TITLE

PETSc - the Portable, Extensible Toolkit for Scientific computation

BYLINE

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL
USA
bsmith@mcs.anl.gov

DEFINITION

The Portable, Extensible Toolkit for Scientific computation (PETSc, pronounced PET-see) is a suite of open source software libraries for the parallel solution of linear and nonlinear algebraic equations. PETSc uses the Message Passing Interface (MPI) for all of its parallelism.

DISCUSSION

The numerical solution of linear systems with sparse matrix representations is at the heart of many numerical simulations, from brain surgery to rocket science. These linear systems arise from the replacement of continuum partial differential equation (PDE) models with suitable discrete models by the use of the finite element, finite volume, finite difference, collocation, or spectral methods and then possibly linearization by Newton's method. The resulting linear systems can range from having a few thousand unknowns to billions of unknowns, thus requiring the largest parallel computers currently available. Large-scale linear systems also arise directly in optimization, economics modeling, and many other non-PDE-based models. The main focus of PETSc is in solving linear systems arising from PDE-based models, though it is applied to other problems as well. PETSc also has limited support for dense matrix computations (through an interface to LAPACK and PLAPACK); but if the computation involves exclusively dense matrices, then PLAPACK or ScaLAPACK are appropriate libraries.

PETSc is a software library intended for use by mathematicians, scientists, and engineers with a solid understanding of programming, some basic understanding of the issues in parallel computing (though they need not have programmed in MPI), a basic understanding of numerical analysis, and an understanding of the basics of linear algebra. It has a higher and deeper learning curve than do software packages such as Matlab. PETSc can be used directly from Fortran 77/90, C/C++, and Python, with bindings that are specialized to each language.

PETSc is used by a variety of parallel PDE solver libraries, including freeCFD, a general-purpose CFD solver; OpenFVM, a finite-volume-based CFD solver; OOFEM, an object-oriented finite element library; libMesh, an adaptive finite element library; and DEAL.II, a sophisticated C++- based finite element simulation package. For large-scale optimization and the scalable computation of eigenvalues, PETSc has two children packages, TAO and SLEPc, that use all of the PETSc parallelism and linear solver infrastructure.

The emphasis of the PETSc solvers is on iterative methods for the solution of linear systems, but it provides its own efficient sequential direct (LU and Cholesky factorization-based) solvers as well as interfaces to several parallel direct solvers; see Table 1. PETSc has a unique configuration system that will automatically download and install the multitude of optional packages that it can use. In addition to the direct solvers, it can use several parallel partitioning packages as well as preconditioners in the hypre and TRILINOS solver packages; see Table 2. A key design feature of PETSc is the composibility of its linear solvers. Two or more solvers may be combined in various ways: by splittings, multigrid and Schur complementing to produce efficient, problem-specific solvers.

The parallelism in PETSc is usually achieved by domain decomposition. The geometry on which the PDE is being solved is divided among the processes, and each process is assigned the unknowns and matrix elements associated with that domain. The communication required during the solution process is then nearest neighbor ghost (halo) point updates and global reductions (using **MPI_Allreduce()**) over a MPI communicator. PETSc has optimized code based on the inspector-executor model to perform the ghost point updates.

In addition to its broad support for linear solvers, PETSc provides robust implementations of Newton's method for nonlinear systems of equations. These include a variety of line-search and trust-region schemes for globalization. The solvers are extensible, allowing easy provision of user convergence tests, line-search strategies, and damping strategies. Several variants of the Eisenstat-Walker convergence criteria for inexact Newton solves are available. There is also support for grid sequencing to efficiently generate high-quality initial solutions for fine grids. To compute the Jacobians commonly needed for Newton's method, PETSc provides coloring of sparse matrices and efficient computation of the Jacobian entries using the coloring with finite differencing, ADIC (automatic differentiation for C programs), and ADIFOR, (automatic differentiation for Fortran 77 programs). All of these run scalably in parallel.

