

## INTRODUCTION TO PETSc PROGRAMMING

### *What is Petsc ?*

PETSc stands for Portable & Extensible Toolkit for Scientific computation. This means that this tool is used for the purpose of scientific computation like efficient solving of matrices and grid data management in CFD etc. This is portable because programs made using Petsc can be implemented across any platform (Windows, Linux, Sun OS, etc.). Petsc functions are available in both C-language and Fortran. All that is required is to install the desired version of Petsc (C or Fortran) and start using it in your programs. This is extensible because one can add his own functions in the Petsc Library and/or make custom changes as per one's needs. Basically Petsc is a library of functions consisting of efficient tools for scientific computation which can be integrated within your source code and build applications over it.

### *How to install Petsc*

Following instructions are meant for installing the C-version of Petsc-2.1.6 in Redhat Linux.

1. Download the package 'petsc.tar.gz' from the site:

<http://www-unix.mcs.anl.gov/petsc/petsc-2/download/>

2. Download patches for the Petsc version downloaded from:

<http://www-unix.mcs.anl.gov/petsc/petsc-2/download/petsc-patches.html>

3. Check if 'blas' and 'lapack' which are common libraries for linear algebra are already installed or not. To do this type the following command:

```
locate libblas.a
locate liblapack.a
```

Both these commands should show up if individual packages are installed. If these are not installed then there will be no output, in which case you should download required libraries (BLAS and LAPACK) and install them. Installation instructions for this are given later.

4. Check if MPI (Message Passing Interface) is already installed or not. This is required for parallel implementation of Petsc programs. This step is optional and can be skipped if mpi implementation is not desired. However care should be taken while configuring Petsc to specify that mpi is not desired. Checking mpi installation:

```
type mpirun
```

An output showing the path of mpirun will tell that mpi is installed. If mpi is desired but not already installed then it can be downloaded from the site:

```
http://www-unix.mcs.anl.gov/mpi/mpich/
```

and installed. Installation instructions are given later.

5. unzip and untar this file using the command:

```
tar -xvzf 'petsc.tar.gz'
```

This will create a directory named 'petsc-2.1.6' and this is where your Petsc will be installed.

6. Go to the directory 'petsc-2.1.6'

```
cd petsc-2.1.6
```

7. Apply patches

```
patch -Np1 < petsc_patches_all-2.1.6
```

8. Run:

```
bin/petscharch -suggest
```

This command will display a list of linux architectures that is to be selected. For Redhat linux, you can safely select linux-gnu as shown below.

9. Run:

```
export PETSC_DIR=`pwd`  
export PETSC_ARCH=linux-gnu
```

This will set the environment variables to be used by Petsc installer.

10. Run automatic configuration

```
config/configure.py --with-blas=/usr/lib/libblas.a --with-  
lapack=/usr/lib/liblapack.a --with-mpi-dir=/usr/local/mpich
```

Here the directories can be changed according to the individual installation of blas, lapack and mpi. If mpi is not desired then following command should be used:

```
config/configure.py --with-mpi=0
```

11. Compile and instal petsc:

```
make BOPT=g all
```

12. Run following to test installation.

```
make BOPT=g test
```

The output of the command will tell if it could successfully run some example programs.

### *Installation of other libraries.*

**Blas:** download 'cblas.tgz' from <http://www.netlib.org/blas/> and run the following commands in order given.

```
tar -xvzf cblas.tgz
cd cblas
configure
make
```

**Lapack:** download 'lapack.tgz' from <http://www.netlib.org/lapack/> and run the following commands

```
tar -xvzf lapack.tgz
cd lapack
configure
make
```

The alternative way to install these is to download .rpm file and install through the OS(here redhat) package manager. But for this you will need the root password.

