Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL-

PETSc Developers Manual

by

The PETSc Team
http://www.mcs.anl.gov/petsc

This document is intended for use with PETSc 3.3

June 2012

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Abstract:

PETSc is a set of extensible software libraries for scientific computation. PETSc is designed using a object-oriented architecture. This means that libraries consist of *objects* that have certain, defined functionality. This document defines how these objects are implemented.

This manual discusses the PETSc library design and describes how to develop new library codes that are compatible with other PETSc components including PETSc 3.0. The idea is not to develop one massive library that everyone shoves code into; rather, to develop an architecture that allows many people to (as painlessly as possible) contribute (and maintain) their own libraries, in a distributed fashion.

The text assumes that you are familiar with PETSc, have a copy of the PETSc users manual, and have access to PETSc source code and documentation (available via http://www.mcs.anl.gov/petsc

Please direct all comments and questions regarding PETSc design and development to `petsc-dev@mcs.anl.gov`. Note that all *bug reports and questions regarding the use of PETSc* should continue to be directed to `petsc-maint@mcs.anl.gov`.

Chapter 1

The PETSc Kernel

PETSc provides a variety of basic services for writing scalable, component based libraries; these are referred to as the PETSc kernel. The source code for the kernel is in `src/sys`. It contains systematic support for

- PETSc types
- error handling
- memory management
- profiling
- object management
- file IO
- options database

Each of these is discussed in a section below.

1.1 PETSc Types

For maximum flexibility, the basic data types `int`, `double` etc are generally not used in source code, rather it has:

- PetscScalar
- PetscInt
- PetscMPIInt
- PetscBLASInt
- PetscBool
- PetscBT - bit storate of logical true and false

PetscInt can be set using ./configure to be either `int` (32 bit) or `long long` (64 bit) to allow indexing into very large arrays. PetscMPIInt are for integers passed to MPI as counts etc, these are always `int` since that is what the MPI standard uses. Similarly PetscBLASInt are for counts etc passed to BLAS and LAPACK routines. These are almost always `int` unless one is using a special "64 bit integer" BLAS/LAPACK (this is available, for example on Solaris systems).

In addition there a special types

- PetscClassId
- PetscErrorCode
- PetscLogEvent

in fact, these are currently always `int` but their use clarifies the code.

1.2 Implementation of Error Handling

PETSc uses a "call error handler; then (depending on result) return error code" model when problems are detected in the running code.

The public include file for error handling is include/petscerror.h, the source code for the PETSc error handling is in `src/sys/error/`.

1.2.1 Simplified Interface

The simplified C/C++ macro-based interface consists of the following three calls

- SETERRQ(comm,error code,"Error message");
- CHKERRQ(ierr);

The macro SETERRQ() is given by

return PetscError(comm,__LINE__,__FUNC__,__FILE__,__SDIR__,specific,"Error message");

It calls the error handler with the current function name and location: line number, file and directory, plus an error codes and an error message. The macro CHKERRQ() is defined by

if (ierr) SETERRQ(PETSC_COMM_SELF,ierr,(char *)0);

In addition to SETERRQ() are the macros SETERRQ1(), SETERRQ2(), SETERRQ3() and SETERRQ4() that allow one to include additional arguments that the message string is formated. For example,

SETERRQ2(PETSC_ERR,"Iteration overflow: its

The reason for the numbered format is because CPP macros cannot handle variable number of arguments.

1.2.2 Error Handlers

The error handling function PetscError() calls the "current" error handler with the code

```
PetscErrorCode PetscError(MPI_Comm,int line,char *func,char* file,char *dir,PetscErrorCode n,int p,cha
{
PetscErrorCode ierr;

PetscFunctionBegin;
if (!eh) ierr = PetscTraceBackErrorHandler(line,func,file,dir,n,p,mess,0);
else ierr = (*eh->handler)(line,func,file,dir,n,p,mess,eh->ctx);
PetscFunctionReturn(ierr);
}
```

The variable `eh` is the current error handler context and is defined in `src/sys/error/err.c` as

```
typedef struct _EH* EH;
struct _EH {
int classid;
int (*handler)(int, char*,char*,char *,int,int,char*,void *);
void *ctx;
EH previous;
};
```

One can set a new error handler with the command

```
int PetscPushErrorHandler(int (*handler)(int,char *,char*,char*,PetscErrorCode,
int,char*,void*),void *ctx )
{
EH neweh = (EH) PetscMalloc(sizeof(struct _EH)); CHKPTRQ(neweh);

PetscFunctionBegin;
if (eh) neweh->previous = eh;
else neweh->previous = 0;
neweh->handler = handler;
neweh->ctx = ctx;
eh = neweh;
PetscFunctionReturn(0);
}
```

which maintains a linked list of error handlers. The most recent error handler is removed via

```
int PetscPopErrorHandler(void)
{
EH tmp;

PetscFunctionBegin;
```

```
if (!eh) PetscFunctionReturn(0);
tmp = eh;
eh = eh->previous;
PetscFree(tmp);
PetscFunctionReturn(0);
}
```

PETSc provides several default error handlers

- PetscTraceBackErrorHandler(),
- PetscAbortErrorHandler(),
- PetscReturnErrorHandler(),
- PetscEmacsClientErrorHandler(),
- PetscMPIAbortErrorHandler(), and
- PetscAttachDebuggerErrorHandler().

