PyKrylov Documentation

Release 0.1

D. Orban

December 15, 2008

CONTENTS

Release 0.1

Date December 15, 2008

This is the general documentation for PyKrylov, a pure Python implementation of common Krylov methods for the solution of systems of linear equations. The aim of this document is to cover usage of the package and how it can serve to benchmark solvers or preconditioners. Because the framework is very general and does not concern any specific application, the style is light and examples keep things simple. This is intentional so users have as much freedom as possible in modifying the example scripts.

Some examples use the efficient Pysparse sparse matrix library to simulate functions that return matrix-vector products.

Comments on this document or the software package should be posted on the LightHouse PyKrylov page.

Contents:

INTRODUCTION TO PYKRYLOV

PyKrylov aims to provide a flexible implementation of the most common Krylov method for solving systems of linear equations in pure Python. The sole requirement is Numpy for fast array operations.

PyKyrlov is in the *very early* stages of development. If you give it a whirl, please let me know what you think at the LightHouse PyKrylov page. Feel free to post bug reports, feature requests and praises.

1.1 Example

The following code snippet solves a linear system with a 1D Poisson coefficient matrix. Since this matrix is symmetric and positive definite, the conjugate gradient algorithm is used:

```python
import numpy as np
from math import sqrt
from pykrylov.cg import CG


def Poisson1dMatvec(x):
    # Matrix-vector product with a 1D Poisson matrix
    y = 2*x
    y[:-1] -= x[1:]
    y[1:] -= x[:-1]
    return y

e = np.ones(100)
rhs = Poisson1dMatvec(e)
cg = CG(Poisson1dMatvec, matvec_max=200)
cg.solve(rhs)


print 'Number of matrix-vector products: ', CG.nMatvec
print 'Residual: %8.2e' % CG.residNorm
print 'Error: %8.2e' % np.linalg.norm(e - CG.bestSolution)/sqrt(n)
```

On my machine, the above script produces:

```
Number of matrix-vector products: 50
Residual: 7.39e-14
Error: 2.06e-15
```

GENERIC TEMPLATE FOR KRYLOV METHODS

2.1 The `generic` Module

A general template for implementing iterative Krylov methods. This module defines the *KrylovMethod* generic class. Other modules subclass *KrylovMethod* to implement specific algorithms.

class KrylovMethod(*matvec, **kwargs*)

> **solve**(*rhs, **kwargs*)
>> This is the `solve()` method of the abstract KrylovMethod class. The class must be specialized and this method overridden.

THE CONJUGATE GRADIENT METHOD

3.1 The `cg` Module

class CG(*matvec, **kwargs*)

    Bases: `pykrylov.generic.generic.KrylovMethod`

    A pure Python implementation of the conjugate gradient (CG) algorithm. The conjugate gradient algorithm may be used to solve symmetric positive definite systems of linear equations, i.e., systems of the form

        A x = b

    where the matrix A is square, symmetric and positive definite. This is equivalent to solving the unconstrained convex quadratic optimization problem

        minimize -<b,x> + 1/2 <x, Ax>

    in the variable x.

    CG performs 1 matrix-vector product, 2 dot products and 3 daxpys per iteration.

    If a preconditioner is supplied, it needs to solve one preconditioning system per iteration.

    solve(*rhs, **kwargs*)

        Solve a linear system with *rhs* as right-hand side by the CG method. The vector *rhs* should be a Numpy array. An optional argument *guess* may be supplied, with an initial guess as a Numpy array. By default, the initial guess is the vector of zeros.

