->rhs,(PetscInt)i,(PetscScalar)add_term,ADD_VALUES);
                    }
                  }
              }
         }
      }
}


    return;

}
#include "fem.h"

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% bookkeep - bookkeeping function for fem
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Argument - int n : number of nodes - 1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/


void  bookkeep(int n){


/*      Set up x array so that xpoints can be made global                    */
        for(i=0;i<n+1;i++)
        {
           x[i] = xpts[i];
           /* printf(" i %d x[i] %g \n",i,x[i]); */
        }

/*      Initialize arrays to hold the tri-diagonal system                    */
        for(i=0;i<n-1;i++)
        {
            b[i]=0;
            for(j=0;j<=2;j++) atri[i][j]=0;
        }


/*      Initialize quadrature weight coefficients                     */
       qdwt[0]=1.0/6.0;
       qdwt[1]=4.0/6.0;
       qdwt[2]=1.0/6.0;


/* Initialize offset array between node numbers and interval numbers     */
       indx[0]=-1;
       indx[n]=-1;
       for(i=1;i<n;i++) indx[i]=i-1;

       iright=0;

/*  Loop over intervals to set up the array holding the endpoints of the      */
/*  intervals, ni_indx, the array holding the points of evaluation for        */
/*  Simpson's rule, xquadpt, and the array to hold the interval lengths, dlen.*/

        for(intrvl=0;intrvl<n;intrvl++)
{
          ileft=iright;
            iright=ileft+1;
            xleft=x[ileft];
            xright=x[iright];
            dintrvl=xright-xleft;
            xquadpt[intrvl][0]=xleft;
            xquadpt[intrvl][1]=.5*(xleft+xright);
            xquadpt[intrvl][2]=xright;
           dlen[intrvl]=fabs(dintrvl);
            ni_indx[intrvl][0]=ileft;
            ni_indx[intrvl][1]=iright;
  }

return;
}
```

```
/*********************************************************************
 * Function to return value of basis function or derivative of basis   *
 *             function.                                              *
 **********************************************************************
 *                                                                    *
 *       Arguments:                                                   *
 *         xx      = point at which the basis function is to be       *
 *                     evaluated.                                     *
 *         intrvl  = interval containing xx.                          *
 *         ij      = indicates which of the two basis functions in    *
 *                     interval intrvl should be used                 *
 *         ni_indx = array containing the endpoints of each interval.  *
 *         id      - If id ~= 2, the value of the basis function      *
 *                     is calculated; if id = 2, the value of the     *
 *                     derivative of the basis function is returned.   *
 *********************************************************************/

#include "fem.h"

double bspl(double xx,int intrvl,int ij,int id)
{
  double  x1,x2,bfcn;
int i1,i2,ij1,ij2;

/*** Determine which basis function in interval intrvl is to be used in
                       calculation                                   ***/

  ij1=ij;
if(ij1 == 0) ij2 = 1;
else ij2 = 0;

/***  Determine endpoint of the interval intrvl                          ***/
       i1=ni_indx[intrvl][ij1];
       i2=ni_indx[intrvl][ij2];

/*** Determine nodal values at the endpoints of the interval intrvl      ***/
       x1=x[i1];
       x2=x[i2];

/*** Evaluate basis function                                         ***/
      if(id == 2) bfcn=1/(x1-x2);
       else bfcn=(xx-x2)/(x1-x2);

       return bfcn;
}
/***********************************************************************
 *    Function to calculate the l-2 norm error and infinity norm error   *
 *      between the finite element solution and the exact solution       *
 ***********************************************************************
 *    Arguments:                                                        *
 *          *el2  = l-2 norm error                                      *
 *          *emax = l - infinity norm error                             *
 *          n    = number of unknowns                                   *
 ***********************************************************************/

#include "fem.h"

void errors(double *el2,double *emax,int n)
{
   int i,intrvl,iquad,iendpt,ibasis,jendpt,jbasis,j;
   double xx,err,ex,basisi,drvi,wght,ej,basisj,drvj,ei,ermax;

