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 3.7

April 2016

This work was supported in part by the Office of Advanced Scientific Computing Research,
Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Abstract:

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

The text assumes that you are familiar with PETSc and have access to PETSc source code and documentation (available via http://www.mcs.anl.gov/petsc).

Before contributing code to PETSc, please read Chapter 2, which contains the source code style guide. http://www.mcs.anl.gov/petsc/developers/index.html contains information on how to submit patches and pull requests to 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`.

Contents

Chapter 1

Answering petsc-maint@mcs.anl.gov and petsc-users@mcs.anl.gov

- Try to be polite. (This is not always easy.)
- Address the person by name (when it is possible to determine their name).
- Apologize for the problem when it is appropriate (but not otherwise).
- Thank the person for their patience if it is more than six hours since the report came in.
- If the person drops the `petsc-maint` or `petsc-users` from the reply list, add it back in.
- Often, it pays to not ask too many questions or give too many suggestions in the same email. The user often only responds to the first of them.

Chapter 2

Style Guide

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

2.1 Names

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

1. All function names and enum types consist of words, each of which is capitalized, for example `KSPSolve()` and `MatGetOrdering()`.
2. All enum elements and macro variables are named with all capital letters. When they consist of several complete words, there is an underscore between each word. For example, `MAT_FINAL_ASSEMBLY` or `PETSC_USE_COMPLEX`.
3. 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 `_Subtype` (for example, `MatMult_SeqAIJ`).

   In addition, functions that are not intended for use outside of a particular file are declared `static`. Also see item 14 in Section 2.2.3.
4. 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()`.
5. Each application usable function begins with the name of the class object, followed by any subclass name, for example, `ISInvertPermutation()`, `MatMult()` or `KSPGMRESSetRestart()`.
6. Functions that PETSc provides as defaults for user providable functions end with `Default` (for example, `KSPMonitorDefault()` or `PetscSignalHandlerDefault()`)
7. Options database keys are lower case, have an underscore between words, and match the function name associated with the option without the word "set" or "get". For example, `-ksp_gmres_restart`.
8. `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`.

2.2 Coding Conventions and Style

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

2.2.1 C Formatting

1. All PETSc function bodies are indented two characters. *No literal tabs* should be used.
2. Each additional level of loops, `if` statements, etc. is indented two more characters.
3. Wrapping lines should be avoided whenever possible.
4. Source code lines do not have a hard length limit; generally, we like them less than 150 characters wide.
5. The local variable declarations should be aligned. For example, use the style

```
PetscScalar a;
PetscInt    i,j;
```

   instead of

```
PetscScalar a;
PetscInt i,j; /* Incorrect */
```

6. The prototypes for functions should not include the names of the variables; for example write

```
PetscErrorCode MyFunction(PetscInt);
```

   not

```
PetscErrorCode MyFunction(PetscInt myvalue); /* Incorrect */
```

7. 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.
8. Equal signs should be aligned in regions where possible.
9. There *must* be a single blank line between the local variable declarations and the body of the function.
10. Indentation for `if` statements *must* be done as as

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

11. *Never* have

```
if ( )
  a single indented line /* Incorrect */
```

or

```
for ( )
  a single indented line /* Incorrect */
```

instead use either

```
if ( ) a single statement
```

or

```
if ( ) {
  a single indented line
}
```

Note that error checking is a separate statement, so the following is *incorrect*

```
if ( ) ierr = XXX();CHKERRQ(ierr); /* Incorrect */
```

and instead one should use

```
if ( ) {
  ierr = XXX();CHKERRQ(ierr);
}
```

12. Always have a space between `if` or `for` and the following `()`.
13. *No* tabs are allowed in *any* of the source code.
14. The open brace should be on the same line as the `if ( )` test, `for ( )`, etc., never on its own line. For example

```
 } else {
```

never

```
 }
 else { /* Incorrect */
```

See item 15 for an exception. The closing brace should *always* be on its own line.

15. In function declarations, the opening brace should be on the *next* line, not on the same line as the function name and arguments. This is an exception to item 14.
16. Do not leave chunks of commented-out code in the source files.
17. Do not use C++-style comments (`// Comment`). Use only C-style comments (`/* Comment */`).
18. Do not include a space after a `(` or before a `)`. Do not write

```
ierr = PetscMalloc1( 10,&a );CHKERRQ(ierr); /* Incorrect */
```

but instead write

```
ierr = PetscMalloc1(10,&a);CHKERRQ(ierr);
```

19. Do not use a space after the) in a cast, or between the type and the * in a cast.

20. Do not include a space before or after a comma in lists. That is, do not write

```
int a,b,c;
ierr = func(a, 22.0);CHKERRQ(ierr); /* Incorrect */
```

but instead write

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

2.2.2 C Usage

1. Array and pointer arguments where the array values are not changed should be labeled as `const` arguments.

2. Scalar values passed to functions should *never* be labeled as `const`.

3. Subroutines that would normally have a `void**` argument to return a pointer to some data should actually be prototyped as `void*`. This prevents the caller from having to put a `(void**)` cast in each function call. See, for example, `DMDAVecGetArray()`.