PETSc also provides a family of implicit and explicit ODE integrators, including an extensive suite of explicit Runge-Kutta methods. The implicit methods support all the functionality of the PETSc nonlinear solvers and use of any of the Krylov methods and preconditioners. The more sophisticated adaptive time-stepping ODE integrators of SUNDIALS can also be used with PETSc and allow use of all available PETSc preconditioners.

Provided in PETSc is an infrastructure for profiling the parallel performance of the application and the solvers it uses, including floating-point operations done, messages, and sizes of messages sent and received. It provides the results in a table that indicates the percentage of time spent in the various parts of the solver and application.

Development of PETSc was started in 1995 by Bill Gropp, Lois Curfman McInnes, and Barry Smith at Argonne National Laboratory. They were joined shortly later by Satish Balay. Aside from a small amount of National Science Foundation funding in the mid-1990s, the U.S. Department of Energy has provide the funding for PETSc development and support. Since its origin, PETSc has received contributions from many of its users.

PETSc was the winner of a 2009 R&D award. It has formed the basis of three Gordon Bell winning application codes in 1999, 2003, and 2004 as well as several Gordon Bell finalists.

Library Design

PETSc follows the distributed-memory SPMD model of MPI, with the flexibility of having different types of computation running on different processes. Specifically PETSc allows users to create their own MPI communicators and designate computations for PETSc to perform on each of these communicators. A typical application code written with PETSc requires very few MPI calls by the developer.

PETSc is written in C using the object-oriented programming techniques of data encapsulation, polymorphism, and inheritance. Opaque objects are defined that contain function tables (using C function pointers) used to call the code appropriate for the underlying data structures. The six main abstract classes in PETSc are the **Vec** vector class for managing the system solutions, the **Mat** matrix class for managing the sparse matrices, the **KSP** Krylov solver class for managing the iterative accelerators, the **PC** preconditioner class, the **SNES** nonlinear solver class, and the **TS** ordinary differential equations (ODE) integrator class. The **DM** helper class manages transferring information about grids and discretizations into the **Vec** and **Mat** classes. Virtually all of the parallel communication required by PETSc (the MPI message passing and collective calls) takes place within these objects. The constructor for each PETSc object takes an MPI communicator, which determines on what processes the object and its computations will reside. The most common are MPI_COMM_WORLD, in which the object is distributed across all the user's processes (and computations involving the object will require communication with that communicator), and MPI_COMM_SELF, in which the object lives on just that process and no communication is ever required for its computations.

A typical application that requires linear solvers has a structure as depicted in Figure 1. In this example, the **DA** object, which is an implementation of the **DM** class for structured grids, is used to construct the needed sparse matrix and vectors to contain the solution and right-hand side; it is said to be a factory for **Vec** and **Mat** objects. Once the numerical values of the matrix are set, in this case by calls to **MatSetValues()**, the matrix is provided to the linear solver via **KSPSetOperators()**. Since **MatSetValues()** may be called with values that belong to any process, the calls to **MatAssemblyBegin/End()** are used to communicate the values to the process where they belong. Values may be set into vectors by using either **VecSetValues()**, with a concluding **VecAssemblyBegin/End()**, as with matrices or by access the array of values using **VecGetArray()**, **VecGetArrayF90()**, or **DAVecGetArray()** and putting values directly into the array. In this case no communication of off-process values is done by PETSc.

A typical application that requires nonlinear solvers has a structure as depicted in Figure 2. In addition to serving as a factory for the Jacobian sparse matrix and solution vector (as in the linear case), the **DA** object is used as a factory for the ghosted representation of the solution **xlocal** and perform the ghost point updates with **DAGlobalToLocal()** in the routines **ComputeFunction()** and **ComputeJacobian()**. These callback routines are registered with the nonlinear solver object **SNES** with the routines **SNESSetFunction()** and **SNESSetJacobian()**. They are called when needed by the solver class.