/*** Compute the L infinity norm error between the exact solution and
        the finite element solution                                  ***/
   ermax=0.0;
   for(i=1;i<n-1;i++)
   {
      ex=exact(x[i]);
        err=fabs(ex-b[i-1]);
      if(ermax < fabs(err)) ermax=fabs(err);
   }
   *emax=ermax;
   /* printf(" *emax %g \n",*emax); */

/*** Compute the L 2 norm error between the exact solution and the
         finite element solution                                     ***/
   err=0.0;
   for(intrvl=0;intrvl<n;intrvl++)
   {
    for(iquad=0;iquad<3;iquad++)
    {
            wght=dlen[intrvl]*qdwt[iquad];
            xx=xquadpt[intrvl][iquad];
            for(ibasis=0;ibasis<2;ibasis++)
          {
                iendpt=ni_indx[intrvl][ibasis];
                basisi=bspl(xx,intrvl,ibasis,1);
                drvi=bspl(xx,intrvl,ibasis,2);
                i=indx[iendpt];
              if(i>=0)
          {
                    ei=(b[i]-exact(x[iendpt]))*basisi;
                    for(jbasis=0;jbasis<2;jbasis++)
                {
                     jendpt=ni_indx[intrvl][jbasis];
                     basisj=bspl(xx,intrvl,jbasis,1);
                       drvj=bspl(xx,intrvl,jbasis,2);
                      j=indx[jendpt];
                       if(j>=0)
                 {
                        ej=(b[j]-exact(x[jendpt]))*basisj;
                        err=err+ei*ej*wght;
                   }
                  }
              }
             }
         }
   }
   *el2=sqrt(err);
   /*   printf(" *el2 %g \n",*el2);  */

   return;
}
/*************************************************************************
 *     Function to return the value of the exact solution of the         *
 *       differential equation - used to set boundary conditions and     *
 *       for error analysis                                              *
 *************************************************************************
 *     arguments:                                                        *
 *                 x = the value at which the function is to be          *
 *                       evaluated                                       *
 *************************************************************************/
double exact(double x)
{
      double exact;

         exact= 1 - x*x;

      return exact;
}
/*******************************************************************
 *   Function to compute the value of the RHS of the              *
 *          original differential equation                      *
 *******************************************************************/

double f(double z)
{
double f;

          f=2+z*(1+z*(6-z));

return f;
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                     driver for program fem                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                     %%
%%       Variables:                                                    %%
%%                   nruns = number of program runs                    %%
%%                   xpts  = xpoints or nodes for fem                  %%
%%                   n = number of nodes - 1                           %%
%%                   el2 = l-2 norm error                              %%
%%                   emax = infinity norm error                        %%
%%                   u = vector containing the n+1 basis coefficients - %%
%%                       a.k.a. the fem solution                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                     %%
%%      Calls the functions assem and bookkeep to generate             %%
%%      the linear system to be solved using PETSc KSP methods.        %%
%%      Prints the output to a file named output.txt              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

#define _FIX
#include "fem.h"
#include <math.h>
#include <time.h>
#include "petscksp.h"

static char help[]="Solves a tri-diagonal system using KSP.\n";

double b[100],x[100],dlen[100],qdwt[3],xquadpt[100][5],atri[100][3],
       xpts[100],xleft,xright,dintrvl,add_term, basisi,drvi,basisj,
       drvj,aij,wght,test,testprime,xx;
int ni_indx[100][2],indx[100],i,j,ileft,iright,intrvl,jj,jbasis,
    jendpt,iquad,ibasis;

int main(int argc, char **args)
{
   double emax,el2;
   int i,in,nruns,n;

   FILE *fout;

   PetscErrorCode ierr;
   PetscViewer viewer;
   PetscScalar *get;

   PETSC_STRUCT sys;


   /*Open the output file for writing*/
   fout = fopen("output.txt","w");