**Blas package:** Download [http://www.netlib.org/lapack/rpms/blas-3\\_0-2\\_i386.rpm](http://www.netlib.org/lapack/rpms/blas-3_0-2_i386.rpm) and install through redhat package manager or run the following command:

```
rpm -i blas-3_0-2_i386.rpm
```

**Lapack package:** download [http://www.netlib.org/lapack/rpms/lapack-3\\_0-2\\_i386.rpm](http://www.netlib.org/lapack/rpms/lapack-3_0-2_i386.rpm) and install through redhat package manager or run the following command:

```
rpm -i lapack-3_0-2_i386.rpmhttp://www-unix.mcs.anl.gov/mpi/mpich/
```

**Installing mpi:** download <ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz> and run the following commands:

```
tar -xvzf mpich.tar.gz
cd mpich-1.2.5
```

```
./configure --prefix=/usr/local/mpich-1.2.5 (requires root permission else try following)
./configure --prefix=$HOME/mpich-1.2.5 (installs mpi in home directory)
```

```
make
```

This completes the downloading and installation of petsc and ready to be used.

### *Writing simple PETSc Programs*

PETSc programs are just normal c programs with function calls that are provided by PETSc libraries and should be used in a specific manner. Similarly there are several PETSc defined data types provided that has to be used within PETSc functions in combination to other normal data types. The programs must include required header files and call proper initialization commands. Here are some important steps to remember.

Before main program

- 1) Include the necessary header files. e.g. header file for vector functions is petscvec.h. So write this:

```
#include "petscvec.h"
```

- 2) The main function should have int return type and its arguments should be defined as follows:

```
int main(int argc, char **argv)
```

argc contains the number of arguments passed to the program when it is executed.  
argv is a string array which contains the value of those arguments.

- 3) Define the help variable that contains the message that is will be displayed when program is compiled with the -help option.

```
char help[]="This is help message";
```

## Initializing PETSc

Before calling any petsc function call the following function:

```
PetscInitialize(&argc,&argv,(char *)0,help);
```

&argc refers to the variable argc which was passed through 'main'  
&argv refers to the argument values which was passed through 'main'  
(char \*)0 typecasting number 0. This is to be put by default.  
help is string message define earlier.

Towards the end of the 'main' program following function should be called:

```
PetscFinalize();
```

## Error Checking

Every Petsc function has an integer return value. This value can be used to detect error status of the function called. This is done using CHKERRQ :

```
ierr = PetscFunction();  
CHKERRQ(ierr);
```

ierr is the integer that stores the return status.  
PetscFunction denotes any PETSc function.  
CHKERRQ is the error checking function which argument as ierr.  
The error reported will be displayed on the screen.

## Vector Routines

These functions require the header file "petscvec.h"  
A new data type 'Vec' is introduced which is a vector. So vectors can be defined as

```
Vec x,y,z;
```

Here the compiler is only told that x,y,z are vector types but its not completely defined yet.  
Complete definition will require creation of vector, setting its size and type. This is done by a set of function which should be called in proper order as shown below:

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

**VecCreate** creates and allocates space to the vector x. &x indicates that the address of x is being passed as argument.

**PETSC\_COMM\_WORLD** is a communicator which is the jargon used by mpi(message passing interface) programmers. For our purpose this argument is to be used as given.

**VecSetSizes** will set the dimension of the vector. This size could be local or global which is relevant only when multiple processors are being used. So we let PETSc to decide the local size (PETSC\_DECIDE) and we only specify the global size (n).

**VecSetFromOptions** sets the type of vector using options given on the command line. There are several types like parallel vector (used for mpis) or sequential vector (single processor). When no option is given it takes some default value. The type can also be set explicitly using other function, VecSet (refer to the table). Once the vector is created and set it is ready to be used by other PETSc functions to operate on these vectors. A list of some commonly required functions are listed in the table provided.