1.2.3 Error Codes

The PETSc error handler take a generic error code. The generic error codes are defined in `include/petscerror.h`, the same generic error code would be used many times in the libraries. For example the generic error code PETSC_ERR_MEM is used whenever requested memory allocation is not available.

1.2.4 Detailed Error Messages

In a modern parallel component oriented application code it does not make sense to simply print error messages to the screen (more than likely there is no "screen", for example with Windows applications). PETSc provides the replaceable function pointer

```
(*PetscErrorPrintf)("Format",...);
```

that, by default prints to standard out. Thus error messages should not be printed with printf() or fprintf() rather it should be printed with (*PetscErrorPrintf)(). One can direct all error messages to stderr with the command line options `-error\_output\_stderr`.

1.3 Implementation of Profiling

This section provides details about the implementation of event logging and profiling within the PETSc kernel. The interface for profiling in PETSc is contained in the file `include/petsclog.h`. The source code for the profile logging is in `src/sys/plog/`.

1.3.1 Profiling Object Create and Destruction

The creation of objects may be profiled with the command

PetscLogObjectCreate(PetscObject h);

which logs the creation of any PETSc object. Just before an object is destroyed, it should be logged with with

PetscLogObjectDestroy(PetscObject h);

These are called automatically by PetscHeaderCreate() and PetscHeaderDestroy() which are used in creating all objects inherited off the basic object. Thus these logging routines should never be called directly.

If an object has a clearly defined parent object (for instance, when a work vector is generated for use in a Krylov solver), this information is logged with the command,

PetscLogObjectParent(PetscObject parent,PetscObject child);

It is also useful to log information about the state of an object, as can be done with the command

PetscLogObjectState(PetscObject h,char *format,...);

For example, for sparse matrices we usually log the matrix dimensions and number of nonzeros.

1.3.2 Profiling Events

Events are logged using the pair

PetscLogEventBegin(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject o4);
PetscLogEventEnd(int event,PetscObject o1,PetscObject o2,PetscObject o3,PetscObject o4);

This logging is usually done in the abstract interface file for the operations, for example, `src/mat/src/matrix.c`.

1.3.3 Controling Profiling

Several routines that control the default profiling available in PETSc are are

PetscLogBegin();
PetscLogAllBegin();
PetscLogDump(char *filename);
PetscLogView(FILE *fd);

These routines are normally called by the PetscInitialize() and PetscFinalize() routines when the option `-log`, `-log\_summary`, or `-log\_all` is given.

1.3.4 Details of the Logging Design

Chapter 2

Basic Object Design

PETSc is designed using strong data encapsulation. Hence, any collection of data (for instance, a sparse matrix) is stored in a way that is completely private from the application code. The application code can manipulate the data only through a well-defined interface, as it does *not* "know" how the data is stored internally.

2.1 Introduction

PETSc is designed around several classes (e.g. Vec (vectors), Mat (matrices, both dense and sparse)). These classs are each implemented using C `structs`, that contain the data and function pointers for operations on the data (much like virtual functions in classes in C++). Each classs consists of three parts:

- a (small) common part shared by all PETSc compatible libraries.
- another common part shared by all PETSc implementations of the class and
- a private part used by only one particular implementation written in PETSc.

For example, all matrix (Mat) classs share a function table of operations that may be performed on the matrix; all PETSc matrix implementations share some additional data fields,including matrix size; while a particular matrix implementation in PETSc (say compressed sparse row) has its own data fields for storing the actual matrix values and sparsity pattern. This will be explained in more detail in the following sections. People providing new class implementations **must** use the PETSc common part.

We will use `$<$class$>$\_$<$implementation$>$` to denote the actual source code and data structures used for a particular implementation of an object that has the `$<$class$>$` interface.

2.2 Organization of the Source Code

Each class has

- Its own, application public, include file `include/petsc$<$class$>$.h`

- Its own directory, `src/$<$class$>$`
- A data structure defined in the file `include/petsc-private/$<$class$>$impl.h`. This data structure is shared by all the different PETSc implementations of the class. For example, for matrices it is shared by dense, sparse, parallel, and sequential formats.
- An abstract interface that defines the application callable functions for the class. These are defined in the directory `src/$<$class$>$/interface`.
- One or more actual implementations of the classs (for example, sparse uniprocessor and parallel matrices implemented with the AIJ storage format). These are each in a subdirectory of `src/$<$class$>$/impls`. Except in rare circumstances data structures defined here should not be referenced from outside this directory.