   /* Initialize PETSc */
   Petsc_Init(argc, args,help);

   fprintf(fout,"%s",help);
   fprintf(fout," \n");
   fprintf(fout,"     finite element solution    \n");
   fprintf(fout," \n");
   fprintf(fout,"n    l-2 error   max error \n");

   nruns = 3;

/*    loop to run three trials, each trial has different # of nodes  */
      for(in=1;in<=nruns;in++)
        {
          /*  set number of nodes - 1  */
       n=(int)(pow(2.,in+2.));

        PetscInt  nn = n-1;

             /*  set evenly spaced nodes */
              for(i=0;i<=n;i++)
                {
               xpts[i]=(double)i/(double)n;
                /*  printf(" %d %g\n",i,xpts[i]); */
                }


         /*Create the vectors and matrix we need for PETSc*/

      Vec_Create(&sys,nn);
      Mat_Create(&sys,nn,nn);


       /*  generate finite element solution */

     /*  first do bookkeeping  */
       bookkeep(n);

     /*  then do assembly  */
      assem(&sys,n);

    /* Call function to do final assembly of PETSc matrix and vectors*/
        Petsc_Assem(&sys);

    /*  Call function to solve the tri-diagonal system*/
      Petsc_Solve(&sys);

    /*  Call function to view the system - outputs a matlab file called results.m*/
      Petsc_View(sys,viewer);

    /*  Call function to get a pointer to the Petsc solution vector*/
      ierr = VecGetArray(sys.sol,&get);

      for(i=0;i<n-1;i++)
          {
            b[i] = get[i];  /* store result in b for error calculation */
          }

    /* Call function to compute the L-2 and L-infinity norm error */
         errors(&el2,&emax,n);

    /* Free all PETSc objects so that we can finalize PETSc     */
      ierr = VecRestoreArray(sys.sol,&get);

    /*Free all PETSc objects created for solve  */
      Petsc_Destroy(&sys);

     /*  print error */
     fprintf(fout,"%d  %g  %g  \n",n,el2,emax);
 }

   /*finalize PETSc*/
   Petsc_End();

   /*close output file*/
   fclose(fout);

   return 0;
}
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PETSc behind the scenes maintenance functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
```

```
#include "mycalls.h"

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Function to initialize Petsc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

void Petsc_Init(int argc, char **args,char *help)
{
PetscErrorCode ierr;
PetscInt n;
PetscMPIInt size;
PetscInitialize(&argc,&args,(char *)0,help);
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);

return;
}


/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function to finalize Petsc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void Petsc_End()
{

PetscErrorCode  ierr;

  ierr = PetscFinalize();

return;

}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function to create the rhs and soln vectors
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void  Vec_Create(PETSC_STRUCT *obj, PetscInt m)
{

PetscErrorCode ierr;

  ierr = VecCreate(PETSC_COMM_WORLD, &obj->rhs);
  ierr = VecSetSizes(obj->rhs, PETSC_DECIDE, m);
  ierr = VecSetFromOptions(obj->rhs);
  ierr = VecDuplicate(obj->rhs, &obj->sol);

return;
}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Function to create the system matrix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void Mat_Create(PETSC_STRUCT *obj, PetscInt m, PetscInt n)
{

PetscErrorCode ierr;

  ierr = MatCreate(PETSC_COMM_WORLD, &obj->Amat);
  ierr = MatSetSizes(obj->Amat,PETSC_DECIDE,PETSC_DECIDE,m,n);
  ierr = MatSetFromOptions(obj->Amat);

return;

}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Function to solve a linear system using KSP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void  Petsc_Solve(PETSC_STRUCT *obj)
{

PetscErrorCode ierr;

ierr = KSPCreate(PETSC_COMM_WORLD,&obj->ksp);
  ierr = KSPSetOperators(obj->ksp,obj->Amat,obj->Amat,DIFFERENT_NONZERO_PATTERN);
  ierr = KSPGetPC(obj->ksp,&obj->pc);
  ierr = PCSetType(obj->pc,PCNONE);
  ierr = KSPSetTolerances(obj->ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);

  ierr = KSPSetFromOptions(obj->ksp);

  ierr = KSPSolve(obj->ksp,obj->rhs,obj->sol);

  ierr = VecAssemblyBegin(obj->sol);
  ierr = VecAssemblyEnd(obj->sol);

return;
}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function to do final assembly of matrix and right hand side vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void  Petsc_Assem(PETSC_STRUCT *obj)
{

