PySparse - A sparse linear algebra extension for Python

Roman Geus

June 7, 2002

This document is a portion of a draft of my PhD thesis. Unfortunately the document contains unresolved references to other parts of my thesis, which are printed as double question marks. Nevertheless the document could be useful as a reference for the PySparse package.

0.1 The PySparse package

The PySparse package extends the Python interpreter by a set of sparse matrix types. PySparse also includes modules that implement

- 6terative methods for solving linear systems of equations,
- a set of standard preconditioners,
- an interface to a direct solver for sparse linear systems of equations,
- and the JDSYM eigensolver.

spmatrix module functions

ll_mat(n, m, sizeHint=1000) Creates a *ll_mat* object, that represents a general, all zero $m \times n$ matrix. The optional *sizeHint* parameter specifies the number of non-zero entries for which space is allocated initially.

If the total number of non-zero elements of the final matrix is known (approximately), this number can be passed as *sizeHint*. This will avoid costly memory reallocations.

ll_mat_sym(n, sizeHint=1000) Creates a *ll_mat* object, that represents a *symmetric*, all zero $n \times n$ matrix. The optional *sizeHint* parameter specifies, how much space is initially allocated for the matrix.

ll_mat_from_mtx(fileName) Creates a *ll_mat* object from a file named *fileName*, which must be in MatrixMarket Coordinate format as described at `http://math.nist.gov/MatrixMarket/formats.html`. Depending on the file content, either a symmetric or a general sparse matrix is generated.

matrixmultiply(A, B) computes the matrix-matrix multiplication

$$C := A\,B$$

and returns the result C as a new *ll_mat* object representing a general sparse matrix. The parameters A and B are expected t212.0935 -50.[(objetes)-252(of)4269ypeed

```
>>> import spmatrix
>>> A = spmatrix.ll_mat(5, 5)
>>> for i in range(5):
...     A[i,i] = i+1
>>> A[2,1] = A[0,0]
>>> print A
ll_mat(general, [5,5], [(0,0): 1, (1,1): 2, (2,1): 1,
(2,2): 3, (3,3): 4, (4,4): 5])
```

A.export_mtx(fileName, precision=6) Exports the matrix A to file named *fileName*

A.update_add_mask_sym(B, ind, mask) This method is provided for efficiently

Example: 2D-Poisson matrix This section illustrates the use of the *spmatrix* module

Function	$n = 100$	$n = 300$	$n = 500$	$n = 1000$

The performance difference between Python's `poisson2d_sym` and `poisson2d_sym_blk` indicates, that a lot of time is spent parsing indices.

0.1.2 The precon module

The *precon* module provides preconditioners, which can be used e.g. for the iterative

```
class diag_prec:
    def __init__(self, A):
        self.shape = A.shape
        n = self.shape[0]
        self.dinv = Numeric.zeros(n, 'd')
```

Example: Solving the poisson system Let's solve the Poisson system

$$L\,x = 1, \tag{1}$$

using the PCG method. L is the 2D Poisson matrix, introduced in Section 0.1.1 and 1 is a vector with all entries equal to one.
The Python solution for this task looks as follows:

```
import Numeric, spmatrix, precon, itsolvers
n = 300
L = poisson2d_sym_blk(n)
b = Numeric.ones(n*n, 'd')
x = Numeric.zeros(n*n, 'd')
info, iter, relres = itsolvers.pcg(L.to_sss(), b, x, 1e-12, 2000)
```

The code makes use of the Python function `poisson2d_sym_blk`, which was defined in Section 0.1.1.
Incorporating e.g. a SSOR preconditioner is straight-forward:

```
import Numeric, spmatrix, precon, itsolvers
n = 300
L = poisson2d_sym_blk(n)
b = Numeric.ones(n*n, 'd')
x = Numeric.zeros(n*n, 'd')
S = L.to_sss()
Kssor = precon.ssor(S)
info, iter, relres = itsolvers.pcg(S, b, x, 1e-12, 2000, Kssor)
```

The Matlab solution (without preconditioner) may look as follows:

```
n = 300;
L = poisson2d_kron(n);
[x,flag,relres,iter] = pcg(L, ones(n*n,1), 1e-12, 2000, ...
                           [], [], zeros(n*n,1));
```

Perf963ar2eeche with a0h

Function	Size	t_{constr}	t_{solv}	t_{tot}
Python	$n = 100$	0.03	1.12	1.15
	$n = 300$	0.21	49.65	49.86
	n			

linsolver is a function that implements an iterative method for solving linear systems of equations. The function *linsolver* is required to conform to the standards mentioned in Section 0.1.3.

The remaining (optional) arguments are specified using keyword arguments:

jmax is an integer that specifies the maximum dimension of the search subspace. (default: 25)

jmin is an integer that specifies dimension of the search subspace after a restart. (default: 10)

blksize is an integer that specifies the block size used in the JDSYM algorithm. (default: 1)

blkwise is an integer that affects the convergence criterion if *blksize* > 1 (cf. Section **??**). (default: 0)

V0 is NumPy array of rank one or two. It specifies the initial search subspace. (default: a randomly generated initial search subspace)

optype is an integer specifying the operator type used in the correction equation. If *optype* $= 1$, the non-symmetric version is used. If *optype* $= 2$, the symmetric version is used. See Section **??** for more information. (default: 2)

linitmax is an integer specifying the maximum number steps taken in the inner iteration (iterative linear solver). (default: 200)

eps_tr is a float value setting the tracking parameter ε_{tr} described in Section **??**. (default: 10^{-3})

tolde250(Sectio)1(nf 8955 9.914.984 Td[2(in)-84 0 Td[(10)]TJ/F10 0 f -229ih)-397(that)-275(id[(1ugencs4 0 T(the

Example: Maxwell problem The following code illustrates the use of the *jdsym* module. Tw matrices A and M are read from files. A Jacobi preconditioner from $A - \tau M$ is built. Then the JDSYM eigensolver is called, calculating 5 eigenvalues near 25.0 and the associated eigenvalues to an accuracy of 10^{-10}. We set *strategy* = 1 to avoid convergence to the high-dimensional null space of (A, M).

```
import spmatrix, itsolvers, jdsym, precon

A = spmatrix.ll_mat_from_mtx('edge6x3x5_A.mtx')
M = spmatrix.ll_mat_from_mtx('edge6x3x5_B.mtx')
tau = 25.0

Atau = A.copy()
Atau.shift(-tau, M)
K = precon.jacobi(Atau)

A = A.to_sss(); M = M.to_sss()
k_conv, lmbd, Q, it  = \
```