A typical application that requires ODE integration has a structure as depicted in Figure 3. This simple example uses the Python interface to **TS** where the entire discretized ODE (in this case using the backward Euler method) is provided directly as the function and Jacobian. It is also possible to provide the function and Jacobian of the right-hand side of the ODE, that is, $u_t = F(u)$, and have the **TS** class manage the ODE discretization, with either an explicit or implicit scheme.

Each PETSc object has a method **XXXSetFromOptions()** that allows runtime control of almost all of the solver options through which is called the PETSc options database. Command-line arguments (as keyword value pairs) are stored in a simple database. The **XXXSetFromOptions()** routines then search the database and select any appropriate options and apply them. For example, the option **-ksp_type gmres** is used by **KSPSetFromOptions()** to call **KSPSetType()** to set the solver type to GMRES. This database may also be used directly by user code.

Also common to all classes are the **XXXView()** methods. These provide a common interface to printing and saving information about any object to a **PetscView** object, which is an abstract representation of a binary file, a text file (like **stdout**), a graphical window for drawing, or a Unix socket. For example, **MatView(Mat A,PetscViewer v)** will present the matrix in a wide variety of ways depending on the viewer type and its state. Calling the viewer method on a solver class, such as **SNES**, displays the type of solver and all its options; see Figure 4 for an example. Note that the figure displays both the nonlinear and linear solver options.

For the **Mat** class, PETSc provides several realizations. The most important of these are the following:

- Compressed sparse row (CSR) format.
- Point-block version of the CSR where a single index is used for small dense blocks of the matrix.
- Symmetric version of the point-block CSR that requires roughly one-half the storage.
- User-provided format (via inheritance).
- "Matrix-free" representations, where the matrix entries are not explicitly stored, but instead matrix-vector products are performed by using one of the following:
  - Finite differencing of the function evaluations.
  - Automatic differentiation of the function evaluations using either ADIC, for C language code or ADIFOR, for Fortran 77 language code.
  - User-provided function.

Since PETSc is focused on PDE problems, row-based storage of the sparse matrices (each process holds a collection of contiguous rows of the matrix) is satisfactory for higher-performance parallel matrix operations. Hence, all of PETSc's built-in sparse matrix implementations use this approach. Custom formats can be provided to handle parallelism for "arrow-head" matrices where row-based distribution does not scale.

PETSc has the point-block-based storage of sparse matrices for faster performance. The speed of sparse matrix computations is essentially always strongly limited by the memory bandwidth of the system, not by the CPU speed. The reason is that sparse matrix computations involve few operations per matrix entry. For example, for matrix vector products there are two floating-point operations (a multiply and an add) for each entry in the matrix. Memory bandwidth-limited computations are sometimes said to hit the memory wall. In the CSR format there is a column index for every nonzero entry in the matrix, and the matrix-vector product is coded as $y[i] = \sum_{j=nz_{i-1}}^{j<nz_i} aa[j] * x[aj[j]]$. For each multiply in the computation a double-precision value of $aa[]$ must be loaded as well as an integer value $aj[]$. Thus, 12 bytes are loaded per multiply. In the point-block CSR format (with block size bs), there is one column index per block and the matrix-vector product may be coded as $y[bs * i + k] = \sum_{j=nz_{i-1}}^{j<nz_i} \sum_{l=0}^{l<bs} aa[bs * (j + l) + k] * x[aj[j] + l]$. Here, for each $(bs * bs)$ multiplies, $(bs * bs)$ loads of $aa[]$ are needed, but only a single integer $aj[]$. For even moderate block size, this approach reduces the loads per multiply from 12 to less than 8.5 bytes. In addition, the same $x[]$ values are used repeatedly for each k, and a smart unrolling can keep the reused values in registers. Using the block CSR when appropriate, depending on the particular processor, can improve the performance of the sparse matrix operators by a factor of 2 to 3.

