Pysparse Documentation

Release 1.0.2

Roman Geus, Daniel Wheeler and Dominique Orban

June 14, 2011

CONTENTS

Release 1.0

Date June 14, 2011

CHAPTER

ONE

INTRODUCTION TO PYSPARSE

PySparse extends the Python interpreter by a set of sparse matrix types holding double precision values. PySparse also includes modules that implement

- Iterative Krylov methods for solving linear systems of equations,
- Diagonal (Jacobi) and SSOR preconditioners,
- Interfaces to direct solvers for sparse linear systems of equations (SuperLU and UMFPACK),
- A Jacobi-Davidson eigenvalue solver for the symmetric, generalised matrix eigenvalue problem (JDSYM),
- Low-level C classes to represent and manipulate sparse matrices,
- High-level Python classes with operator overloading to perform usual operations on matrices.

Most of the above modules are implemented as C extension modules for maximum performance.

PySparse uses NumPy for handling dense vectors and matrices and uses SuperLU and UMFPACK for factorizing general sparse matrices.

1.1 Module Overview

1.1.1 `spmatrix`

The `spmatrix` module is the foundation of the Pysparse package. It extends the Python interpreter by three new types named `ll_mat`, `csr_mat` and `sss_mat`. These types represent sparse matrices in the LL-, the CSR- and SSS-formats respectively (see *Sparse Matrix Formats*). For all three formats, double precision values (C type double) are used to represent the nonzero entries. The common way to use the spmatrix module is to first build a matrix in the LL-format. The LL-matrix is manipulated until it has its final shape and content. Afterwards it may be converted to either the CSR- or SSS-format, which needs less memory and allows for fast matrix-vector multiplications. A ll_mat object can be created from scratch, by reading data from a file (in MatrixMarket format) or as a result of matrix operation (as e.g. a matrix-matrix multiplication). The ll_mat object supports manipulating (reading, writing, add-updating) single entries or sub-matrices. On the other hand, `csr_mat` and `sss_mat` matrices are not constructed directly, instead they are created by converting `ll_mat` objects. Once created, `csr_mat` and `sss_mat` objects cannot be manipulated. Their purpose is to support efficient matrix-vector multiplications.

1.1.2 `itsolvers`

The `itsolvers` module provides a set of iterative methods for solving linear systems of equations. The iterative methods are callable like ordinary Python functions. All these functions expect the same parameter list, and all function

return values also follow a common standard. Any user-defined iterative solvers should also follow these conventions, since other PySparse modules rely on them (e.g. the `jdsym` module).

Currently the `itsolvers` module contains the following iterative methods: PCG, MINRES, QMRS, BICGSTAB and CGS.

1.1.3 precon

The `precon` module provides preconditioners, which can be used e.g. for the iterative methods implemented in the `itsolvers` module or the JDSYM eigensolver (in the `jdsym` module). In the PySparse framework, any Python object that has the following properties can be used as a preconditioner:

- a `shape` attribute, which returns a 2-tuple describing the dimension of the preconditioner,
- a `precon` method, that accepts two vectors x and y, and applies the preconditioner to x and stores the result in y. Both x and y are double precision, rank-1 NumPy arrays of appropriate size.

The `precon` module currently implements m-step Jacobi and m-step SSOR preconditioners.

1.1.4 superlu

The `superlu` module interfaces the SuperLU library to make it usable by Python code. SuperLU is a software package written in C for the direct solution of a general linear system of equations. SuperLU computes LU-factorizations of general non-symmetric, sparse matrices with partial pivoting. It is also applicable to rectangular systems of equations.

1.1.5 umfpack

The `umfpack` module interfaces the UMFPACK factorization package. UMFPACK computes the LU factorization of a general matrix with partial pivoting. It is also applicable to rectangular systems.

The main difference between the `superlu` and `umfpack` modules resides in the way the factorization is performed internally. SuperLU works with the concept of *supernodes* leading naturally to parallelism in the factorization. If available, a custom-built SuperLU library for multi-core processors can be supplied to Pysparse in place of the default library. Both factorization packages rely intensively on the BLAS to operate on dense sub-matrices. Provided the BLAS library supplied to Pysparse was compiled with multi-threading, some level of parallelism will also be available in UMFPACK. However, a rough empirical observation is that UMFPACK is often faster than SuperLU on mono-processor machines.

1.1.6 jdsym

The `jdsym` module provides an implementation of the JDSYM algorithm, that is conveniently callable from Python. The JDSYM algorithm computes solutions of large sparse symmetric (genralised or standard) eigenvalue problems. JDSYM is an implementation of the Jacobi-Davidson method, optimized for symmetric matrices.

1.2 Prerequisites

- NumPy
- Optionally, a custom-built UMFPACK and/or SuperLU

1.3 Installing Pysparse

```
python setup.py install
```

1.4 Testing Pysparse

From the `Test` directory, `testSuperLU` runs a series of tests to exercise the various options of the SuperLU direct solver:

```
$ python testSuperlu.py
          Test    RelErr       Tol    nnz(A)  nnz(L+U)    Fact   Solve
    ------------------------------------------------------------------
    poi1d-dflt  1.78e-12  2.25e-07     99999    199998    0.06    0.00
.   poi1d-size  1.78e-12  2.25e-07     99999    199998    0.05    0.00
.   poi1d-relx  1.78e-12  2.25e-07     99999    200758    0.06    0.00
.   poi1d-trsh  1.78e-12  2.25e-07     99999    199998    0.06    0.00
.   poi1d-prm0  1.44e-12  2.25e-07     99999    199998    0.04    0.00
.   poi1d-prm1  1.78e-12  2.25e-07     99999    199998    0.07    0.00
.   poi1d-prm2  1.78e-12  2.25e-07     99999    199998    0.06    0.00
.   poi1d-prm3  1.44e-12  2.25e-07     99999    200000    0.06    0.00
.   poi2d-dftl  2.55e-16  3.60e-12    119600   1952434    0.47    0.02
.   poi2d-size  3.39e-16  3.60e-12    119600   1952434    0.42    0.02
.   poi2d-relx  2.60e-16  3.60e-12    119600   2000252    0.48    0.02
.   poi2d-trsh  2.55e-16  3.60e-12    119600   1952434    0.48    0.02
.   poi2d-prm0  2.69e-15  3.60e-12    119600  16000398    5.48    0.11
.   poi2d-prm1  7.00e-16  3.60e-12    119600   3506336    0.89    0.03
.   poi2d-prm2  2.55e-16  3.60e-12    119600   1952434    0.47    0.02
.   poi2d-prm3  2.24e-15  3.60e-12    119600   3472176    0.82    0.03
.   spdgs-trsh  4.44e-16  2.22e-14     29998     39998    0.01    0.00
.   spdgs-prm0  4.44e-16  2.22e-14     29998     40000    0.01    0.00
.   spdgs-prm1  4.44e-16  2.22e-14     29998     40002    0.01    0.00
.   spdgs-prm3  4.44e-16  2.22e-14     29998     40002    0.01    0.00
.
----------------------------------------------------------------------
Ran 20 tests in 12.675s

OK
```

There is a corresponding test script for UMFPACK, `testUmfpack`:

```
$python testUmfpack.py
      RelErr       Tol    nnz(A)    nnz(L)    nnz(U)    Fact   Solve
    ------------------------------------------------------------------
    1.44e-12  2.25e-07    149998     99999     99999    0.13    0.01
.   1.44e-12  2.25e-07    149998     99999     99999    0.12    0.01
.   1.44e-12  2.25e-07    149998     99999     99999    0.12    0.01
.   1.44e-12  2.25e-07    149998     99999     99999    0.12    0.01
.   1.44e-12  2.25e-07    149998     99999     99999    0.12    0.01
.   1.44e-12  2.25e-07    149998     99999     99999    0.14    0.01
.   1.55e-17  3.60e-12    199200   1081911   1081911    0.54    0.03
.   1.55e-17  3.60e-12    199200   1081911   1081911    0.53    0.03
.   2.50e-17  3.60e-12    199200   1081911   1081911    0.53    0.03
.   2.50e-17  3.60e-12    199200   1081911   1081911    0.53    0.03
.   1.55e-17  3.60e-12    199200   1081911   1081911    0.53    0.03
.   1.64e-17  3.60e-12    199200   1489438   2166768    1.00    0.04
.   4.44e-16  2.22e-14     29998     19999     19999    0.03    0.00
```

```
.   4.44e-16  2.22e-14     29998     19999     19999    0.02    0.00
.   4.44e-16  2.22e-14     29998     19999     19999    0.02    0.00
.   4.44e-16  2.22e-14     29998     19999     19999    0.02    0.00
.   4.44e-16  2.22e-14     29998     19999     19999    0.03    0.00
.   4.44e-16  2.22e-14     29998     19999     19999    0.02    0.00
.
----------------------------------------------------------------------
Ran 18 tests in 8.486s
```

1.5 Generating the Documentation

To re-generate the documentation, Sphinx version 0.5 or higher must be installed. The jsMath package must also be installed. Edit `$PYSPARSE/Doc/pysparse/source/conf.py` to specify the location of `jsMath`.

To re-generate the html documentation,

```
cd $PYSPARSE/Doc/pysparse
make html
```

where `Doc` is a subdirectory of the top Pysparse directory. You can then point your browser to the file `build/index.html`.

Similarly, to re-generate the pdf documentation,

```
cd $PYSPARSE/Doc/pysparse
make latex
cd build/latex
make all-pdf
```

This creates `Pysparse.pdf` in the current directory. Obviously, you need to have a working LaTeX distribution installed.

CHAPTER

TWO

SPARSE MATRIX FORMATS

This section describes the sparse matrix storage schemes available in Pysparse. It also covers sparse matrix creation, population and conversion.

- Linked-list format (LL): a convenient format for creating and populating a sparse matrix, whether symmetric or general.
- Compressed sparse row format (CSR): a format designed to speed up matrix-vector products, but not well suited to matrix population and manipulation.
- Sparse Skyline format (SSS): a format for symmetric matrices designed to speed up matrix-vector products, but not well suited to matrix population and manipulation.

2.1 Linked-List Format

The linked-list format allows insertion and lookup of nonzero elements in moderate time and without having to move too much data around. Internally, the nonzero entries of a matrix are stored row by row in a linked list. Within a given row, column indices are sorted in ascending order.

