PETSc for Python

http://petsc4py.googlecode.com

Lisandro Dalcin
dalcinl@gmail.com

Centro Internacional de Métodos Computacionales en Ingeniería
Consejo Nacional de Investigaciones Científicas y Técnicas
Santa Fe, Argentina

January, 2011
Python for parallel scientific computing
PASI, Valparaíso, Chile

Outline

What is **petsc4py**?

Python bindings for **PETSc**, the *Portable Extensible Toolkit for Scientific Computation*.

A *good friend* of **petsc4py** is:

- **mpi4py**: Python bindings for **MPI**, the *Message Passing Interface*.

Other two projects depend on petsc4py:

- **slepc4py**: Python bindings for **SLEPc**, the *Scalable Library for Eigenvalue Problem Computations*.
- **tao4py**: Python bindings for **TAO**, the *Toolkit for Advanced Optimization*.

Implementation

Implemented with Cython `http://www.cython.org`

- Code base far easier to write, maintain, and extend.
- Faster than other solutions (mixed Python and C codes).
- Easier to cross language boundaries (reuse C/C++/Fortran).

Features – PETSc components

- **Index Sets**: permutations, indexing into vectors, renumbering.
- **Vectors**: sequential and distributed.
- **Matrices**: sequential and distributed, sparse and dense.
- **Distributed Arrays**: regular grid-based problems.
- **Linear Solvers**: Krylov subspace methods.
- **Preconditioners**: sparse direct solvers, multigrid
- **Nonlinear Solvers**: line search, trust region, matrix-free.
- **Timesteppers**: time-dependent, linear and nonlinear PDE's.

Features – Interoperability

Support for wrapping other PETSc-based codes.

- You can use **SWIG** (*typemaps* provided).
- You can use **F2Py** (`fortran` attribute).

Features – Interoperability – **SWIG**

```
%module MyPDE

%include petsc4py/petsc4py.i

%{
#include "MyPDE.h"
%}

typedef struct Params {
  double alpha;
  double beta;
  double gamma;
} Params;

PetscErrorCode FormInitGuess(DA da, Vec x, Params *p);
PetscErrorCode FormFunction(DA da, Vec x, Vec F, Params *p);
PetscErrorCode FormJacobian(DA da, Vec x, Mat J, Params *p);
```

Features – Interoperability – **F2Py**

```
1   python module MyPDE
2   interface
3
4       subroutine FormInitGuess(da, x, params, ierr)
5         integer, intent(in)       :: da, x
6         real(kind=8), intent(in) :: params(3)
7         integer, intent(out)      :: ierr
8       end subroutine FormInitGuess
9
10      subroutine FormFunction(da, x, F, params, ierr)
11        integer, intent(in)       :: da, x, F
12        real(kind=8), intent(in) :: params(3)
13        integer, intent(out)      :: ierr
14      end subroutine FormFunction
15
16      subroutine FormJacobian(da, x, J, params, ierr)
17        integer, intent(in)       :: da, x, J
18        real(kind=8), intent(in) :: params(3)
19        integer, intent(out)      :: ierr
20      end subroutine FormJacobian
21
22  end interface
23  end python module MyPDE
```

Vectors (Vec) – Conjugate Gradients Method

$cg(A, x, b, i_{max}, \epsilon)$:

$i \Leftarrow 0$

$r \Leftarrow b - Ax$

$d \Leftarrow r$

$\delta_0 \Leftarrow r^T r$

$\delta \Leftarrow \delta_0$

while $i < i_{max}$ and

$\delta > \delta_0 \epsilon^2$ do :