The **KSP** Krylov accelerator class provides over a dozen Krylov methods; see Table 3. The data encapsulation and polymorphic design of the **Vec**, **Mat**, and **PC** classes in PETSc allow the immediate use of any of their implementations with any of the Krylov solvers. When possible, these are implemented to allow left, right, or symmetric preconditioning and the use of various norms of the residual in the convergence tests including the "natural" (energy) norm. Custom convergence tests and monitoring routines can be provided to any of the solvers.

The **PC** preconditioners class contains a variety of both classical and modern preconditioners including incomplete factorizations, domain decomposition methods, and multigrid methods. See Table 2 for a partial list. In addition, several preconditioner classes are designed to allow composition of solvers. These include **PCKSP**, which allows using a Krylov method as a preconditioner; **PCFieldSplit**, which allows constructing solvers by composing solvers for different fields of the solution; **KSPCOMPOSITE** which allows combining arbitrary solvers; and **PCGALERKIN** which constructs preconditioners by the Galerkin process. **PCFieldSplit** preconditioners are often called block preconditioners, for example, when one field is velocity and another pressure, the resulting Stokes solver is often solved with one block for velocities and one for pressure.

Applications

A wide variety of simulation applications have been written by using PETSc. These include fluid flow for aircraft, ship, and automobile design; blood flow simulation for medical device design; porous media flow for oil reservoir simulation for energy development and groundwater contamination modeling; modeling of materials properties; economic modeling; structural mechanics for construction design; combustion modeling; and nuclear fission and fusion for energy development.

PETSc-Fun3d was an early application based on Kyle Anderson's NASA code, Fun3d, that solves the Euler and Navier-Stokes equations including both compressible and incompressible on unstructured grids. PETSc-Fun3d won a Gordon Bell special prize in 1999 running on over 6,000 of the ASCI Red processors. This application, the dissertation work of Dinesh Kaushik, motivated many of the early optimizations of PETSc.

The forward and inverse modeling of earthquakes using the PETSc algebraic solvers resulting in 2003 Gordon Bell special prize.

The algebraic multigrid solver Prometheus was written by Mark Adams using the PETSc **Vec**, **Mat**, **KSP**, and **PC** classes. It takes advantage of the block CSR sparse format in PETSc to maximize per-

formance. It was used in the simulation of whole-bone micromechanics with over half a billion degrees of freedom, resulting in a 2004 Gordon Bell special prize.

PETSc has been used by several research groups in the simulation of heart arrhythmias, which are the cause of the majority of sudden cardiac deaths. This application involves solving the nonlinear bidomain equations, which are two coupled partial differential equations that model the intracellular , and extracellular potential of the heart. Numerical solutions to these equations (and more sophisticated models) explains much of the electrical behavior of the heart, including defibrillation.

PFLOTRAN, led by Peter Lichtner of Los Alamos National Laboratory, is a subsurface flow and contaminant transport simulator that uses the PETSc **DM** class to manage the parallelism of its mesh, the **SNES** nonlinear solver class for the solutions needed at each time step, the **Mat** class to contain the sparse Jacobians, and the **Vec** class for its flow and contaminant's solutions. It has been run on up to 64,000 cores of the Cray XT5 and has been used to more accurately model uranium plumes at DOE's Hanford site.

The UNIC neutronics package developed by Mike Smith and Dinesh Kaushik of Argonne National Laboratory has run full reactor core simulations on 160,000 cores of the IBM Blue Gene/P. It supports both the second-order Pn and Sn methods with dozens of energy groups. It parallelizes simultaneously over the geometry by means of domain decomposition and angles using a hierarchy of MPI communicators and PETSc solver objects.

RELATED ENTRIES

MPI
Distributed Memory Computing
SPMD Programs
Domain Decomposition
BLAS
ScaLAPACK
LAPACK
UMFPACK
SPAI
PLAPACK
Super LU
Multigrid
METIS
CHACO
Scalability
Memory wall

BIBLIOGRAPHIC NOTES AND FURTHER READING

The PETSc website is the best location for up-to-date information on PETSc [8]. A complete list of external packages that PETSc can use is given in [5]. More details of the applications developed by using PETSc can be found at [7]. Further details on the design decisions made in PETSc may be found in [2].