Sample Function Call	Variable definition	Function
VecCopy(x,y)	Vec x , Vec y	y=x
VecCreate( PETSC_COMM_WORLD , &Vec x	Vec x	Create new vector x
VecDestroy(x)	Vec x	Destroy x
VecDuplicate(x,&y)	Vec x , Vec y	y=x
VecGetArray(x,&a)	Vec x, PetscScalar *a	*a=x[0] , *(a+1)=x[1] , *(a+2)=x[2] ..
VecGetLocalSize(x,&n)	Vec x, int n	n=num. of elements in x locally
VecGetSize(x,&n)	Vec x, int n	n=num. of elements in x globally
VecPermute(x,row,PETSC_TRUE)	Vec x, IS row	Rearrange elements of x as defined the vector row
VecRestoreArray(x,&a)	Vec x, PetscScalar *a	Must be called after VecGetArray
VecSet(&alpha,x)	PetscScalar alpha, Vec x	x[i]=alpha, for all i
VecSetFromOptions(x)	Vec x	To be called after VecCreate
VecSetValue(x,row,val,ADD_VALUES)	Vec x, int row, PetscScalar val	x[row]=val
VecSqrt(x)	Vec x	x[i]=sqrt(x[i])
VecSum(x,&sum)	Vec x, PetscScalar sum	Sum=x[0]+x[1]+...+x[n]
VecView(x,PETSC_VIEWER_STDOUT_SELF)	Vec x	Print x
VecAXPY(&a,&b,x,y)	Vec x,y; PetscScalar a,b;	y(i)=a*x(i)+b*(y(i))
VecXPY(&a,x,y)	Vec x,y; PetscScalar a;	y(i)=a*x(i)+y(i)
VecYPX(&a,x,y)	Vec x,y; PetscScalar a;	y(i)=a*y(i)+x(i)
VecAbs(x)	Vec x	x(i)=abs(x[i])
VecConjugate(x)	Vec x	Find conjugate of complex vector x
VecCreateSeq(PETSC_COMM_WORLD,n,&x)	Vec x, int n;	Create a sequential array
VecEqual(x,y,&flag)	Vec x,y; PetscTruth flag	If x==y then flag=0 else flag=1
VecMAXPY(n,&a,y,&x)	int n; Vec x[n],y; PetscScalar a[n];	y=y+a[0]*x[0]+a[1]*x[1]+...a[n]
VecMDot(n,x,y,&val)	Int n; Vec x,y[n]; PetscScalar val;	val(i)=dot(x,y[i])
VecMax(x,&p,&val)	Vec x, int p, PetscReal val	val=Max. Component at location p
VecMin(x,&p,&val)	Vec x, int p, PetscReal val	val=Min. Component at location p
VecNorm(Vec x,NORM_2,&val)	Vec x, PetscReal val;	val=norm(x)
VecNormalize(x,&val)	Vec x, PetscReal val;	Normalize x with its 2-norm
VecReciprocal(x)	Vec x	x[i]=1/x[i]

VecScale(&alpha,x)	PetscScalar alpha, Vec x	$x[i]=\alpha*x[i]$
VecSetRandom(rnd,x)	PetscRandom rnd, Vec x	Set x to random elements
VecShift(&shift,x)	PetscScalar shift, Vec x	$x[i]=x[i]+shift$
VecViewFromOptions(x,&title)	Vec x, char title[];	Print x with title
VecWXPY(&a,x,y,w)	PetscScalar alpha, Vec x,y,ww=a*x+y	
VecSetType(x,VECSEQ)	Vec x	Set x as a sequential array

## Sample Program on vector operations

---

```

#include "petscvec.h"
void vecprint(Vec v,int n)
{
    int i;
    PetscScalar *array;
    VecGetArray(v,&array);
    for(i=0;i<n;i++) PetscPrintf(PETSC_COMM_WORLD," %d ",(int)*(array+i));
    printf("\n");
    VecRestoreArray(v,&array);
}
int main(int argc, char **argv)
{
    Vec x,y;
    int n=10,err;
    char help[]="gone case";
    PetscScalar num=2;

    err=PetscInitialize(&argc,&argv,(char*)0,help);CHKERRQ(err);

    err=VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(err);
    //VecSetType(x,VECSEQ);
    err=VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(err);
    err=VecSetFromOptions(x);CHKERRQ(err);

    VecDuplicate(x,&y);

    VecSet(&num,x);
    VecSet(&num,y);

    printf("x =");
    vecprint(x,n);
    printf("y =");
    vecprint(y,n);

    VecScale(&num,x);
    printf("x=%d*x=",(int)num);