PetscErrorCode   ierr;

ierr = MatAssemblyBegin(obj->Amat, MAT_FINAL_ASSEMBLY);
ierr = MatAssemblyEnd(obj->Amat, MAT_FINAL_ASSEMBLY);
ierr = VecAssemblyBegin(obj->rhs);
ierr = VecAssemblyEnd(obj->rhs);

return;

}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function to Destroy the matrix and vectors that have been created
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void  Petsc_Destroy(PETSC_STRUCT *obj)
{
PetscErrorCode   ierr;

ierr = VecDestroy(obj->rhs);
ierr = VecDestroy(obj->sol);
ierr = MatDestroy(obj->Amat);
ierr = KSPDestroy(obj->ksp);

return;

}


/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function to View the matrix and vectors that have been created in an m-file
%% Note:  Assumes all final assemblies of matrices and vectors have been performed
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void  Petsc_View(PETSC_STRUCT obj, PetscViewer viewer)
{
PetscErrorCode   ierr;

ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD, "results.m", &viewer);
ierr = PetscViewerPushFormat(viewer,PETSC_VIEWER_ASCII_MATLAB);
ierr = PetscObjectSetName((PetscObject)obj.Amat,"Amat");
ierr = PetscObjectSetName((PetscObject)obj.rhs,"rhs");
ierr = PetscObjectSetName((PetscObject)obj.sol,"sol");
ierr = MatView(obj.Amat,viewer);
ierr = VecView(obj.rhs, viewer);
ierr = VecView(obj.sol, viewer);

return;

}

/*************************************************************************
 * Function to compute the coefficient values of the first order term
 *  of the differential equation
 *************************************************************************/

double p(double z)
{
double p;

  p=1+z*z;

return p;
}

/**********************************************************************
 *Function to compute the coefficient values of the zero order term
 *    in the original differential equation
 **********************************************************************/

double q(double z)
{
return z;
}

/*** Header file                                                  ***/

/*** Include standard libraries                                   ***/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mycalls.h"
```

```
/*** Global variables                                           ***
 *        b = left side of the tri-diagonal system Ax=b           *
 *        x = xpoints or nodal values                             *
 *        dlen = length of the intervals between adjacent nodes   *
 *        qdwt = quadrature weights                               *
 *        xquadpt = points used in Simpson's rule for numerical   *
 *              integration of each interval                      *
 *        atri = array to hold the tri-diagonal A matrix in the   *
 *                system Ax = b                                   *
 *        ni_indx = array to hold the location of the endpoints of *
 *                each interval                                   *
 *        indx = array to hold the offset between interval number  *
 *                and node number                                 *
 ******************************************************************/


/*** Function Prototypes                                        ***/
void  bookkeep();
void  assem(PETSC_STRUCT *obj, int n);
void  errors(double *el2,double *emax,int n);
double exact(double x);
double f(double z);
double p(double z);
double q(double z);
double bspl(double xx,int intrvl,int ij,int id);

#ifndef _FIX

extern double b[100],x[100],dlen[100],qdwt[3],xquadpt[100][5],atri[100][3],
    xpts[100],xleft,xright,dintrvl,add_term,
basisi,drvi,basisj,drvj,aij,wght,test,testprime,xx;
extern int ni_indx[100][2],indx[100],i,j,ileft,iright,intrvl,jj,
    jbasis,jendpt,iquad,ibasis;

#endif
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Header file for Petsc Functions to execute all Petsc maintenance
%% behind the scenes.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

#define _PETSC
#include "share.h"