Each type of object, for instance a vector, is defined in its own public include file, by

typedef _p_<class>* <class>; (for example, typedef _p_Vec* Vec;).

This organization allows the compiler to perform type checking on all subroutine calls while at the same time completely removing the details of the implementation of `_p_<class>` from the application code. This capability is extremely important because it allows the library internals to be changed without altering or recompiling the application code.

Polymorphism is supported through the directory `src/$<$class$>$/interface`, which contains the code that implements the abstract interface to the operations on the object. Essentially, these routines do some error checking of arguments and logging of profiling information and then call the function appropriate for the particular implementation of the object. The name of the abstract function is `$<$class$>$Operation`, for instance, MatMult() or PCCreate(), while the name of a particular implementation is `$<$class$>$Operation\_$<$implementation$>$`, for instance, `MatMult\_SeqAIJ()` or `PCCreate\_ILU()`. These naming conventions are used to simplify code maintenance.

2.3 Common Object Header

All PETSc/PETSc objects have the following common header structures (in `include/petsc-private/petscimpl.h`)

```
/* Function table common to all PETSc compatible classs */

typedef struct {
int (*getcomm)(PetscObject,MPI_Comm *);
int (*view)(PetscObject,Viewer);
int (*destroy)(PetscObject);
int (*query)(PetscObject,char *,PetscObject *);
int (*compose)(PetscObject,char*,PetscObject);
int (*composefunction)(PetscObject,char *,char *,void *);
```

```
int (*queryfunction)(PetscObject,char *, void **);
} PetscOps;

/* Data structure header common to all PETSc compatible classs */

struct _p_<class> {
PetscClassId classid;
PetscOps *bops;
<class>Ops *ops;
MPI_Comm comm;
PetscLogDouble flops,time,mem;
int id;
int refct;
int tag;
DLList qlist;
OList olist;
char *type_name;
PetscObject parent;
char *name;
char *prefix;
void *cpp;
void **fortran_func_pointers;
.....
CLASS-SPECIFIC DATASTRUCTURES
};
```

Here `$<$class$>$ops` is a function table (like the `PetscOps` above) that contains the function pointers for the operations specific to that class. For example, the PETSc vector class object looks like

```
/* Function table common to all PETSc compatible vector objects */

typedef struct _VecOps* VecOps;
struct _VecOps {
PetscErrorCode (*duplicate)(Vec,Vec*), /* get single vector */
(*duplicatevecs)(Vec,int,Vec**), /* get array of vectors */
(*destroyvecs)(Vec*,int), /* free array of vectors */
(*dot)(Vec,Vec,Scalar*), /* z = xĤ * y */
(*mdot)(int,Vec,Vec*,Scalar*), /* z[j] = x dot y[j] */
(*norm)(Vec,NormType,double*), /* z = sqrt(xĤ * x) */
(*tdot)(Vec,Vec,Scalar*), /* x'*y */
(*mtdot)(int,Vec,Vec*,Scalar*), /* z[j] = x dot y[j] */
(*scale)(Scalar*,Vec), /* x = alpha * x */
(*copy)(Vec,Vec), /* y = x */
(*set)(Scalar*,Vec), /* y = alpha */
```

```
(*swap)(Vec,Vec), /* exchange x and y */
(*axpy)(Scalar*,Vec,Vec), /* y = y + alpha * x */
(*axpby)(Scalar*,Scalar*,Vec,Vec),/* y = y + alpha * x + beta * y*/
(*maxpy)(int,Scalar*,Vec,Vec*), /* y = y + alpha[j] x[j] */
(*aypx)(Scalar*,Vec,Vec), /* y = x + alpha * y */
(*waxpy)(Scalar*,Vec,Vec,Vec), /* w = y + alpha * x */
(*pointwisemult)(Vec,Vec,Vec), /* w = x .* y */
(*pointwisedivide)(Vec,Vec,Vec), /* w = x ./ y */
(*setvalues)(Vec,int,int*,Scalar*,InsertMode),
(*assemblybegin)(Vec), /* start global assembly */
(*assemblyend)(Vec), /* end global assembly */
(*getarray)(Vec,Scalar**), /* get data array */
(*getsize)(Vec,int*),(*getlocalsize)(Vec,int*),
(*getownershiprange)(Vec,int*,int*),
(*restorearray)(Vec,Scalar**), /* restore data array */
(*max)(Vec,int*,double*), /* z = max(x); idx=index of max(x) */
(*min)(Vec,int*,double*), /* z = min(x); idx=index of min(x) */
(*setrandom)(PetscRandom,Vec), /* set y[j] = random numbers */
(*setoption)(Vec,VecOption),
(*setvaluesblocked)(Vec,int,int*,Scalar*,InsertMode),
(*destroy)(Vec),
(*view)(Vec,Viewer);
};

/* Data structure header common to all PETSc vector classs */

struct _p_Vec {
PetscClassId classid;
PetscOps *bops;
VecOps *ops;
MPI_Comm comm;
PetscLogDouble flops,time,mem;
int id;
int refct;
int tag;
DLList qlist;
OList olist;
char *type_name;
PetscObject parent;
char* name;
char *prefix;
void** fortran_func_pointers;
void *data; /* implementation-specific data */
int N, n; /* global, local vector size */
int bs;
ISLocalToGlobalMapping mapping; /* mapping used in VecSetValuesLocal() */
```