4. Do not use the `register` directive.

5. Never use a local variable counter like `PetscInt flops = 0;` to accumulate flops and then call `PetscLogFlops();` *always* just call `PetscLogFlops()` directly when needed.

6. Do not use `if (rank == 0)` or `if (v == NULL)` or `if (flg == PETSC_TRUE)` or `if (flg == PETSC_FALSE)`. Instead, use `if (!rank)` or `if (!v)` or `if (flg)` or `if (!flg)`.

7. Do not use `#ifdef` or `#ifndef`. Rather, use `#if defined(...` or `#if !defined(...`

2.2.3 Usage of PETSc Functions and Macros

1. Public PETSc includes, `petsc*.h`, should not include private PETSc `petsc/private/*impl.h` includes.

2. Public and private PETSc includes cannot include include files in the PETSc source tree.

3. The first line of the executable statements in a function must be `PetscFunctionBegin;`

4. Use `PetscFunctionReturn(returnvalue);` not `return(returnvalue);`

5. *Never* put a function call in a `return` statement; do not do

```
PetscFunctionReturn( somefunction(...) ); /* Incorrect */
```

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

7. Do not use `sqrt()`, `pow()`, `sin()`, etc. directly in PETSc source code or examples. Rather, use `PetscSqrtScalar()`, `PetscSqrtReal()`, etc., depending on the context. See `petscmath.h` for expressions to use.

8. `assert.h` should not be included in PETSc source and `assert()` should not be used. It doesn't play well in the parallel MPI world.

9. 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.

10. Do not include a space before `CHKXXX()`. That is, do not write

```
ierr = PetscMalloc1(10,&a); CHKERRQ(ierr); /* Incorrect */
```

    but instead write

```
ierr = PetscMalloc1(10,&a);CHKERRQ(ierr);
```

11. Except in code that may be called before PETSc is fully initialized, always use `PetscMallocN()` (for example `PetscMalloc1()`), `PetscCallocN()`, `PetscNew()`, and `PetscFree()`, not `malloc()` and `free()`.

12. 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. Code should also be provided that works if the MPI feature is not available. For example,

```
#if defined(PETSC_HAVE_MPI_IN_PLACE)
  ierr = MPI_Allgatherv(MPI_IN_PLACE,0,MPI_DATATYPE_NULL,lens,
                        recvcounts,displs,MPIU_INT,comm);CHKERRQ(ierr);
#else
  ierr = MPI_Allgatherv(lens,sendcount,MPIU_INT,lens,recvcounts,
                        displs,MPIU_INT,comm);CHKERRQ(ierr);
#endif
```

13. 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.

14. Library functions should be declared `PETSC_INTERN` if they are intended to be visible only within a single shared library. They should be declared `PETSC_EXTERN` if intended to be visible across shared libraries. Note that PETSc can be configured to build a separate shared library for each top-level class (`Mat`, `Vec`, `KSP`, etc.) and that plugin implementations of these classes can be included as separate shared libraries; thus, private functions may be marked `PETSC_EXTERN`. For example,

    - `MatStashCreate_Private` is marked `PETSC_INTERN` as it is used across compilation units, but only within the `Mat` package.
    - All functions, such as `KSPCreate()`, included in the public headers (`include/petsc*.h`) should be marked `PETSC_EXTERN`.

- `MatHeaderReplace()` is not intended for users (it is in `include/petsc/private/matimpl.h`) but is marked `PETSC_EXTERN` since it is used both by implementations of the `Mat` class (which could be defined in plugin implementations) and by functions in the `DM` and `KSP` packages.

2.3 Formatted Comments

PETSc uses formatted comments and the Sowing packages to generate documentation (manual pages) and the Fortran interfaces. Documentation for Sowing and the formatting may be found at http://wgropp.cs.illinois.edu/projects/software/sowing/; in particular, see the documentation for `doctext`.