Other related parallel solver packages include TRILINOS [9], hypre [10], and SUNDIALS [11]. TRILINOS is a large, general-purpose solver package much in the spirit of PETSc and written largely in C++; it currently has little support for use from Fortran. The hypre package specializes in high-performance preconditioners and includes a scalable algebraic multigrid solver BoomerAMG. SUNDIALS specializes in nonlinear solvers and adaptive ODE integrators; it expects the required linear solver to be provided by the user or another package. Many of the solvers in these other packages can be called through PETSc.

Acknowledgments

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

BIBLIOGRAPHY

1. S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith and H. Zhang. PETSc Users Manual, Argonne National Laboratory Technical Report ANL0-95/11 - Revision 3.0.0, 2008.

2. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset and H. P. Langtangen, eds., Birkhauser Press, pp. 163–202, 1997.

3. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Software for the Scalable Solution of PDEs, in CRPC Handbook of Parallel Computing, J. Dongarra, I. Foster, G. Fox, B. Gropp, K. Kennedy, L. Torczon, A. White, eds., Morgan Kaufmann Publishers, 2002.

4. J. Dongarra's freely available software for linear algebra, http://www.netlib.org/utk/people/JackDongarra/la-sw.html.

5. List of external software packages available from PETSc, http://www.mcs.anl.gov/petsc/petsc-as/miscellaneous/external.html.

6. Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition, SIAM, 2003.

7. Partial list of applications written using PETSc, http://www.mcs.anl.gov/petsc/petsc-as/publications/petscapps.html.

8. PETSc's webpage, http://www.mcs.anl.gov/petsc.

9. TRILINOS's webpage, http://trilinos.sandia.gov.

10. hypre's webpage, https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html.

11. SUNDIAL's webpage, https://computation.llnl.gov/casc/sundials/main.html.

.

Table 1: Partial list of Direct Solvers Available in PETSc

Factorization	Package	Complex Numbers Support	Parallel Support
LU	PETSc	x	
	SuperLU	x	
	SuperLU_Dist	x	x
	MUMPS	x	x
	Spooles	x	x
	PaStiX	x	x
	IBM's ESSL		
	UMFPACK		
	LUSOL		
Cholesky	PETSc	x	
	Spooles	x	x
	MUMPS	x	x
	PaStiX	x	x
	DSCPACK	x	

