

PETSc Developers Guide

Revision 3.11

Mathematics and Computer Science Division

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free at OSTI.GOV (<http://www.osti.gov/>), a service of the US Dept. of Energy's Office of Scientific and Technical Information.

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: orders@ntis.gov

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: reports@osti.gov

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

PETSc Developers Manual

Revision 3.11

Prepared by
S. Kruger¹, P. Sanan², and B. Smith³

¹Tech-X Corporation

²Institute of Geophysics, ETH Zurich

³Mathematics and Computer Science Division, Argonne National Laboratory

March 2019

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

Abstract

PETSc is an extensible software library for scientific computation. This document provides information for PETSc developers and those wishing to contribute to PETSc. 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>) including the PETSc users manual [2]. Higher-level views of PETSc can be found in [10], [1], [8], [5], and [4].

Before contributing code to PETSc, please read Chapter 2, which contains the source code style guide. Information on how to submit patches and pull requests to PETSc can be found at <http://www.mcs.anl.gov/petsc/developers/index.html>.

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 be directed to `petsc-maint@mcs.anl.gov`.

Contents

1	Responding to User Requests and Questions	9
2	Style Guide	10
2.1	Names	10
2.2	Coding Conventions and Style	11
2.2.1	C Formatting	11
2.2.2	C Usage	13
2.2.3	Usage of PETSc Functions and Macros	14
2.3	Formatted Comments	16
3	The PETSc Kernel	18
3.1	PETSc Types	18
3.2	Implementation of Error Handling	19
3.2.1	Simplified Interface	19
3.2.2	Error Handlers	19
3.2.3	Error Codes	20
3.2.4	Detailed Error Messages	20
3.3	Memory Management	21
3.4	Implementation of Profiling	22
3.4.1	Profiling Object Creation and Destruction	22
3.4.2	Profiling Events	22
3.4.3	Controlling Profiling	22
4	Basic Object Design and Implementation	23
4.1	Introduction	23
4.2	Organization of the Source Code	23
4.3	Common Object Header	24
4.4	Common Object Functions	26
4.5	Object Function Implementation	27
4.5.1	Compose and Query Objects	27
4.5.2	Compose and Query Functions	28
4.5.3	Simple PETSc Objects	29
4.6	PETSc Packages	29
5	How the Solvers Handle User Provided Callbacks	30

6	The Various Matrix Classes	35
6.1	Matrix Blocking Strategies	35
6.2	Assorted Matrix Types	36
6.2.1	Sequential AIJ Sparse Matrices	36
6.2.2	Parallel AIJ Sparse Matrices	36
6.2.3	Sequential Block AIJ Sparse Matrices	37
6.2.4	Parallel Block AIJ Sparse Matrices	37
6.2.5	Sequential Dense Matrices	37
6.2.6	Parallel Dense Matrices	37
7	PETSc Testing System	38
7.1	PETSc Test Description Language	38
7.1.1	Runtime Language Options	39
7.1.2	Additional Specifications	41
7.1.3	Test Block Examples	41
7.1.4	Build Language Options	43
7.2	PETSC Test Harness	44
7.3	Test Output Standards: TAP	44
7.4	Test Harness Implementation	45
7.5	Using the Test Harness for Regression Testing	46
7.5.1	Additional Tips	47

Chapter 1

Responding to User Requests and Questions

PETSc users communicate to the PETSc team via two mailing lists: `petsc-maint@mcs.anl.gov` and `petsc-users@mcs.anl.gov`. When responding to such inquiries, please follow the guidelines given below.

- Be polite.
- Address the person by name (when you can determine the name).
- Apologize for the problem when it is appropriate (but not otherwise).
- Thank the person for his or her 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.
- Don't ask too many questions or give too many suggestions in the same email. The user often responds only to the first of them or becomes confused.

Chapter 2

Style Guide

The PETSc team uses certain conventions to make the source code consistent and hence easier to maintain. We will interchangeably use the terminology *subclass*, *implementation*, or *type*¹ to refer to a concrete realization of an abstract base class. For example, `KSPGMRES` is a type for the base class `KSP`.

2.1 Names

Consistency of names for variables, functions, and so on is extremely important. We use several conventions

1. All function names and enum types consist of acronyms or 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`.
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 15 in Section 2.2.3.

4. Function names in structures (for example, `_matops`) are the same as the base application function name without the object prefix and are in small letters. For example, `MatMultTranspose()` has a structure name of `multtranspose`.
5. Names of implementations of class functions should begin with the function name, an underscore, and the name of the implementation, for example, `KSPSolve_GMRES()`.
6. Each application-usable function begins with the name of the class object, followed by any subclass name, for example, `ISInvertPermutation()`, `MatMult()`, or `KSPGMRESRestart()`.
7. Functions that PETSc provides as defaults for user-providable functions end with `Default` (for example, `KSPMonitorDefault()` or `PetscSignalHandlerDefault()`).

¹Type also refers to the string name of the subclass.

8. 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`.
9. Specific XXXType values (for example, `MATSEQAIJ`) 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 maintained.

2.2.1 C Formatting

1. No tabs are allowed in *any* of the source code.
2. All PETSc function bodies are indented two characters.
3. Each additional level of loops, `if` statements, and so on is indented two more characters.
4. Wrapping lines should be avoided whenever possible.
5. Source code lines do not have a hard length limit; generally, we like them less than 150 characters wide.
6. 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 */
```

7. Assignment and comparison operations, for example, `x = 22.0` or `x < 22.0`, should have single spaces around the operator. This convention is true even when assignments are given directly in a line that declares the variable, such as `PetscReal r = 22.3`. The exception is when these symbols are used in a `for` loop; then, there should be no spaces, for example, `for (i=0; i<m; i++)`. Comparisons in `while()` constructs should have the spaces.
8. When declaring variables there should be no space between multiple variables, for example, `PetscReal a,b,c`, not `PetscReal a, b, c`.
9. The prototypes for functions should not include the names of the variables; for example, write

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

not

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

10. All local variables of a particular type (for example, `PetscInt`) should be listed on the same line if possible; otherwise, they should be listed on adjacent lines.
11. Equal signs should be aligned in regions where possible.
12. There *must* be a single blank line between the local variable declarations and the body of the function.
13. Indentation for `if` statements *must* be done as follows.

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

14. *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 you should use

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

15. Always have a space between `if` or `for` and the following `()`.
16. The open brace should be on the same line as the `if ()` test, `for ()`, and so forth, not on its own line, for example,