- `/*@`
  indicates a formatted comment of a function that will be used for both documentation and a Fortran interface.
- `/*@C`
  a formatted comment of a function that will be used only for documentation
- `/*E`
  a formatted comment of an enum used for documentation only, note that each of these needs to be listed in `lib/petsc/conf/bfort-petsc.txt` as a native and defined in the corresponding `include/petsc/finclude/petscxxx.h` fortran include file and the values set as parameters in the file `include/petsc/finclude/petscxxx.h`
- `/*S`
  a formatted comment for a data type such as `KSP`, note that each of these needs to be listed in `lib/petsc/conf/bfort-petsc.txt` as a `nativeptr`.
- `/*M`
  a formatted comment of a CPP macro used for both documentation and a Fortran interface.
- `/*MC`
  a formatted comment of a CPP macro for documentation.

Functions that take `char*` or function pointer arguments must have the `C` symbol. The Fortran interface generator cannot handle them, so the Fortran interface for them must be created manually.

The Fortran interface files go into the three directories depending on how they are created: `ftn-auto`, `ftn-custom`, `ftn-f90`.

2.3.1 Man Page Format

Each function, typedef, class, macro, enum, etc. in the public API should include the following data, correctly formatted in a block (see above) to generate complete man pages and Fortran interfaces with Sowing. All entries below should be separated by blank lines.

- The item's name, followed by a dash and brief (one-sentence) description
- (Optional for simple items) A longer description of the function. This should include literature references if relevant.
- If documenting a function, a description of the function's "collectivity" (whether all ranks in an MPI communicator need to participate)

  - `Not Collective` if the function need not be called on all MPI ranks
  - `Collective [on XXX]` if the function is a collective operation [with respect to the data of class `XXX`]
  - `Logically Collective [on XXX]` if the function is collective but does not require any actual synchronization (say, setting class parameters uniformly).

- If documenting a function with input parameters, a list of input parameter descriptions in an `Input Parameters:` section
- If documenting a function with output parameters, a list of output parameter descriptions in an `Output Parameters:` section
- If documenting a function which interacts with the options database, a list of options database keys in an `Options Database Keys:` section
- (Optional) a `Notes:` section. In-depth discussion, technical caveats, special cases, and so on should be listed here. If it is ambiguous whether returned pointers need to be freed by the user or not, this information should be mentioned here.
- (If applicable) a `Fortran Notes:` section, detailing any relevant differences in calling or using the item.
- `Level:` followed by `beginner`, `intermediate`, `advanced`, or `developer`
- (Optional) a list of `Concepts:`
- (Optional) a list of `Keywords:`
- The `.seealso:` keyword and a list of related man pages. These man pages should usually also point back to this man page.

Chapter 3

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
- an options database
- basic objects for viewing and drawing.

Each of these is discussed in a section below.

3.1 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 storage of logical true and false

PetscInt can be set using `./configure` to be either `int` (32 bit, the default) or `long long` (64 bit, with `configure --with-64-bit-indices`) to allow indexing into very large arrays. PetscMPIInt are used for integers passed to MPI as counts, etc. These are always `int` since that is what the MPI standard uses. Similarly, PetscBLASInt is 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 system and with Intel's MKL).

In addition, there are special types:

- PetscClassId

- PetscErrorCode
- PetscLogEvent

These are currently always `int` but their use clarifies the code.

3.2 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, and the source code for the PETSc error handling is in `src/sys/error/`.

3.2.1 Simplified Interface

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

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

The macro SETERRQ() is given by

```
return PetscError(comm,__LINE__,PETSC_FUNCTION_NAME,__FILE__,error code,error
   type,"Error message");
```

It calls the error handler with the current function name and location: line number, file and directory, plus an error code and an error message. Normally `comm` is PETSC_COMM_SELF; it can only be another communicator if one is absolutely sure the same error will be generated on all processes in the communicator. This is to prevent the same error message from being printed by many processes. The `error type` is PETSC_ERROR_INITIAL on detection of the initial error and PETSC_ERROR_REPEAT for any additional calls. This is so that the detailed error information is only printed once instead of for all levels of returned errors.