ISLocalToGlobalMapping bmapping; /* mapping used in VecSetValuesBlockedLocal() */
};

Each PETSc object begins with a PetscClassId which is used for error checking. Each different class of objects has its value for the classid; these are used to distinguish between classes. When a new class is created one needs to call

ierr = PetscClassIdRegister(char *classname,PetscClassId *classid);CHKERRQ(ierr);

For example,

ierr = PetscClassIdRegister("index set",&IS_CLASSID);CHKERRQ(ierr);

Question: Why is a fundamental part of PETSc objects defined in PetscLog when PETSc Log is something that can be "turned off" One can verify that an object is valid of a particular class with

PetscValidHeaderSpecific(x,VEC_CLASSID,1);

The third argument to this macro indicates the position in the calling sequence of the function the object was passed in. This is generate more complete error messages.

To check for an object of any type use

PetscValidHeader(x,1);

Several routines are provided for manipulating data within the header, including

int PetscObjectGetComm(PetscObject object,MPI_Comm *comm);

which returns in `comm` the MPI communicator associated with the specified object.

2.4 Common Object Functions

We now discuss the specific functions in the PETSc common function table.

- `getcomm(PetscObject,MPI\_Comm *)` obtains the MPI communicator associated with this object.
- `view(PetscObject,Viewer)` allows one to store or visualize the data inside an object. If the Viewer is null than should cause the object to print information on the object to standard out. PETSc provides a variety of simple viewers.
- `destroy(PetscObject)` causes the reference count of the object to be decreased by one or the object to be destroyed and all memory used by the object to be freed when the reference count drops to zero. If the object has any other objects composed with it then they are each sent a `destroy()`, i.e. the `destroy()` function is called on them also.

- `compose(PetscObject,char *name, PetscObject)` associates the second object with the first object and increases the reference count of the second object. If an object with the same name was previously composed that object is dereferenced and replaced with the new object. If the second object is null and and object with the same name has already been composed that object is dereferenced (the `destroy()` function is called on it, and that object is removed from the first object); i.e. this is a way to remove, by name, an object that was previously composed.

- `query(PetscObject,char *name, PetscObject *)` retrieves an object that was previously composed with the first object. Retreives a null if no object with that name was previously composed.

- `composefunction(PetscObject,char *name,char *fname,void *func)` associates a function pointer to an object. If the object already had a composed function with the same name, the old one is replace. If the fname is null it is removed from the object. The string `fname` is the character string name of the function; it may include the path name or URL of the dynamic library where the function is located. The argument `name` is a "short" name of the function to be used with the `queryfunction()` call. On systems that support dynamic libraries the `func` argument is ignored; otherwise `func` is the actual function pointer.

  For example, `fname` may be `libpetscksp:PCCreate\_LU` or `http://www.mcs.anl.gov/petsc/libpetscksp:PCCreate\_LU`.

- `queryfunction(PetscObject,char *name,void **func)` retreives a function pointer that was associated with the object. If dynamic libraries are used the function is loaded into memory at this time (if it has not been previously loaded), not when the `composefunction()` routine was called.

Since the object composition allows one to **only** compose PETSc objects with PETSc objects rather than any arbitrary pointer, PETsc provides the convenience object PetscContainer, created with the routine PetscContainerCreate(MPI_Comm,PetscContainer) to allow one to wrap any kind of data into a PETSc object that can then be composed with a PETSc object.

2.5 PETSc Implementation of the Object Functions

This sections discusses how PETSc implements the `compose()`, `query()`, `composefunction()`, and `queryfunction()` functions for its object implementations. Other PETSc compatible class implementations are free to manage these functions in any manner; but generally they would use the PETSc defaults so that the library writer does not have to "reinvent the wheel."