```
} else {
```

instead of

```

}
else { /* Incorrect */

```

See item 17 for an exception. The closing brace should *always* be on its own line.

17. 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 16.
18. Do not leave sections of commented-out code in the source files.
19. Do not use C++-style comments (`// Comment`). Use only C-style comments (`/* Comment */`).
20. All variables must be declared at the beginning of the code block (C89 style), never mixed in with code.
21. 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);
```

22. Do not use a space after the `)` in a cast or between the type and the `*` in a cast.
23. Do not include a space before or after a comma in lists. That is, do not write

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

but instead write

```
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. 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)`.
6. Do not use `#ifdef` or `#ifndef`. Rather, use `#if defined(...)` or `#if !defined(...)`.
7. Never use system random number generators such as `rand()` in PETSc code or examples because these can produce different results on different systems thus making portability testing difficult. Instead use `PetscRandom` which produces the exact same results regardless of system it is used on.

2.2.3 Usage of PETSc Functions and Macros

1. Public PETSc include files, `petsc*.h`, should not reference private PETSc `petsc/private/*impl.h` include files.
2. Public and private PETSc include files cannot reference include files located in the PETSc source tree.
3. The first line of the executable statements in functions must be `PetscFunctionBegin`;
4. Use `PetscFunctionReturn(returnvalue)`, not `return(returnvalue)`;
5. *Never* put a function call in a `return` statement; do not write

```
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()`, and so on directly in PETSc C/C++ source code or examples (usage is fine in Fortran source code). Rather, use `PetscSqrtScalar()`, `PetscSqrtReal()`, and so on, depending on the context. See `petscmath.h` for expressions to use.
8. Do not include `assert.h` in PETSc source code. Do not use `assert()`, 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 doing so 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,
                          recvcnts,displs,MPIU_INT,comm);CHKERRQ(ierr);
#else
    ierr = MPI_Allgatherv(lens,sendcount,MPIU_INT,lens,recvcnts,
                          displs,MPIU_INT,comm);CHKERRQ(ierr);
#endif
```

13. Do not introduce PETSc routines that provide essentially the same functionality as an available MPI routine. For example, do not write a routine `PetscGlobalSum()` that takes a scalar value and performs an `MPI_Allreduce()` on it. Instead, use the MPI routine `MPI_Allreduce()` directly in the code.
14. Never use a local variable counter such as `PetscInt flops = 0;` to accumulate flops and then call `PetscLogFlops()`; *always* just call `PetscLogFlops()` directly when needed.
15. Library functions should be declared `PETSC_INTERN` if they are intended to be visible only within a single PETSc 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`, and so on) and that plugin implementations of these classes can be included as separate shared libraries; thus, private functions may need to 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.
16. Before removing or renaming an API function, type, or enumerator, `PETSC_DEPRECATED()` should be used in the relevant header file to indicate the new, correct usage and the version number where the deprecation will first appear. For example,

```
typedef NewType OldType PETSC_DEPRECATED("Use NewType (since v3.9)");
PETSC_DEPRECATED("Use NewFunction (since v3.9)") PetscErrorCode
    OldFunction();
#define OLD_ENUMERATOR_DEPRECATED    OLD_ENUMERATOR
    PETSC_DEPRECATED_ENUM("Use NEW_ENUMERATOR (since v3.9)")
typedef enum {
    OLD_ENUMERATOR_DEPRECATED = 3,
    NEW_ENUMERATOR = 3
} MyEnum;
```

The old function or type, with the deprecation warning, should remain for at least one major release. The function or type's manual page should be updated (see 2.3, item 8).

17. Before removing or renaming an options database key, `PetscOptionsDeprecated()` should be used for at least one major release.
18. The format strings in PETSc ASCII output routines, such as `PetscPrintf`, take a `%D` for all PETSc variables of type `PetscInt`, not a `%d`.
19. All arguments of type `PetscReal` to PETSc ASCII output routines, such as `PetscPrintf`, must be cast to `double`, for example,