In Pysparse, matrices in linked-list format are created by using the `ll_mat` class.

This format resembles a sorted version of the coordinate format but with a data structure that lends itself to fast insertion, removal and lookup.

Typically, a new matrix should be created as an `ll_mat` and populated. If necessary, it can then be converted to compressed sparse row or sparse skyline format using the `to_csr()` and `to_sss()` methods.

The data structure for a matrix in linked-list format has the following components:

`val` The double precision array `val` of length `nalloc` contains the non-zero entries of matrix.

`col` The integer array `col` of length `nalloc` contains the column indices of the non-zero entries stored in `val`.

`link` the integer array `link` of length `nalloc` stores the pointer (index) to the next non-zero entry of the same row. A value of -1 indicates that there is no next entry.

`root` The integer array `root` of length `n` contains the pointers to the first entry of each row. The other entries of the same row can be located by following the `link` array.

`free` The integer `free` points to the first entry of the *free list*, i.e. a linked list of unoccupied spots in the `val` and `col` arrays. This list is populated when non-zero entries are removed from the matrix.

Here `n` is the number of rows of the matrix and `nalloc` is number of allocated elements in the arrays `val`, `col` and `link`. Note that the number of nonzero entries stored is less than or equal to `nalloc`, but the `val`, `col` and `link` arrays can be enlarged dynamically if necessary.

2.2 Compressed Sparse Row Format

In CSR format, a sparse matrix is represented via three arrays:

va The double precision array `va` of length `nnz` contains the non-zero entries of the matrix, stored row by row.

ja The integer array `ja` of length `nnz` contains the column indices of the non-zero entries stored in `va`.

ia The integer array `ia` of length `n + 1` contains the pointers (indices) to the beginning of each row in the arrays `va` and `ja`. The last element of `ia` always has the value `nnz + 1`.

Here `n` is the number of rows of the matrix and `nnz` is its number of nonzero entries.

This format is particularly interesting for computing matrix-vector products. Even though the order of the entries is not prescribed in this format, we sort the entries of each row by ascending column indices. This enables us to use more efficient algorithms for certain operations.

2.3 Sparse Skyline Format

The SSS format is closely related to the CSR format. It is often used for sparse *symmetric* matrices. The diagonal is stored in a separate (full) vector and the strict lower triangle is stored in CSR format:

va The double precision array `va` of length `nnz` contains the non-zero entries of the strict lower triangle, stored row by row.

ja The integer array `ja` of length `nnz` contains the column indices of the non-zero entries stored in `va`.

ia The integer array `ia` of length `n + 1` contains the pointers (indices) to the beginning of each row in the arrays `va` and `ja`. The last element of `ia` always has the value `nnz + 1`.

da The double precision array `da` of length `n` stores all diagonal entries of the matrix.

Here `n` is the order of the matrix and `nnz` is the number of nonzero entries in the strict lower triangle.

We sort the entries of each row by ascending column indices, like we do with the CSR format. The SSS format has the advantage over the CSR format, that it requires roughly half of the storage space and that the matrix-vector multiplication can be implemented more efficiently

CHAPTER

THREE

LOW-LEVEL SPARSE MATRIX TYPES

3.1 The `spmatrix` Module

The `spmatrix` module is the foundation of the PySparse package. It extends the Python interpreter by three new types named `ll_mat`, `csr_mat` and `sss_mat`. These types represent sparse matrices in the LL-, the CSR- and SSS-formats respectively (see *Sparse Matrix Formats*). For all three formats, double precision values (C type `double`) are used to represent the non-zero entries.

The common way to use the `spmatrix` module is to first build a matrix in the LL-format. The LL-matrix is manipulated until it has its final shape and content. Afterwards it may be converted to either the CSR- or SSS-format, which needs less memory and allows for fast matrix-vector multiplications.

A `ll_mat` object can be created from scratch, by reading data from a file (in MatrixMarket format) or as a result of matrix operation (as e.g.a matrix-matrix multiplication). The `ll_mat` object supports manipulating (reading, writing, add-updating) single entries or sub-matrices.

`csr_mat` and `sss_mat` are not constructed directly, instead they are created by converting `ll_mat` objects. Once created, `csr_mat` and `sss_mat` objects cannot be manipulated. Their purpose is to support efficient matrix-vector multiplications.

3.1.1 `spmatrix` module functions

`spmatrix.`**`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.

`spmatrix.`**`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.

`spmatrix.`**`ll_mat_from_mtx`**(*fileName*)
: Creates a `ll_mat` object from a file named `fileName`, which must be in MatrixMarket Coordinate format. Depending on the file content, either a symmetric or a general sparse matrix is generated.

`spmatrix.`**`matrixmultiply`**(*A*, *B*)
: Computes the matrix-matrix multiplication $\mathbf{C} := \mathbf{AB}$ and returns the result $\mathbf{C}$ as a new `ll_mat` object representing a general sparse matrix. The parameters $\mathbf{A}$ and $\mathbf{B}$ are expected to be objects of type `ll_mat`.

spmatrix.**dot**(*A*, *B*)
Computes the *dot-product* $\mathbf{C} := \mathbf{A}^T\mathbf{B}$ and returns the result $\mathbf{C}$ as a new ll_mat object representing a general sparse matrix. The parameters $\mathbf{A}$ and $\mathbf{B}$ are expected to be objects of type ll_mat.

3.1.2 ll_mat objects

ll_mat objects represent matrices stored in the LL format, which are described in *Sparse Matrix Formats*. ll_mat objects come in two flavours: general matrices and symmetric matrices. For symmetric matrices only the non-zero entries in the lower triangle are stored. Write operations to the strictly upper triangle are prohibited for the symmetric format. The issym attribute of an ll_mat object can be queried to find out whether or not the symmetric storage format is used.

The entries of a matrix can be accessed conveniently using two-dimensional array indices. In the Python language, subscripts can be of any type (as it is customary for dictionaries). A two-dimensional index can be regarded as a 2-tuple (the brackets do not have to be written, so A[1,2] is the same as A[(1,2)]). If both tuple elements are integers, then a single matrix element is referenced. If at least one of the tuple elements is a slice (which is also a Python object), then a submatrix is referenced.

Subscripts have to be decoded at runtime. This task includes type checks, extraction of indices from the 2-tuple, parsing of slice objects and index bound checks. Following Python conventions, indices start with 0 and wrap around (so -1 is equivalent to the last index).

The following code creates an empty 5×5 matrix A, sets all diagonal elements to their respective row/column index and then copies the value of A[0,0] to A[2,1]:

```
>>> from pysparse.sparse 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])
```

The Python slice notation can be used to conveniently access sub-matrices.

```
>>> print A[:2,:]     # the first two rows
ll_mat(general, [2,5], [(0,0): 1, (1,1): 2])
>>> print A[:,2:5]    # columns 2 to 4
ll_mat(general, [5,3], [(2,0): 3, (3,1): 4, (4,2): 5])
>>> print A[1:3,2:5]  # submatrix from row 1 col 2 to row 2 col 4
ll_mat(general, [2,3], [(1,0): 3])
```

The slice operator always returns a new ll_mat object, containing a **copy** of the selected submatrix.

Write operations to slices are also possible:

```
>>> B = ll_mat(2, 2)            # create 2-by-2
>>> B[0,0] = -1; B[1,1] = -1    # diagonal matrix
>>> A[:2,:2] = B                # assign it to upper
>>>                             # diagonal block of A
>>> print A
ll_mat(general, [5,5], [(0,0): -1, (1,1): -1, (2,1): 1,
(2,2): 3, (3,3): 4, (4,4): 5])
```

Fancy Indexing

There is flexibility in the way submatrices of `ll_mat` objects can be accessed. In particular, rows and columns can be permuted arbitrarily and submatrices need not be composed of consecutive rows or indices. Let's look at an example. Below, the `poisson1d()` function assembles a Poisson matrix. We come back to Poisson matrices later in this section.

```
>>> from pysparse.sparse import poisson
>>> n = 5
>>> A = poisson.poisson1d(n)
>>> print A   # Original matrix
ll_mat(general, [5,5]):
 2.000000 -1.000000  --------  --------  --------
-1.000000  2.000000 -1.000000  --------  --------
 -------- -1.000000  2.000000 -1.000000  --------
 --------  -------- -1.000000  2.000000 -1.000000
 --------  --------  -------- -1.000000  2.000000

>>> print A[n-1:1:-1,1:n-1]  # Rows 2 through n-1 in reverse order,
>>>                         # second through one before last col
ll_mat(general, [3,3]):
 --------  -------- -1.000000
 -------- -1.000000  2.000000
-1.000000  2.000000 -1.000000

>>> print A[::-1,:]  # Reverse row order
ll_mat(general, [5,5]):
 --------  --------  -------- -1.000000  2.000000
 --------  -------- -1.000000  2.000000 -1.000000
 -------- -1.000000  2.000000 -1.000000  --------
-1.000000  2.000000 -1.000000  --------  --------
 2.000000 -1.000000  --------  --------  --------

>>> print A[:,::-1]  # Reverse col order (same as above b/c A is symmetric)
ll_mat(general, [5,5]):
 --------  --------  -------- -1.000000  2.000000
 --------  -------- -1.000000  2.000000 -1.000000
 -------- -1.000000  2.000000 -1.000000  --------
-1.000000  2.000000 -1.000000  --------  --------
 2.000000 -1.000000  --------  --------  --------

>>> print A[::-1,::-1]  # Reverse row and col order (same as original matrix)
ll_mat(general, [5,5]):
 2.000000 -1.000000  --------  --------  --------
-1.000000  2.000000 -1.000000  --------  --------
 -------- -1.000000  2.000000 -1.000000  --------
 --------  -------- -1.000000  2.000000 -1.000000
 --------  --------  -------- -1.000000  2.000000

>>> print A[1:3,3:]   # Rows 1 and 2, cols 3 and up
ll_mat(general, [2,2]):
 --------  --------
-1.000000  --------

>>> print A[::2,::2]  # Every other row and col
ll_mat(general, [3,3]):
 2.000000  --------  --------
 --------  2.000000  --------
```

```
 --------  --------  2.000000
```

Keep in mind that as always with Python slices, the final index is never included. Note also that slicing always returns a general matrix. Even though it might be symmetric, both triangles are stored. Finally, slicing should be applied to general matrices. If applied to symmetric matrices, only a partial result is returned.