2.5.1 Compose and Query

In `src/sys/objects/olist.c` PETSc defines a C struct

```
typedef struct _PetscOList *PetscOList;
struct _PetscOList {
char name[128];
PetscObject obj;
PetscOList next;
};
```

from which linked lists of composed objects may be constructed. The routines to manipulate these elementary objects are

```
int PetscOListAdd(PetscOList *fl,char *name,PetscObject obj );
int PetscOListDestroy(PetscOList fl );
int PetscOListFind(PetscOList fl, char *name, PetscObject *obj)
int PetscOListDuplicate(PetscOList fl, PetscOList *nl);
```

The function PetscOListAdd() will create the initial PetscOList if the argument `fl` points to a null.

The PETSc object `compose()` and `query()` functions are then simply (defined in `src/sys/objects/inherit.c`)

```
PetscErrorCode PetscObjectCompose_Petsc(PetscObject obj,char *name,PetscObject ptr)
{
PetscErrorCode ierr;

PetscFunctionBegin;
ierr = PetscOListAdd(&obj->olist,name,ptr); CHKERRQ(ierr);
PetscFunctionReturn(0);
}

PetscErrorCode PetscObjectQuery_Petsc(PetscObject obj,char *name,PetscObject *ptr)
{
PetscErrorCode ierr;

PetscFunctionBegin;
ierr = PetscOListFind(obj->olist,name,ptr);CHKERRQ(ierr);
PetscFunctionReturn(0);
}
```

2.5.2 Compose and Query Function

PETSc allows one to compose functions by string name (to be loaded later from a dynamic library) or by function pointer. In `src/sys/dll/reg.c` PETSc defines the C structure

```
typedef struct _PetscFList* PetscFList;
struct _PetscFList {
int (*routine)(void *);
char *path;
```

```
char *name;
char *rname; /* name of create function in link library */
PetscFList *next;
};
```

The PetscFList object is a linked list of function data; each of which contains

- a function pointer (if it has already been loaded into memory from the dynamic library)
- the "path" (directory and library name) where the function exists (if it is loaded from a dynamic library)
- the "short" name of the function,
- the actual name of the function as a string (for dynamic libraries this string is used to load in the actual function pointer).

Each PETSc object contains a PetscFList object. The `composefunction()` and `queryfunction()` are given by

```
PetscErrorCode PetscObjectComposeFunction_Petsc(PetscObject obj,char *name,char *fname,void *ptr)
{
PetscErrorCode ierr;

PetscFunctionBegin;
ierr = PetscFListAdd(&obj->qlist,name,fname,(int (*)(void *))ptr);CHKERRQ(ierr);
PetscFunctionReturn(0);
}

PetscErrorCode PetscObjectQueryFunction_Petsc(PetscObject obj,char *name,void **ptr)
{
PetscErrorCode ierr;

PetscFunctionBegin;
ierr = PetscFListFind(obj->qlist,obj->comm,name,PETSC_FALSE,( int(**)(void *)) ptr);CHKERRQ(ier
PetscFunctionReturn(0);
}
```

Because we need to support function composition on systems both **with** and **without** dynamic link libraries the actual source code is a little messy. The idea is that on systems with dynamic libraries all PETSc "register" and "composefunction" function calls that take the actual function pointer argument must eliminate this argument in the preprocessor step before the code is compiled. Otherwise, since the compiler sees the function pointer, it will compile it in and link in all those functions; thus one could not take advantage of the dynamic libraries. This is done with macros like the following

```
#if defined(USE_DYNAMIC_LIBRARIES)
#define PetscFListAdd(a,b,p,c) PetscFListAdd_Private(a,b,p,0)
```

#else
#define PetscFListAdd(a,b,p,c) PetscFListAdd_Private(a,b,p,(int (*)(void *))c)
#endif

Thus when the code is compiled with the dynamic link library flag the function pointer argument is removed from the code; otherwise it is retained. Ugly, but neccessary.

The `PetscFListAdd\_Private()` and all related routines can be found in the directory `src/sys/dll`.

In addition to using the PetscFList mechanism to compose functions into PETSc objects, it is also used to allow registration of new class implementations; for example, new preconditioners, see Section 4.2.3.

2.5.3 Simple PETSc Objects

There are some simple PETSc objects that do not need PETSCHEADER - and the associated functionality. These objects are internally named as _n_<class> as opposed to _p_<class>. For eg: _n_PetscTable vs _p_Vec.

Chapter 3

Mimimal Class Standards

This chapter discusses the miminal functionality and format required of any class that is compatible with PETSc.

Chapter 4

PetscObjects

4.1 Elementary Objects: IS, Vec, Mat

4.2 Solver Objects: PC, KSP, SNES, TS

4.2.1 Preconditioners: PC

The base PETSc PC object is defined in the `include/petsc-private/pcimpl.h` include file. A carefully commented implementation of a PC object can be found in `src/ksp/pc/impls/jacobi/jacobi.c`.