```
PetscPrintf(PETSC_COMM_WORLD, 'Norm %g\n', (double)norm);
```

2.3 Formatted Comments

PETSc uses formatted comments and the Sowing packages [6, 7] 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.

- `/*@`
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, not to generate a Fortran interface. In general, such labeled C functions should have a custom Fortran interface provided. Functions that take `char*` or function pointer arguments must have the `C` symbol and a custom Fortran interface provided.
- `/*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 `src/SECTION/f90-mod/petscSUBSECTION.h`, for example, `src/vec/f90-mod/petscisc.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`.
- `/*MC`
a formatted comment of a CPP macro or enum value for documentation.

The Fortran interface files supplied by the user go into the two directories `ftn-custom` and `f90-custom`, while those generated by Sowing go into `ftn-auto`.

Manual Page Format

Each function, typedef, class, macro, enum, and so on in the public API should include the following data, correctly formatted (see codes section) to generate complete manual pages and Fortran interfaces with Sowing. All entries below should be separated by blank lines. Except where noted, add a newline after the section headings.

1. The item's name, followed by a dash and brief (one-sentence) description.
2. If documenting a function, a description of the function's "collectivity" (whether all ranks in an MPI communicator need to participate). Unless otherwise noted, it's assumed that this collectivity is with respect to the MPI communicator associated with the first argument.
 - `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 MPI communicator associated with argument `XXX`)

- **Logically Collective** [on XXX] []; **YYY must contain common value** if the function is collective but does not require any actual synchronization (e.g. setting class parameters uniformly). Any argument YYY which must have the same value on all ranks of the MPI communicator should be noted here.
3. If documenting a function with input parameters, a list of input parameter descriptions in an **Input Parameters:** section.
 4. If documenting a function with output parameters, a list of output parameter descriptions in an **Output Parameters:** section.
 5. If documenting a function that interacts with the options database, a list of options database keys in an **Options Database Keys:** section.
 6. (Optional) a **Notes** section containing in-depth discussion, technical caveats, special cases, and so on. If it is ambiguous whether returned pointers/objects need to be freed/destroyed by the user or not, this information should be mentioned here.
 7. (If applicable) a **Fortran Notes:** section detailing any relevant differences in calling or using the item from Fortran.
 8. **Level:** (no newline) followed by **beginner**, **intermediate**, **advanced**, **developer**, or **deprecated**.
 9. (Optional) **Concepts:** (no newline), followed by a list of concepts.
 10. (Optional) **Keywords:** (no newline), followed by a list of keywords.
 11. **.seealso:** (no newline), followed by a list of related manual pages. These manual pages should usually also point back to this manual page in their **seealso:** sections.

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 [5]. The source code for the kernel is in `src/sys`. It contains systematic support for

- managing PETSc types,
- error handling,
- memory management,
- profiling,
- object management,
- Fortran interfaces (see [3])
- mechanism for generating appropriate citations for algorithms and software used in PETSc (see [9])
- file I/O,
- an options database, and
- objects and code for viewing, drawing, and displaying data and solver objects.

Each of these is discussed in a section below.

3.1 PETSc Types

For maximum flexibility, the basic data types `int`, `double`, and so on are not used in PETSc source code. Rather, it has

- `PetscScalar`,
- `PetscInt`,
- `PetscMPIInt`,
- `PetscBLASInt`,
- `PetscBool`, and
- `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` is used for integers passed to MPI as counts and sizes. These are always `int` since that is what the MPI standard uses. Similarly, `PetscBLASInt` is for counts, and so on 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, with Intel’s MKL and OpenBLAS).

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 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,PETSC_ERROR_INITIAL,"Error message");
```

It calls the error handler with the current function name and location: line number, and file, plus an error code and an error message. Normally `comm` is `PETSC_COMM_SELF`; it can be another communicator only if one is absolutely sure the same error will be generated on all processes in the communicator. This feature is to prevent the same error message from being printed by many processes.

The macro `CHKERRQ()` is defined by

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

In addition to `SETERRQ()`, the macros `SETERRQ1()`, `SETERRQ2()`, `SETERRQ3()`, and `SETERRQ4()` allow one to provide additional arguments to a formatted message string, for example,

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

The reason for the numbered format is that C89 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,error code,error type,const char *mess)
{
```

```

    PetscErrorCode ierr;

    PetscFunctionBegin;
    if (!eh) ierr = PetscTraceBackErrorHandler(line,func,file,error code,error
        type,mess,0);
    else      ierr = (*eh->handler)(line,func,file,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 _PetscEH* PetscEH;
struct _PetscEH {
    PetscErrorCode handler(MPI_Comm,int,const char*,const char*,const char*,
        PetscErrorCode,PetscErrorType,const char*,void*);

    void          *ctx;
    PetscEH       previous;
};

```

You 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()`, the default;
- `PetscAbortErrorHandler()`, called with `-on_error_abort`, useful when running in the debugger;
- `PetscReturnErrorHandler()`, which returns up the stack without printing error messages;
- `PetscEmacsClientErrorHandler()`;
- `PetscMPIAbortErrorHandler()`, which calls `MPI_Abort()` after printing the error message; and
- `PetscAttachDebuggerErrorHandler()`, called with `-on_error_attach_debugger`.

3.2.3 Error Codes

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

3.2.4 Detailed Error Messages

In a modern parallel component-oriented application code, it does not always make sense to simply print error messages to the terminal (and more than likely there is no “terminal”, for example, with Microsoft Windows or Apple iPad applications). PETSc provides the replaceable function pointer

```

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

```

which, by default, prints to standard out. Thus, error messages should not be printed with `printf()` or `fprintf()`. Rather, they should be printed with `(*PetscErrorPrintf)()`. You can direct all error messages to `stderr`, instead of the default `stdout`, with the command line option `-error_output_stderr`.

3.3 Memory Management

PETSc provides simple wrappers for the system `malloc()`, `calloc()`, and `free()` routines. The public interface for these is provided in `petscsys.h`, while the implementation code is in `src/sys/memory`. The most basic interfaces are

```
#define PetscMalloc(a,b)
    ((*PetscTrMalloc)((a),__LINE__,PETSC_FUNCTION_NAME,__FILE__,(void**)(b)))
#define PetscFree(a)
    ((*PetscTrFree)((void*)(a),__LINE__,PETSC_FUNCTION_NAME,__FILE__) || ((a) =
    0,0))
PetscErrorCode PetscMallocA(int n,PetscBool clear,int lineno,const char
    *function,const char *filename,size_t bytes0,void *ptr0,...)
PetscErrorCode PetscFreeA(int n,int lineno,const char *function,const char
    *filename,void *ptr0,...)
```

which allow the use of any number of profiling and error-checking wrappers for `malloc()`, `calloc()`, and `free()`. Both `PetscMallocA()` and `PetscFreeA()` call the function pointer values `(*PetscTrMalloc)` and `(*PetscTrFree)`. `PetscMallocSet()` is used to set these function pointers. The functions are guaranteed to support requests for zero bytes of memory correctly. Freeing memory locations also sets the pointer value to zero, preventing later code from accidentally using memory that has been freed. All PETSc memory allocation calls are memory aligned on at least double-precision boundaries; the macro generated by configure `PETSC_MEMALIGN` indicates in bytes what alignment all allocations have. This can be controlled at configure time with the option `-with-memalign=<4,8,16,32,64>`.

`PetscMallocA()` supports a request for up to 7 distinct memory locations of possibly different types. This serves two purposes: it reduces the number of system `malloc()` calls, thus potentially increasing performance, and it clarifies in the code related memory allocations that should be freed together.

The following macros are the preferred way to obtain and release memory in the PETSc source code. They automatically manage calling `PetscMallocA()` and `PetscFreeA()` with the appropriate location information.