The macro CHKERRQ() is defined by

```
if (ierr) PetscError(PETSC_COMM_SELF,__LINE__,PETSC_FUNC__,__FILE__,
                     ierr,PETSC_ERROR_REPEAT," ");
```

In addition to SETERRQ(), there are macros SETERRQ1(), SETERRQ2(), SETERRQ3() and SETERRQ4() that allow one to provide additional arguments to a formatted message string. For example,

```
SETERRQ2(comm,PETSC_ERR,"Iteration overflow: its \%d norm \%g",its,norm);
```

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

3.2.2 Error Handlers

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

```
PetscErrorCode PetscError(MPI_Comm,int line,const char *func,const char *file,
                          const char *dir,error code,error type,
                          const char *mess)
{
  PetscErrorCode ierr;
```

```
  PetscFunctionBegin;
  if (!eh) ierr = PetscTraceBackErrorHandler(line,func,file,dir,error code,
                                             error type,mess,0);
  else     ierr = (*eh->handler)(line,func,file,dir,error code,error type,
                                 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 {
  PetscErrorCode handler(MPI_Comm,int,const char*,const char*,const char*,
                         PetscErrorCode,PetscErrorType,const char*,void*);
  void           *ctx;
  EH             previous;
};
```

One can set a new error handler with the command `PetscPushErrorHandler()`, which maintains a linked list of error handlers. The most recent error handler is removed via `PetscPopErrorHandler()`.

PETSc provides several default error handlers:

- `PetscTraceBackErrorHandler()`,
- `PetscAbortErrorHandler()`, called with `-on_error_abort`,
- `PetscReturnErrorHandler()`,
- `PetscEmacsClientErrorHandler()`,
- `PetscMPIAbortErrorHandler()`, and
- `PetscAttachDebuggerErrorHandler()`, called with `-on_error_attach_debugger`.

3.2.3 Error Codes

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

3.2.4 Detailed Error Messages

In a modern parallel component-oriented application code, it does not make sense to simply print error messages to the screen (and 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, they should be printed with `(*PetscErrorPrintf)()`. One can direct all error messages to stderr with the command line options `-error_output_stderr`.

3.3 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/.

3.3.1 Profiling Object Creation and Destruction

The creation of objects is profiled with the command PetscLogObjectCreate()

```
PetscLogObjectCreate(PetscObject h);
```

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

```
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().

```
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().

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

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

3.3.2 Profiling Events

Events are logged using the pair PetscLogEventBegin()

```
PetscLogEventBegin(PetscLogEvent event,PetscObject o1,...,PetscObject o4);
PetscLogEventEnd(PetscLogEvent event,PetscObject o1,...,PetscObject o4);
```

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

3.3.3 Controlling Profiling

Routines that control the default profiling available in PETSc include:

- PetscLogDefaultBegin();
- PetscLogAllBegin();
- PetscLogDump(const char *filename);
- PetscLogView(PetscViewer);

These routines are normally called by the PetscInitialize() and PetscFinalize() routines when the option -log_view is given.

Chapter 4

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.

4.1 Introduction

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

1. A (small) common part shared by all PETSc classes (for example both `KSP` and `PC` have this same header),
2. another common part shared by all PETSc implementations of the class (for example both `KSP_GMRES` and `KSP_CG` have this common sub-header), and
3. a private part used by only one particular implementation written in PETSc.

For example, all matrix (`Mat`) classes 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.

4.2 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`. This is how polymorphism is supported with 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 (Also see Section 2.1).
- One or more actual implementations of the class (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.

4.3 Common Object Header

All PETSc/PETSc objects have the following common header structures defined in `include/petsc/private/petscimpl.h`:

Listing 4.1: Function table common to all PETSc compatible classes

```
typedef struct {
  PetscErrorCode (*getcomm)(PetscObject,MPI_Comm*);
  PetscErrorCode (*view)(PetscObject,Viewer);
  PetscErrorCode (*destroy)(PetscObject);
  PetscErrorCode (*query)(PetscObject,const char*,PetscObject*);
  PetscErrorCode (*compose)(PetscObject,const char*,PetscObject);
  PetscErrorCode (*composefunction)(PetscObject,const char*,void(*)(void));
  PetscErrorCode (*queryfunction)(PetscObject,const char*,void (**)(void));
} PetscOps;
```

Listing 4.2: Data structure header common to all PETSc compatible classes

```
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 operations in include/petsc/private/vecimpl.h include the following:

Listing 4.3: Function table common to all PETSc compatible vector objects (truncated)