```

```

    vecprint(x,n);

    VecAXPY(&num,x,y);
    printf("y=%d*x+y=",(int)num);
    vecprint(y,n);

    VecAYPX(&num,x,y);
    printf("y=%d*y+x=",(int)num);
    vecprint(y,n);

    VecWXPY(&num,x,y,x);
    printf("x=%d*x+y=",(int)num);
    vecprint(x,n);

    VecDestroy(x);
    VecDestroy(y);

    return 0;
}

```

The output:

```

x = 2 2 2 2 2 2 2 2 2 2
y = 2 2 2 2 2 2 2 2 2 2
x=2*x= 4 4 4 4 4 4 4 4 4 4
y=2*x+y= 10 10 10 10 10 10 10 10 10 10
y=2*y+x= 24 24 24 24 24 24 24 24 24 24
x=2*x+y 32 32 32 32 32 32 32 32 32 3

```

### *Creating Makefile for PETSc Programs*

Makefiles are required by the program 'make' which is used to compile the source files. The advantage of using make is that it only compiles those files in which changes have been made so the compile time will be lesser if small changes are made in the sourcefiles. This is useful when there are large number of sourcefiles and corresponding object files. For Petsc it is compulsory to use 'make' command to compile the programs. When make is invoked, it first searches for a file named 'Makefile' and if not found, it searches for the file 'makefile' in the local directory. So a makefile should always be named 'Makefile' or 'makefile'. In the makefile there are macros and commands required for compiling. Macros are short names that can be used withing commands to replace long expressions and to avoid making frequent changes in the main part of the makefile. There are several builtin macros like CFLAGS, FLAGS, OPPFLAGS etc. There could be user-defined macros also. Macros are evaluated by prefixing them with '\$', so \$CFLAG will finally evaluate to the expression that was defined earlier. Placing the macro in braces will allow the eavaluated expression to be used as a part of other bigger expression. For our purpose the required macros are LOCDIR which should be assigned the path of the current directory. Care should be taken that the path name should not end in a '/' after the directory name. e.g.:

```
LOCDIR=/home/tanuj/petsc
```

This should be followed by the following line:

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

This is followed by the compiling commands. It has the following format :

```
myex1: myex1.o chkopts
    -${CLINKER} -o myex1 myex1.o ${PETSC_VEC_LIB}
    ${RM} -f myex1.o
```

'myex1' is the called the target which is an executable to be created after compilation is done. This target file may depend on several other files (object files). These files are called dependencies. The dependencies must be specified to the makefile by placing them in front of the target file separated by a colon. The lines next to this line should always begin with a tab entry. `-${CLINKER}` tells 'make' which compiler to choose. `CLINKER` is a macro that is predefined and is to be used as it is. Thing to note is a hyphen '-' preceding the \$ sign. It tells 'make' to ignore the errors that the following command makes so that 'make' can continue evaluating other commands or else it will stop and report the error when it receives any error. '-o' option is used to specify the name of the executable to be created. This name follows the -o option. e.g. in the above extract output file will be myex1. This is followed by the name of the object files that are required to be linked. In the above case, myex1.o is one of the object files.

There are several other files that are required by PETSc. These files are specified by pointing to the libraries given by PETSc. Libraries are specific to PETSc functions that are used within the program. So if the program contains calls to PETSc vector functions then libraries corresponding to vectors should be used. To use a library simply put the predefined macro corresponding to the required library. In the above case this is `-${PETSC_VEC_LIB}`.

After successful compilation, the object files created can be removed using `"${RM} -f objectfiles.o"`. -f option is used to forcibly remove the file without asking for confirmation. This way there could be several targets can be compiled through one makefile.

Here is a sample makefile that has 3 targets.

