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

- iterative 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.

All these modules are implemented as C extension modules for maximum performance. In the following sections all modules of PySparse are described in detail.

0.1.1 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 (cf. Appendix **??**). 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 Matrix-Market 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.

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

$$\boldsymbol{C} := \boldsymbol{A}\boldsymbol{B}$$

and returns the result $\boldsymbol{C}$ as a new *ll_mat* object representing a general sparse matrix. The parameters $\boldsymbol{A}$ and $\boldsymbol{B}$ are expected to be objects of type *ll_mat*.

dot(A, B) computes the "dot-product"

$$\boldsymbol{C} := \boldsymbol{A}^T\boldsymbol{B}$$

and returns the result $\boldsymbol{C}$ as a new *ll_mat* object representing a general sparse matrix. The parameters $\boldsymbol{A}$ and $\boldsymbol{B}$ are expected to be objects of type *ll_mat*.

ll_mat objects *ll_mat* objects represent matrices stored in the LL format, which are described in Appendix **??**. *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[1]. 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]`.

[1]The standard Python language does not know multidimensional indices. However, thanks to Python's clever design, its easy to provide multidimensional indices for extension types, without any dirty hacks.

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.

```
>>> 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
                      # starting at row 1 col 2,
                      # ending at 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])
```

ll_mat object attributes

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

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

A.issym Returns true (a non-zero integer) if matrix $\boldsymbol{A}$ 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 $\boldsymbol{A}$ is stored in the general LL format.

ll_mat object methods

A.to_csr() Returns a newly allocated *csr_mat* object, which results from converting matrix $\boldsymbol{A}$.

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

A.export_mtx(fileName, precision=6) Exports the matrix $\boldsymbol{A}$ to file named *fileName*. The matrix is stored in MatrixMarket Coordinate format as described at `http://math.nist.gov/MatrixMarket/formats.html`. Depending on the properties of *ll_mat* object $\boldsymbol{A}$ 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.

A.shift(sigma, M) Performs a daxpy-like operation on matrix $\boldsymbol{A}$,

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \sigma \boldsymbol{M}.$$

The parameter σ is expected to be a Python Float object. The parameter $\boldsymbol{M}$ is expected to an object of type *ll_mat*.

A.copy() Returns a new *ll_mat* object, that represents a copy of the *ll_mat* object $\boldsymbol{A}$. 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`.

A.update_add_mask(B, ind0, ind1, mask0, mask1) This method is provided for efficiently assembling global finite element matrices. The method adds the matrix $\boldsymbol{B}$ to entries of matrix $\boldsymbol{A}$. The indices of the entries to be updated are specified by *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. $\boldsymbol{B}$ is an array of rank two. The four remaining parameters are rank-1 arrays. Their length corresponds to either the number of rows or the number of columns of $\boldsymbol{B}$.

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.

A.update_add_mask_sym(B, ind, mask) This method is provided for efficiently assembling symmetric global finite element matrices. The method adds the matrix $\boldsymbol{B}$ to entries of matrix $\boldsymbol{A}$. The indices of the entries to be updated are specified by *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. $\boldsymbol{B}$ is an array of rank two representing a square matrix. The four remaining parameters are rank-1 arrays. Their length corresponds to the order of matrix $\boldsymbol{B}$.

csr_mat and sss_mat objects *csr_mat* objects represent matrices stored in the CSR format, which are described in Appendix **??**. *sss_mat* objects represent matrices stored in the SSS format (c.f. Appendix **??**). 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 *to_sss()* 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

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

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

csr_mat and sss_mat object methods

A.matvec(x, y) Computes the sparse matrix-vector product

$$\mathbf{y} \leftarrow \boldsymbol{A}\mathbf{x}.$$

$\mathbf{x}$ and $\mathbf{y}$ are two double precision, rank-1 NumPy arrays of appropriate size.

A.matvec_transp(x, y) Computes the transposed sparse matrix-vector product

$$\mathbf{y} \leftarrow \boldsymbol{A}^T\mathbf{x}.$$

$\mathbf{x}$ and $\mathbf{y}$ are two double precision, rank-1 NumPy arrays of appropriate size. For *sss_mat* objects *matvec_transp* is equivalent to *matvec*.