```
typedef struct _VecOps* VecOps;
struct _VecOps {
  PetscErrorCode (*duplicate)(Vec,Vec*); /* get single vector */
  PetscErrorCode (*duplicatevecs)(Vec,PetscInt,Vec**); /* get array of vectors */
  PetscErrorCode (*destroyvecs)(PetscInt,Vec[]); /* free array of vectors */
  PetscErrorCode (*dot)(Vec,Vec,PetscScalar*); /* z = x^H * y */
  PetscErrorCode (*mdot)(Vec,PetscInt,const Vec[],PetscScalar*); /* z[j] = x dot
     y[j] */
  PetscErrorCode (*norm)(Vec,NormType,PetscReal*); /* z = sqrt(x^H * x) */
  PetscErrorCode (*tdot)(Vec,Vec,PetscScalar*); /* x'*y */
  PetscErrorCode (*mtdot)(Vec,PetscInt,const Vec[],PetscScalar*);/* z[j] = x dot
     y[j] */
  PetscErrorCode (*scale)(Vec,PetscScalar);  /* x = alpha * x   */
  PetscErrorCode (*copy)(Vec,Vec); /* y = x */
  PetscErrorCode (*set)(Vec,PetscScalar); /* y = alpha  */
  PetscErrorCode (*swap)(Vec,Vec); /* exchange x and y */
  PetscErrorCode (*axpy)(Vec,PetscScalar,Vec); /* y = y + alpha * x */
  PetscErrorCode (*axpby)(Vec,PetscScalar,PetscScalar,Vec); /* y = alpha * x +
     beta * y*/
  PetscErrorCode (*maxpy)(Vec,PetscInt,const PetscScalar*,Vec*); /* y = y +
     alpha[j] x[j] */
  ... (ETC.) ...
};
```

Listing 4.4: Data structure header common to all PETSc vector classes

```
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 */
  PetscLayout            map;
  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 `classid`; these are used to distinguish between classes. When a new class is created one needs to call

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

For example,

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

One can verify that an object is valid of a particular class with `PetscValidHeaderSpecific`, for example

```
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 to generate more complete error messages.

To check for an object of any type use

```
PetscValidHeader(x,1);
```

4.4 Common Object Functions

Several routines are provided for manipulating data within the header. These include 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,const 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 an 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,const char *name,PetscObject*)` retrieves an object that was previously composed with the first object. Retrieves a null if no object with that name was previously composed.
- `composefunction(PetscObject,const char *name,const 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 replaced. If `func` is `NULL` the existing function 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,const char *name,void **func)` retrieves 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.

4.5 Object Function Implementation

This section 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."

4.5.1 Compose and Query

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

```
typedef struct _PetscObjectList* PetscObjectList;
struct _PetscObjectList {
    char             name[128];
    PetscObject      obj;
    PetscObjectList  next;
};
```

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

```
int PetscObjectListAdd(PetscObjectList *fl,const char *name,PetscObject obj);
int PetscObjectListDestroy(PetscObjectList fl);
int PetscObjectListFind(PetscObjectList fl,const char *name,PetscObject *obj)
int PetscObjectListDuplicate(PetscObjectList fl,PetscObjectList *nl);
```

The function PetscObjectListAdd() will create the initial PetscObjectList 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,const char
    *name,PetscObject ptr)
{
  PetscErrorCode ierr;

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

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

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

4.5.2 Compose and Query Function

PETSc allows one to compose functions by specifying a name and function pointer. In src/sys/dll/reg.c, PETSc defines the linked list structure

```
struct _n_PetscFunctionList {
  void              (*routine)(void);    /* the routine */
  char              *name;               /* string to identify routine */
  PetscFunctionList next;                /* next pointer */
  PetscFunctionList next_list;           /* used to maintain list of all lists
     for freeing */
};
```

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

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

```
  PetscFunctionBegin;
  ierr = PetscFunctionListAdd(&obj->qlist,name,fname,ptr);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}

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

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

In addition to using the PetscFunctionList mechanism to compose functions into PETSc objects, it is also used to allow registration of new class implementations; for example, new preconditioners - see Section 5.2.4.

4.5.3 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 example, _n_PetscTable vs _p_Vec.

Chapter 5

PetscObjects

5.1 Elementary Objects: IS, Vec, Mat

5.2 Solver Objects: PC, KSP, SNES, TS

5.2.1 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`.

5.2.2 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`.

5.2.3 ODE and DAE Solvers (Timesteppers): TS

The base PETSc `TS` object is defined in the `include/petsc/private/tsimpl.h` include file.

5.2.4 Registering New Methods

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

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.

6.1 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); and

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

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, 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.

6.1.1 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).

6.1.2 Parallel AIJ Sparse Matrices

This matrix type, which is the default parallel matrix format; additional implementation details are given in [1].

6.1.3 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 nnz=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.

6.1.4 Parallel Block AIJ Sparse Matrices

Parallel block AIJ matrices with block size nb can be created with the command MatCreateBAIJ()

```
ierr = MatCreateBAIJ(MPIComm 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 PETSC_DECIDE 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=NULL, and o_nnz=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.

6.1.5 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.

6.1.6 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.

Chapter 7

PETSc Testing System

The PETSc test system consists of:

1. A language contained within the source files that describes the tests to be run
2. The *test generator* (`config/gmakegentest.py`) that at the `make` step parses the source files and generates the makefiles and shell scripts that compose:
3. The *petsc test harness*: a harness consisting of makefile and shell scripts that runs the executables with several logging and reporting features.

7.1 PETSc Test description language

PETSc tests and tutorials contain within their file a simple language to describe tests and subtests required to run executables associated with compilation of that file. The general skeleton of the file is:

```
static char help[] = "A simple MOAB example\n\

...
<source code>
...