3.2 Example

The following script runs through a list of symmetric positive definite matrices in Matrix Market format, builds the right-hand side so that the exact solution of the system is the vector of ones, and applies the conjugate gradient to each of them. The sparse matrices are handled by way of the Pysparse package.

```python
from pykrylov.cg import CG
from pysparse import spmatrix
from pysparse.pysparseMatrix import PysparseMatrix as sp
import numpy as np
from math import sqrt

matrices = ['1138bus.mtx', 'bcsstk08.mtx', 'bcsstk09.mtx', 'bcsstk10.mtx']
matrices += ['bcsstk11.mtx', 'bcsstk18.mtx', 'bcsstk19.mtx' ]

hdr = '%15s  %5s  %5s  %8s  %8s  %8s' % ('Name', 'Size', 'Mult',
```

```
                                         'Resid0', 'Resid', 'Error')
print hdr

for matName in matrices:
    A = sp( matrix=spmatrix.ll_mat_from_mtx(matName) )
    n = A.shape[0]
    e = np.ones(n)
    rhs = A * e
    cg = CG(lambda v: A*v, matvec_max=2*n)
    cg.solve(rhs)
    err = np.linalg.norm(cg.bestSolution - e)/sqrt(n)
    print '%15s  %5d  %5d  %8.2e  %8.2e  %8.2e' % (matName, n, cg.nMatvec,
                                                   cg.residNorm0,
                                                   cg.residNorm, err)
```

The script above produces the following output:

```
        Name    Size    Mult    Resid0      Resid     Error
 1138bus.mtx    1138    1759  1.46e+03  1.44e-03  1.30e-05
bcsstk08.mtx    1074    1255  8.74e+10  7.41e+04  7.54e-02
bcsstk09.mtx    1083     180  3.17e+08  1.84e+02  8.59e-06
bcsstk10.mtx    1086    1753  8.10e+07  7.30e+01  1.04e-03
bcsstk11.mtx    1473    1689  5.43e+09  4.88e+03  5.68e-02
bcsstk18.mtx   11948    9729  2.79e+11  2.49e+05  2.85e-01
bcsstk19.mtx     817     468  1.34e+15  1.26e+09  8.09e-01
```

Note that the default relative stopping tolerance is *1.0e-6* and is achieved in all cases.

THE CONJUGATE GRADIENT SQUARED METHOD

4.1 The `cgs` Module

class CGS (*matvec, **kwargs*)

Bases: pykrylov.generic.generic.KrylovMethod

A pure Python implementation of the conjugate gradient squared (CGS) algorithm. CGS may be used to solve unsymmetric systems of linear equations, i.e., systems of the form

A x = b

where the matrix A may be unsymmetric.

CGS requires 2 matrix-vector products with A, 3 dot products and 7 daxpys per iteration. It does not require products with the transpose of A.

If a preconditioner is supplied, CGS needs to solve two preconditioning systems per iteration.

solve (*rhs, **kwargs*)

Solve a linear system with *rhs* as right-hand side by the CGS method. The vector *rhs* should be a Numpy array. An optional argument *guess* may be supplied, with an initial guess as a Numpy array. By default, the initial guess is the vector of zeros.

4.2 Example

Here is an example using CGS on a linear system. The coefficient matrix is read from file in Matrix Market format:

```python
import numpy as np
from pykrylov.cgs import CGS as KSolver
from pysparse import spmatrix
from pysparse.pysparseMatrix import PysparseMatrix as sp

A = sp(matrix=spmatrix.ll_mat_from_mtx('jpwh_991.mtx'))
n = A.shape[0]
e = np.ones(n)
rhs = A*e

ks = KSolver( lambda v: A*v,
              matvec_max=2*n,
              verbose=False,
              outputStream=sys.stderr,
```

```
            reltol = 1.0e-5 )
ks.solve(rhs, guess = 1+np.arange(n, dtype=np.float))

print 'Number of matvecs: ', ks.nMatvec
print 'Initial/final res: %8.2e/%8.2e' % (ks.residNorm0, ks.residNorm)
print 'Direct error: %8.2e' % (np.linalg.norm(ks.bestSolution-e)/sqrt(n))
```

Running this script produces the following output:

```
Number of matvecs:  64
Initial/final res: 8.64e+03/4.72e-03
Direct error: 1.47e-04
```

THE BI-CONJUGATE GRADIENT STABILIZED METHOD

5.1 The `bicgstab` Module

class `BiCGSTAB`(*matvec, **kwargs*)

    Bases: `pykrylov.generic.generic.KrylovMethod`

    A pure Python implementation of the bi-conjugate gradient stabilized (Bi-CGSTAB) algorithm. Bi-CGSTAB may be used to solve unsymmetric systems of linear equations, i.e., systems of the form

        A x = b

    where the matrix A is unsymmetric and nonsingular.