4.2.2 Krylov Solvers: KSP

The base PETSc KSP object is defined in the `include/petsc-private/kspimpl.h` include file. A carefully commented implementation of a KSP object can be found in `src/ksp/ksp/impls/cg/cg.c`.

4.2.3 Registering New Methods

See `src/ksp/examples/tutorials/ex12.c` for an example of registering a new preconditioning (PC) method.

Chapter 5

Style Guide

The PETSc team uses certain conventions to make our source code consistent. Groups developing classs compatible with PETSc are, of course, free to organize their own source code anyway they like.

5.1 Names

Consistency of names for variables, functions, etc. is extremely important in making the package both usable and maintainable. We use several conventions:

- All function names and enum types consist of words, each of which is capitalized, for example KSPSolve() and MatGetOrdering().
- All enum elements and macro variables are capitalized. When they consist of several complete words, there is an underscore between each word.
- Functions that are private to PETSc (not callable by the application code) either
  - have an appended `_Private` (for example, `StashValues_Private`) or
  - have an appended `_$<$class$>$Subtype` (for example, `MatMult_SeqAIJ`).

  In addition, functions that are not intended for use outside of a particular file are declared static.
- Function names in structures are the same as the base application function name without the object prefix, and all are in small letters. For example, MatMultTranspose() has a structure name of `multtranspose()`.
- Each application usable function begins with the name of the class object, followed by any subclass name, for example, ISInvertPermutation(), MatMult() or KSPGMRESSetRestart().
- Options database keys are lower case, have an underscore between words and match the function name associated with the option without the word set. For example, `-ksp_gmres_restart`.

5.2 Coding Conventions and Style Guide

Within the PETSc source code, we adhere to the following guidelines so that the code is uniform and easily maintainable:

- All PETSc function bodies are indented two characters.
- Each additional level of loops, if statements, etc. is indented two more characters.
- Wrapping lines should be avoided whenever possible.
- Source code lines do not have a hard length limit, generally we like then less than 150 characters wide.
- The local variable declarations should be aligned. For example, use the style

  int i,j;
  Scalar a;

  instead of

  int i,j;
  Scalar a;

- All local variables of a particular type (e.g., `int`) should be listed on the same line if possible; otherwise, they should be listed on adjacent lines.
- Equal signs should be aligned in regions where possible.
- There *must* be a single blank line between the local variable declarations and the body of the function.
- The first line of the executable statments must be PetscFunctionBegin;
- The following text should be before each function

  #undef __FUNC__
  #define __FUNC__ "FunctionName"

  this is used by various macros (for example the error handlers) to always know what function one is in.

- Use PetscFunctionReturn(returnvalue); not `return(returnvalue);`
- *Never* put a function call in a return statment; do not do

  PetscFunctionReturn(somefunction(...));

- Do **not** put a blank line immediately after PetscFunctionBegin; or a blank line immediately before PetscFunctionReturn(0);.

- Indentation for `if` statements *must* be done as as

  if ( ) {
  ....
  } else {
  ....
  }

- *Never* have

  if ( )
  a single indented line

  or

  for ( ) a single indented line

  instead use either

  if ( ) a single line

  or

  if ( ) {
  a single indented line
  }

- *No* tabs are allowed in *any* of the source code.

- The open bracket { should be on the same line as the *if ()* test, *for ()*, etc. never on its own line. The closing bracket } should **always** be on its own line.

- In function declaration the open bracket { should be on the **next** line, not on the same line as the function name and arguments. This is an exception to the rule above.

- The macros SETERRQ() and CHKERRQ() should be on the same line as the routine to be checked unless this violates the 150 character width rule. Try to make error messages short, but informative.

- *No* space after a (or before a). No space before the CHKXXX(). That is, do not write

  ierr = PetscMalloc(10*sizeof(int),&a); CHKERRQ(ierr);

  instead write

  ierr = PetscMalloc(10*sizeof(int),&a);CHKERRQ(ierr);

- *No* space after the) in a cast, no space between the type and the * in a caste.
- *No* space before or after a , in lists That is, do not write

  int a, b,c;
  ierr = func(a, 22.0);CHKERRQ(ierr);

  instead write

  int a,b,c;
  ierr = func(a,22.0);CHKERRQ(ierr);