Table 2: Partial list of Preconditioners Available in PETSc

Preconditioner	Package	Complex Numbers Support	Parallel Support
ICC(k)	PETSc	x	
ILU(k)	PETSc	x	
	Euclid/hypre		x
ILUdt	pilut/hypre		x
Jacobi	PETSc	x	x
SOR	PETSc	x	
Block Jacobi	PETSc	x	x
Additive Schwarz	PETSc	x	x
Geometric multigrid	PETSc	x	x
Algebraic multigrid	BoomerAMG/hypre		x
	ML/TRILINOS		x
Approximate inverse	SPAI		x
	Parasails/hypre		x

```
      program main   !   Solves the linear system  J x = f
#include "finclude/petscalldef.h"
      use petscksp; use petscda
      Vec x,f; Mat J; DA da; KSP ksp; PetscErrorCode ierr
      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)

      call DACreate1d(MPI_COMM_WORLD,DA_NONPERIODIC,8,1,1,PETSC_NULL_INTEGER,da,ierr)
      call DACreateGlobalVector(da,x,ierr); call VecDuplicate(x,f,ierr)
      call DAGetMatrix(da,MATAIJ,J,ierr)

      call ComputeRHS(da,f,ierr)
      call ComputeMatrix(da,J,ierr)

      call KSPCreate(MPI_COMM_WORLD,ksp,ierr)
      call KSPSetOperators(ksp,J,J,SAME_NONZERO_PATTERN,ierr)
      call KSPSetFromOptions(ksp,ierr)
      call KSPSolve(ksp,f,x,ierr)

      call MatDestroy(J,ierr); call VecDestroy(x,ierr); call VecDestroy(f,ierr)
      call KSPDestroy(ksp,ierr); call DADestroy(da,ierr)
      call PetscFinalize(ierr)
      end
      subroutine  ComputeRHS(da,x,ierr)
#include "finclude/petscalldef.h"
      use petscda
      DA da; Vec x;PetscErrorCode ierr;PetscInt xs,xm,i,mx; PetscScalar hx;PetscScalar, pointer::xx(:)
      call DAGetInfo(da,PETSC_NULL_INTEGER,mx,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,...
      call DAGetCorners(da,xs,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,xm,PETSC_NULL_INTEGER,...
      hx     = 1.d0/(mx-1)
      call VecGetArrayF90(x,xx,ierr)
      do i=xs,xs+xm-1
          xx(i) = i*hx
      enddo
      call VecRestoreArrayF90(x,xx,ierr)
      return
      end
      subroutine ComputeMatrix(da,J,ierr)
#include "finclude/petscalldef.h"
      use petscda
      Mat J; DA da; PetscErrorCode ierr; PetscInt xs,xm,i,mx; PetscScalar hx
      call DAGetInfo(da,PETSC_NULL_INTEGER,mx,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,...
      call DAGetCorners(da,xs,PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,xm,PETSC_NULL_INTEGER,...
      hx     = 1.d0/(mx-1)
      do i=xs,xs+xm-1
        if ((i .eq. 0) .or. (i .eq. mx-1)) then
          call MatSetValue(J,i,i,1d0,INSERT_VALUES,ierr)
        else
          call MatSetValue(J,i,i-1,-hx,INSERT_VALUES,ierr)
          call MatSetValue(J,i,i+1,-hx,INSERT_VALUES,ierr)
          call MatSetValue(J,i,i,2*hx,INSERT_VALUES,ierr)
        endif
      enddo
      call MatAssemblyBegin(J,MAT_FINAL_ASSEMBLY,ierr); call MatAssemblyEnd(J,MAT_FINAL_ASSEMBLY,ierr)
      return
      end
```