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%   Function prototypes  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

void  Petsc_Init(int argc,char **args,char *help);
void  Petsc_End();
void  Petsc_Solve(PETSC_STRUCT *obj);
void  Vec_Create(PETSC_STRUCT *obj, PetscInt m);
void  Mat_Create(PETSC_STRUCT *obj, PetscInt m, PetscInt n);
void  Petsc_Assem(PETSC_STRUCT *obj);
void  Petsc_Destroy(PETSC_STRUCT *obj);
void  Petsc_View(PETSC_STRUCT obj, PetscViewer viewer);
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% header file for PETSc struct
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This is the header file defining a PETSc struct for use in adding PETSc
%% to existing codes.  The struct containing the linear system matrix, the
%% right hand side and solution vectors, the linear solver context, and the
%% preconditioner context associated with a tridiagonal system.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

#ifdef _PETSC
#include "petscmat.h"
#include "petscksp.h"
typedef Mat PETSC_MAT;
typedef Vec PETSC_VEC;

typedef struct
{
    PETSC_MAT     Amat; /*linear system matrix*/
    KSP         ksp; /*linear solver context*/
    PC         pc; /*preconditioner context*/
    PETSC_VEC rhs; /*petsc rhs vector*/
    PETSC_VEC sol; /*petsc solution vector*/

}PETSC_STRUCT;