---

```
CFLAGS      =
FFLAGS      =
CPPFLAGS    =
FPPFLAGS    =
LOCDIR      = /home/tanuj/petsc
MANSEC      = Vec Mat
include ${PETSC_DIR}/bmake/common/base
```

```
myex1: myex1.o chkopts
      -${CLINKER} -o myex1 myex1.o ${PETSC_VEC_LIB}
      ${RM} -f myex1.o
```

```
helloworld: helloworld.o chkopts
           -${CLINKER} -o helloworld helloworld.o ${PETSC_VEC_LIB}
           ${RM} -f helloworld.o
```

```
mymat: mymat.o chkopts
      -${CLINKER} -o mymat mymat.o ${PETSC_SLES_LIB}
      ${RM} -f mymat.o
```

---

### *Creating Static PETSc Libraries*

Libraries are a collection of related object files (.o extension) which can be linked later and can be supplied to the user in a single file. There are two types of libraries: static(.a extension) and dynamic(.so extension). Static libraries are linked during compilation while dynamic libraries are linked during runtime.

#### **Creating library files in unix:**

Static libraries(command name 'ar' standing for archive)

```
ar [options] [library_name.a] [list of object files]
options could be the following:
```

```
c - Create a new library
q - add the named file to the end of the archive
r - replace a named archive/library member
t - print a table of archive contents
```

e.g. creating a new library named mylib.a from all object files:

```
ar cq mylib.a *.o
```

#### **Static libraries and makefile**

Let the object files to be archived be "obj1.o" "obj2.o" "obj3.o" etc.

Let the archived filename be "mylib.a".

Let the petsc program that uses this library be "myprog".

Then, the makefile entry should look like the following lines:

```
mylib.a: obj1.o obj2.o obj3.o
    -rm -f $@
    ar cq $@ obj1.o obj2.o obj3.o
```

```
myprog: mylib.a
    ${CLINKER} -o petout mylib.a ${PETSC_LIB}
```

Note: Other things should not change in the makefile and you should remember to export environment variables ( PETSC\_DIR and PETSC\_ARCH )

### Now, how to make object files from source files?

When no specific option is given to the compiler, it compiles all the source files into object files and then links them into an executable. Compiler can be instructed to stop just after creating the object files by using option "-c" as shown here:

```
g++ -c source1.c source2.c source3.c ...
```

This will create files: source1.o source2.o source3.o and so on.

Care should be taken that the source-files(.c files) are not included into other other source-files.

Then how are the source-files integrated? The source files usually contain several function calls and function definitions. The programmer should make a header file which has all the declarations of all the functions used and this header file should be included in all those source files where the functions are called. Its the compiler's duty to find out where functions are defined among the given source files. If this is done properly there should be no problem in creating objects from these files and then linking them.

### Example programs

---

#### **file1:print.c**

```
#include"head.h"
void print(Vec x,Vec y)
{
    VecView(x,PETSC_VIEWER_STDOUT_SELF);
    VecView(y,PETSC_VIEWER_STDOUT_SELF);
}
```

**file:exec.c**

```
#include "head.h"
char help[]="library creation example";

int main(int argc, char **argv)
{
    Vec x,y;
    int n=3;
    PetscScalar a=3.1;

    PetscInitialize(&argc,&argv,(char *)0,help);
    VecCreate(PETSC_COMM_WORLD,&x);
    VecSetSizes(x,PETSC_DECIDE,n);
    VecSetFromOptions(x);
    VecDuplicate(x,&y);
    VecSet(&a,x);
    a=a-4;
    VecSet(&a,y);

    print(x,y);

    VecDestroy(x);
    VecDestroy(y);

    return 0;
}
```

**file: head.h**

```
#include "petscvec.h"
void print(Vec x,Vec y);
```

**file:makefile**

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

mylib.a: func.o exec.o
-rm -f $@
ar cq $@ func.o exec.o

petout: mylib.a
${CLINKER} -o petout mylib.a ${PETSC_VEC_LIB}
```

**execution commands:**

```
export PETSC_DIR=<petsc directory path>  
export PETSC_ARCH=linux-gnu  
make BOPT=g mylib.a  
make BOPT=g petout  
./petout
```