Fancy indexing can also be done with Python lists:

```
>>> print A[ [1,4,2,0], ::2]
ll_mat(general, [4,3]):
-1.000000 -1.000000  --------
 --------  --------  2.000000
 --------  2.000000  --------
 2.000000  --------  --------
>>> p = [1,4,2,0]
>>> q = [0,2,4]
>>> print A[p,q]
ll_mat(general, [4,3]):
-1.000000 -1.000000  --------
 --------  --------  2.000000
 --------  2.000000  --------
 2.000000  --------  --------
```

or with integer Numpy arrays:

```
>>> idx0 = numpy.array([1,4,2,0], dtype=numpy.int)
>>> idx1 = numpy.array([0,2,4], dtype=numpy.int)
>>> print A[idx0,idx1]
ll_mat(general, [4,3]):
-1.000000 -1.000000  --------
 --------  --------  2.000000
 --------  2.000000  --------
 2.000000  --------  --------
```

Finally, fancy indexing can be used to assign the same numerical value to a submatrix:

```
>>> A[:3,:3] = 7   # Assign value 7.0 to a principal submatrix
>>> print A
ll_mat(general, [5,5]):
 7.000000  7.000000  7.000000  --------  --------
 7.000000  7.000000  7.000000  --------  --------
 7.000000  7.000000  7.000000 -1.000000  --------
 --------  --------  -1.000000  2.000000 -1.000000
 --------  --------  --------  -1.000000  2.000000
```

Notice however that although the slice `[0:3]` appears to amount to the list `[0,1,2]`, the assignments `A[:3,:3]=7` and `A.put([7,7,7], [0,1,2], [0,1,2])` produce **very different** results.

> **Warning:** For large-scale matrices, fancy indexing is most efficient when both index sets have the same type: two Python slices or two Python lists. When the index sets have different types, index arrays are built internally and this results in a performance hit.

`ll_mat` Object Attributes and Methods

class spmatrix.**ll_mat**

A general sparse matrix class in linked-list format which also allows the representation of symmetric matrices. Only the lower triangle of a symmetric matrix is kept in memory for efficiency.

shape
Returns a 2-tuple containing the shape of the matrix $\mathbf{A}$, i.e. the number of rows and columns.

nnz
Returns the number of non-zero entries stored in matrix $\mathbf{A}$. If $\mathbf{A}$ is stored in symmetric format, only the number of non-zero entries in the lower triangle (including the diagonal) are returned.

issym
Returns true (a non-zero integer) if matrix $\mathbf{A}$ is stored in the symmetric LL format, i.e. only the non-zero entries in the lower triangle are stored. Returns false (zero) if matrix $\mathbf{A}$ is stored in the general LL format.

matvec (*x*, *y*)
Computes the sparse matrix-vector product $\mathbf{y} := \mathbf{A}\mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ are double precision, rank-1 NumPy arrays of appropriate size.

matvec_transp (*x*, *y*)
Computes the transposed sparse matrix-vector product $\mathbf{y} := \mathbf{A}^T\mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ are double precision, rank-1 NumPy arrays of appropriate size. For `sss_mat` objects `matvec_transp` is equivalent to `matvec`.

to_csr ()
This method converts a sparse matrix in linked list format to compressed sparse row format. Returns a newly allocated `csr_mat` object, which results from converting matrix $\mathbf{A}$.

to_sss ()
This method converts a sparse matrix in linked list format to sparse skyline format. Returns a newly allocated `sss_mat` object, which results from converting matrix $\mathbf{A}$. This function works for `ll_mat` objects in both the symmetric and the general format. If $\mathbf{A}$ is stored in general format, only the entries in the lower triangle are used for the conversion. No check whether $\mathbf{A}$ is symmetric is performed.

export_mtx (*fileName*, *precision=6*)
Exports the matrix $\mathbf{A}$ to file named `fileName`. The matrix is stored in MatrixMarket Coordinate format. Depending on the properties of the `ll_mat` object $\mathbf{A}$ the generated file either uses the symmetric or a general MatrixMarket Coordinate format. The optional parameter `precision` specifies the number of decimal digits that are used to express the non-zero entries in the output file.

shift (*sigma*, *M*)
Performs the daxpy operation $\mathbf{A} \leftarrow \mathbf{A} + \sigma\mathbf{M}$. The parameter σ is expected to be a Python Float object. The parameter $\mathbf{M}$ is expected to an object of type `ll_mat` of compatible shape.

copy ()
Returns a new `ll_mat` object that is a (deep) copy of the `ll_mat` object $\mathbf{A}$. So:

```
>>> B = A.copy()
```

is equivalent to:

```
>>> B = A[:,:]
```

On the other hand:

```
>>> B = A.copy()
```

is *not* the same as:

```
>>> B = A
```

The latter version only returns a reference to the same object and assigns it to `B`. Subsequent changes to `A` will therefore also be visible in `B`.

update_add_mask (*B*, *ind0*, *ind1*, *mask0*, *mask1*)
This method is provided for efficiently assembling global finite element matrices. The method adds the

matrix $\mathbf{B}$ to entries of matrix $\mathbf{A}$. The indices of the entries to be updated are specified by the integer arrays `ind0` and `ind1`. The individual updates are enabled or disabled using the `mask0` and `mask1` arrays.

The operation is equivalent to the following Python code:

```
for i in range(len(ind0)):
    for j in range(len(ind1)):
        if mask0[i] and mask1[j]:
            A[ind0[i],ind1[j]] += B[i,j]
```

All five parameters are NumPy arrays. $\mathbf{B}$ is a rank-2 array. The four remaining parameters are rank-1 arrays. Their length corresponds to either the number of rows or the number of columns of $\mathbf{B}$.

This method is not supported for `ll_mat` objects of symmetric type, since it would generally result in an non-symmetric matrix. `update_add_mask_sym` must be used in that case. Attempting to call this method using a `ll_mat` object of symmetric type will raise an exception.

update_add_mask_sym(*B*, *ind*, *mask*)

This method is provided for efficiently assembling symmetric global finite element matrices. The method adds the matrix $\mathbf{B}$ to entries of matrix $\mathbf{A}$. The indices of the entries to be updated are specified by the integer array `ind`. The individual updates are enabled or disabled using the `mask` array.

The operation is equivalent to the following Python code:

```
for i in range(len(ind)):
    for j in range(len(ind)):
        if mask[i]:
            A[ind[i],ind[j]] += B[i,j]
```

The three parameters are all NumPy arrays. $\mathbf{B}$ is a rank-2 array representing a square matrix. The two remaining parameters are rank-1 arrays. Their length corresponds to the order of matrix $\mathbf{B}$.

update_add_at(*val*, *irow*, *jcol*)

Add in place the elements of the vector `val` at the indices given by the two arrays `irow` and `jcol`. The operation is equivalent to:

```
for i in range(len(val)):
    A[irow[i],jcol[i]] += val[i]
```

generalize()

Convert `ll_mat` object to non-symmetric form in place.

compress()

Frees memory by reclaiming unused space in the internal data structure. Returns the number of elements freed.

norm(*p*)

Returns the `p`-norm of a matrix, where `p` is a string. If `p='1'`, the 1-norm is returned, if `p='inf'`, the infinity-norm is returned, and if `p='fro'`, the Frobenius norm is returned.

Note: The 1 and infinity norm are not yet implemented for symmetric matrices.

---

keys()

Return a list of tuples `(i,j)` of the nonzero matrix entries.

values()

Return a list of the nonzero matrix entries as floats.

items()
Return a list of tuples `(indices, value)` of the nonzero entries keys and values. The indices are themselves tuples `(i,j)` of row and column values.

scale(*sigma*)
Scale each element in the matrix by the constant `sigma`.

take(*val*, *irow*, *jcol*)
Extract elements at positions `(irow[i], jcol[i])` and place them in the array `val`. In other words:

```
for i in range(len(val)): val[i] = A[irow[i],jcol[i]]
```

put(*val*, *irow*, *jcol*)
`put` takes the opposite tack to `take`. Place the values in `val` at positions given by `irow` and `jcol`:

```
for i in range(len(val)): A[irow[i],jcol[i]] = val[i]
```

Here, `irow` and `jcol` can be Python lists or integer Numpy arrays. If either `irow` or `jcol` is omitted, it is replaced with `[0, 1, 2, ...]`. Similarly, `val` can be a Python list, an integer Numpy array or a single scalar. If `val` is a scalar `v`, it has the same effect as if it were the constant list or array `[v, v, ..., v]`.

delete_rows(*mask*)
Delete rows in place. If `mask[i] == 0`, the `i`-th row is deleted. This operation does not simply zero out rows, they are *removed*, i.e., the resulting matrix is *smaller*.

delete_cols(*mask*)
Similar to `delete_rows` only with columns.

delete_rowcols(*mask*)
If `mask[i] == 0` both the `i`-th row and the `i`-th column are deleted.

find()
Returns a triple `(val,irow,jcol)` of Numpy arrays containing the matrix in coordinate format. There is a nonzero element with value `val[i]` in position `(irow[i],jcol[i])`.

3.1.3 `csr_mat` and `sss_mat` Objects

`csr_mat` objects represent matrices stored in the CSR format, which are described in *Sparse Matrix Formats*. `sss_mat` objects represent matrices stored in the SSS format (c.f. *Sparse Matrix Formats*). The only way to create a `csr_mat` or a `sss_mat` object is by conversion of a `ll_mat` object using the `to_csr()` or the `to_sss()` method respectively. The purpose of the `csr_mat` and the `sss_mat` objects is to provide fast matrix-vector multiplications for sparse matrices. In addition, a matrix stored in the CSR or SSS format uses less memory than the same matrix stored in the LL format, since the `link` array is not needed.

`csr_mat` and `sss_mat` objects do not support two-dimensional indices to access matrix entries or sub-matrices. Again, their purpose is to provide fast matrix-vector multiplication.

`csr_mat` and `sss_mat` Object Attributes and Methods

class `spmatrix.`**`csr_mat`**
A general sparse matrix class in compressed sparse row format which also allows the representation of symmetric matrices. Only the lower triangle of a symmetric matrix is kept in memory for efficiency.

class `spmatrix.`**`sss_mat`**
A general sparse matrix class in sparse skyline format which also allows the representation of symmetric matrices. Only the lower triangle of a symmetric matrix is kept in memory for efficiency.

shape
Returns a 2-tuple containing the shape of the matrix $\mathbf{A}$, i.e. the number of rows and columns.

nnz
Returns the number of non-zero entries stored in matrix $\mathbf{A}$. If $\mathbf{A}$ is an `sss_mat` object, the non-zero entries in the strictly upper triangle are not counted.

matvec (x, y)
Computes the sparse matrix-vector product $\mathbf{y} := \mathbf{A}\mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ are double precision, rank-1 NumPy arrays of appropriate size.

matvec_transp (x, y)
Computes the transposed sparse matrix-vector product $\mathbf{y} := \mathbf{A}^T\mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ are double precision, rank-1 NumPy arrays of appropriate size. For `sss_mat` objects `matvec_transp` is equivalent to `matvec`.

3.2 Example: 2D-Poisson matrix

This section illustrates the use of the `spmatrix` module to build the well known 2D-Poisson matrix resulting from a $n \times n$ square grid:

```
from pysparse.sparse import spmatrix

def poisson2d(n):
    n2 = n*n
    L = spmatrix.ll_mat(n2, n2, 5*n2-4*n)
    for i in range(n):
        for j in range(n):
            k = i + n*j
            L[k,k] = 4
            if i > 0:
               L[k,k-1] = -1
            if i < n-1:
               L[k,k+1] = -1
            if j > 0:
               L[k,k-n] = -1
            if j < n-1:
               L[k,k+n] = -1
    return L
```

Using the symmetric variant of the `ll_mat` object, this gets even shorter:

```
def poisson2d_sym(n):
    n2 = n*n
    L = spmatrix.ll_mat_sym(n2, 3*n2-2*n)
    for i in range(n):
        for j in range(n):
            k = i + n*j
            L[k,k] = 4
            if i > 0:
               L[k,k-1] = -1
            if j > 0:
               L[k,k-n] = -1
    return L
```

To illustrate the use of the slice notation to address sub-matrices, let's build the 2D Poisson matrix using the diagonal and off-diagonal blocks:

```
def poisson2d_sym_blk(n):
    n2 = n*n
    L = spmatrix.ll_mat_sym(n2, 2*n2-2*n)
    I = spmatrix.ll_mat_sym(n, n)
    P = spmatrix.ll_mat_sym(n, 2*n-1)
    for i in range(n):
        I[i,i] = -1
    for i in range(n):
        P[i,i] = 4
    if i > 0: P[i,i-1] = -1
    for i in range(0, n*n, n):
        L[i:i+n,i:i+n] = P
        if i > 0: L[i:i+n,i-n:i] = I
    return L
```

3.3 Vectorization

The `put` method of `ll_mat` objects allows us to operate on entire arrays at a time. This is advantageous because the loop over the elements of an array is performed at C level instead of in the Python script.

If you need to `put` the same value in many places, `put` lets you specify this value as a floating-point number instead of an array, e.g.:

```
A.put(4.0, range(n), range(n))
```

is perfectly equivalent to:

```
A.put(4*numpy.ones(n), range(n), range(n))
```

Moreover, if the second index set is omitted, it defaults to `range(n)` where `n` is the appropriate matrix dimension. So the above is again perfectly equivalent to:

```
A.put(4.0, range(n))
```

For illustration, let's rewrite the `poisson2d`, `poisson2d_sym` and `poisson2d_sym_blk` constructors.

The `put` method can be used in `poisson2d` as so:

```
from pysparse.sparse import spmatrix
import numpy

def poisson2d_vec(n):
    n2 = n*n
    L = spmatrix.ll_mat(n2, n2, 5*n2-4*n)
    d = numpy.arange(n2, dtype=numpy.int)
    L.put(4.0, d)
    L.put(-1.0, d[:-n], d[n:])
    L.put(-1.0, d[n:], d[:-n])
    for i in xrange(n):
        di = d[i*n:(i+1)*n]
        L.put(-1.0, di[1:], di[:-1])
        L.put(-1.0, di[:-1], di[1:])
    return L
```

And similarly in the symmetric version:

```
def poisson2d_sym_vec(n):
    n2 = n*n
```

```
    L = spmatrix.ll_mat_sym(n2, 3*n2-2*n)
    d = numpy.arange(n2, dtype=numpy.int)
    L.put(4.0, d)
    L.put(-1.0, d[n:], d[:-n])
    for i in xrange(n):
        di = d[i*n:(i+1)*n]
        L.put(-1.0, di[:-1], di[1:])
    return L
```

The time differences to construct matrices with and without vectorization can be dramatic. The following timings were generated on a 2.4GHz Intel Core2 Duo processor:

```
In [1]: from pysparse.tools import poisson, poisson_vec

In [2]: %timeit -n10 -r3 L = poisson.poisson2d(100)
10 loops, best of 3: 38.2 ms per loop
In [3]: %timeit -n10 -r3 L = poisson_vec.poisson2d_vec(100)
10 loops, best of 3: 4.26 ms per loop

In [4]: %timeit -n10 -r3 L = poisson.poisson2d(300)
10 loops, best of 3: 352 ms per loop
In [5]: %timeit -n10 -r3 L = poisson_vec.poisson2d_vec(300)
10 loops, best of 3: 31.7 ms per loop

In [6]: %timeit -n10 -r3 L = poisson.poisson2d(500)
10 loops, best of 3: 980 ms per loop
In [7]: %timeit -n10 -r3 L = poisson_vec.poisson2d_vec(500)
10 loops, best of 3: 86.4 ms per loop

In [8]: %timeit -n10 -r3 L = poisson.poisson2d(1000)
10 loops, best of 3: 4.02 s per loop
In [9]: %timeit -n10 -r3 L = poisson_vec.poisson2d_vec(1000)
10 loops, best of 3: 333 ms per loop
```

and for the symmetric versions:

```
In [18]: %timeit -n10 -r3 L = poisson.poisson2d_sym(100)
10 loops, best of 3: 22.6 ms per loop
In [19]: %timeit -n10 -r3 L = poisson_vec.poisson2d_sym_vec(100)
10 loops, best of 3: 2.48 ms per loop

In [20]: %timeit -n10 -r3 L = poisson.poisson2d_sym(300)
10 loops, best of 3: 202 ms per loop
In [21]: %timeit -n10 -r3 L = poisson_vec.poisson2d_sym_vec(300)
10 loops, best of 3: 20 ms per loop

In [22]: %timeit -n10 -r3 L = poisson.poisson2d_sym(500)
10 loops, best of 3: 561 ms per loop
In [23]: %timeit -n10 -r3 L = poisson_vec.poisson2d_sym_vec(500)
10 loops, best of 3: 53.8 ms per loop

In [24]: %timeit -n10 -r3 L = poisson.poisson2d_sym(1000)
10 loops, best of 3: 2.26 s per loop
In [25]: %timeit -n10 -r3 L = poisson_vec.poisson2d_sym_vec(1000)
10 loops, best of 3: 205 ms per loop
```

From these numbers, it is obvious that vectorizing is crucial, especially for large matrices. The gain in terms of time seems to be a factor of at least four or five. Note that the last system has order one million.

Finally, the block version could be written as:

```
def poisson2d_vec_sym_blk(n):
    n2 = n*n
    L = spmatrix.ll_mat_sym(n2, 3*n2-2*n)
    D = spmatrix.ll_mat_sym(n, 2*n-1)
    d = numpy.arange(n, dtype=numpy.int)
    D.put(4.0, d)
    D.put(-1.0, d[1:], d[:-1])
    P = spmatrix.ll_mat_sym(n, n-1)
    P.put(-1,d)
    for i in xrange(n-1):
        L[i*n:(i+1)*n, i*n:(i+1)*n] = D
        L[(i+1)*n:(i+2)*n, i*n:(i+1)*n] = P
    # Last diagonal block
    L[-n:,-n:] = D
    return L
```

Here, `put` is sufficiently efficient that the benefit of constructing the matrix by blocks is not apparent anymore. The slicing and block notation can nevertheless be used for clarity. It could also be implemented as a combination of `find` and `put`, at the expense of memory consumption.

```
In [9]: %timeit -n10 -r3 L = poisson.poisson2d_sym_blk(1000)
10 loops, best of 3: 246 ms per loop
In [10]: %timeit -n10 -r3 L = poisson_vec.poisson2d_sym_blk_vec(1000)
10 loops, best of 3: 232 ms per loop
```

3.4 Matlab Implementation

Let's compare the performance of three python codes above with the following Matlab functions:

The Matlab function `poisson2d` is equivalent to the Python function with the same name

```
function L = poisson2d(n)
         L = sparse(n*n);
         for i = 1:n
             for j = 1:n
                 k = i + n*(j-1);
                 L(k,k) = 4;
                 if i > 1, L(k,k-1) = -1; end
                 if i < n, L(k,k+1) = -1; end
                 if j > 1, L(k,k-n) = -1; end
                 if j < n, L(k,k+n) = -1; end
             end
         end
```

The function `poisson2d_blk` is an adaption of the Python function `poisson2d_sym_blk` (except for exploiting the symmetry, which is not directly supported in Matlab).

```
function L = poisson2d_blk(n)
         e = ones(n,1);
         P = spdiags([-e 4*e -e], [-1 0 1], n, n);
         I = -speye(n);
         L = sparse(n*n);
         for i = 1:n:n*n
             L(i:i+n-1,i:i+n-1) = P;
             if i > 1, L(i:i+n-1,i-n:i-1) = I; end
```

```
        if i < n*n - n, L(i:i+n-1,i+n:i+2*n-1) = I; end
    end
```

The function `poisson2d_kron` demonstrates one of the most efficient ways to generate the 2D Poisson matrix in Matlab.

```
function L = poisson2d_kron(n)
    e = ones(n,1);
    P = spdiags([-e 2*e -e], [-1 0 1], n, n);
    L = kron(P, speye(n)) + kron(speye(n), P);
```

The Matlab functions above were place in a Matlab script names `poisson.m` which takes `n` as argument. It then calls `poisson2d`, `poisson2d_blk` and `poisson2d_kron` successively, surrounding each call by `tic` and `toc`. The tests were performed on a 2.4GHz Intel Core2 Duo running Matlab 7.6.0.324 (R2008a).

The results are as follows:

```
>> poisson(100)
poisson2d      Elapsed time is 1.731940 seconds.
poisson2d_blk  Elapsed time is 0.804837 seconds.
poisson2d_kron Elapsed time is 0.019118 seconds.

>> poisson(300)
poisson2d      Elapsed time is 145.979044 seconds.
poisson2d_blk  Elapsed time is 32.785585 seconds.
poisson2d_kron Elapsed time is 0.215165 seconds.

>> poisson(500)
poisson2d      Elapsed time is 2318.512099 seconds.
poisson2d_blk  Elapsed time is 292.355093 seconds.
poisson2d_kron Elapsed time is 0.627137 seconds.

>> poisson(1000)
poisson2d_kron Elapsed time is 2.317660 seconds.
```

It is striking to see how slow the straightforward `poisson2d` version is in Matlab. As we see in the next section, the Python version is faster by several orders of magnitude.

3.5 Comparison with Matlab

First, consider the simple `Poisson2D` function. The *table below* summarizes the results of the previous section by giving timing ratios between the Python and Matlab Poisson constructors.

Table 3.1: Matlab vs. Python: Construction of 2D Poisson matrices.

n	Matlab	Python	Python_vec	Matlab/Python	Matlab/Python_vec
100	1.73	0.0382	0.00426	45.53	406.1
300	145.98	0.3520	0.0317	414.72	4605.0
500	2318.51	0.9800	0.0864	2365.8	26834.6
1000	∞	4.02	0.333	∞	∞

Unfortunately, since Matlab does not explicitly support symmetric matrices, we cannot compare the other functions. For information only, we compare the block version of the Python constructor with the Kronecker-product version of the Matlab constructor. The results are in the *next table*.

Table 3.2: Matlab vs. Python: Construction of 2D Poisson matrices—Fastest methods.

n	Matlab	Python_vec	Matlab/Python_vec
100	0.01912	0.0025	7.65
300	0.2152	0.0219	9.83
500	0.6271	0.0631	9.94
1000	2.318	0.232	9.99

CHAPTER

FOUR

PRECONDITIONERS

4.1 The `precon` Module

The `precon` module provides preconditioners, which can be used e.g.for the iterative methods implemented in the in the `itsolvers` module or the JDSYM eigensolver (in the `jdsym` module).

In the Pysparse framework, any Python object that has the following properties can be used as a preconditioner:

- a `shape` attribute, which returns a 2-tuple describing the dimension of the preconditioner,
- a `precon` method, that accepts two vectors $\mathbf{x}$ and $\mathbf{y}$, and applies the preconditioner to $\mathbf{x}$ and stores the result in $\mathbf{y}$. Both $\mathbf{x}$ and $\mathbf{y}$ are double precision, rank-1 NumPy arrays of appropriate size.

The `precon` module implements two new object types `jacobi` and `ssor`, representing Jacobi and SSOR preconditioners.

4.1.1 `precon` Module Functions

`precon.`**`jacobi`**(*A*, *omega=1.0*, *steps=1*)

Creates a `jacobi` object, representing the Jacobi preconditioner. The parameter $\mathbf{A}$ is the system matrix used for the Jacobi iteration. The matrix needs to be subscriptable using two-dimensional indices, so e.g.an `ll_mat` object would work. The optional parameter ω, which defaults to 1.0, is the weight parameter. The optional `steps` parameter (defaults to 1) specifies the number of iteration steps.

`precon.`**`ssor`**(*A*, *omega=1.0*, *steps=1*)

Creates a `ssor` object, representing the SSOR preconditioner. The parameter $\mathbf{A}$ is the system matrix used for the SSOR iteration. The matrix $\mathbf{A}$ has to be an object of type `sss_mat`. The optional parameter ω, which defaults to 1.0, is the relaxation parameter. The optional `steps` parameter (defaults to 1) specifies the number of iteration steps.

4.1.2 `jacobi` and `ssor` Objects

Both `jacobi` and `ssor` objects provide the `shape` attribute and the `precon` method, that every preconditioner object in the PySparse framework must implement. Apart from that, there is nothing noteworthy to say about these objects.

4.1.3 Example: Diagonal Preconditioner

The diagonal preconditioner is just a special case of the Jacobi preconditioner, with `omega=1.0` and `steps=1`, which happen to be the default values of these parameters.

It is however easy to implement the diagonal preconditioner using a Python class:

```
class diag_prec:
      def __init__(self, A):
          self.shape = A.shape
          n = self.shape[0]
          self.dinv = numpy.empty(n)
          for i in xrange(n):
              self.dinv[i] = 1.0 / A[i,i]
      def precon(self, x, y):
          numpy.multiply(x, self.dinv, y)
```

So:

```
>>> D1 = precon.jacobi(A, 1.0, 1)
```

and:

```
>>> D2 = diag_prec(A)
```

yield functionally equivalent preconditioners. `D1` is probably faster than `D2`, because it is fully implemented in C.

CHAPTER

FIVE

ITERATIVE SOLVERS

5.1 The `itsolvers` Module

The `itsolvers` module provides a set of iterative methods for solving linear systems of equations.

The iterative methods are callable like ordinary Python functions. All these functions expect the same parameter list, and all function return values also follow a common standard.

Any user-defined iterative solvers should also follow these conventions, since other PySparse modules rely on them (e.g. the `jdsym` module; see *Eigenvalue Solver*).

Let's illustrate the calling conventions, using the PCG method.

`info, iter, relres = pcg(A, b, x, tol, maxit[, K])`

Solve a linear system `A x = b` with the preconditioned conjugate gradient algorithm. Overwrite `x` with the best estimate of the solution found.

Parameters

A The coefficient matrix of the linear system of equations. `A` must provide the `shape` attribute and the `matvec` and `matvec_transp` methods for multiplying with a vector.

b The right-hand-side of the linear system as a rank-1 NumPy array.

x A rank-1 NumPy array. Upon entry, `x` holds the initial guess. On exit, `x` holds an approximate solution of the linear system.

tol A float value representing the requested error tolerance. The exact meaning of this parameter depends on the actual iterative solver.

maxit An integer that specifies the maximum number of iterations to be executed.

K A preconditioner object that supplies the `shape` attribute and the `precon` method.

Returns

info an integer that contains the exit status of the iterative solver. `info >= 0` indicates, that `x` holds an acceptable solution, and `info < 0` indicates an error condition. `info` has one of the following values:

+2. iteration converged, residual is as small as seems reasonable on this machine,

+1. iteration converged, `b = 0`, so the exact solution is `x = 0`.

+0. iteration converged, relative error appears to be less than `tol`.

-1. iteration did not converge, maximum number of iterations was reached.

-2. iteration did not converge, the system involving the preconditioner was ill-conditioned.

-3. iteration did not converge, an inner product of the form $\mathbf{x}^T\mathbf{K}^{-1}\mathbf{x}$ was not positive, so the preconditioning matrix $\mathbf{K}$ does not appear to be positive definite.

-4. iteration did not converge, the matrix `A` appears to be very ill-conditioned.

-5. iteration did not converge, the method stagnated.

-6. iteration did not converge, a scalar quantity became too small or too large to continue computing.

iter the number of iterations performed.

relres the relative residual at the approximate solution computed by the iterative method. What this actually is depends on the actual iterative method used.

The iterative solvers may accept additional parameters, which are passed as keyword arguments.

Note that not all iterative solvers check for all above error conditions.

5.1.1 `itsolvers` Module Functions

The module functions defined in the `precon` module implement various iterative methods (PCG, MINRES, QMRS and CGS). The parameters and return values conform to the conventions described above.

`info, iter, relres = pcg(A, b, x, tol, maxit[, K])`
Implements the Preconditioned Conjugate Gradient method.

`info, iter, relres = minres(A, b, x, tol, maxit[, K])`
Implements the MINRES method.

`info, iter, relres = qmrs(A, b, x, tol, maxit[, K])`
Implements the QMRS method.

`info, iter, relres = cgs(A, b, x, tol, maxit[, K])`
Implements the CGS method.

5.1.2 Example: Solving the Poisson System

Let's solve the Poisson system

$$\mathbf{L}\mathbf{x} = \mathbf{1}, \tag{5.1}$$

using the PCG method. $\mathbf{L}$ is the 2D Poisson matrix, introduced in *Low-Level Sparse Matrix Types*, and $\mathbf{1}$ is a vector with all entries equal to one.

The Python solution for this task looks as follows:

```
from pysparse.sparse import spmatrix
from pysparse.precon import precon
from pysparse.itsolvers import krylov
import numpy
n = 300
L = poisson2d_sym_blk(n)
b = numpy.ones(n*n)
x = numpy.empty(n*n)
info, iter, relres = krylov.pcg(L.to_sss(), b, x, 1e-12, 2000)
```

The code makes use of the Python function `poisson2d_sym_blk`, which was defined in *Low-Level Sparse Matrix Types*.

Incorporating e.g. a SSOR preconditioner is straightforward:

```
from pysparse.sparse import spmatrix
from pysparse.precon import precon
from pysparse.itsolver import krylov
import numpy
n = 300
L = poisson2d_sym_blk(n)
b = numpy.ones(n*n)
x = numpy.empty(n*n)
S = L.to_sss()
Kssor = precon.ssor(S)
info, iter, relres = krylov.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));
```

5.1.3 Performance comparison with Matlab and native C

Todo

Update the timings below.

Warning: The timings below are Roman's old benchmarks. I don't know on which machine they were run. We should update them.

To evaluate the performance of the Python implementation we solve the 2D Poisson system (5.1) using the PCG method. The Python timings are compared with results of a Matlab and a native C implementation.

The native C and the Python implementation use the same core algorithms for PCG method and the matrix-vector multiplication. On the other hand, C reads the matrix from an external file instead of building it on the fly. In contrast to the Python implementation, the native C version does not suffer from the overhead generated by the runtime argument

parsing and calling overhead.

Table 5.1: Performance comparison of Python, Matlab and native C implementations to solve the linear system (1) without preconditioning. The execution times are given in seconds. Assembly is the time for constructing the matrix (or reading it from a file in the case of native C). Solve is the time spent in the PCG solver. Total is the sum of Assembly and Solve. Matlab version 6.0 Release 12 was used for these timings.

Function	Size	Assembly	Solve	Total
Python	n=100	0.03	1.12	1.15
	n=300	0.21	49.65	49.86
	n=500	0.62	299.39	300.01
Native C	n=100	0.30	0.96	1.26
	n=300	3.14	48.38	51.52
	n=500	10.86	288.67	299.53
Matlab	n=100	0.21	8.85	9.06
	n=300	2.05	387.26	389.31
	n=500	6.23	1905.67	1911.8

This table shows the execution times for the Python, the Matlab and the native C implementation for solving the linear system (5.1). Matlab is not only slower when building the matrix, also the matrix-vector multiplication seems to be implemented inefficiently. Considering *Solve*, the performance of Python and native C is comparable. The Python overhead is under a factor of 4.

CHAPTER

SIX

DIRECT SOLVERS

6.1 The Low-Level C Modules

6.1.1 The `superlu` Module

The `superlu` module interfaces the SuperLU library to make it usable by Python code. SuperLU is a software package written in C, that is able to compute an LU-factorisation of a general non-symmetric sparse matrix with partial pivoting.

The `superlu` module exports a single function, called `factorize`.

`superlu.`**`factorize`**(*A*, ***kwargs*)

The factorize function computes an LU-factorisation of the matrix `A`.

Parameters

A A `csr_mat` object that represents the matrix to be factorized.

Keywords

diag_pivot_thresh the partial pivoting threshold, in the interval $[0, 1]$. `diag_pivot_thresh=0` corresponds to no pivoting. `diag_pivot_thresh=1` corresponds to partial pivoting (default: 1.0).

drop_tol the drop tolerance, in the interval $[0, 1]$. `drop_tol=0` corresponds to the exact factorization (default: 0.0).

relax the degree of relaxing supernodes (default: 1).

panel_size the maximum number of columns that form a panel (default: 10).

permc_spec the matrix ordering used to control sparsity of the factors:

0. natural ordering
1. MMD applied to the structure of $\mathbf{A}^T\mathbf{A}$
2. MMD applied to the structure of $\mathbf{A}^T + \mathbf{A}$
3. COLAMD, approximate minimum degree column ordering

(default: 2).

Return type an object of type `superlu_context`. This object encapsulates the L and U factors of `A` (see below).

Note: The `drop_tol` has no effect in SuperLU version 2.0 and below. In SuperLU version 3.0 and above, the default value of `permc_spec` is 3.

`superlu_context` Object Attributes and Methods

class `superlu.`**`superlu_context`**
An abstract encapsulation of the LU factorization of a matrix by SuperLU.

`shape`
A 2-tuple describing the dimension of the matrix factorized. It is equal to `A.shape`.

`nnz`
The `nnz` attribute holds the total number of nonzero entries stored in both the L and U factors.

`solve`(*b*, *x*, *trans*)
The `solve` method accepts two rank-1 NumPy arrays `b` and `x` of appropriate size and assigns the solution of the linear system $\mathbf{Ax} = \mathbf{b}$ to `x`. If the optional parameter `trans` is set to the string `'T'`, the transposed system $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ is solved instead.

Example: Solving a 2D Poisson System with SuperLU

Let's now solve the 2D Poisson system $\mathbf{Ax} = \mathbf{1}$ using an LU factorization. Here, $\mathbf{A}$ is the 2D Poisson matrix, introduced in *Low-Level Sparse Matrix Types* and $\mathbf{1}$ is a vector with all entries equal to one.

The Python solution for this task looks as follows:

```
from pysparse.sparse import spmatrix
from pysparse.direct import superlu
import numpy
n = 100
A = poisson2d_sym_blk(n)
b = numpy.ones(n*n)
x = numpy.empty(n*n)
LU = superlu.factorize(A.to_csr(), diag_pivot_thresh=0.0)
LU.solve(b, x)
```

The code makes use of the Python function `poisson2d_sym_blk()`, which was defined in *Low-Level Sparse Matrix Types*.

Example: An Incomplete LU Factorization Preconditioner

Warning: SuperLU 3.0 and above accept a `drop_tol` argument although the source files mention that incomplete factorization is *not implemented*. Therefore, changing `drop_tol` has no effect on the factorization at the moment and we must wait for it to be implemented. In the meantime, we can still demonstrate in this section how to implement an incomplete factorization preconditioner in Pysparse, even though in the present situation, it will be a *complete* factorization preconditioner!

Versions of SuperLU above 3.0 accept the `drop_tol` argument that allows the computation of incomplete factors, realizing a tradeoff between computational cost and factor density. The following example show how to use an incomplete LU factorization as a preconditioner in any of the iterative methods of the `itsolvers` module:

```
from pysparse.tools import poisson
from pysparse.direct import superlu
from pysparse.itsolvers import krylov
import numpy

class ILU_Precon:
    """
    A preconditioner based on an
    incomplete LU factorization.

    Input: A matrix in CSR format.
    Keyword argument: Drop tolerance.
    """
    def __init__(self, A, drop=1.0e-3):
        self.LU = superlu.factorize(A, drop_tol=drop)
        self.shape = self.LU.shape

    def precon(self, x, y):
        self.LU.solve(x,y)


n = 300
A = poisson.poisson2d_sym_blk(n).to_csr()    # Convert right away
b = numpy.ones(n*n)
x = numpy.empty(n*n)

K = ILU_Precon(A)
info, niter, relres = krylov.pcg(A, b, x, 1e-12, 2000, K)
```

Note: Note that the 2D Poisson matrix is symmetric and positive definite, although barely. Indeed its smallest eigenvalue is $2(1-\cos(\pi/(n+1))) \approx (\pi/(n+1))^2$. Therefore, a Cholesky factorization would be more appropriate. In the future, we intend to interface the Cholmod library.

6.1.2 The `umfpack` Module

The `umfpack` module interfaces the UMFPACK library to make it usable by Python code. UMFPACK is a software package written in C, that is able to compute an LU factorization of a general non-symmetric sparse matrix with partial pivoting.

Note: The major difference with the `superlu` modules is that `umfpack` receives a matrix in `ll_mat` format instead of *csr_mat* format.

The `umfpack` module exports a single function, called `factorize`.

`superlu.`**`factorize`**(*A*, ***kwargs*)

The factorize function computes an LU-factorisation of the matrix `A`.

Parameters

A A `ll_mat` object that represents the matrix to be factorized.

Keywords

strategy Pivoting strategy. Possible values are:

- "UMFPACK_STRATEGY_AUTO"

- "UMFPACK_STRATEGY_UNSYMMETRIC"
- "UMFPACK_STRATEGY_SYMMETRIC"
- "UMFPACK_STRATEGY_2BY2"

tol2by2 Tolerance for the 2-by-2 strategy.

scale Scaling used during factorization. Possible values are:

- "UMFPACK_SCALE_NONE"
- "UMFPACK_SCALE_SUM"
- "UMFPACK_SCALE_MAX"

tolpivot Relative pivot tolerance for threshold partial pivoting with row interchanges.

tolsympivot If diagonal pivoting is attempted, this parameter controls when the diagonal is selected in a given pivot column.

Return type an object of type `umfpack_context`. This object encapsulates the L and U factors of `A` (see below).

`umfpack_context` Object Attributes and Methods

class `superlu.`**`umfpack_context`**

An abstract encapsulation of the LU factorization of a matrix by UMFPACK.

`shape`

A 2-tuple describing the dimension of the matrix factorized. It is equal to `A.shape`.

`nnz`

The `nnz` attribute holds the total number of nonzero entries stored in the input matrix. It is equal to `A.nnz`. To obtain the number of nonzero element in the factors, see `lunz()`.

`solve`(*b*, *x*, *method*, *irsteps*)

Parameters

b The right-hand side of the system $\mathbf{A}x = b$ as a Numpy array.

x A Numpy array to hold the solution of $\mathbf{A}x = b$.

method (optional) Different systems may be solved by setting the `method` argument appropriately. See the documentation of the `pysparseUmfpackSolver` below for more details.

irsteps (optional) The number of iterative refinement steps to perform.

`lu``()`

Return the factors, permutation and scaling information. See the documentation of the `pysparseUmfpackSolver` below for more details.

`lunz``()`

Return the number of nonzeros in factors, i.e., in $\mathbf{L} + \mathbf{U}$.

Example: Solving a 2D Poisson System with UMFPACK

We now solve again the 2D Poisson system $\mathbf{A}\mathbf{x} = \mathbf{1}$ using an LU factorization. Here, $\mathbf{A}$ is the 2D Poisson matrix, introduced in *Low-Level Sparse Matrix Types* and $\mathbf{1}$ is a vector with all entries equal to one.

The Python solution using UMFPACK looks as follows:

```
from pysparse.sparse import spmatrix
from pysparse.direct import umfpack
import numpy
n = 100
A = poisson2d_sym_blk(n)
b = numpy.ones(n*n)
x = numpy.empty(n*n)
LU = umfpack.factorize(A, strategy="UMFPACK_STRATEGY_SYMMETRIC")
LU.solve(b, x)
```

The code makes use of the Python function `poisson2d_sym_blk()`, which was defined in *Low-Level Sparse Matrix Types*.

6.2 Higher-Level Python Interfaces

This section anticipates on *Higher-Level Sparse Matrix Classes* and shows usage of higher-level interfaces to the LU factorization packages.

6.2.1 The Abstract `directSolver` Module

A framework for solving sparse linear systems of equations using a direct factorization.

class `pysparse.direct.directSolver.`**`PysparseDirectSolver`**(*A*, ***kwargs*)

PysparseDirectSolver is a generic class and should be subclassed.

`solve`(*b*, ***kwargs*)

6.2.2 The `pysparseSuperLU` Module: A Higher-Level SuperLU Interface

A framework for solving sparse linear systems of equations using an LU factorization, by means of the supernodal sparse LU factorization package SuperLU ([DEGLL99], [DGL99], [LD03]).

This package is appropriate for factorizing sparse square unsymmetric or rectangular matrices.

See [SLU] for more information.

References:

class `pysparse.direct.pysparseSuperLU.`**`PysparseSuperLUSolver`**(*A*, ***kwargs*)

Bases: `pysparse.direct.directSolver.PysparseDirectSolver`

PysparseSuperLUSolver is a wrapper class around the SuperLu library for the factorization of full-rank n-by-m matrices. Only matrices with real coefficients are currently supported.

Parameters

A The matrix to be factorized, supplied as a PysparseMatrix instance.

Keywords

symmetric a boolean indicating that the user wishes to use symmetric mode. In symmetric mode, `permc_spec=2` must be chosen and `diag_pivot_thresh` must be small, e.g., 0.0 or 0.1. Since the value of `diag_pivot_thresh` is up to the user, setting `symmetric` to `True` does *not* automatically set `permc_spec` and `diag_pivot_thresh` to appropriate values.

diag_pivot_thresh a float value between 0 and 1 representing the threshold for partial pivoting (0 = no pivoting, 1 = always perform partial pivoting). Default: 1.0.

drop_tol the value of a drop tolerance, between 0 and 1, if an incomplete factorization is desired (0 = exact factorization). This keyword does not exist if using SuperLU version 2.0 and below. In more recent version of SuperLU, the keyword is accepted but has no effect. Default: 0.0

relax an integer controling the degree of relaxing supernodes. Default: 1.

panel_size an integer specifying the maximum number of columns to form a panel. Default: 10.

permc_spec an integer specifying the ordering strategy used during the factorization.

0. natural ordering,
1. MMD applied to the structure of $\mathbf{A}^T\mathbf{A}$
2. MMD applied to the structure of $\mathbf{A}^T + \mathbf{A}$
3. COLAMD.

Default: 2.

LU
A `superlu_context` object encapsulating the factorization.

sol
The solution of the linear system after a call to `solve()`.

factorizationTime
The CPU time to perform the factorization.

solutionTime
The CPU time to perform the forward and backward sweeps.

lunz
The number of nonzero elements in the factors L and U together after a call to `fetch_lunz()`.

fetch_factors()
Not yet available.

fetch_lunz()
Retrieve the number of nonzeros in the factors L and U together. The result is stored in the member `lunz` of the class instance.

solve(*rhs*, *transpose=False*)
Solve the linear system `A x = rhs`, where `A` is the input matrix and `rhs` is a Numpy vector of appropriate dimension. The result is placed in the `sol` member of the class instance.

If the optional argument `transpose` is `True`, the transpose system `A^T x = rhs` is solved.

6.2.3 Example: The 2D Poisson System with SuperLU

The solution of a 2D Poisson system with `PysparseSuperLUSolver` may look like this:

```
from pysparse.sparse.pysparseMatrix import PysparseMatrix
from pysparse.direct.pysparseSuperLU import PysparseSuperLUSolver
from pysparse.tools.poisson_vec import poisson2d_sym_blk_vec
from numpy import ones
from numpy.linalg import norm
```

```
n = 200
A = PysparseMatrix( matrix=poisson2d_sym_blk_vec(n) )
x_exact = ones(n*n)/n
b = A * x_exact
LU = PysparseSuperLUSolver(A)
LU.solve(b)
print 'Factorization time: ', LU.factorizationTime
print 'Solution time: ', LU.solutionTime
print 'Error: ', norm(LU.sol - x_exact)/norm(x_exact)
```

The above script produces the output:

```
Factorization time:  0.494116
Solution time:  0.017096
Error: 2.099685128150953e-14
```

Note that this example uses the vectorized Poisson constructors of *Low-Level Sparse Matrix Types*.

6.2.4 The `pysparseUmfpack` Module: A Higher-Level UMFPACK Interface

A framework for solving sparse linear systems of equations using an LU factorization, by means of the unsymmetric multifrontal sparse LU factorization package UMFPACK ([D04a], [D04b], [DD99], [DD97]).

This package is appropriate for factorizing sparse square unsymmetric or rectangular matrices.

See [UMF] for more information.

References:

class pysparse.direct.pysparseUmfpack.**PysparseUmfpackSolver**(*A*, ***kwargs*)

Bases: pysparse.direct.directSolver.PysparseDirectSolver

PysparseUmfpackSolver is a wrapper class around the UMFPACK library for the factorization of full-rank n-by-m matrices. Only matrices with real coefficients are currently supported.

Parameters

A A PysparseMatrix instance representing the matrix to be factorized.

Keywords

strategy string that specifies what kind of ordering and pivoting strategy UMFPACK should use. Valid values are 'auto', 'unsymmetric', 'symmetric' and '2by2'. Default: 'auto'

tol2by2 tolerance for the 2 by 2 strategy. Default: 0.1

scale string that specifies the scaling UMFPACK should use. Valid values are 'none', 'sum', and 'max'. Default: 'sum'.

tolpivot relative pivot tolerance for threshold partial pivoting with row interchanges. Default: 0.1

tolsympivot if diagonal pivoting is attempted, this parameter is used to control when the diagonal is selected in a given pivot column. Default: 0.0

LU

An `umfpack_context` object encapsulating the factorization.

sol

The solution of the linear system after a call to `solve()`.

L
The L factor of the input matrix.

U
The U factor of the input matrix.

P
The row permutation used for the factorization.

Q
The column permutation used for the factorization.

R
The row scaling used during the factorization. See the documentation of `fetch_factors()`.

factorizationTime
The CPU time to perform the factorization.

solutionTime
The CPU time to perform the forward and backward sweeps.

do_recip
Nature of the row scaling. See `fetch_factors()`.

lnz
The number of nonzero elements in the factor L.

unz
The number of nonzero elements in the factor U from which the diagonal was removed.

nz_udiag
The number of nonzero elements on the diagonal of the factor U.

fetch_factors()
Retrieve the L and U factors of the input matrix along with the permutation matrices P and Q and the row scaling matrix R such that

$$\mathbf{PRAQ} = \mathbf{LU}.$$

The matrices P, R and Q are stored as Numpy arrays. L and U are stored as PysparseMatrix instances and are lower triangular and upper triangular, respectively.

R is a row-scaling diagonal matrix such that

- the i-th row of A has been divided by R[i] if `do_recip = True`,
- the i-th row of A has been multiplied by R[i] if `do_recip = False`.

fetch_lunz()
Retrieve the number of nonzeros in the factors. The results are stored in the members `lnz`, `unz` and `nz_udiag` of the class instance.

solve(*rhs*, ***kwargs*)
Solve the linear system `A x = rhs`. The result is placed in the `sol` member of the class instance.

Parameters

rhs a Numpy vector of appropriate dimension.

Keywords

method specifies the type of system being solved:

"UMFPACK_A"	$\mathbf{A}x = b$ (default)
"UMFPACK_At"	$\mathbf{A}^T x = b$
"UMFPACK_Pt_L"	$\mathbf{P}^T\mathbf{L}x = b$
"UMFPACK_L"	$\mathbf{L}x = b$
"UMFPACK_Lt_P"	$\mathbf{L}^T\mathbf{P}x = b$
"UMFPACK_Lt"	$\mathbf{L}^T x = b$
"UMFPACK_U_Qt"	$\mathbf{U}\mathbf{Q}^T x = b$
"UMFPACK_U"	$\mathbf{U}x = b$
"UMFPACK_Q_Ut"	$\mathbf{Q}\mathbf{U}^T x = b$
"UMFPACK_Ut"	$\mathbf{U}^T x = b$

irsteps number of iterative refinement steps to attempt. Default: 2

6.2.5 Example: The 2D Poisson System with UMFPACK

The solution of a 2D Poisson system with `PysparseUmfpackSolver` may look like this:

```
from pysparse.tools.poisson_vec import poisson2d_sym_blk_vec
from numpy import ones
from numpy.linalg import norm

n = 200
A = PysparseMatrix( matrix=poisson2d_sym_blk_vec(n) )
x_exact = ones(n*n)/n
b = A * x_exact
LU = PysparseUmfpackSolver(A)
LU.solve(b)
print 'Factorization time: ', LU.factorizationTime
print 'Solution time: ', LU.solutionTime
print 'Error: ', norm(LU.sol - x_exact)/norm(x_exact)
```

This script produces the output:

```
Factorization time:  0.520043
Solution time:  0.031086
Error: 1.10998989668e-15
```

CHAPTER

SEVEN

EIGENVALUE SOLVER

7.1 The `jdsym` Module

The `jdsym` module provides an implementation of the JDSYM algorithm, that is conveniently callable from Python. JDSYM is an eigenvalue solver to compute eigenpairs of a generalised matrix eigenvalue problem of the form

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{M}\mathbf{x} \tag{7.1}$$

or a standard eigenvalue problem of the form

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{7.2}$$

where $\mathbf{A}$ is symmetric and $\mathbf{M}$ is symmetric positive definite.

The module exports a single function:

`jdsym.`**`jdsym`**(*A*, *M*, *K*, *kmax*, *tau*, *jdtol*, *itmax*, *linsolver*, ***kwargs*)

Implements Jacobi-Davidson iterative method to identify a given number of eigenvalues near a target value.

Parameters

A the matrix $\mathbf{A}$ in (7.1) or (7.2). `A` must provide the `shape` attribute and the `matvec` and `matvec_transp` methods.

M the matrix $\mathbf{M}$ in (7.1). $\mathbf{M}$ must provide the `shape` attribute and the `matvec` and `matvec_transp` methods. If the standard eigenvalue problem (7.2) is to be solved, `M` should be set to `None`.

K a preconditioner object that supplies the `shape` attribute and the `precon` method. If no preconditioner is to used, then the `None` value can be passed for this parameter.

kmax the number of eigenpairs to be computed.

tau the target value τ. Eigenvalues in the vicinity of τ will be computed.

jdtol the convergence tolerance for eigenpairs $(\lambda, \mathbf{x})$. The converged eigenpairs have a residual $\|\mathbf{A}\mathbf{x} - \lambda\mathbf{M}\mathbf{x}\|_2$ less than `jdtol`.

itmax an integer that specifies the maximum number of Jacobi-Davidson iterations to perform.

linsolver 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 *Iterative Solvers*.

Keywords

jmax the maximum dimension of the search subspace (default: 25).

jmin the dimension of the search subspace after a restart (default: 10).

blksize the block size used in the JDSYM algorithm (default: 1).

blkwise is an integer that affects the convergence criterion if `blksize` is larger than 1 (default: 0).

V0 a 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 (default: 2).

linitmax the maximum number of steps taken in the inner iteration (iterative linear solver) (default: 200).

eps_tr the tracking parameter (default: 1.0e-3).

toldecay is a float value that influences the dynamic adaptation of the stopping criterion of the inner iteration (default: 1.5).

clvl verbosity level. The higher the `clvl` parameter, the more output is sent to the standard output. `clvl=0` produces no output (default: 0).

strategy is an integer specifying shifting and sorting strategy of JDSYM. `strategy=0` enables the default JDSYM algorithm. `strategy=1` enables JDSYM to avoid convergence to eigenvalues smaller than τ (default: 0).

projector is used to keep the search subspace and the eigenvectors in a certain subspace. The parameter `projector` can be any Python object that has a `shape` attribute and a `project` method. The `project` method takes a vector (a rank-1 NumPy array) as its sole argument and projects that vector in-place. This parameter can be used to implement the DIRPROJ and SAUG methods (default: no projection).

Returns

kconv the number of converged eigenpairs.

lambda a rank-1 NumPy array containing the converged eigenvalues.

Q a rank-2 NumPy array containing the converged eigenvectors. The i-th eigenvector is accessed by `Q[:,i]`.

it an integer indicating the number of Jacobi-Davidson steps (outer iteration steps) performed.

7.1.1 Example: Maxwell Problem

Todo

Update the timings below.

> **Warning:** The timings below are Roman's old benchmarks. We should run them again.

The following code illustrates the use of the `jdsym` module. Two matrices $\mathbf{A}$ and $\mathbf{M}$ are read from files. A Jacobi preconditioner from $\mathbf{A} - \tau\mathbf{M}$ 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 ($\mathbf{A}$, $\mathbf{M}$):

```
from pysparse.sparse import spmatrix
from pysparse.itsolvers import krylov
from pysparse.eig import jdsym
from pysparse.precon import 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  = jdsym.jdsym(A, M, K, 5, tau,
                                   1e-10, 150, krylov.qmrs,
                                   jmin=5, jmax=10, clvl=1, strategy=1)
```

This code takes 33.71 seconds to compute the five eigenpairs. A native C version, using the same computational kernels, takes 35.64 for the same task. We expected the Python version to be slower due to the overhead generated when calling the matrix-vector multiplication and the preconditioner, but surprisingly the Python code was even a bit faster. New in version 1.0.1.

CHAPTER

EIGHT

HIGHER-LEVEL SPARSE MATRIX CLASSES

8.1 The `pysparseMatrix` module

This module defines a few convenience classes as wrappers around ll_mat objects. Being proper Python classes, they are subclassable. PysparseMatrix objects have hooks for all methods of ll_mat objects.

class `pysparse.sparse.pysparseMatrix.`**`PysparseMatrix`**(***kwargs*)

Bases: `pysparse.sparse.sparseMatrix.SparseMatrix`

A PysparseMatrix is a class wrapper for the pysparse spmatrix sparse matrix type. This class facilitates matrix populating and allows intuitive operations on sparse matrices and vectors.

Keywords

nrow The number of rows of the matrix

ncol The number of columns of the matrix

size The common number of rows and columns, for a square matrix

bandwidth The bandwidth (if creating a band matrix)

matrix The starting *spmatrix* if there is one

sizeHint A guess on the number of nonzero elements of the matrix

symmetric A boolean indicating whether the matrix is symmetric.

storeZeros A boolean indicating whether to store zero values.

`addAt`(*vector*, *id1*, *id2*)

Add elements of `vector` to the positions in the matrix corresponding to `(id1,id2)`

```
>>> L = PysparseMatrix(size = 3)
>>> L.put([3.,10.,numpy.pi,2.5], [0,0,1,2], [2,1,1,0])
>>> L.addAt((1.73,2.2,8.4,3.9,1.23), (1,2,0,0,1), (2,2,0,0,2))
>>> print L
12.300000  10.000000   3.000000
    ---     3.141593   2.960000
 2.500000      ---     2.200000
```

`addAtDiagonal`(*vector*)

Add the components of vector `vector` to the diagonal elements of the matrix.

col_scale(*v*)
Apply in-place column scaling. Each column is scaled by the corresponding component of `v`, i.e., `A[:,i] *= v[i]`.

copy()
Returns a (deep) copy of a sparse matrix

exportMmf(*filename*)
Exports the matrix to a Matrix Market file of the given filename.

find()
Returns three Numpy arrays to describe the sparsity pattern of `self` in so-called coordinate (or triplet) format:

```
>>> L = PysparseMatrix(size = 3)
>>> L.put([3.,10.,numpy.pi,2.5], [0,0,1,2], [2,1,1,0])
>>> (val,irow,jcol) = L.find()
>>> val
array([ 10.        ,   3.        ,   3.14159265,   2.5       ])
>>> irow
array([0, 0, 1, 2])
>>> jcol
array([1, 2, 1, 0])
```

getMatrix()
Returns the underlying *ll_mat* sparse matrix of *self*

getNnz()
Returns the number of nonzero elements of *self*

getNumpyArray()
Convert a sparse matrix to a dense Numpy matrix.

getShape()
Returns the shape `(nrow,ncol)` of a sparse matrix

isSymmetric()
Returns *True* is *self* is a symmetric matrix or *False* otherwise

matvec(*x*)
This method is required for scipy solvers.

put(*value*, *id1=None*, *id2=None*)
Put elements of `value` at positions of the matrix corresponding to `(id1, id2)`

```
>>> L = PysparseMatrix(size = 3)
>>> L.put( [3.,10.,numpy.pi,2.5], [0,0,1,2], [2,1,1,0] )
>>> print L
    ---    10.000000   3.000000
    ---     3.141593      ---
 2.500000      ---        ---
>>> L.put(2*numpy.pi, range(3), range(3))
>>> print L
 6.283185  10.000000   3.000000
    ---     6.283185      ---
 2.500000      ---     6.283185
```

If `value` is a scalar, it has the same effect as the vector of appropriate length with all values equal to `value`. If `id1` is omitted, it is replaced with `range(nrow)`. If `id2` is omitted, it is replaced with `range(ncol)`.

putDiagonal(*vector*)
Put elements of `vector` along diagonal of matrix

```
>>> L = PysparseMatrix(size = 3)
>>> L.putDiagonal([3.,10.,numpy.pi])
>>> print L
 3.000000      ---        ---
    ---    10.000000      ---
    ---        ---     3.141593
>>> L.putDiagonal([10.,3.])
>>> print L
10.000000      ---        ---
    ---     3.000000      ---
    ---        ---     3.141593
>>> L.putDiagonal(2.7182)
>>> print L
 2.718200      ---        ---
    ---     2.718200      ---
    ---        ---     2.718200
```

row_scale(*v*)
Apply in-place row scaling. Each row is scaled by the corresponding component of `v`, i.e., `A[i,:] *= v[i]`.

take(*id1=None*, *id2=None*)
Extract elements at positions `(irow[i], jcol[i])` and place them in the array `val`. In other words:

```
for i in range(len(val)): val[i] = A[irow[i],jcol[i]]
```

takeDiagonal()
Extract the diagonal of a matrix and place it in a Numpy array.

class pysparse.sparse.pysparseMatrix.**PysparseIdentityMatrix**(*size*)
Bases: pysparse.sparse.pysparseMatrix.PysparseMatrix

Represents a sparse identity matrix for pysparse.

```
>>> print PysparseIdentityMatrix(size = 3)
 1.000000      ---        ---
    ---     1.000000      ---
    ---        ---     1.000000
```

class pysparse.sparse.pysparseMatrix.**PysparseSpDiagsMatrix**(*size*, *vals*, *pos*, ***kwargs*)
Bases: pysparse.sparse.pysparseMatrix.PysparseMatrix

Represents a banded matrix with specified diagonals.

Example: Create a tridiagonal matrix with 1's on the diagonal, 2's above the diagonal, and -2's below the diagonal.

```
>>> from numpy import ones
>>> e = ones(5)
>>> print PysparseSpDiagsMatrix(size=5, vals=(-2*e,e,2*e), pos=(-1,0,1))
 1.000000   2.000000      ---        ---        ---
-2.000000   1.000000   2.000000      ---        ---
    ---    -2.000000   1.000000   2.000000      ---
    ---        ---    -2.000000   1.000000   2.000000
    ---        ---        ---    -2.000000   1.000000
```

Note that since the *pos[k]*-th diagonal has *size*-|*pos[k]*| elements, only that many first elements of *vals[k]* will be inserted.

If the banded matrix is requested to be symmetric, elements above the main diagonal are not inserted.

8.1.1 Fancy Indexing

Fancy indexing carries over to `PysparseMatrix` objects and is used exactly in the same way as with `ll_mat` objects. Refer to Section *Low-Level Sparse Matrix Types* for details.

CHAPTER

NINE

OTHER SPARSE MATRIX PACKAGES FOR PYTHON

- Scipy provides a sparse matrix module featuring several storage formats.
- CVXOPT is a package for convex optimization and allows creation and manipulation of sparse matrices in compressed sparse row format.

CHAPTER

TEN

LICENSE

Parts of this documentation are scavenged from Roman Geus' original Pysparse website.

Some pages of this documentation display equations via the jsMath package. They should look reasonably good with most setups but the best rendering is obtained by installing the TeX fonts. Please refer to http://www.math.union.edu/~dpvc/jsMath/users/welcome.html.

CHAPTER

ELEVEN

TODO LIST

Todo

Update the timings below.

(The *original entry* is located in itsolvers.rst, line 162.)

Todo

Update the timings below.

(The *original entry* is located in jdsym.rst, line 106.)

CHAPTER

TWELVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[DEGLL99] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li and J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM Journal on Matrix Analysis and Applications **20**(3), pp. 720-755, 1999.

[DGL99] J. W. Demmel, J. R. Gilbert and X. S. Li, *An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination*, SIAM Journal on Matrix Analysis and Applications **20**(4), pp. 915-952, 1999.

[LD03] X. S. Li and J. W. Demmel, *SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*, ACM Transactions on Mathematical Software **29**(2), pp. 110-140, 2003.

[SLU] http://crd.lbl.gov/~xiaoye/SuperLU

[D04a] T. A. Davis, *A column pre-ordering strategy for the unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, **30**(2), pp. 165-195, 2004.

[D04b] T. A. Davis, *Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, **30**(2), pp. 196-199, 2004.

[DD99] T. A. Davis and I. S. Duff, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, ACM Transactions on Mathematical Software, **25**(1), pp. 1-19, 1999.

[DD97] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM Journal on Matrix Analysis and Applications, **18**(1), pp. 140-158, 1997.

[UMF] http://www.cise.ufl.edu/research/sparse/umfpack

PYTHON MODULE INDEX

INDEX

M

N

P

Q

R

S

T

U

V