#endif
```

A.4 1D Parallel PETSc Solve

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                    driver for program fem
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%       Variables:
%%                  nruns = number of program runs
%%                  xpts  = xpoints or nodes for fem
%%                   n = number of nodes - 1
%%                   el2 = l-2 norm error
%%                   emax = infinity norm error
%%                   u = vector containing the n+1 basis coefficients -
%%                       a.k.a. the fem solution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%      Calls the functions assem and bookkeep to generate
%%      the linear system to be solved using PETSc KSP methods.
%%      Prints the output to a file named output.txt
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

#define _FIX
#include "fem.h"
#include <math.h>
#include <time.h>
#include "petscksp.h"

static char help[]="Solves a tri-diagonal system using KSP.\n";

double bbb[100],bb[100],b[100],x[100],dlen[100],qdwt[3],xquadpt[100][5],atri[100][3],
       xpts[100],xleft,xright,dintrvl,add_term,basisi,drvi,basisj,drvj,aij,
       wght,test,testprime,xx;
int ni_indx[100][2],indx[100],i,j,ileft,iright,intrvl,jj,jbasis,jendpt,iquad,ibasis;

int main(int argc, char **args)
{
   double emax,el2;
   int i,in,nruns,n;

   FILE *fout;

   PetscErrorCode ierr;
   PetscViewer viewer;
   PetscScalar *get;
   PetscScalar    *sendbuf,*recvbuf;
   PetscMPIInt    rank,size;

   PETSC_STRUCT sys;


   /*Open the output file for writing*/
   fout = fopen("output.txt","w");

   /* Initialize PETSc */
   Petsc_Init(argc, args,help);
   MPI_Comm_size(PETSC_COMM_WORLD,&size);
   MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
   PetscSynchronizedPrintf(PETSC_COMM_WORLD,"Number of processors = %d, rank = %d\n",size,rank);
   PetscSynchronizedFlush(PETSC_COMM_WORLD);
   fprintf(fout,"%s",help);
   fprintf(fout," \n");
   fprintf(fout,"     finite element solution    \n");
   fprintf(fout," \n");
   fprintf(fout,"rank\t n\t l-2 error\t max error \n");

      nruns = 4;

/*    loop to run three trials, each trial has different # of nodes  */
      for(in=1;in<=nruns;in++)
        {
             /*  set number of nodes - 1  */

             n=(int)(pow(2.,in+2.));

                PetscInt  nn = n-1;


             /*  set evenly spaced nodes */
              for(i=0;i<=n;i++)
                {
              xpts[i]=(double)i/(double)n;
               /*  printf(" %d %g\n",i,xpts[i]); */
                }


      /*Create the vectors and matrix we need for PETSc*/

       Vec_Create(&sys,nn);
       Mat_Create(&sys,nn,nn);


       /*  generate finite element solution */

     /*  first do bookkeeping  */
        bookkeep(n);

     /*  then do assembly  */
        assem(&sys,n);

    /* Call function to do final assembly of PETSc matrix and vectors*/
        Petsc_Assem(&sys);
    /*  Call function to solve the tri-diagonal system*/
        Petsc_Solve(&sys);

    /*  Call function to view the system - outputs a matlab file called results.m*/
        Petsc_View(sys,viewer);

    /*  Call function to get a pointer to the Petsc solution vector*/
        int sendcount,recvcount;
    ierr = VecGetArray(sys.sol,&get);

for(i=0;i<n-1;i++)
                b[i] = get[i];  // store result in b for error calculation

PetscInt low,high,otherlow,otherhigh;
                MPI_Status status;
                PetscInt count;
                int tag = 9999;
                VecGetOwnershipRange(sys.sol,&low,&high);
                VecGetLocalSize(sys.sol,&count);
                // first set correct order on local node according to low&high
                for(j=0;j<count;j++)
                   bbb[low+j] = b[j];
                for(i=0;i<size;i++)
                {
                   if(i != rank) {
                      MPI_Send( &count, 1, MPI_INT, i,tag, MPI_COMM_WORLD );
                      MPI_Send( &low, 1, MPI_INT, i,tag, MPI_COMM_WORLD );
                      MPI_Send( &high, 1, MPI_INT, i,tag, MPI_COMM_WORLD );
                      MPI_Send( b, count, MPI_DOUBLE, i,tag, MPI_COMM_WORLD );
                   }
                }
                // each processor will receive #size-1 message
                for(i=0;i<size;i++)
                {
                   if(i!=rank) {
                      MPI_Recv (&recvcount,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
                      MPI_Recv (&otherlow,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
                      MPI_Recv (&otherhigh,1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
                      MPI_Recv (&bb[0],recvcount,MPI_DOUBLE,i,tag,MPI_COMM_WORLD,&status);
                      for(j=0;j<recvcount;j++) {
                         bbb[otherlow+j] = bb[j];
                      }
                      //PetscSynchronizedFlush(PETSC_COMM_WORLD);
                  }

                }
                //synchronization
                MPI_Barrier(MPI_COMM_WORLD);


      for(i=0;i<n-1;i++)
         {
                b[i] = bbb[i];  // store result in b for error calculation
                //PetscSynchronizedPrintf(MPI_COMM_WORLD,"bbb[%d] = %12.4e\n",i,bbb[i]);
                //PetscSynchronizedFlush(PETSC_COMM_WORLD);
         }

    /* Call function to compute the L-2 and L-infinity norm error */
      errors(&el2,&emax,n);


    /* Free all PETSc objects so that we can finalize PETSc     */
      ierr = VecRestoreArray(sys.sol,&get);

    /*Free all PETSc objects created for solve  */
      Petsc_Destroy(&sys);



  PetscSynchronizedPrintf(MPI_COMM_WORLD,"Processor[%d] %d  %g  %g  \n",rank,n,el2,emax);
      PetscSynchronizedFlush(PETSC_COMM_WORLD);
             /*  print error */
fprintf(fout,"[%d]\t%d\t%g\t%g  \n",rank,n,el2,emax);
}

   /*finalize PETSc*/
   Petsc_End();

   /*close output file*/
   fclose(fout);

   return 0;
}
```

