# Extensibility in PETSc    http://www.mcs.anl.gov/petsc

**Matthew G. Knepley**
Computation Institute, University of Chicago
knepley@gmail.com

**Jed Brown**
Argonne National Lab & University of Colorado Boulder
jed@jedbrown.org

We acknowledge support from the Department of Energy Office of Advanced Scientific Computing.

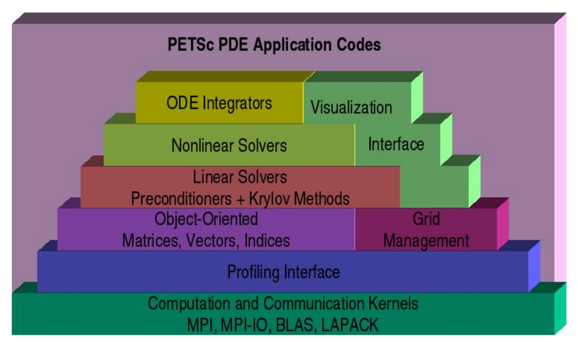
## PETSc

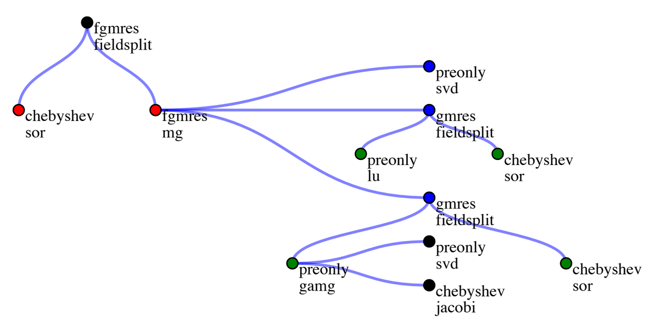
PETSc is a set of libraries for the efficient, scalable, solution of systems of nonlinear algebraic equations, and an extensible platform for scientific computing.

- **Modular**: Functionality is cleanly separated into interacting interfaces
- **Scalable**: Code runs efficiently on 1–1,000,000+ cores
- **Extensible**: Users can easily add functionality to solve their problems

## Clean Hierarchy of Interfaces

For each PETSc class, there is a toplevel interface, using only other toplevel interfaces, and internally we only us these interface when employing other classes.




PETSc dispatches to an implementation class from the toplevel interface,

```
PetscErrorCode MatMult(Mat mat,Vec x,Vec y) {
  PetscErrorCode ierr;

  if (!mat->ops->mult) SETERRQ(PetscObjectComm((PetscObject) mat), PETSC_ERR_SUP, "This matrix type does not have a multiply defined");
  ierr = (*mat->ops->mult)(mat, x, y);CHKERRQ(ierr);
  return 0;
}
```

allowing us to **switch storage formats** on the fly, or in the case of solvers to **change algorithms**. We can seamlessly manage external hardware and storage,

```
PetscErrorCode VecAXPY_SeqCUSP(Vec yin, PetscScalar alpha, Vec xin) {
  CUSPARRAY      *xarray, *yarray;
  PetscErrorCode ierr;

  ierr = VecCUSPGetArrayRead(xin, &xarray);CHKERRQ(ierr);
  ierr = VecCUSPGetArrayReadWrite(yin, &yarray);CHKERRQ(ierr);
  try {
    cusp::blas::axpy(*xarray, *yarray, alpha);
    ierr = WaitForGPU();CHKERRCUSP(ierr);
  } catch(char *ex) {
    SETERRQ1(PETSC_COMM_SELF, PETSC_ERR_LIB, "CUSP error: %s", ex);
  }
  ierr = VecCUSPRestoreArrayRead(xin, &xarray);CHKERRQ(ierr);
  ierr = VecCUSPRestoreArrayReadWrite(yin, &yarray);CHKERRQ(ierr);
  return 0;
}
```

### Why Use Dynamic Dispatch?

- Runtime customization of implementation
- Small, abstract interface
- No runtime type checks for customization
- Single breakpoint for debugging

### Why Not Use Templates?

- Lack of encapsulation (above and below template library interfaces)
- Crowds namespace and interfaces with instantiated names
- Template expansion code not available to the user
- Compile time and error message explosion
- Convoluted instantiation and resolution logic