/*TEST
   build:
     requires: moab
   testset:
     suffix: 1
     requires: !complex
   testset:
     suffix: 2
     args: -debug -fields v1,v2,v3
     test:
     test:
       args: -foo bar
TEST*/
```

For our language, a *test* is associated with a:

Single shell script

Single makefile

Single output file that represents the *expected results*.

Two or more command tests. Usually, one (or more) mpiexec test that runs the executable, and one (or more) diff tests to compare output with the expected result.

Our language also supports that a *testset* that specifies specifies either a new test entirely, or multiple executable/diff tests within a single test. At the core, the executable/diff test combination will look something like this:

```
mpiexec -n 1 ../ex1 1> ex1.tmp 2> ex1.err
diff ex1.tmp output/ex1.out 1> diff-ex1.tmp 2> diff-ex1.err
```

In practice, we want to do various logging and counting by the test harness, but this is explained further below. The input language supports a simple, yet flexible, tests control and we begin by describing this language.

7.1.1 Runtime language options

At the end of each test, a marked comment block that uses YAML is inserted that describes the test to be run. The elements of the test are done with a set of supported key words that sets up the test.

The goals of the language are to:

1. Be as minimal as possible with the simplest test requiring only one keyword
2. Be independent of the filename such that a file can be renamed without rewriting the tests
3. Be intuitive

To enable the second bullet, the *basestring* of the filename is defined as the filename without the extention; i.e., if the filename is `ex1.c` then `basestring=ex1`.f
With this background, these keywords are are:

testset or **test**: (*Required*)

At the top level, either a single test, or a testset must be specified. All other are subsets of this keyword.

suffix: (*Optional*; *Default:* `suffix=''`)

- The testname is given by: `testname='run'+basestring` if suffix is set to an empty string, and by `testname='run'+basestring+'_'+suffix`
- This can only be specified for top level test nodes

output_file: (*Optional*; *Default:* `output_file=testname+'.out'`)

- The output of the test is to be compared to an *expected result* whose name is given by output_file.
- This file is described relative to the source directory of the source file and should be in the output subdirectory (e.g., `output/ex1.out`)

nsize: (*Optional*; *Default:* `nsize=1`)

- The integer that is passed to mpiexec; i.e., `mpiexec -n nsize`

args: (*Optional*; *Default:* `""`)

- The arguments to pass to the executable

TODO: (*Optional*; *Default:* `False`)

- Setting this boolean to True will tell the test to appear in the test harness, but report only TODO per the TAP standard.
- A runscript will be generated and can easily be modified by hand to run

filter: (*Optional*; *Default:* `""`)

- Sometimes only a subset of the output is meant to be tested against the expected result. If this keyword is used, it processes the executable output and puts it into the file to be actually compared with output_file.
- The value of this is the command to be run; e.g., `grep foo` or `sort -nr`
- A skeleton example of the resultant commands to be run is:

```
mpiexec -n 1 ../ex1 | grep residual 1> ex1.tmp 2>
   ex1.err
diff ex1.tmp output/ex1.out 1> diff-ex1.tmp 2>
   diff-ex1.err
```

- If the filter begins with `Error:`, then the test is assumed to be testing the error output, and the error code and output is set up to be tested.

filter_output: (*Optional*; *Default:* `""`)

- Sometimes filtering the output is useful for standardizing tests; for example, to handle the issues related to parallel output. This works the same as filter to implement this feature

localrunfiles: (*Optional*; *Default:* `""`)

- The tests are run under `PETSC_ARCH/tests`, but some tests require runtime files that are maintained in the source tree. Files in this (space-delimited) list will be copied over.
- The copying is done by the file generator and not by creating makefile dependencies.

requires: (*Optional*; *Default:* `""`)

- A space-delimited list of run requirements (not build requirements. See Build requirements below)
- In general, the language supports `and` and `not` constructs using `! => not` and `, => and`
- MPIUNI should work for all -n 1 examples so this need not be in the requirements list
- Inputs sometimes include external matrices that are found in the DATAFILES path. `requires: DATAFILES` can be specifed for these tests.

- Packages are specified with lower case specification; e.g., `requires: superlu\_dist`
- Any defined variable in petscconf.h can be specified with the `defined(...)` syntax; e.g., `defined(PETSC_USE_INFO)`
- Any define of the form `PETSC_HAVE_FOO` can just use `requires: foo` similar to how third-party packages are handled.

timeoutfactor: (*Optional*; *Default:* `"1"`)

- Tests are limited to a set time that is found at the top of `"config/petsc_harness.sh"`. and can be overwritten by passing in the `TIMEOUT` argument to `gmakefile` (see `make -f gmakefile help`.
- This parameters allows one to extend the default timeout for an individual test such that the new timeout time is `timeout=(default timeout) x (timeoutfactor)`.

7.1.2 Additional specifications

In addition to the above keywords, other language features are supported:

- for loops: Specifying {{...}*sharedoutput*} or {{...}*separateoutput*} will create for loops over enclosed space-delmited list. If the loop causes a different output, then separate output would be used. If the loop does not cause separate output, then the shared (or {{...}}) syntax must be used.
  For loops are supported within nsize and args. An example would be:

  ```
  args: -matload_block_size {{2,3}}
  ```

  In this case, two execution lines would be addded with two different arguments. Associated `diff` lines would be added as well automatically. See examples below for how it works in practice.

7.1.3 Test block examples

This is the simplest test block:

```
/*TESTS
  test:
TESTS*/
```

which is equivalent to:

```
/*TESTS
  testset:
    test:
TESTS*/
```

which is equivalent to:

```
/*TESTS
  testset:
TESTS*/
```

If this block is in ex1.c, then it will create a `runex1` test that requires only one processor/thread, with no arguments, and diff the resultant output with `output/ex1.out`.

For fortran, the equivalent is:

```
!/*TESTS
!  test:
!TESTS*/
```

A fuller example would be:

```
/*TESTS
  test:
  test:
    suffix: 1
    nsize: 2
    args:  -t 2 -pc_type jacobi -ksp_monitor_short -ksp_type gmres
       -ksp_gmres_cgs_refinement_type refine_always -s2_ksp_type bcgs
       -s2_pc_type jacobi -s2_ksp_monitor_short
    requires: x
*/TESTS
```

This creates two tests. Assuming that this is `ex1.c`, the tests would be `runex1` and `runex1\_1`.

An example of how one tests a permutuation of arguments against the same output file:

```
/*TESTS
  testset:
      suffix: 19
      requires: datafilespath
      args: -f0 DATAFILESPATH/matrices/poisson1args:  -ksp_type cg -pc_type
         icc -pc_factor_levels 2test:test:args:  -mat_type seqsbaij*/TESTS
```

Assuming that this is `ex10.c`, there would be two mpiexec/diff invocations in `runex10_19.sh`.

Here is a similar example, but the permation of arguments create different output:

```
/*TESTS
   testset:
      requires: datafilespath
      args: -f0 DATAFILESPATH/matrices/mediumargs:  -ksp_type
         bicgtest:suffix:  4args:  -pc_type lutest:suffix:  5*/TESTS
```

Assuming that this is `ex10.c`, there would be two shell scripts created: `runex10_4.sh` and `runex10_5.sh`.

An example using a for loop would be:

```
/*TESTS
  testset:
       suffix: 1
       args:   -f DATAFILESPATH/matrices/small -mat_type aijrequires:
          datafilespathtestset:suffix:  2output_file:
          output/ex138_1.outargs:  -f {DATAFILESPATH}/matrices/small
       args: -mat\_type baij -matload\_block\_size {{2,3}shared output}
       requires: datafilespath
*/TESTS
```

In this example, runex138_2 will invoke ex138 twice with two different arguments, but both are diffed with the same file.

An example for showing the hierarchial nature of the test specification is:

```
testset:
  suffix:2
  output\_file: output/ex138\_1.out
  args: -f DATAFILESPATH/matrices/small -mat_type baijtest:args:
     -matload_block_size 2test:args:  -matload_block_size 3
```

This is functionally equivalent to the for loop shown above.

Here is a more complex example using for loops:

```
testset:
   suffix: 19
   requires: datafilespath
   args: -f0 DATAFILESPATH/matrices/poisson1args:  -ksp_type cg -pc_type
      iccargs:  -pc_factor_levels 0 2 4separate outputtest:test:args:
      -mat_type seqsbaij
```

If this is in `ex10.c`, then the shell scripts generated would be:

```
runex10\_19\_pc\_factor\_levels-0.sh
runex10\_19\_pc\_factor\_levels-2.sh
runex10\_19\_pc\_factor\_levels-4.sh
```

Each shell script would invoke mpiexec twice.

7.1.4 Build language options

It is possible to specify issues related to the compilation of the source file with the `build:` block. The language is:

requires: (Optional; *Default* 1) Same as the runtime requirements (e.g., can include requires: fftw) but also requirements related to types: A. Precision types: single, double, quad, int32 B. Scalar types: complex (and !complex)

In addition, `TODO` is available to allow one to skip the build of this file but still maintain it in the source tree.

depends: (Optional; *Default* 1) List any dependencies required to compile the file

A typical example for compiling for real/double only is:

```
/*T
  requires: !complex
T*/
```

7.2 PETSC Test Harness

The goals of the PETSc Test Harness are to:

- Provide standard output used by other testing tools

- Lightweight as possible and easily fit within the PETSc build chain
- Provide information on all tests, even those that are not built or run because they do not meet the configuration requirements

Before understanding the test harness, it is first important to understand the desired requirements for reporting and logging.

7.2.1 Testing the parsing

After inserting the language into the file, one can test the parsing by executuing:

```
${PETSC_DIR}/bin/maint/testparse.py -t <test src file>
```

A dictionary will be pretty-printed out. From this dictionary print-out, it is usually obvious if there is a problem in the parsing. This python file is used by

```
${PETSC_DIR}/config/gmakegentest.py
```

in generating the test harness.

7.3 Test output standards: TAP

The PETSc test system is designed to be compliant with the Test Anything Protocal (TAP): See https://testanything.org/tap-specification.html

This is a very simple standard designed to allow testing tools to work together easily. There are libraries to enable the output to be used easily including sharness, which is used by the git team. However, the simplicity of the petsc tests and TAP specification means that we use our own simple harness given by a single shell script that each file sources: `petsc\_harness.sh`.

As an example, consider this test input:

```
test:
    suffix: 2
    output_file: output/ex138.out
    args: -f ${DATAFILESPATH}/matrices/small -mat_type {{aij,baij,sbaij}} -matload_block_s
    requires: datafilespath
```

A sample output would be:

```
ok 1 In mat...tests: "./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type aij -matload_
ok 2 In mat...tests: "Diff of ./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type aij -
ok 3 In mat...tests: "./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type aij -matload_
ok 4 In mat...tests: "Diff of ./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type aij -
ok 5 In mat...tests: "./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type baij -matload
ok 6 In mat...tests: "Diff of ./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type baij
...

ok 11 In mat...tests: "./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type saij -matloa
ok 12 In mat...tests: "Diff of ./ex138 -f ${DATAFILESPATH}/matrices/small -mat_type aij
```

7.4 Test harness implementation

Most of the requirements for being TAP-compliant lie in the shell scripts so we focus on that description.

A sample shell script is given by:

```
#!/bin/sh
. petsc_harness.sh

petsc_testrun ./ex1 ex1.tmp ex1.err
petsc_testrun 'diff ex1.tmp output/ex1.out' diff-ex1.tmp diff-ex1.err

petsc_testend
```

`petsc_harness.sh` is a small shell script that provides the logging and reporting functions `petsc_testrun` and `petsc_testend`.

A small sample of the output from the test harness would be:

```
ok 1 ./ex1
ok 2 diff ex1.tmp output/ex1.out
not ok 4 ./ex2
#ex2: Error: cannot read file
not ok 5 diff ex2.tmp output/ex2.out
ok 7 ./ex3 -f /matrices/small -mat_type aij -matload_block_size 2
ok 8 diff ex3.tmp output/ex3.out
ok 9 ./ex3 -f /matrices/small -mat_type aij -matload_block_size 3
ok 10 diff ex3.tmp output/ex3.out
ok 11 ./ex3 -f /matrices/small -mat_type baij -matload_block_size 2
ok 12 diff ex3.tmp output/ex3.out
ok 13 ./ex3 -f /matrices/small -mat_type baij -matload_block_size 3
ok 14 diff ex3.tmp output/ex3.out
ok 15 ./ex3 -f /matrices/small -mat_type sbaij -matload_block_size 2
ok 16 diff ex3.tmp output/ex3.out
ok 17 ./ex3 -f /matrices/small -mat_type sbaij -matload_block_size 3
ok 18 diff ex3.tmp output/ex3.out
# FAILED   4 5
# failed 2/16 tests; 87.500% ok
```

For developers, modifying the lines that get written to the file can be done by modifying:

```
${PETSC_DIR}/config/example_template.py
```

To modify the test harness, one can modify this file

```
${PETSC_DIR}/config/petsc_harness.sh
```

Bibliography

[1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.