- Subroutines that would normally have void ** argument to return a pointer to some data, should actually be protyped as as void*. This prevents the caller from having to put a (void**) caste in each function call. See, for example, DMDAVecGetArray().
- Do not use the *register* directive.
- Never use a local variable counter like PetscInt flops = 0; to accumulate flops and then call PetscLogFlops() *always* just call PetscLogFlops() directly when needed.
- Do not use *if (rank == 0)* or *if (v == PETSC_NULL)* or *if (flg == PETSC_TRUE)* or *if (flg == PETSC_FALSE)* instead use *if (!rank)* or *if (!v)* or *if (flg)* or *if (!flg)*.
- Do not use *#ifdef* or *#ifndef* rather use *#if defined(...* or *#if !defined(...*
- MPI routines and macros that are not part of the 1.0 or 1.1 standard should not be used in PETSc without appropriate ./configure checks and #if defined() checks the code. Code should also be provided that works if the MPI feature is not available. For example,

```
#if defined(PETSC_HAVE_MPI_IN_PLACE)
    ierr  = \href{http://www.mcs.anl.gov/petsc/petsc-3.3/docs/http://www.mcs.anl.go
#else
    ierr  = \href{http://www.mcs.anl.gov/petsc/petsc-3.3/docs/http://www.mcs.anl.go
#endif
```

- There shall be no PETSc routines introduced that provide essentially the same functionality as an available MPI routine. For example, one should not write a routine PetscGlobalSum() that takes a scalar value and performs an MPI_Allreduce() on it. One should use MPI_Allreduce() directly in the code. (Note: PetscGlobalSum() does exist but is ONLY to be used in code that is to be run through ADIC (since ADIC cannot handle MPI calls)).
- XXXTypes (for example KSPType) do not have an underscore in them, unless they refer to another package that uses an underscore, for example MATSOLVERSUPERLU_DIST.

5.3 Option Names

Since consistency simplifies usage and code maintenance, the names of PETSc routines, flags, options, etc. have been selected with great care. The default option names are of the form `-$<$class$>$\_sub$<$class$>$\_name`. For example, the option name for the basic convergence tolerance for the KSP package is `-ksp\_atol`. In addition, operations in different packages of a similar nature have a similar name. For example, the option name for the basic convergence tolerance for the SNES package is `-snes\_atol`.

When a `Set` is included in a function name, it is dropped in the options key. For example `KSPGMRESSetRestart()` becomes `-ksp_gmres_restart`.

Chapter 6

The Various Matrix Classes

PETSc provides a variety of matrix implementations, since no single matrix format is appropriate for all problems. This section first discusses various matrix blocking strategies, and then describes the assortment of matrix types within PETSc.

6.0.1 Matrix Blocking Strategies

In today's computers, the time to perform an arithmetic operation is dominated by the time to move the data into position, not the time to compute the arithmetic result. For example, the time to perform a multiplication operation may be one clock cycle, while the time to move the floating point number from memory to the arithmetic unit may take 10 or more cycles. To help manage this difference in time scales, most processors have at least three levels of memory: registers, cache, and random access memory, RAM. (In addition, some processors have external caches, and the complications of paging introduce another level to the hierarchy.)

Thus, to achieve high performance, a code should first move data into cache, and from there move it into registers and use it repeatedly while it remains in the cache or registers before returning it to main memory. If one reuses a floating point number 50 times while it is in registers, then the "hit" of 10 clock cycles to bring it into the register is not important. But if the floating point number is used only once, the "hit" of 10 clock cycles becomes very noticeable, resulting in disappointing flop rates.

Unfortunately, the compiler controls the use of the registers, and the hardware controls the use of the cache. Since the user has essentially no direct control, code must be written in such a way that the compiler and hardware cache system can perform well. Good quality code is then be said to respect the memory hierarchy.

The standard approach to improving the hardware utilization is to use blocking. That is, rather than working with individual elements in the matrices, one employs blocks of elements. Since the use of implicit methods in PDE-based simulations leads to matrices with a naturally blocked structure (with a block size equal to the number of degrees of freedom per cell), blocking is extremely advantageous. The PETSc sparse matrix representations use a variety of techniques for blocking, including

- storing the matrices using a generic sparse matrix format, but storing additional information about adjacent rows with identical nonzero structure (so called I-nodes); this

I-node information is used in the key computational routines to improve performance (the default for the MATSEQAIJ and MATMPIAIJ formats);

- storing the matrices using a fixed (problem dependent) block size (via the MATSEQBAIJ and MATMPIBAIJ formats); and

The advantage of the first approach is that it is a minimal change from a standard sparse matrix format and brings a large percent of the improvement one obtains via blocking. Using a fixed block size gives the best performance, since the code can be hardwired with that particular size (for example, in some problems the size may be 3, in others 5, etc.), so that the compiler will then optimize for that size, removing the overhead of small loops entirely.

The following table presents the floating point performance for a basic matrix-vector product using these three approaches: a basic compressed row storage format (using the PETSc runtime options `-mat\_seqaij -mat\_no\_unroll`); the same compressed row format using I-nodes (with the option -mat_seqaij); and a fixed block size code, with a block size of three for these problems (using the option -mat_seqbaij). The rates were computed on one node of an older IBM SP, using two test matrices. The first matrix (ARCO1), courtesy of Rick Dean of Arco, arises in multiphase flow simulation; it has 1501 degrees of freedom, 26,131 matrix nonzeros and, a natural block size of 3, and a small number of well terms. The second matrix (CFD), arises in a three-dimensional Euler flow simulation and has 15,360 degrees of freedom, 496,000 nonzeros, and a natural block size of 5. In addition to displaying the flop rates for matrix-vector products, we also display them for triangular solve obtained from an ILU(0) factorization.