```
#define PetscMalloc1(m1,r1)
    PetscMallocA(1,PETSC_FALSE,__LINE__,PETSC_FUNCTION_NAME,__FILE__,
    (size_t)(m1)*sizeof(**(r1)),(r1))
#define PetscMalloc2(m1,r1,m2,r2)
    PetscMallocA(2,PETSC_FALSE,__LINE__,PETSC_FUNCTION_NAME,__FILE__,
    (size_t)(m1)*sizeof(**(r1)),(r1),(size_t)(m2)*sizeof(**(r2)),(r2))
...
#define PetscMalloc7(...)
#define PetscFree2(m1,m2)    PetscFreeA
    (2,__LINE__,PETSC_FUNCTION_NAME,__FILE__,&(m1),&(m2))
...
#define PetscFree7(...)
```

Similar routines, `PetscCalloc1()` to `PetscCalloc7()`, provide memory initialized to zero. The size requests for these macros are in number of data items requested, not in bytes. This decreases the number of errors in the code since the compiler determines their sizes from the object type instead of requiring the user to provide the correct value with `sizeof()`.

The routines `PetscTrMallocDefault()` and `PetscTrFreeDefault()`, which are set with the routine `PetscSetUseTrMalloc_Private()` (and are used by default for the debug version of PETSc), provide simple logging and error checking versions of memory allocation.

3.4 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/`.

3.4.1 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 `PetscLogObjectDestroy()`

```
PetscLogObjectDestroy(PetscObject h);
```

These are called automatically by `PetscHeaderCreate()` and `PetscHeaderDestroy()`, which are used in creating all objects inherited from the basic object. Thus, these logging routines need 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(PetscObject h,const char *format,...);
```

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

3.4.2 Profiling Events

Events are logged by using the pair

```
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`.

3.4.3 Controlling Profiling

Routines that control the default profiling available in PETSc include the following

- `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 and Implementation

PETSc is designed by 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, since it does *not* “know” how the data is stored internally.

4.1 Introduction

PETSc is designed around several classes including **Vec** (vectors) and **Mat** (matrices, both dense and sparse). Each class is implemented by 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 **KSPGMRES** and **KSPCG** have this common subheader).
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 parallel layout, 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. 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 the following organization.

- Its own, application-public, include file `include/petsc<class>.h`.
- Its own directory, `src/<class>` or `src/<package>/<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 re-compiling 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] */
    ... (AND SO ON) ...
};

```

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() */
};

```

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 you need to call

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

For example,

```
PetscClassIdRegister("index set",&IS_CLASSID);
```

you 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. The function pointers are not called directly; rather you should call **PetscObjectFunctionName()**, where **FunctionName** is one of the functions listed below with the first letter of each word capitalized.

- **getcomm(PetscObject,MPI_Comm*)** obtains the MPI communicator associated with this object.
- **view(PetscObject,PetscViewer)** allows you to store or visualize the data inside an object. If the Viewer is NULL, then it should cause the object to print information on the object to **lstinlinedstdout**.
- **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, they are each sent a `destroy()`; that is, 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). 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 via `PetscObjectCompose()`. It retrieves a NULL if no object with that name was previously composed.
- `composefunction(PetscObject, const char *name, void *func)` associates a function pointer with 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 `name` is the character string name of the function.
For example, `fname` may be `PCCreate_LU`.
- `queryfunction(PetscObject, const char *name, void **func)` retrieves a function pointer that was associated with the object via `PetscObjectComposeFunction()`. 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 compose PETSc objects *only* with PETSc objects rather than any arbitrary pointer, PETSc provides the convenience object `PetscContainer`, created with the routine `PetscContainerCreate(MPI_Comm, PetscContainer*)`, to allow wrapping 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 unless there is a specific reason, they should use the PETSc defaults so that the library writer does not have to “reinvent the wheel.”

4.5.1 Compose and Query Objects

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 as follows (defined in `src/sys/objects/inherit.c`).

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

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

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

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

4.5.2 Compose and Query Functions

PETSc allows you to compose functions by specifying a name and function pointer. In `src/sys/dll/reg.c`, PETSc defines the following 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 the following.

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

    PetscFunctionBegin;
    PetscFunctionListAdd(&obj->qlist,name,fname,ptr);
```

```

    PetscFunctionReturn(0);
}

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

    PetscFunctionBegin;
    PetscFunctionListFind(obj->qlist,name,ptr);
    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.

4.5.3 Simple PETSc Objects

Some simple PETSc objects 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`.

4.6 PETSc Packages

The PETSc source code is divided into the following library-level packages: `sys`, `Vec`, `Mat`, `DM`, `KSP`, `SNES`, `TS`, `TAO`. Each of these has a directory under the `src` directory in the PETSc tree and, optionally, can be compiled into separate libraries. Each package defines one or more classes; for example, the `KSP` package defines the `KSP` and `PC` classes, as well as several utility classes. In addition, each library-level package may contain several class-level packages associated with individual classes in the library-level package. In general, most “important” classes in PETSc have their own class level package. Each package provides a registration function `XXXInitializePackage()`, for example `KSPInitializePackage()`, which registers all the classes and events for that package. Each package also registers a finalization routine, `XXXFinalizePackage()`, that releases all the resources used in registering the package, using `PetscRegisterFinalize()`. The registration for each package is performed “on demand” the first time a class in the package is utilized. This is handled, for example, with code such as

```

PetscErrorCode VecCreate(MPI_Comm comm, Vec *vec)
{
    Vec v;