    Bi-CGSTAB requires 2 matrix-vector products, 6 dot products and 6 daxpys per iteration.

    In addition, if a preconditioner is supplied, it needs to solve 2 preconditioning systems per iteration.

    This implementation is a preconditioned version of that given in [Kelley].

    `solve`(*rhs, **kwargs*)

        Solve a linear system with *rhs* as right-hand side by the Bi-CGSTAB method. The vector *rhs* should be a Numpy array. An optional argument *guess* may be supplied, with an initial guess as a Numpy array. By default, the initial guess is the vector of zeros.

5.2 Example

Here is an example using Bi-CGSTAB on a linear system. The coefficient matrix is read from file in Matrix Market format:

```python
import numpy as np
from pykrylov.cgs import BiCGSTAB as KSolver
from pysparse import spmatrix
from pysparse.pysparseMatrix import PysparseMatrix as sp

A = sp(matrix=spmatrix.ll_mat_from_mtx('jpwh_991.mtx'))
n = A.shape[0]
e = np.ones(n)
rhs = A*e

ks = KSolver( lambda v: A*v,
              matvec_max=2*n,
```

```
              verbose=False,
              outputStream=sys.stderr,
              reltol = 1.0e-5 )
ks.solve(rhs, guess = 1+np.arange(n, dtype=np.float))

print 'Number of matvecs: ', ks.nMatvec
print 'Initial/final res: %8.2e/%8.2e' % (ks.residNorm0, ks.residNorm)
print 'Direct error: %8.2e' % (np.linalg.norm(ks.bestSolution-e)/sqrt(n))
```

Running this script produces the following output:

```
Number of matvecs:  57
Initial/final res: 8.64e+03/5.18e-02
Direct error: 3.35e-03
```

THE TRANSPOSE-FREE QUASI-MINIMUM RESIDUAL METHOD

6.1 The `tfqmr` Module

class TFQMR (*matvec, **kwargs*)

Bases: `pykrylov.generic.generic.KrylovMethod`

A pure Python implementation of the transpose-free quasi-minimum residual (TFQMR) algorithm. TFQMR may be used to solve unsymmetric systems of linear equations, i.e., systems of the form

$$A \, x = b$$

where the matrix *A* may be unsymmetric.

TFQMR requires 2 matrix-vector products with *A*, 4 dot products and 10 daxpys per iteration. It does not require products with the transpose of *A*.

If a preconditioner is supplied, TFQMR needs to solve 2 preconditioning systems per iteration.

solve (*rhs, **kwargs*)

Solve a linear system with *rhs* as right-hand side by the TFQMR method. The vector *rhs* should be a Numpy array. An optional argument *guess* may be supplied, with an initial guess as a Numpy array. By default, the initial guess is the vector of zeros.

6.2 Example

Here is an example using TFQMR on a linear system. The coefficient matrix is read from file in Matrix Market format:

```python
import numpy as np
from pykrylov.tfqmr import TFQMR as KSolver
from pysparse import spmatrix
from pysparse.pysparseMatrix import PysparseMatrix as sp

A = sp(matrix=spmatrix.ll_mat_from_mtx('jpwh_991.mtx'))
n = A.shape[0]
e = np.ones(n)
rhs = A*e

ks = KSolver( lambda v: A*v,
              matvec_max=2*n,
              verbose=False,
              outputStream=sys.stderr,
```

```
                reltol = 1.0e-5 )
ks.solve(rhs, guess = 1+np.arange(n, dtype=np.float))

print 'Number of matvecs: ', ks.nMatvec
print 'Initial/final res: %8.2e/%8.2e' % (ks.residNorm0, ks.residNorm)
print 'Direct error: %8.2e' % (np.linalg.norm(ks.bestSolution-e)/sqrt(n))
```

Running this script produces the following output:

```
Number of matvecs:  70
Initial/final res: 8.64e+03/6.23e-04
Direct error: 2.77e-05
```

BENCHMARKING SOLVERS

You may have noticed that in the examples of sections *The Conjugate Gradient Method*, *The Conjugate Gradient Squared Method* and *The Bi-Conjugate Gradient Stabilized Method*, the only line that differs in the example scripts has the form

```python
from pykrylov.tfqmr import TFQMR as KSolver
```

The rest of the code fragment is exactly the same across all examples. This similarity could be used, for instance, to benchmark solvers on a set of test problems.