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.

```
def poisson2d(n):
    L = spmatrix.ll_mat(n*n, n*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):
    L = spmatrix.ll_mat_sym(n*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):
    L = spmatrix.ll_mat_sym(n*n)
    I = spmatrix.ll_mat_sym(n)
    P = spmatrix.ll_mat_sym(n)
    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
```

Performance comparison with Matlab 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	$n = 100$	$n = 300$	$n = 500$	$n = 1000$
Python `poisson2d`	0.44	4.11	11.34	45.50
Python `poisson2d_sym`	0.26	2.34	6.55	26.33
Python `poisson2d_sym_blk`	0.03	0.21	0.62	2.22
Matlab `poisson2d`	28.19	3464.9	38859	∞
Matlab `poisson2d_blk`	6.85	309.20	1912.1	∞
Matlab `poisson2d_kron`	0.21	2.05	6.23	29.96

Table 1: Performance comparison of Python and Matlab functions to generate the 2D Poisson matrix
The execution times are given in seconds. Matlab version 6.0 Release 12 was used for these timings.

```
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 execution times reported in Tab. 1 clearly show, that the Python implementation is superior to the Matlab implementation. If the fastest versions are compared for both languages, Python is approximately 10 times faster. Comparing the straight forward `poisson2d` versions, one is struck by the result that, the Matlab function is incredibly slow. The Python version is more then three orders of magnitude faster! This result really raises the doubt, whether Matlab's sparse matrix format is appropriately chosen.

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 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,
- and 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 the SSOR preconditioners as described in Sections **??** and **??**.

precon module functions

jacobi(A, omega=1.0, steps=1) Creates a *jacobi* object, representing the Jacobi preconditioner. The parameter $\boldsymbol{A}$ 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 as described in Section **??**. The optional *steps* parameter (defaults to 1) specifies the number of iteration steps.

ssor(A, omega=1.0, steps=1) Creates a *ssor* object, representing the SSOR preconditioner. The parameter $\boldsymbol{A}$ is the system matrix used for the SSOR iteration. The matrix $\boldsymbol{A}$ has to be an object of type *sss_mat*. The optional parameter ω, which defaults to 1.0, is the relaxation parameter as described in Section **??**. The optional *steps* parameter (defaults to 1) specifies the number of iteration steps.

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.

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 = Numeric.zeros(n, 'd')
        for i in xrange(n):
            self.dinv[i] = 1.0 / A[i,i]
    def precon(self, x, y):
        Numeric.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.

0.1.3 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)

Parameter list Let's illustrate the calling conventions, using the PCG method defined as *info*, *iter*, *relres* = *pcg*($\boldsymbol{A}$, $\mathbf{b}$, $\mathbf{x}$, *tol*, *maxit*, $\boldsymbol{K}$).

- $\boldsymbol{A}$ The parameter $\boldsymbol{A}$ represents the coefficient matrix of the linear system of equations. $\boldsymbol{A}$ must provide the *shape* attribute and the *matvec* and *matvec_transp* methods for multiplying with a vector.
- $\mathbf{b}$ The parameter $\mathbf{b}$, representing the right-hand-side of the linear system, is a rank-1 NumPy array.
- $\mathbf{x}$ The parameter $\mathbf{x}$ is also a rank-1 NumPy array. On input, $\mathbf{x}$ holds the initial guess. On output, $\mathbf{x}$ holds the approximate solution of the linear system.
- *tol* The *tol* parameter is a float value representing the requested error tolerance. The exact meaning of this parameter depends on the actual iterative solver.
- *maxit* The *maxit* parameter is an integer that specifies the maximum number of iterations to be executed.
- $\boldsymbol{K}$ The *optional* $\boldsymbol{K}$ parameter represents a preconditioner object that supplies the *shape* attribute and the *precon* method.

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

Return value All iterative solvers return a tuple with three elements (*info*, *iter*, *relres*):

info is an integer that contains the exit status of the iterative solver. *info* has one of the following values

- 2 iteration converged, residual is as small as seems reasonable on this machine.
- 1 iteration converged, $\mathbf{b} = \mathbf{0}$, so the exact solution is $\mathbf{x} = \mathbf{0}$.
- 0 iteration converged, relative error appears to be less than *tol*.
- -1 iteration not converged, maximum number of iterations was reached.
- -2 iteration not converged, the system involving the preconditioner was ill-conditioned.
- -3 iteration not converged, an inner product of the form $\mathbf{x}^T \boldsymbol{K}^{-1}\mathbf{x}$ was not positive, so the preconditioning matrix $\boldsymbol{K}$ does not appear to be positive definite.
- -4 iteration not converged, the matrix $\boldsymbol{A}$ appears to be very ill-conditioned
- -5 iteration not converged, the method stagnated
- -6 iteration not converged, a scalar quantity became too small or too large to continue computing

So, $info >= 0$ indicates, that $\mathbf{x}$ holds an acceptable solution, and $info < 0$ indicates an error condition.

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

iter holds the of iterations performed.

relres holds relative error of the solution computed by the iterative method. What this actually is, depends on the actual iterative method used.

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

pcg(A, b, x, tol, maxit, K) The *pcg* function implements the Preconditioned Conjugate Gradients method.

minres(A, b, x, tol, maxit, K) The *minres* function implements the MINRES method.

qmrs(A, b, x, tol, maxit, K) The *qmrs* function implements the QMRS method.

cgs(A, b, x, tol, maxit, K) The *min* function implements the CGS method.

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

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

using the PCG method. $\boldsymbol{L}$ is the 2D Poisson matrix, introduced in Section 0.1.1 and $\mathbf{1}$ 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));
```