    PetscFunctionBegin;
    PetscValidPointer(vec,2);
    *vec = NULL;
    VecInitializePackage();
    ...
}

```

Chapter 5

How the Solvers Handle User Provided Callbacks

The solver objects in PETSc, **KSP** (optionally), **SNES**, and **TS** require user provided callback functions (and contexts for the functions) that define the problem to be solved. These functions are supplied by the user with calls such as **SNESSetFunction(SNES,...)** and **TSSetRHSFunction(TS,...)**. One would naturally think that the functions provided would be attached to the appropriate solver object, that is, that the **SNES** callbacks would be attached to the **SNES** object and **TS** callbacks to the **TS** object. This is not the case. Or possibly one might think the callbacks would be attached to the **DM** object associated with the solver object. This is also not the case. Rather, the callback functions are attached to an inner nonpublic DMXXX object (XXX is **KSP**, **SNES**, or **TS**) that is attached to the **DM** that is attached to the XXX solver object. This convoluted design is to support multilevel and multidomain solvers where different levels and different domains may (or may not) share the same callback function or callback context. You can control exactly what XXX/DM objects share a common DMXXX object.

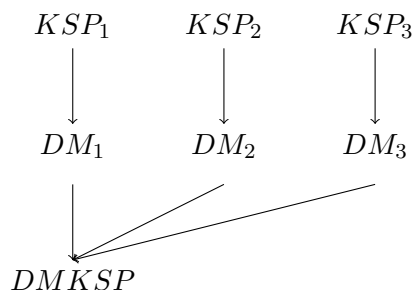


Figure 1: Three levels of KSP/DM share the same DMKSP

In Figure 1 we depict how three levels of **KSP** objects share a common DMKSP object. The code to access the inner DMKSP object is

```
DM    dm_2;  
DMKSP dmksp;  
KSPGetDM(ksp_2,&dm_2);  
DMGetDMKSP(dm_2,&dmksp);
```

To obtain a new DMKSP object for which you can change the callback functions (or their contexts) without affecting the original DMKSP, call

```

DM    dm_2;
DMKSP dmksp;
KSPGetDM(ksp_2,&dm_2);
DMGetDMKSPWrite(dm_2,&dmksp_2);
    
```

This results in the object organization as indicated in Figure 2

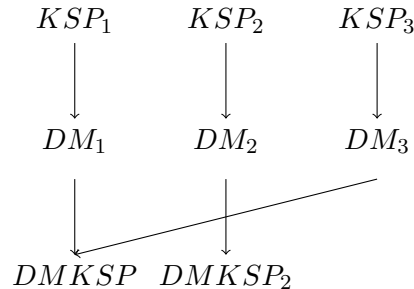


Figure 2: Two levels of KSP/DM share the same DMKSP; one has its own private copy

The DMKSP object is essentially the list of callback functions and their contexts, for example,

```

typedef struct _p_DMKSP *DMKSP;
typedef struct _DMKSP_Ops *DMKSP_Ops;
struct _DMKSP_Ops {
    PetscErrorCode (*computeoperators)(KSP,Mat,Mat,void*);
    PetscErrorCode (*computerhs)(KSP,Vec,void*);
    PetscErrorCode (*computeinitialguess)(KSP,Vec,void*);
    PetscErrorCode (*destroy)(DMKSP*);
    PetscErrorCode (*duplicate)(DMKSP,DMKSP);
};

struct _p_DMKSP {
    PETSCHEADER(struct _DMKSP_Ops);
    void *operatorsctx;
    void *rhsctx;
    void *initialguessctx;
    void *data;
    DM originaldm;

    void (*fortran_func_pointers[3])(void); /* Store our own function pointers so
        they are associated with the DMKSP instead of the DM */
};
    
```

We now explore in more detail exactly how the solver calls set by the user are passed down to the inner DMKSP object. For each user level solver routine for setting a callback a similar routine exists at the DM level. Thus, XXXSetY(XXX,...) has a routine DMXXXSetY(DM,...).

```

PetscErrorCode KSPSetComputeOperators(KSP ksp,PetscErrorCode
    (*func)(KSP,Mat,Mat,void*),void *ctx)
{
    PetscErrorCode ierr;
    
```

```

DM          dm;

PetscFunctionBegin;
PetscValidHeaderSpecific(ksp,KSP_CLASSID,1);
ierr = KSPGetDM(ksp,&dm);CHKERRQ(ierr);
ierr = DMKSPSetComputeOperators(dm,func,ctx);CHKERRQ(ierr);
if (ksp->setupstage == KSP_SETUP_NEWRHS) ksp->setupstage = KSP_SETUP_NEWMATRIX;
PetscFunctionReturn(0);
}

```

The implementation of `DMXXXSetY(DM,...)` gets a “writable” version of the `DMXXX` object via `DMGetDMXXXWrite(DM,DMXXX*)` and sets the function callback and its context into the `DMXXX` object.

```

PetscErrorCode DMKSPSetComputeOperators(DM dm,PetscErrorCode
    (*func)(KSP,Mat,Mat,void*),void *ctx)
{
    PetscErrorCode ierr;
    DMKSP          kdm;

    PetscFunctionBegin;
    PetscValidHeaderSpecific(dm,DM_CLASSID,1);
    ierr = DMGetDMKSPWrite(dm,&kdm);CHKERRQ(ierr);
    if (func) kdm->ops->computeoperators = func;
    if (ctx) kdm->operatorsctx = ctx;
    PetscFunctionReturn(0);
}

```

The routine for `DMGetDMXXXWrite(DM,DMXXX*)` entails a duplication of the object unless the `DM` associated with the `DMXXX` object is the original `DM` that the `DMXXX` object was created with. This can be seen in the following code.