A.5 1D results.m file

```
% Size = 7 7
% Nonzeros = 19
```

```
zzz = zeros(19,3);
zzz = [
1 1  1.6343750000000000e+01
1 2  -8.2877604166666661e+00
2 1  -8.2877604166666661e+00
2 2  1.7104166666666664e+01
2 3  -8.7851562500000000e+00
3 2  -8.7851562500000000e+00
3 3  1.8364583333333336e+01
3 4  -9.5325520833333321e+00
4 3  -9.5325520833333321e+00
4 4  2.0125000000000000e+01
4 5  -1.0529947916666666e+01
5 4  -1.0529947916666666e+01
5 5  2.2385416666666664e+01
5 6  -1.1777343750000000e+01
6 5  -1.1777343750000000e+01
6 6  2.5145833333333332e+01
6 7  -1.3274739583333330e+01
7 6  -1.3274739583333330e+01
7 7  2.8406249999999996e+01
];
Amat = spconvert(zzz);
rhs = [
8.3192952473958339e+00
3.2788085937500000e-01
3.9733886718749994e-01
4.8583984374999994e-01
5.9191894531250000e-01
7.1411132812500000e-01
8.5095214843750000e-01
];
sol = [
9.8465653196096148e-01
9.3797172650023541e-01
8.5994446026019922e-01
7.5058122917808645e-01
6.0989304867213834e-01
4.3789244249952697e-01
2.3459134127625003e-01
];
```

B Queue Files

B.1 Job Script Example

```
#!/bin/bash
############################################################
## Team HPC
## PBS Job Script
## (c) 2001-2002 Team HPC, All rights reserved
############################################################

# $Id: cpi.gcc.eth.pbs,v 1.2 2005/03/31 20:42:01 dinkel Exp $

### Set the job name
#PBS -N petsc_1d

### Set the queue to submit this job.
#PBS -q default

### Set the number of nodes that will be used.
#PBS -l nodes=2:ppn=2

#PBS -k oe

# Ensure to parse the script to set the env vars for the compiler you used.
. ~/bin/switch_gcc_eth.sh

export NPROCS=‘wc -l $PBS_NODEFILE |gawk ’//{print $1}’‘

echo The master node of this job is ‘hostname‘
echo The working directory is ‘echo $PBS_O_WORKDIR‘
echo The node file is $PBS_NODEFILE
echo "=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
echo This job runs on the following nodes:
echo ‘cat $PBS_NODEFILE‘
echo "=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
echo This job has allocated $NPROCS nodes

echo "=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"
echo
echo command to EXE:
echo
echo
```

```
/usr/local/mpich2/bin/mpdboot -n 15 -r "ssh" -f "/home/jsalyer/mympd.hosts"
/usr/local/mpich2/bin/mpiexec -n 2 ./driver
/usr/local/mpich2/bin/mpdallexit

sleep 5
```

B.2 Hosts File

```
############################################################
# Team HPC
# machines.LINUX file
# $Id: machines.LINUX,v 1.1 2004/08/17 14:55:40 dinkel Exp $
############################################################
# WARNING: This file is automatically edited by Cluster-H.Q.
# Anything below the << CLUSTER-H.Q. DMZ >> comment will be
# removed and regenerated each time the node database is
# updated.
# If you wish to edit this file by hand, be sure that your
# changes are listed ABOVE the --<< CLUSTER-H.Q. DMZ >>--
# comment, or else they will be lost.
############################################################

# --<< CLUSTER-H.Q. DMZ >>--
# Last generated on Wed, 12 Oct 2005 11:26:09 -0400
node001:2
node003:2
node004:2
node005:2
node006:2
node007:2
node008:2
node009:2
node010:2
node011:2
node012:2
node013:2
node014:2
node015:2
node016:2
```

B.3 Queue output file for 1D Parallel Run

```
The master node of this job is node016
The working directory is /home/jsalyer
The node file is /var/spool/PBS/aux/1318.head
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
This job runs on the following nodes:
node016 node016 node015 node015
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
This job has allocated 4 nodes
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

command to EXE:

Number of processors = 2, rank = 0
Number of processors = 2, rank = 1
Processor[0] 8  0.000417727  0.000581229
Processor[1] 8  0.000417727  0.000581229
Processor[0] 16  0.000105368  0.000146215
Processor[1] 16  0.000105368  0.000146215
Processor[0] 32  2.62932e-05  3.6437e-05
Processor[1] 32  2.62932e-05  3.6437e-05
Processor[0] 64  3.43242e-06  4.60698e-06
Processor[1] 64  3.43242e-06  4.60698e-06
```