Performance comparison with Matlab and native C To evaluate the performance of the Python implementation we solve the 2D Poisson system (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.

Tab. 2 shows the execution times for the Python, the Matlab and the native C implementation for solving the linear system (1). Matlab is not only slower when building the matrix, also the matrix-vector multiplication seems to be implemented inefficiently. Considering t_{solv}, the performance of Python and native C is comparable. The Python overhead is under 4% for this case.

0.1.4 The jdsym module

The *jdsym* module provides an implementation of the JDSYM algorithm (cf. Algorithm **??**), that is conveniently callable from Python. The module exports a single function called *jdsym*.

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
	$n = 500$	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

Table 2: Performance comparison of Python, Matlab and native C implementations to solve the linear system (1) without preconditioning
The execution times are given in seconds. t_{constr} is the time for constructing the matrix (or reading it from a file in the case of native C). t_{solv} is the time spent in the PCG solver. t_{tot} is the sum of t_{constr} and t_{solv}. Matlab version 6.0 Release 12 was used for these timings.

jdsym(A, M, K, kmax, tau, jdtol, itmax, linsolver, **keywords) Invokes the JDSYM eigenvalue solver (cf. Section **??**). JDSYM computes eigenpairs of a generalised matrix eigenvalue problem of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \boldsymbol{M}\mathbf{x} \tag{2}$$

or a standard eigenvalue problem of the form

$$\boldsymbol{A}\mathbf{x} = \lambda \mathbf{x}, \tag{3}$$

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

Arguments The *jdsym* function has seven mandatory arguments

$\boldsymbol{A}$ This parameter represents the matrix $\boldsymbol{A}$ in (2) or (3). $\boldsymbol{A}$ must provide the *shape* attribute and the *matvec* and *matvec_transp* methods.

$\boldsymbol{M}$ This parameter represents the matrix $\boldsymbol{M}$ in (2). $\boldsymbol{M}$ must provide the *shape* attribute and the *matvec* and *matvec_transp* methods. If the standard eigenvalue problem (3) is to be solved, the `None` value can be passed for this parameter.