```

PetscErrorCode DMGetDMKSPWrite(DM dm,DMKSP *kspdm)
{
    PetscErrorCode ierr;
    DMKSP          kdm;

    PetscFunctionBegin;
    PetscValidHeaderSpecific(dm,DM_CLASSID,1);
    ierr = DMGetDMKSP(dm,&kdm);CHKERRQ(ierr);
    if (!kdm->originaldm) kdm->originaldm = dm;
    if (kdm->originaldm != dm) { /* Copy on write */
        DMKSP oldkdm = kdm;
        ierr = PetscInfo(dm,"Copying DMKSP due to write\n");CHKERRQ(ierr);
        ierr = DMKSPCreate(PetscObjectComm((PetscObject)dm),&kdm);CHKERRQ(ierr);
        ierr = DMKSPCopy(oldkdm,kdm);CHKERRQ(ierr);
        ierr = DMKSPDestroy((DMKSP*)&dm->dmksp);CHKERRQ(ierr);
        dm->dmksp = (PetscObject)kdm;
        kdm->originaldm = dm;
    }
    *kspdm = kdm;
    PetscFunctionReturn(0);
}

```



```
}

```

The routine `DMGetDMXXX(DM, DMXXX*)` has the following form.

```
PetscErrorCode DMGetDMKSP(DM dm, DMKSP *kspdm)
{
    PetscErrorCode ierr;

    PetscFunctionBegin;
    PetscValidHeaderSpecific(dm, DM_CLASSID, 1);
    *kspdm = (DMKSP) dm->dmksp;
    if (!*kspdm) {
        ierr = PetscInfo(dm, "Creating new DMKSP\n"); CHKERRQ(ierr);
        ierr =
            DMKSPCreate(PetscObjectComm((PetscObject) dm), kspdm); CHKERRQ(ierr);
        dm->dmksp = (PetscObject) *kspdm;
        (*kspdm)->originaldm = dm;
        ierr = DMCoarsenHookAdd(dm, DMCoarsenHook_DMKSP, NULL, NULL); CHKERRQ(ierr);
        ierr = DMRefineHookAdd(dm, DMRefineHook_DMKSP, NULL, NULL); CHKERRQ(ierr);
    }
    PetscFunctionReturn(0);
}
```

This routine uses `DMCoarsenHookAdd()` and `DMRefineHookAdd()` to attach to the `DM` object two functions that are automatically called when the object is coarsened or refined. The hooks `DMCoarsenHook_DMXXX()` and `DMRefineHook_DMXXX()` have the same form:

```
static PetscErrorCode DMCoarsenHook_DMKSP(DM dm, DM dmc, void *ctx)
{
    PetscErrorCode ierr;

    PetscFunctionBegin;
    ierr = DMCopyDMKSP(dm, dmc); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}
```

where

```
PetscErrorCode DMCopyDMKSP(DM dmsrc, DM dmdest)
{
    PetscErrorCode ierr;

    PetscFunctionBegin;
    PetscValidHeaderSpecific(dmsrc, DM_CLASSID, 1);
    PetscValidHeaderSpecific(dmdest, DM_CLASSID, 2);
    ierr = DMKSPDestroy((DMKSP*)&dmdest->dmksp); CHKERRQ(ierr);
    dmdest->dmksp = dmsrc->dmksp;
    ierr = PetscObjectReference(dmdest->dmksp); CHKERRQ(ierr);
    ierr =
        DMCoarsenHookAdd(dmdest, DMCoarsenHook_DMKSP, NULL, NULL); CHKERRQ(ierr);
    ierr =
        DMRefineHookAdd(dmdest, DMRefineHook_DMKSP, NULL, NULL); CHKERRQ(ierr);
}
```

```
PetscFunctionReturn(0);  
}
```

ensures that the new **DM** shares the same **DMXXX** as the parent **DM** and also inherits the hooks if it is refined or coarsened.

If you provide callbacks to a solver *after* the **DM** associated with a solver has been refined or coarsened, those child **DMs** will not share a common **DMXXX**.

The **TS** object manages its callback functions in a way similar to **KSP** and **SNES**, although there are no multilevel **TS** implementations so in theory the **DMTS** object is currently unneeded.

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 in 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. In order to help manage this difference in time scales, most processors have at least three levels of memory: registers, cache, and random access memory. (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 a floating-point number is reused 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 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 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, you employ 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 advantageous. The PETSc sparse matrix representations use a variety of techniques for blocking, including the following:

- 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).

The advantage of the first approach is that it is a minimal change from a standard sparse matrix format and brings a large percentage of the improvement obtained 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, and so on), 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 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 3 for these problems (using the option `-mat_seqbaij`). The rates were computed on one node of an older IBM Power processor based system, using two test matrices. The first matrix (ARCO1), courtesy of Rick Dean of Arco, arises in multiphase flow simulation; it has 1,501 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 display them for triangular solves obtained from an ILU(0) factorization.

Problem	Block size	Basic	I-node version	Fixed block size
<i>Matrix-Vector Product (Mflop/sec)</i>				
Multiphase	3	27	43	70
Euler	5	28	58	90
<i>Triangular Solves from ILU(0) (Mflop/sec)</i>				
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.

6.2 Assorted Matrix Types

PETSc offers a variety of both sparse and dense matrix types.

6.2.1 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).

6.2.2 Parallel AIJ Sparse Matrices

The AIJ sparse matrix type, is the default parallel matrix format; additional implementation details are given in [4].

6.2.3 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 $\text{nb} \times \text{nb}$ 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

```
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, you 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.