Consider the script:

```python
import numpy as np
from pykrylov.cgs import CGS
from pykrylov.tfqmr import TFQMR
from pykrylov.bicgstab import BiCGSTAB
from pysparse import spmatrix
from pysparse.pysparseMatrix import PysparseMatrix as sp

from math import sqrt

if __name__ == '__main__':

    hdr = '%10s  %6s  %8s  %8s' % ('Name', 'Matvec', 'Resid', 'Error')
    fmt = '%10s  %6d  %8.2e  %8.2e'
    print hdr

    A = sp(matrix=spmatrix.ll_mat_from_mtx('jpwh_991.mtx'))

    n = A.shape[0]
    e = np.ones(n)
    rhs = A*e

    # Loop through solvers using default stopping tolerance
    for KSolver in [CGS, TFQMR, BiCGSTAB]:
        ks = KSolver(lambda v: A*v, matvec_max=2*n)
        ks.solve(rhs, guess = 1+np.arange(n, dtype=np.float))

        err = np.linalg.norm(ks.bestSolution-e)/sqrt(n)
        print fmt % (ks.acronym, ks.nMatvec, ks.residNorm, err)
```

Executing the script above produces the formatted output:

```
     Name  Matvec      Resid      Error
      CGS      64  4.72e-03  1.47e-04
    TFQMR      70  6.23e-04  2.77e-05
Bi-CGSTAB      62  6.34e-03  4.05e-04
```

7.1 Example with Preconditioning

A preconditioner can be supplied to any Krylov solver via the *precon* keyword argument upon instantiation.

For example, we could supply a simple (and naive) diagonal preconditioner by modifying the benchmarking script as:

```
import numpy as np
from pykrylov.cgs import CGS
from pykrylov.tfqmr import TFQMR
from pykrylov.bicgstab import BiCGSTAB
from pysparse import spmatrix
from pysparse.pysparseMatrix import PysparseMatrix as sp

from math import sqrt

if __name__ == '__main__':

    hdr = '%10s  %6s  %8s  %8s' % ('Name', 'Matvec', 'Resid', 'Error')
    fmt = '%10s  %6d  %8.2e  %8.2e'
    print hdr

    A = sp(matrix=spmatrix.ll_mat_from_mtx('jpwh_991.mtx'))

    # Extract diagonal of A and make it sufficiently positive
    diagA = np.maximum(np.abs(A.takeDiagonal()), 1.0)

    n = A.shape[0]
    e = np.ones(n)
    rhs = A*e

    # Loop through solvers using default stopping tolerance
    for KSolver in [CGS, TFQMR, BiCGSTAB]:
        ks = KSolver(lambda v: A*v,
                     matvec_max=2*n
                     precon = lambda u: u/diagA)
        ks.solve(rhs, guess = 1+np.arange(n, dtype=np.float))

        err = np.linalg.norm(ks.bestSolution-e)/sqrt(n)
        print fmt % (ks.acronym, ks.nMatvec, ks.residNorm, err)
```

This time, the output is a bit better than before:

```
     Name  Matvec      Resid      Error
      CGS      56  3.77e-03  7.42e-05
    TFQMR      59  1.07e-03  6.03e-05
Bi-CGSTAB      54  4.62e-03  3.02e-04
```

Much in the same way, a modification of the script above could be used to loop through preconditioners with a given solver.

Todo

Should preconditioners be objects instead of just functions?

TODO LIST

Todo

Should preconditioners be objects instead of just functions?

(The original entry is located in bmark.rst, line 115 and can be found *here*.)

INDICES AND TABLES

- *Index*

- *Module Index*

- *Search Page*

BIBLIOGRAPHY

[Kelley] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*, number 16 in *Frontiers in Applied Mathematics*, SIAM, Philadelphia, 1995.

MODULE INDEX

B

C

G

P

T

INDEX