$\boldsymbol{K}$ The $\boldsymbol{K}$ parameter represents 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 is an integer that specifies the number of eigenpairs to be computed.

tau is a float value that specifies the target value τ. Eigenvalues in the vicinity of τ will be computed.

jdtol is a float value that specifies the convergence tolerance for eigenpairs $(\lambda, \mathbf{x})$. The converged eigenpairs satisfy $\|\boldsymbol{A}\mathbf{x} - \lambda\boldsymbol{M}\mathbf{x}\|_2 < $ *jdtol*.

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

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})

toldecay is a float value, that influences the dynamic adaption of the stopping criterion of the inner iteration. *toldecay* corresponds to the value γ in Section **??**. (default: 1.5)

clvl is an integer specifying the "verbosity" of the *jdsym* function. 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 actually be any Python object, that provides 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 described in Sections **??** and **??**. (Default: no projection)

Return value The *jdsym* module function returns a tuple with four elements (*kconv*, *lambda*, *Q*, *it*):

kconv is an integer that indicates the number of converged eigenpairs.

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

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

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

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

```
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  = \
      jdsym.jdsym(A, M, K, 5, tau, 1e-10, 150, itsolvers.qmrs,
                  jmin=5, jmax=10, clvl=1, strategy=1)
```

This code takes 33.71 seconds to compute the five wanted 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.

0.1.5 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 a LU-factorisation of a general non-symmetric, sparse matrix with partial pivoting.

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

factorize(A, diag_pivot_thresh, drop_tol, relax, panel_size, permc_spec) The factorise module function computes a LU-factorisation of the matrix $\boldsymbol{A}$. All but the first parameter are optional and can be specified using keyword arguments.

$\boldsymbol{A}$ is a *csr_mat* object that represents the matrix to be factorised.

diag_pivot_thresh is a float value in the interval $[0, 1]$ representing the threshold for partial pivoting. *diag_pivot_thresh* $= 0$ corresponds to no pivoting. *diag_pivot_thresh* $=$ 1 corresponds to partial pivoting. (default: 1.0)

drop_tol is a float value in the interval $[0, 1]$ representing the drop tolerance parameter. *drop_tol* $= 0$ corresponds to the exact factorisation.
CAUTION: the *drop_tol* has no effect in the current and all older SuperLU releases (versions 2.0 and below). (default: 0.0)

relax is an integer that controls the degree of relaxing supernodes. (default: 1)

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

permc_spec is an integer specifying the matrix ordering used for the factorisation:

0 natural ordering

1 MMD applied to the structure of $\boldsymbol{A}^T\boldsymbol{A}$

2 MMD applied to the structure of $\boldsymbol{A}^T + \boldsymbol{A}$

3 COLAMD, approximate minimum degree column ordering

(default: 2)

The *factorize* function returns an object of type *superlu_context*. This object encapsulates the LU-factors computed during the factorisation.

superlu_context object attributes

shape The *shape* attribute, returns a 2-tuple describing the dimension of the factorised matrix $\boldsymbol{A}$.

nnz The *nnz* attribute returns an integer holding the total number of non-zero entries stored in both the $\boldsymbol{L}$ and the $\boldsymbol{U}$ factors.

superlu_context object methods

solve(b, x, trans) The *solve* method accepts two rank-1 NumPy arrays $\mathbf{b}$ and $\mathbf{x}$ of appropriate size and assigns the solution of the linear system

$$\boldsymbol{A}\mathbf{x} = \mathbf{b}$$

to $\mathbf{x}$. If the optional parameter *trans* is set to `'T'`, then the transposed system

$$\boldsymbol{A}^T\mathbf{x} = \mathbf{b}$$

is solved instead.

Example: 2D Poisson matrix Let's now solve the 2D Poisson system

$$\boldsymbol{L}\mathbf{x} = \mathbf{1},$$

using a factorisation. $\boldsymbol{L}$ is the 2D Poisson matrix, introduced in Section 0.1.1 and $\mathbf{1}$ is a vector with all one entries.
The Python solution for this task looks as follows:

```
import Numeric, spmatrix, superlu
n = 100
L = poisson2d_sym_blk(n)
b = Numeric.ones(n*n, 'd')
x = Numeric.zeros(n*n, 'd')
LU = superlu.factorize(L.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 Section 0.1.1.