## Plugin Architecture

Since PETSc can change implementation on the fly, using the *Delegator* pattern, new class implementations can be provided at runtime by the user. The user creates a dynamic library, say `libmysnes.so` with a registration function that PETSc calls on load ( `PetscDLLibraryRegister_mynes`). This function registers a constructor for the new class, and the constructor sets up the virtual table of functions which direct interface calls down the to implementation. Below is an implementation of a new nonlinear solver library:

```
#define MYSNES "mysnes"

typedef struct {
  /* Data for user solver */
} SNES_MySNES;

/* Other implemented functions, e.g. SNESView_MySNES(), go here */

static PetscErrorCode SNESSolve_MySNES(SNES snes)
{
  SNES_MySNES *my = (SNES_MySNES *) snes->data;

  /* User solver */
  return 0;
}

PetscErrorCode SNESCreate_MySNES(SNES snes)
{
  SNES_MySNES    *my;
  PetscErrorCode ierr;

  snes->ops->solve = SNESSolve_MySNES;
  /* Other vtable entries here */
  ierr = PetscNewLog(snes, &my);CHKERRQ(ierr);
  snes->data = (void*) my;
  return 0;
}
```

```
static PetscBool MySNESPackageInitialized = PETSC_FALSE;

PetscErrorCode MySNESFinalizePackage(void)
{

  MySNESPackageInitialized = PETSC_FALSE;
  return 0;
}

PetscErrorCode MySNESInitializePackage(void)
{
  PetscErrorCode ierr;

  if (MySNESPackageInitialized) return 0;
  MySNESPackageInitialized = PETSC_TRUE;
  ierr = SNESRegister(SNESSHELL, SNESCreate_MySNES);CHKERRQ(ierr);
  ierr = PetscRegisterFinalize(MySNESFinalizePackage);CHKERRQ(ierr);
  return 0;
}

PetscErrorCode PetscDLLibraryRegister_mynes(void)
{
  PetscErrorCode ierr;

  ierr = SNESInitializePackage();CHKERRQ(ierr);
  return 0;
}
```

### Why Use Plugins?

- Inversion of Dependencies

  The `MatSchur` matrix implementation uses a `KSP` object internally, so it cannot sit in a matrix library, but it does not belong in the solver library. We can put it in a third library that uses the solver library dynamically.

- Extensibility without recompilation or reinstallation
- Static compilation does not provide a performance boost

PETSc objects can be configured at runtime with Options Database, the *Service Locator* pattern. For example, an optimized solver for the 2D Allen-Cahn problem (SNES ex55) can be constructed entirely from options:

Run flexible GMRES with 5 levels of multigrid as the preconditioner

```
./ex55 -ksp_type fgmres -pc_type mg -pc_mg_levels 5 -da_grid_x 65 -da_grid_y 65
```

Use the Galerkin process to compute the coarse grid operators

```
-pc_mg_galerkin
```

Use SVD as the coarse grid saddle point solver

```
-mg_coarse_ksp_type preonly -mg_coarse_pc_type svd
```

Smoother: Flexible GMRES (2 iterates) with a Schur complement PC

```
-mg_levels_ksp_type fgmres -mg_levels_pc_fieldsplit_detect_saddle_point -mg_levels_ksp_max_it 2 -mg_levels_pc_type fieldsplit
-mg_levels_pc_fieldsplit_type schur -mg_levels_pc_fieldsplit_factorization_type full -mg_levels_pc_fieldsplit_schur_precondition diag
```

Schur complement solver: GMRES (5 iterates) with no preconditioner

```
-mg_levels_fieldsplit_1_ksp_type gmres -mg_levels_fieldsplit_1_pc_type none -mg_levels_fieldsplit_ksp_max_it 5
```

Schur complement action: Use only the lower diagonal part of A00

```
-mg_levels_fieldsplit_0_ksp_type preonly -mg_levels_fieldsplit_0_pc_type sor -mg_levels_fieldsplit_0_pc_sor_forward
```

### Why Use Options?

- Code without configuration is simpler and more maintainable
- Configuration of deep hierarchies of objects

  Sovler configurations, such as the one shown on the left, can be deeply nested, and changed in response to problem conditions. Maintaining optimality across problem configurations requires large changes to the solver. Destruction and recreation of objects with different types, as you would have with templates, is not feasible for these large collections of interrelated objects.

## Major New Capabilities

- Advanced time stepping
- Nonlinear preconditioning
- Unstructured mesh managment
- Scalable communication infrastructure

## Library Behavior

PETSc endeavors to make the library consistent across architectures, environments, and builds. It has a consistent Application Binary Interface (ABI) across different builds, for example debugging and optimized.

We try to avoid Bill Gropp's List of Component Mistakes:

- Namespace pollution and monolithic library structure
- Printing error messages or exiting
- Requiring interactive input or `main()`
- Requiring running on all processes
- Lack of portability, testing, documentation, examples
- Ignorance of standards

## Configure Extensibility

PETSc Configure can be extended with user modules at runtime. The tests for many packages are now contributed, and work automatically when dropped in the moduels directory.

Advantages of our system over others, such as Autoconf and SCons:

1. Namespacing:

   Tests are wrapped up in modules, which also hold the test results. Thus you get the normal Python namespacing of results. As simple as this sounds, SCons does not do it, nor CMake, nor Autoconf. They all use one flat namespace. Also, when we build up command lines, you can see where options came from, whereas in the others, all flags are dumped into reservoirs like INCLUDE and LIBS.

2. Explicit control flow

   The modules are organized explicitly in a DAG. The user indicates dependence with a single call, `requires('path.to.other.test')`, which not only structures the DAG, but returns the object so that the module can use the results of the test it depends on.

3. Multi-languages tests

   We have explicit pushing and popping of languages, so builds can use any one they want, all with their own compilers, flags, libraries, etc. Thus its easy for us to do cross-language checks in a few lines, whereas this is very difficult in other systems.

4. Subpackages

   We have a template package (and a GNU specialization) so that PETSc downloads, builds, and tests it for inclusion. In some cases now, people use packages through PETSc because it will get it and build it automatically. Most other systems have no idea of hierarchy.