TECHNICAL REPORT

Library Reading Room, the British Museum

The PETSc Library :: Portable, Extensible Toolkit for Scientific Computing

Dr. Michael Browne
ICHEC Computational Scientist

PETSc

Introduction	1
Important features	1
Hello world	3
Vectors	3
PETSc on ICHEC	4
Executing PETSc programs	5
Sample code	7

Introduction

The PETSc (pronounced PET-see) library is designed to work in both parallel and sequential codes, it is object orientated in design and is available for Unix and windows. It consists of both data structures and functions that are intended for building scientific applications. Fortran, C/C++ and Python are supported.

While PETSc is built on MPI it provides an abstraction layer above MPI allowing users to write parallel code using high-level routines with little concern for low level MPI operations. This short guide provides a quick introduction to some of PETSc's capabilities.

Important features

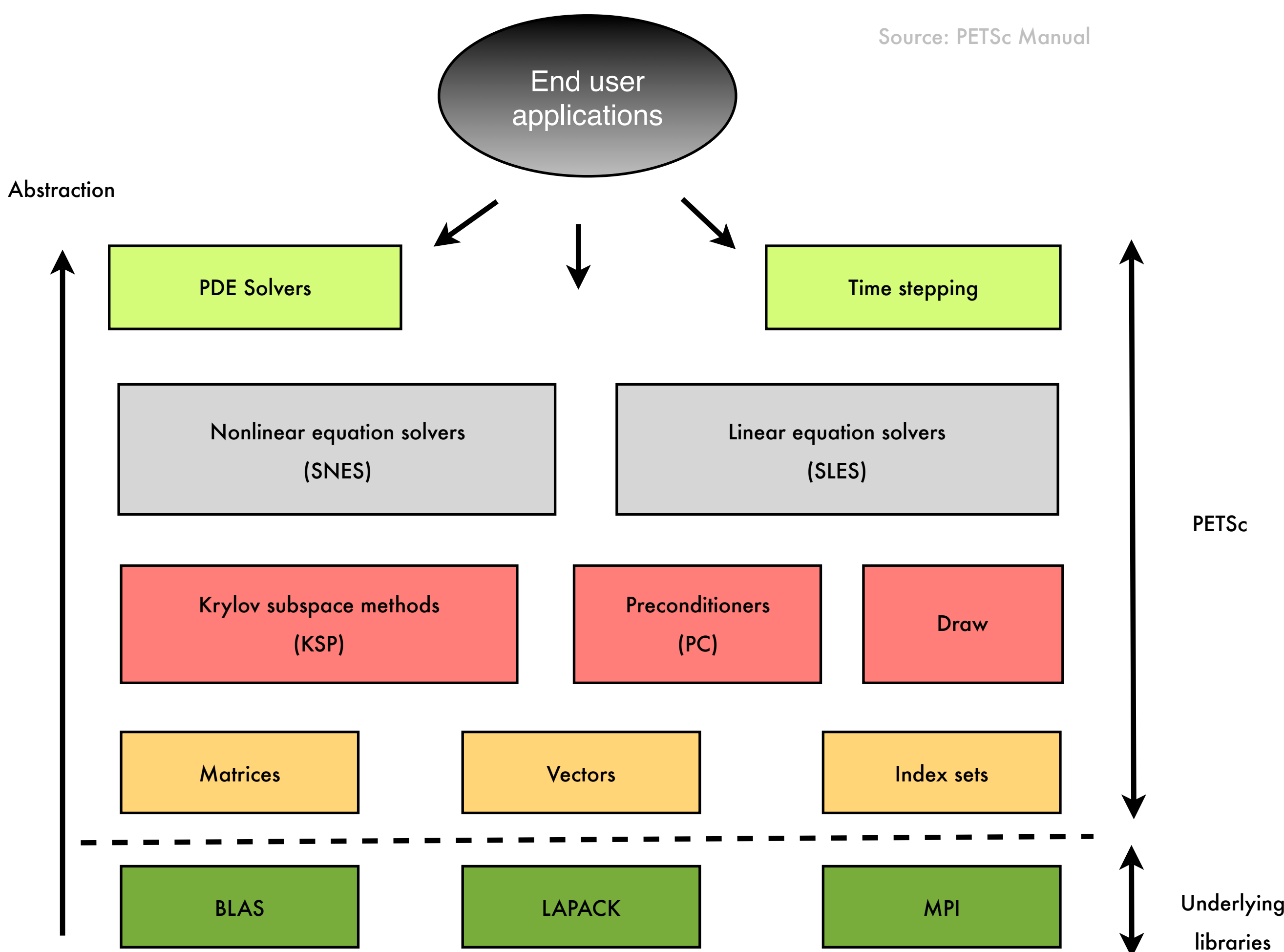
PETSc has built in profiling for both memory usage and floating point calculation this accompanied with progress reporting features common in many of the solvers mean that one can get a reasonable picture of a codes execution profile merely by supplying a few additional command line arguments.

One of the strengths of the PETSc library is the large number of code examples it ships with. These examples are cross-referenced throughout the documentation so one can readily see an example of most significant functions along with related functions.

PETSc's key feature is the number of highly regarded solvers etc. it brings under one roof: parallel timestepping ODE solvers, parallel preconditioners, Krylov subspace methods. parallel Newton-based nonlinear solvers and interfaces to numerous other 3rd party packages.

For full details on PETSc visit:
http://www-unix.mcs.anl.gov/petsc/petsc-2/

The PETSc manual is well put together and highly recommended it can be download from:
http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manual.pdf



The above diagram aims to give an impression of how PETSc is structured. At the lowest level are a number of proven and optimised libraries common to many scientific computing projects. These are largely interchangeable so one can build PETSc against a particular set of low-level libraries, for example vendor optimised BLAS libraries. One can still make calls to these libraries from within an application indeed it is often necessary to be able to make some MPI calls.

Above these PETSc implements a stack of so called 'modules' providing high-level solvers. Modules are accessed via abstract interfaces consisting of a set of calling sequences. This high level approach means that it is possible to build generic code which allows one to change the algorithms being used, often at runtime using command line flags. This encourages experimentation such as changing preconditioners or Krylov methods.

The solvers rely on the underlying data structures and may support more than one type, i.e. if a solver is not parallel then it will not support distributed matrices. PETSc is also designed to facilitate adding custom solvers.

PETSc Hello World

```
#include "petsc.h"

int main( int argc, char *argv[] ) {
      PetscInitialize( &argc, &argv, 0, 0 );
      PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
      PetscFinalize();
      return 0;
}
```

Hello World

PETSc programs generally begin with a call to PetscInitialize() it allows PETSc to initialise the underlying MPI libraries if required. Similarly PetscFinalize() is called at the end of a PETSc program and as you might expect it calls MPI_Finalize().

PetscPrintf() operates on a communicator in a very similar fashion to many MPI functions. In this case PETSC_COMM_WORLD which by default is set to MPI_COMM_WORLD. The first process in the communicator only prints the message, "Hello World".

Vectors

PETSc implements a number of data types e.g. PetscInt, PetscReal and PetscScalar.

PetscInt and PetscReal as you would expect are merely aliases for the conventional Int and Double datatypes (in C/C++ parlance), however fully embracing the PETSc versions when using the library is safer.

PetscScalar is interesting as it can be a real number or a complex number, the actual type being determined at library compile time. It means that with a little care it is possible to build and test code with one type of scalar and convert to the other by merely recompiling and linking against an alternate PETSc build.

For data types beyond the simple PetscScalar datatype which can simply be declared, create() and destroy() functions must be used. The principal data structures available to PETSc users are vectors and matrices.

Vectors can be parallel or sequential. To create a vector one first declares a vector 'object', myvec.

```
Vec myvec;
```

One then creates the vector, in this case whether the vector is parallel or distributed is decided at runtime by checking for the -vec_type mpi command line argument with the call to VecSetFromOptions(). Otherwise the vector will default

```
VecCreate(PETSC_COMM_WORLD,&myvec);
VecSetSizes(myvec,PETSC_DECIDE,n);
VecSetFromOptions(myvec);
```

to sequential. The size of the vector is set by VecSetSizes() in this case the vector has n elements. The PETSC_DECIDE argument means that should the vector be set to parallel the distribution of elements across processes is left to PETSc. VecCreate() takes two arguments, as you might expect a pointer to the vector, but also a PETSc communicator in this case PETSC_COMM_WORLD. This determines across which processes the creation will take place. The creation is a collective operation and so must happen in all processes in the communicator. There are many calls which result in the creation of a vector. These can be more specific and compact than the above example where one does not desire verbosity or flexibility; VecCreate(), VecCreateSeq(), VecCreateMPI, VecDuplicate(), VecDuplicateVecs(), VecCreateGhost(), VecCreateMPIWithArray(), VecCreateGhostWithArray().

To set values in the vector one generally calls VecSetValues() specifying the vector, the number of elements to set, the values to use and one of several possible insertion modes. Note if the program is parallel it is important to remember to take steps to avoid multiple processes setting the same vector elements at the same time. The vector is then assembled and ready for use.

```
VecSetValues(myvec,
    1,&row,&value,INSERT_VALUES);
VecAssemblyBegin(x);
VecAssemblyEnd(x);
```

Direct Access

While the high-level global memory addressing model provided by PETSc can be very convenient there are times, particularly when writing HPC code, where for performance reasons it is important to ensure that you are only working with locally stored elements of a distributed structure. For instance if one wants direct access to the local elements of a vector a number of functions exist to facilitate this:

VecGetArray() returns a pointer to the array containing the locally stores values in the vector. Normally, this will be a pointer to the storage that PETSc is using. Following a call to this function local modifications can take place. A call to VecRestoreArray() must then be made before the local contents can be used be other PETSc processes. In effect the call "gives back" the local values to the distributed array. Along with these function VecGetLocalSize() can be useful in determining the number of elements in the locally stored part of the vector. One can also use VecGetOwnershipRange() to return the lower and upper bounds of the portion of a parallel vector which is stored locally. Generally it is good practice to leave the decision about data distribution to PETSc itself. As you might expect similar functions exist for handling PETSc matrices.

Distributed Arrays & Index Sets

Distributed Array (DA) objects are built on top of PETSc vectors. They can be used when working with regular grids, where one processes needs neighbouring data elements that are held by another process. This type of neighbouring data is often called 'ghost' or 'halo' data. Distributed arrays are not designed for storing conventional matrices or for working with unstructured data.

A DA does not itself store data rather it is a mechanism for managing and communicating vectors which store data. There are local and global vectors and a mechanisms for propagating updates from one view to the other as well as indices mapping functions.

Index Sets (IS) provide a more general way of specifying indices. An IS is simply a set of integer indices. They can be useful when working with unstructured data, allowing one to scatter to and from arbitrarily defined ghost points.

PETSc does NOT do:

- Load balancing
- Mesh generation
- Advanced visualisation
- Generic message passing
- Thread safety

Some PETSc Matrix Types [:::]	
mpiaij	Distributed sparse
seqaij	Sequential sparse
mpidense	Distributed dense
mpibaij	Distributed block sparse
seqbaij	Local symmetric block sparse
mpibdiag	Distributed block diagonal
superlu	sequential direct solver matrix for external SuperLU package
aijmumps	sequential or distributed direct solver matrix for external mumps package
See ${PETSC_DIR}/include/petscmat.h for a complete listing	

Executing PETSc Programs

Before running a PETSc based program one must set the PETSC_DIR environment variable to the path of the PETSc directory. Within this directory there is a lib directory which will have at least one subdirectory corresponding to a set of PETSc libraries built with a given configuration.

A second environment variable PETSC_ARCH is used to specify which library build within the PETSC_DIR to use. This allows you to prepare a variety of PETSc builds e.g. optimised, debug differing MPI libraries etc. and create and run the corresponding executables while only changing the PETSC_ARCH variable.

As PETSc uses MPI a PETSc program can be started using mpiexec as you would another MPI program.

PETSc has extensive command line argument processing (see: PetscOptionsGetXYZ() and XYZSetFromOptions() functions) which can used within programs to pass values and flags but also to alter more fundamental aspects such as matrix types and solvers. When debugging the following command line arguments can be particularly helpful:

For debugging and profiling features to be available PETSc must be built with the following configure flag set:

–with-debugging=1

One should not make assumptions about the order in which output from printf()s and PetscPrintf()s will appear in based on their respective positions with the code because of the uncertainty involved in buffer flushes particularly in a parallel environment.

As PETSc is highly configurable it is hard to produce a 'one size fits all' build or even set of builds and so ICHEC does not currently provide PETSc as a prebuilt package, rather we provide configuration files which you can use as the basis for your own build, these can be found here:
http://www.ichec.ie/infrastructure/software/PETSc
If you wish to use PETSc on our systems please feel free to contact us via the helpdesk and we will assist with custom builds where necessary.

However in the case of the Blue Gene/L and Blue Gene/P PETSc builds produced by IBM are available. They are placed in the IBM Packages directory to use them set PETSC_DIR variable as follows:
export PETSC_DIR=/ichec/work/ibm-packages/petsc-2.3.3-p13

There are three PETSc builds:

- bgl-ibm-debug is compiled with Scalasca instrumentation it can run on the /L backend and will generate trace files which can be used by Scalasca to examine performance. To use it, set the PETSC_ARCH variable as follows: export PETSC_ARCH=bgl-ibm-debug at compile time.
- bgl-ibm-opt is compiled with IBM XL compilers for the /L, to use it: export PETSC_ARCH=bgl-ibm-opt
- bgp-ibm-opt is compiled with IBM XL compilers for the /P. To use it: export PETSC_ARCH=bgp-ibm-opt

At compile time on the Blue Gene/P some messages may be displayed relating to shared libraries, these can be ignored.

Debugging Command Line Arguments	
-fp_trap	Halt execution on a floating point exception
-malloc_dump	Produce a list of unfreed memory when a program ends
-malloc_debug	Enable memory tracing
-start_in_debugger	Start all processes in a debugger
-on_error_attach_debugger	Start debugger only when there is an error
-log_summary	Produce a summary execution profile
-info	Output verbose information from PETSc functions
-log_trace	Output a function call trace
The PetscError() function is a good place to set a breakpoint.	

Threading

PETSc itself is not thread safe so one must be cautious if using it in a multithreaded code whether the threading is implemented using OpenMP, pthreads or some other mechanism. A simple rule of thumb is to use just one thread in a giving process to handle all interactions with PETSc. However this is very restrictive and a better solution maybe to eschew threads in favour of a multiprocess approach even if operating on multi-processor hardware.

SLEPc

SLEPc, the Scalable Library for Eigenvalue Problem Computations, is a popular library for solving large, sparse eigenvalue problems in parallel. It can deal with Hermitian and non-Hermitian forms using real or complex arithmetic. It also features a number functions for solving singular value decomposition problems.

A number of eigensolvers are available:

- Krylov-Schur
- Lanczos
- Arnoldi
- Subspace Iteration
- Power/RQI.

Spectral transformations such as shift-and-invert or spectrum folding are also available. SLEPC is designed to be easily understood by a PETSc user. It too is well documented, supports C/C++ and Fortran and uses PETSc's object-oriented style.

If you find regular PETSc does not provide all of the linear algebra functions you require you may find that SLEPc can help: http://www.grycap.upv.es/slepc

Sample Code

The best way to explain how PETSc works in practice is to examine some well commented sample code. The following pages reproduce an example code directly from the PETSc documentation (${PETSC_DIR}/src/ksp/ksp/examples/tutorials/ex1.c). The code solves a tridiagonal linear system using the KSP object discussed earlier. As mentioned there are many more examples within the documentation.

The PETSc project operates a number of very responsive mailing lists, the archives of which are a valuable resource they are available here: http://www-unix.mcs.anl.gov/petsc/petsc-2/miscellaneous/mailing-lists.html

Source: ${PETSC_DIR}/src/ksp/ksp/examples/tutorials/ex1.c

```
/* Program usage: mpiexec ex1 [-help] [all PETSc options] */

static char help[] = "Solves a tridiagonal linear system with KSP.\n\n";

/*T
    Concepts: KSP^solving a system of linear equations
    Processors: 1
T*/

/*
    Include "petscksp.h" so that we can use KSP solvers. Note that this file
    automatically includes:
    petsc.h - base PETSc routines petscvec.h - vectors
    petscsys.h - system routines petscmat.h - matrices
    petscis.h - index sets petscksp.h - Krylov subspace methods
    petscviewer.h - viewers petscpc.h - preconditioners
    Note: The corresponding parallel example is ex23.c
*/

#include "petscksp.h"
#undef __FUNCT__
#define __FUNCT__ "main"

int main(int argc,char **args)
{
    Vec x, b, u; /* approx solution, RHS, exact solution */
    Mat A; /* linear system matrix */
    KSP ksp; /* linear solver context */
    PC pc; /* preconditioner context */
    PetscReal norm; /* norm of solution error */
    PetscErrorCode ierr;
    PetscInt i,n = 10,col[3],its;
    PetscMPIInt size;
    PetscScalar neg_one = -1.0,one = 1.0,value[3];

    PetscInitialize(&argc,&args,(char *)0,help);
    ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
    if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
    ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Compute the matrix and right-hand-side vector that define
    the linear system, Ax = b.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

    /*
    Create vectors. Note that we form 1 vector from scratch and
```

```
  then duplicate as needed.
  */
  ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
  ierr = PetscObjectSetName((PetscObject) x, "Solution");CHKERRQ(ierr);
  ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
  ierr = VecSetFromOptions(x);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

  /*
  Create matrix. When using MatCreate(), the matrix format can
  be specified at runtime.
  Performance tuning note: For problems of substantial size,
  preallocation of matrix memory is crucial for attaining good
  performance. See the matrix chapter of the users manual for details.
  */
  ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
  ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,n,n);CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);

  /*
  Assemble matrix
  */
  value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
  for (i=1; i<n-1; i++) {
      col[0] = i-1; col[1] = i; col[2] = i+1;
      ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
  }
  i = n - 1; col[0] = n - 2; col[1] = n - 1;
  ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
  i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
  ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  /*
  Set exact solution; then compute right-hand-side vector.
  */
  ierr = VecSet(u,one);CHKERRQ(ierr);
  ierr = MatMult(A,u,b);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Create the linear solver and set various options
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  /*
  Create linear solver context
  */
  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
```

```
/*
Set operators. Here the matrix that defines the linear system
also serves as the preconditioning matrix.
*/
ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);

/*
Set linear solver defaults for this problem (optional).
- By extracting the KSP and PC contexts from the KSP context,
we can then directly call any KSP and PC routines to set
various options.
- The following four statements are optional; all of these
parameters could alternatively be specified at runtime via
KSPSetFromOptions();
*/
ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);

/*
Set runtime options, e.g.,
-ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
These options will override those specified above as long as
KSPSetFromOptions() is called _after_ any other customization
routines.
*/
ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Solve the linear system
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
Solve linear system
*/
ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

/*
View solver info; we could instead use the option -ksp_view to
print this info to the screen at the conclusion of KSPSolve().
*/
ierr = KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Check solution and clean up
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
Check the error
*/
ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
```

```
    ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
    ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A, Iterations %D\n",
    norm,its);CHKERRQ(ierr);

    /*
    Free work space. All PETSc objects should be destroyed when they
    are no longer needed.
    */
    ierr = VecDestroy(x);CHKERRQ(ierr); ierr = VecDestroy(u);CHKERRQ(ierr);
    ierr = VecDestroy(b);CHKERRQ(ierr); ierr = MatDestroy(A);CHKERRQ(ierr);
    ierr = KSPDestroy(ksp);CHKERRQ(ierr);

    /*
    Always call PetscFinalize() before exiting a program. This routine
    - finalizes the PETSc libraries as well as MPI
    - provides summary and diagnostic information if certain runtime
    options are chosen (e.g., -log_summary).
    */
    ierr = PetscFinalize();CHKERRQ(ierr);
    return 0;
}
```