Figure 1: Example of Linear Solver Usage in PETSc in Fortran 90

```
static char help[] = "Solves -Laplacian u - exp(u) = 0,  0 < x < 1\n\n";
#include "petscda.h"
#include "petscsnes.h"
extern PetscErrorCode ComputeFunction(SNES,Vec,Vec,void*),ComputeJacobian(SNES,Vec,Mat*,Mat*,...

int main(int argc,char **argv) {
  SNES snes; Vec x,f; Mat J; DA da;
  PetscInitialize(&argc,&argv,(char *)0,help);

  DACreate1d(PETSC_COMM_WORLD,DA_NONPERIODIC,8,1,1,PETSC_NULL,&da);
  DACreateGlobalVector(da,&x); VecDuplicate(x,&f);
  DAGetMatrix(da,MATAIJ,&J);

  SNESCreate(PETSC_COMM_WORLD,&snes);
  SNESSetFunction(snes,f,ComputeFunction,da);
  SNESSetJacobian(snes,J,J,ComputeJacobian,da);
  SNESSetFromOptions(snes);
  SNESSolve(snes,PETSC_NULL,x);

  MatDestroy(J); VecDestroy(x); VecDestroy(f); SNESDestroy(snes); DADestroy(da);
  PetscFinalize();
  return 0;}
PetscErrorCode ComputeFunction(SNES snes,Vec x,Vec f,void *ctx) {
  PetscInt i,Mx,xs,xm; PetscScalar *xx,*ff,hx; DA da = (DA) ctx; Vec xlocal;
  DAGetInfo(da,PETSC_IGNORE,&Mx,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,...
  hx     = 1.0/(PetscReal)(Mx-1);
  DAGetLocalVector(da,&xlocal);DAGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal);DAGlobalToLocalEnd(da,x,...
  DAVecGetArray(da,xlocal,&xx); DAVecGetArray(da,f,&ff);
  DAGetCorners(da,&xs,PETSC_NULL,PETSC_NULL,&xm,PETSC_NULL,PETSC_NULL);

  for (i=xs; i<xs+xm; i++) {
    if (i == 0 || i == Mx-1) ff[i] = xx[i]/hx;
    else  ff[i] =  (2.0*xx[i] - xx[i-1] - xx[i+1])/hx - hx*PetscExpScalar(xx[i]);
  }
  DAVecRestoreArray(da,xlocal,&xx); DARestoreLocalVector(da,&xlocal);DAVecRestoreArray(da,f,&ff);
  return 0;}
PetscErrorCode ComputeJacobian(SNES snes,Vec x,Mat *J,Mat *B,MatStructure *flag,void *ctx){
  DA da = (DA) ctx; PetscInt i,Mx,xm,xs; PetscScalar hx,*xx; Vec xlocal;
  DAGetInfo(da,PETSC_IGNORE,&Mx,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,...
  hx = 1.0/(PetscReal)(Mx-1);
  DAGetLocalVector(da,&xlocal);DAGlobalToLocalBegin(da,x,INSERT_VALUES,xlocal);DAGlobalToLocalEnd(da,x,...
  DAVecGetArray(da,xlocal,&xx);
  DAGetCorners(da,&xs,PETSC_NULL,PETSC_NULL,&xm,PETSC_NULL,PETSC_NULL);

  for (i=xs; i<xs+xm; i++) {
    if (i == 0 || i == Mx-1) { MatSetValue(*J,i,i,1.0/hx,INSERT_VALUES);}
    else {
      MatSetValue(*J,i,i-1,-1.0/hx,INSERT_VALUES);
      MatSetValue(*J,i,i,2.0/hx - hx*PetscExpScalar(xx[i]),INSERT_VALUES);
      MatSetValue(*J,i,i+1,-1.0/hx,INSERT_VALUES);
    }
  }
  MatAssemblyBegin(*J,MAT_FINAL_ASSEMBLY);MatAssemblyEnd(*J,MAT_FINAL_ASSEMBLY);*flag = SAME_NONZERO_...
  DAVecRestoreArray(da,xlocal,&xx);DARestoreLocalVector(da,&xlocal);
  return 0;}
```

Figure 2: Example of Nonlinear Solver Usage in PETSc in C

```
import sys, petsc4py
petsc4py.init(sys.argv)
from petsc4py import PETSc
import math

class MyODE:
  def __init__(self,da):
    self.da = da
  def function(self, ts,t,x,f):
    mx = da.getSizes(); mx = mx[0]; hx = 1.0/mx
    (xs,xm) = da.getCorners(); xs = xs[0]; xm = xm[0]
    xx = da.createLocalVector()
    da.globalToLocal(x,xx)
    dt = ts.getTimeStep()
    x0 = ts.getSolution()
    if xs == 0: f[0] = xx[0]/hx; xs = 1;
    if xs+xm >= mx: f[mx-1] = xx[xm-(xs==1)]/hx; xm = xm-(xs==1);
    for i in range(xs,xs+xm-1):
      f[i] = (xx[i-xs+1]-x0[i])/dt + (2.0*xx[i-xs+1]-xx[i-xs]-xx[i-xs+2])/hx - hx*math.exp(xx[i-xs+1])
    f.assemble()
  def jacobian(self,ts,t,x,J,P):
    mx = da.getSizes(); mx = mx[0]; hx = 1.0/mx
    (xs,xm) = da.getCorners(); xs = xs[0]; xm = xm[0]
    xx = da.createLocalVector()
    da.globalToLocal(x,xx)
    x0 = ts.getSolution()
    dt = ts.getTimeStep()
    P.zeroEntries()
    if xs == 0: P.setValues([0],[0],1.0/hx); xs = 1;
    if xs+xm >= mx: P.setValues([mx-1],[mx-1],1.0/hx); xm = xm-(xs==1);
    for i in range(xs,xs+xm-1):
      P.setValues([i],[i-1,i,i+1],[-1.0/hx,1.0/dt+2.0/hx-hx*math.exp(xx[i-xs+1]),-1.0/hx])
    P.assemble()
    return True # same_nz

da = PETSc.DA().create([9],comm=PETSc.COMM_WORLD)
f = da.createGlobalVector()
x = f.duplicate()
J = da.getMatrix(PETSc.MatType.AIJ);

ts = PETSc.TS().create(PETSc.COMM_WORLD)
ts.setProblemType(PETSc.TS.ProblemType.NONLINEAR)
ts.setType('python')

ode = MyODE(da)
ts.setFunction(ode.function, f)
ts.setJacobian(ode.jacobian, J, J)

ts.setTimeStep(0.1)
ts.setDuration(10, 1.0)
ts.setFromOptions()
x.set(1.0)
ts.solve(x)
```

Figure 3: Example of ODE Usage in PETSc in Python

```
SNES Object:
  type: ls
    line search variant: SNESLineSearchCubic
    alpha=0.0001, maxstep=1e+08, minlambda=1e-12
  maximum iterations=50, maximum function evaluations=10000
  tolerances: relative=1e-08, absolute=1e-50, solution=1e-08
  KSP Object:
    type: fgmres
      GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization
      GMRES: happy breakdown tolerance 1e-30
    maximum iterations=10000, initial guess is zero
    tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
    right preconditioning
    using UNPRECONDITIONED norm type for convergence test
  PC Object:
    type: mg
      MG: type is FULL, levels=2 cycles=v
    Coarse grid solver -- level 0 presmooths=1 postsmooths=1 -----
      KSP Object:(mg_coarse_)
        type: preonly
      PC Object:(mg_coarse_)
        type: lu
          LU: out-of-place factorization
            matrix ordering: nd
          LU: tolerance for zero pivot 1e-12
          LU: factor fill ratio needed 1.875
        Matrix Object:
          type=seqaij, rows=64, cols=64
          total: nonzeros=1024, allocated nonzeros=1024
            using I-node routines: found 16 nodes, limit used is 5
    Down solver (pre-smoother) on level 1 smooths=1 --------------------
      KSP Object:(mg_levels_1_)
        type: gmres
          GMRES: restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization
          GMRES: happy breakdown tolerance 1e-30
        maximum iterations=1
        tolerances:  relative=1e-05, absolute=1e-50, divergence=10000
        left preconditioning
        using nonzero initial guess
        using PRECONDITIONED norm type for convergence test
      PC Object:(mg_levels_1_)
        type: ilu
          ILU: 0 levels of fill
          ILU: factor fill ratio allocated 1
          ILU: tolerance for zero pivot 1e-12
        Matrix Object:
          type=seqaij, rows=196, cols=196
          total: nonzeros=3472, allocated nonzeros=3472
            using I-node routines: found 49 nodes, limit used is 5
    Matrix Object:
      type=seqaij, rows=196, cols=196
      total: nonzeros=3472, allocated nonzeros=3472
        using I-node routines: found 49 nodes, limit used is 5
```

Figure 4: Example of Output Using **SNESView()**

Table 3: Partial list of Krylov Methods Available in PETSc

Richardson (simple) iteration, $x^{n+1} = x^n + B(b - Ax^n)$
Chebychev iteration
Conjugate gradient method
Bi-conjugate gradient
Bi-conjugate gradient stabilized (bi-CG-stab)
Conjugate residuals
Conjugate gradient squared
Minimum residuals (MINRES)
Generalized minimal residual (GMRES)
Flexible GMRES (fGMRES)
transpose free quasi minimal residuals (QMR)