6.2.4 Parallel Block AIJ Sparse Matrices

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

```
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 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 you must ensure that they are chosen to be compatible with the vectors. To do so, you first consider the product $y = Ax$. The `m` that used 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`.

You 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.

6.2.5 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 Fortran style.

6.2.6 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 example source files that describes the tests to be run
2. The *test generator* (`config/gmakegentest.py`) that at the `make` step parses the example source files and generates the makefiles and shell scripts.
3. The *petsc test harness* that consists of makefile and shell scripts that runs the executables with several logging and reporting features.

Details on using the harness may be found in the main PETSc manual.

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 the following

- A single shell script
- A single makefile
- A single output file that represents the *expected results*
- Two or more command tests, usually, one or more `mpiexec` tests that run the executable, and one or more `diff` tests to compare output with the expected result.

Our language also supports a *testset* that 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; as are explained further below. The input language supports simple yet flexible test control, and we begin by describing this language.

7.1.1 Runtime Language Options

At the end of each test file, a marked comment block, using YAML, is inserted to describe the test(s) 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 be

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

In order to enable the second goal, the *basestring* of the filename is defined as the filename without the extension; for example, if the filename is `ex1.c`, then `basestring=ex1`. With this background, these keywords are as follows.

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

At the top level either a single test or a test set must be specified. All other keywords are sub-entries of this keyword.

suffix: (*Optional; Default: suffix=""*)

- The test name is given by `testname = basestring` if the suffix is set to an empty string, and by `testname = basestring + "_" + suffix` otherwise.
- This can be specified only for top level test nodes.

output_file: (*Optional; Default: output_file = "output/" + testname + ".out"*)

- The output of the test is to be compared with 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 (for example, `output/ex1.out`)

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

- This integer is passed to `mpirun`; i.e., `mpirun -n nsiz`

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

- These arguments are passed 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, for example, `grep foo` or `sort -nr`.
- If the filter begins with `Error:`, then the test is assumed to be testing the `stderr` output, and the error code and output are set up to be tested.

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

- Sometimes filtering the output file is useful for standardizing tests. For example, in order to handle the issues related to parallel output, both the output from the test example and the output file need to be sorted (since `sort` does not produce the same output on all machines). 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. If you list a directory instead of files, it will copy the entire directory (this is limited currently to a single directory)
- The copying is done by the test generator and not by creating makefile dependencies.

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

- This is 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 require external matrices that are found in the `DATAFILES` path. For these tests `requires: datafiles` can be specified.
- Packages are indicated with lower-case specification, for example, `requires: superlu_dist`.

- Any defined variable in `petscconf.h` can be specified with the `defined(...)` syntax, for example, `defined(PETSC_USE_INFO)`.
- Any definition of the form `PETSC_HAVE_FOO` can just use `requires: foo` similar to how third-party packages are handled.

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

- This parameter allows you to extend the default timeout for an individual test such that the new timeout time is `timeout=(default timeout) x (timeoutfactor)`.
- 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`).

7.1.2 Additional Specifications

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

- for loops: Specifying `{{ ... }shared output}` or `{{ ... }separate output}` will create for loops over an enclosed space-delimited 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 (in shorthand notation, `{{ ... }}`) syntax must be used.

For loops are supported within `nsiz` and `args`. An example is

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

In this case, two execution lines would be added with two different arguments. Associated `diff` lines would be added as well automatically.

Here the output for each `matload_block_size` is assumed to give the same output so that only one diff file is needed. If the variables produced different output, then the `separate output` option would be added. In this case, each loop variable and value become a separate script.

See examples below for how it works in practice.

7.1.3 Test Block Examples

The following is the simplest test block:

```
/*TEST
  test:
TEST*/
```

which is equivalent to

```
/*TEST
  testset:
    test:
TEST*/
```

which is equivalent to

```
/*TEST
  testset:
TEST*/
```

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

For Fortran, the equivalent is

```
!/*TEST
! test:
!TEST*/
```

A more complete example is

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

This creates two tests. Assuming that this is `src/a/b/examples/tutorials/ex1.c`, the tests would be `a_b_tutorials-ex1` and `a_b_tutorials-ex1_1`.

Following is an example of how to test a permutation of arguments against the same output file:

```
/*TEST
testset:
  suffix: 19
  requires: datafilepath
  args: -f0 ${DATAFILESPATH}/matrices/poisson1
  args: -ksp_type cg -pc_type icc -pc_factor_levels 2
  test:
  test:
    args: -mat_type seqsbaij
*/TEST
```

Assuming that this is `ex10.c`, there would be two `mpiexec/diff` invocations in `runex10_19.sh`. Here is a similar example, but the permutation of arguments creates different output:

```
/*TEST
testset:
  requires: datafilepath
  args: -f0 ${DATAFILESPATH}/matrices/medium
  args: -ksp_type bicg
  test:
    suffix: 4
    args: -pc_type lu
  test:
    suffix: 5
*/TEST
```

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

An example using a for loop is:

```
/*TEST
testset:
  suffix: 1
  args: -f ${DATAFILESPATH}/matrices/small -mat_type aij
```

```

    requires: datafilespath
testset:
    suffix: 2
    output_file: output/ex138_1.out
    args: -f ${DATAFILESPATH}/matrices/small
    args: -mat_type baij -matload_block_size {{2 3}shared output}
    requires: datafilespath
*/TEST

```

In this example, `ex138_2` will invoke `runex138_2.sh` twice with two different arguments, but both are diffed with the same file.

Following is an example showing the hierarchical nature of the test specification.

```

testset:
    suffix:2
    output_file: output/ex138_1.out
    args: -f ${DATAFILESPATH}/matrices/small -mat_type baij
    test:
        args: -matload_block_size 2
    test:
        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/poisson1
    args: -ksp_type cg -pc_type icc
    args: -pc_factor_levels {{0 2 4}separate output}
    test:
    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

You can specify issues related to the compilation of the source file with the `build:` block. The language is as follows.

- **requires:** (*Optional; Default: ""*)
 - Same as the runtime requirements (for example, can include `requires: fftw`) but also requirements related to types:
 1. Precision types: `single`, `double`, `quad`, `int32`
 2. Scalar types: `complex` (and `!complex`)
 - In addition, `TODO` is available to allow you to skip the build of this file but still maintain it in the source tree.
- **depends:** (*Optional; Default: ""*)

- List any dependencies required to compile the file

A typical example for compiling for only real numbers is

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

7.2 PETSC Test Harness

The goals of the PETSc test harness are threefold.

- Provide standard output used by other testing tools
- Be as 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, you should first understand the desired requirements for reporting and logging.

Testing the Parsing

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

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

A dictionary will be pretty-printed. From this dictionary printout, any problems in the parsing are usually obvious. 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 Protocol (TAP); see <https://testanything.org/tap-specification.html>

This is a 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 ${DATAFILES_PATH}/matrices/small -mat_type {{aij baij sbaij}} -matload_block_size {{2 3}}
  requires: datafiles_path
```

A sample output follows.