$q \Leftarrow Ad$

$\alpha \Leftarrow \dfrac{\delta}{d^T q}$

$x \Leftarrow x + \alpha d$

$r \Leftarrow r - \alpha q$

$\delta_{old} \Leftarrow \delta$

$\delta \Leftarrow r^T r$

$\beta \Leftarrow \dfrac{\delta}{\delta_{old}}$

$d \Leftarrow r + \beta d$

$i \Leftarrow i + 1$

```
1   def cg(A, b, x, imax=50, eps=1e-6):
2       """
3       A, b, x   : matrix, rhs, solution
4       imax      : maximum iterations
5       eps       : relative tolerance
6       """
7       # allocate work vectors
8       r = b.duplicate()
9       d = b.duplicate()
10      q = b.duplicate()
11      # initialization
12      i = 0
13      A.mult(x, r)
14      r.aypx(-1, b)
15      r.copy(d)
16      delta_0 = r.dot(r)
17      delta = delta_0
18      # enter iteration loop
19      while (i < imax and
20             delta > delta_0 * eps**2):
21          A.mult(d, q)
22          alpha = delta / d.dot(q)
23          x.axpy(+alpha, d)
24          r.axpy(-alpha, q)
25          delta_old = delta
26          delta = r.dot(r)
27          beta = delta / delta_old
28          d.aypx(beta, r)
29          i = i + 1
30      return i, delta**0.5
```

Matrices (Mat) [1]

```
1  from petsc4py import PETSc
2
3  # grid size and spacing
4  m, n  = 32, 32
5  hx = 1.0/(m-1)
6  hy = 1.0/(n-1)
7
8  # create sparse matrix
9  A = PETSc.Mat()
10 A.create(PETSc.COMM_WORLD)
11 A.setSizes([m*n, m*n])
12 A.setType('aij') # sparse
13
14 # precompute values for setting
15 # diagonal and non-diagonal entries
16 diagv = 2.0/hx**2 + 2.0/hy**2
17 offdx = -1.0/hx**2
18 offdy = -1.0/hy**2
```

Matrices (Mat) [2]

```
# loop over owned block of rows on this
# processor and insert entry values
Istart, Iend = A.getOwnershipRange()
for I in range(Istart, Iend) :
    A[I,I] = diagv
    i = I//n     # map row number to
    j = I - i*n # grid coordinates
    if i> 0  : J = I-n; A[I,J] = offdx
    if i< m-1: J = I+n; A[I,J] = offdx
    if j> 0  : J = I-1; A[I,J] = offdy
    if j< n-1: J = I+1; A[I,J] = offdy

# communicate off-processor values
# and setup internal data structures
# for performing parallel operations
A.assemblyBegin()
A.assemblyEnd()
```

Linear Solvers (KSP+PC)

```
# create linear solver,
ksp = PETSc.KSP()
ksp.create(PETSc.COMM_WORLD)

# use conjugate gradients method
ksp.setType('cg')
# and incomplete Cholesky
ksp.getPC().setType('icc')

# obtain sol & rhs vectors
x, b = A.getVecs()
x.set(0)
b.set(1)

# and next solve
ksp.setOperators(A)
ksp.setFromOptions()
ksp.solve(b, x)
```

Nonlinear Solvers (SNES) [1]

```python
from petsc4py import PETSc
from numpy import exp

m, n = 32, 32 # grid sizes
alpha = 6.8   # parameter

def Bratu2D(snes, X, F, alpha, m, n):
    # NumPy array <- Vec
    x = X.array.reshape(m, n)
    f = F.array.reshape(m, n)
    # setup 5-points stencil
    u  = x[1:-1, 1:-1] # center
    uN = x[1:-1,  :-2] # north
    uS = x[1:-1, 2:  ] # south
    uW = x[ :-2, 1:-1] # west
    uE = x[2:,   1:-1] # east
    # compute nonlinear function
    hx = 1.0/(m-1) # x grid spacing
    hy = 1.0/(n-1) # y grid spacing
    f[:,:] = x
    f[1:-1, 1:-1] = \
         (2*u - uE - uW) * (hy/hx) \
       + (2*u - uN - uS) * (hx/hy) \
       - alpha * exp(u)  * (hx*hy)
```

Nonlinear Solvers (SNES) [2]

```
# create  nonlinear solver
snes = PETSc.SNES().create()
# register the function in charge of
# computing the nonlinear residual
f = PETSc.Vec().createSeq(m*n)
snes.setFunction(Bratu2D, f,
                 args=(alpha, m, n))

# configure the nonlinear solver
# to use a matrix-free Jacobian
snes.setUseMF(True)
snes.getKSP().setType('cg')
snes.setFromOptions()

# solve the nonlinear problem
b = None # rhs = 0
x = f.duplicate() # solution
snes.solve(b, x)
```

Do not hesitate to ask for help . . .

- Mailing List: `petsc-users@mcs.anl.gov`
- Mail&Chat: `dalcinl@gmail.com`

Thanks!