Problem	Block size	Basic	I-node version	Fixed block size
Matrix-Vector Product (Mflop/sec)				
Multiphase	3	27	43	70
Euler	5	28	58	90
Triangular Solves from ILU(0) (Mflop/sec)				
Multiphase	3	22	31	49
Euler	5	22	39	65

These examples demonstrate that careful implementations of the basic sequential kernels in PETSc can dramatically improve overall floating point performance, and users can immediately benefit from such enhancements without altering a single line of their application codes. Note that the speeds of the I-node and fixed block operations are several times that of the basic sparse implementations. The disappointing rates for the variable block size code occur because even on a sequential computer, the code performs the matrix-vector products and triangular solves using the coloring introduced above and thus does not utilize the cache particularly efficiently. This is an example of improving the parallelization capability at the expense of using each processor less efficiently.

6.0.2 Sequential AIJ Sparse Matrices

The default matrix representation within PETSc is the general sparse AIJ format (also called the Yale sparse matrix format or compressed sparse row format, CSR).

6.0.3 Parallel AIJ Sparse Matrices

This matrix type, which is the default parallel matrix format; additional implementation details are given in [**?**].

6.0.4 Sequential Block AIJ Sparse Matrices

The sequential and parallel block AIJ formats, which are extensions of the AIJ formats described above, are intended especially for use with multiclass PDEs. The block variants store matrix elements by fixed-sized dense `nb` × `nb` blocks. The stored row and column indices begin at zero.

The routine for creating a sequential block AIJ matrix with `m` rows, `n` columns, and a block size of `nb` is

ierr = MatCreateSeqBAIJ(MPI_Comm comm,int nb,int m,int n,int nz,int *nnz, Mat *A)

The arguments `nz` and `nnz` can be used to preallocate matrix memory by indicating the number of *block* nonzeros per row. For good performance during matrix assembly, preallocation is crucial; however, the user can set `nz=0` and `nzz=PETSC\_NULL` for PETSc to dynamically allocate matrix memory as needed. The PETSc users manual discusses preallocation for the AIJ format; extension to the block AIJ format is straightforward.

Note that the routine MatSetValuesBlocked() can be used for more efficient matrix assembly when using the block AIJ format.

6.0.5 Parallel Block AIJ Sparse Matrices

Parallel block AIJ matrices with block size n̄b can be created with the command

ierr = MatCreateBAIJ(MPI_Comm comm,int nb,int m,int n,int M,int N,int d_nz,
int *d_nnz, int o_nz,int *o_nnz,Mat *A);

`A` is the newly created matrix, while the arguments `m`, `n`, `M`, and `N`, indicate the number of local rows and columns and the number of global rows and columns, respectively. Either the local or global parameters can be replaced with PETSC_DECIDE, so that PETSc will determine them. The matrix is stored with a fixed number of rows on each processor, given by `m`, or determined by PETSc if `m` is PETSC_DECIDE.

If PETSC_DECIDE is not used for `m` and `n` then the user must ensure that they are chosen to be compatible with the vectors. To do this, one first considers the product $y = Ax$. The `m` that one uses in MatCreateBAIJ() must match the local size used in the VecCreateMPI() for `y`. The `n` used must match that used as the local size in VecCreateMPI() for `x`.

The user must set `d\_nz=0`, `o\_nz=0`, `d\_nnz=PETSC\_NULL`, and `o\_nnz=PETSC\_NULL` for PETSc to control dynamic allocation of matrix memory space. Analogous to `nz` and `nnz` for the routine MatCreateSeqBAIJ(), these arguments optionally specify block nonzero information for the diagonal (`d\_nz` and `d\_nnz`) and off-diagonal (`o\_nz` and `o\_nnz`) parts of the matrix. For a square global matrix, we define each processor's diagonal portion to be its local rows and the corresponding columns (a square submatrix); each processor's off-diagonal portion encompasses the remainder of the local matrix (a rectangular submatrix). The PETSc users manual gives an example of preallocation for the parallel AIJ matrix format; extension to the block parallel AIJ case is straightforward.

6.0.6 Sequential Dense Matrices

PETSc provides both sequential and parallel dense matrix formats, where each processor stores its entries in a column-major array in the usual Fortran77 style.

6.0.7 Parallel Dense Matrices

The parallel dense matrices are partitioned by rows across the processors, so that each local rectangular submatrix is stored in the dense format described above.