```

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

ok 11 In mat...tests: "./ex138 -f ${DATAFILES_PATH}/matrices/small -mat_type saij -matload_block_size 2"
ok 12 In mat...tests: "Diff of ./ex138 -f ${DATAFILES_PATH}/matrices/small -mat_type aij -matload_block_size 2"

```

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 the following.

```

#!/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 is as follows.

```

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, you can modify the following file:

```

${PETSC_DIR}/config/petsc_harness.sh

```

7.5 Using the Test Harness for Regression Testing

The test system can also be helpful for contributing developers in regression testing. A typical workflow might proceed as follows. Note that the exact syntax of all input and output in this section is subject to change.

1. Run the full test suite on the development machine, using the branch of PETSc you want to base your changes on (usually `master`).

```
git checkout master
make -f gmakefile test
```

Confirm that all the tests pass.

2. Create a new branch, and make your changes.

```
git checkout -b myname/my-feature
# make and commit changes
```

3. Run the full test suite on your branch ¹.

```
make -f gmakefile test
```

4. Examine the output for any failed tests, for example,

```
# -----
# Summary
# -----
# FAILED ts_tutorials-ex11_adv_2d_quad_0 diff-sys_classes_viewer_tests-ex4_4 ts_tutorials-
# ex11_adv_2d_quad_1
# success 3051/3915 tests (77.9%)
# failed 3/3915 tests (0.1%)
# todo 91/3915 tests (2.3%)
# skip 770/3915 tests (19.7%)
#
# Approximate time (not incl. build time): 429 sec
#
# To rerun failed tests:
# /opt/local/bin/gmake -f gmakefile test search='ts_tutorials-ex11_adv_2d_quad_0
# sys_classes_viewer_tests-ex4_4 ts_tutorials-ex11_adv_2d_quad'
```

This output indicates that three tests have failed and should be investigated.

5. Rerun one of the failed tests, with additional verbosity.

```
/opt/local/bin/gmake V=1 -f gmakefile test
search='sys_classes_viewer_tests-ex4_4'
```

6. Examine the output.

```
arch-darwin-master-double-debug/tests/sys/classes/viewer/examples/tests/runex4_4.sh -v
ok sys_classes_viewer_tests-ex4_4 cat ex4_4.tmp | > /dev/null; cat ex4a1.tmp > ex4_4.tmp.tmp
2>> runex4_4.err && mv ex4_4.tmp.tmp ex4_4.tmp
```

¹For information on how to run a subset of tests, consult the users manual and other sources referenced therein.

```

not ok diff-sys_classes_viewer_tests-ex4_4 /Users/patrick/petsc-master/lib/petsc/bin/
petscdiff /Users/patrick/petsc-master/src/sys/classes/viewer/examples/tests/output/ex4a.
out ex4_4.tmp > diff-runex4_4.out 2> diff-runex4_4.out
#2a3,14
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1
#> Testing PetscViewerASCIIPrintf 0
#> Testing PetscViewerASCIIPrintf 1

```

This indicates that the output does not match the reference output.

7. Look in the directory from which the test was run.

```
cd $PETSC_ARCH/tests/sys/classes/viewer/examples/tests/
```

8. The name of the test indicates that the test was defined in `ex4.c`, with the suffix 4. This is defined in `ex4.c` as such:

```

test:
    suffix: 4
    args: -myviewer ascii:ex4a1.tmp::append
    filter: cat ex4a1.tmp
    output_file: output/ex4a.out

```

(Note that the default number of MPI processes is 1.)

9. We can deduce now that the command to rerun the example is

```

make ex4
$PETSC_DIR/$PETSC_ARCH/bin/mpiexec -n 1 ./ex4 -myviewer ascii:ex4a1.tmp::append
cat ex4a1.tmp

```

10. Similarly, we can deduce that the reference output is in

```
$PETSC_DIR/src/sys/classes/viewer/examples/tests/output/ex4a.out
```

(Note that there is a default output location in `output/testname_suffix`.)

11. This should provide enough information to iteratively debug the problem by making changes, recompiling, and rerunning this example directly.
12. Repeat for other errors.

7.5.1 Additional Tips

To rerun just the reporting use

```
config/report_tests.py
```

To see the full options use

```
config/report_tests.py -h
```

To see the full timing information for the five most expensive tests use

```
config/report_tests.py -t 5
```


Bibliography

- [1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. *CRPC Handbook of Parallel Computing*, chapter Software for the Scalable Solution of PDEs. Morgan Kaufmann Publishers, 2002.
- [2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual: Revision 3.11. Technical Report ANL-95/11 - Rev 3.11, Argonne National Laboratory, 2019.
- [3] Satish Balay, Jed Brown, Matthew G. Knepley, Lois McInnes, and Barry Smith. *Software Engineering for Science*, chapter Providing Mixed Language and Legacy Support within a Library. Taylor & Francis, 2015.
- [4] 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.
- [5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. A microkernel design for component-based parallel numerical software systems. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 58–67. SIAM, 1998.
- [6] W Gropp. Users manual for bfort: Producing Fortran interfaces to C source code. Technical Report ANL/MCS-TM-208, Argonne National Laboratory, 1995.
- [7] W Gropp. Users manual for doctext: Producing documentation from source code. Technical Report ANL/MCS-TM-206, Argonne National Laboratory, 1995.
- [8] William D. Gropp and Barry F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 87–93, Mississippi State University, 1994. IEEE.
- [9] Matthew G. Knepley, Jed Brown, Lois Curfman McInnes, and Barry F. Smith. Accurately citing software and algorithms used in publications. In *First Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE), held at SC13*, 2013.
- [10] B. Smith. *Encyclopedia of Parallel Computing*, chapter PETSc, the Portable, Extensible Toolkit for Scientific computing. Springer, 2011.



Mathematics and Computer Science Division

Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov