CONTENTS

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,
-

CHAPTER

TWO

SPARSE MATRIX FORMATS

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

CHAPTER THREE

3.1.2 ll_mat objects

`ll_mat`

rank-1 NumPy arrays of appropriate size. For `sss_mat` objects `matvec_transp` is equivalent to `matvec`.

`to_csr()`

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. `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 `B`.

`update_add_at`(*val*, *irow*, *jcol*)
Add in place the elements of the vector `val`

3.1.3 `csr_mat` and `sss_mat` Objects

`csr_mat` objects repreaent matrices stored in the CSR format, which are described in *Sparse Matrix Formats*. `sss_mat` objects repreaent matrices stored in the SSS format (c.f.

```
            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*poisson2d_sym(n):

          if
```

if

= -1

```
In [9]: %timeit -n10 -r3 L = poisson.poisson2d(1000)
10 loops, best of 3: 4.02 s per loop
In [10]: %timeit -n10 -r3 L = poisson_vec.poisson2d_vec(1000)
10 loops, best of 3: 398 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: 5.05 ms per loop

In [20]: %timeit -n10 -r3 L = poisson.poisson2d_sym(3ps, best of 3: 5.05 ms per loop
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: 27 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: 63.7 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: 224 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 I50(a6 8.seems)-2(actk)-600([18]:)]TJ0 g 0 G0.40 0.40 0.40 rg 0.40 0.40 0.40 I

CHAPTER

FOUR

PRECONDITIONERS

4.1 The `precon` Module

CHAPTER

FIVE

ITERATIVE SOLVERS

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.

`nnz`

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

`solve`

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


n = .00
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 = itsolvers.pcg(A, b, x, 1e-12, 2000, K)
```

Note:

6.2.2 The

`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

scale string that specifies the scaling UMFPACK should use. Valid values are ‘none’,

PysparsolvePyspar(*rhs, x* **)**PysparSolve the linear systemPyspar $A x = rhs$Pyspar. The result is placed in the

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

$$Ax = \quad Mx \tag{7.1}$$

or a standard eigenvalue problem of the form

$$Ax = \quad x \tag{7.2}$$

where A 0(an)-25symmetric and M 0(an)-25symmetric positive definite.

The module exports a single function:

`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 A in (7.1) or (7.2). A must provide the `shape` attribute and the `matvec` and `matvec_transp` methods.
- **M** the matrix M in (7.1). M must provide the `shape` attribute and the `matvec` and `matvec_transp` methods. If the standard eigenvalue problem (7.2

blkwise is an integer that affects the convergence criterion if `blksize`

```
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)
```

CHAPTER

EIGHT

HIGHER-LEVEL SPARSE MATRIX CLASSES

8.1 The `pysparseMatrix` module

class `PysparseMatrix`(***kwargs*)

Bases: `sparseMatrix.SparseMatrix`

```
>>> L = PysparseMatrix(size = 3)
```

8.1.1 Creating an Identity Matrix

class `PysparseIdentityMatrix`(*size*)

Bases: `pysparseMatrix.PysparseMatrix`

Represents a sparse identity matrix for pysparse.

CHAPTER

NINE

CHAPTER
ELEVEN

CHAPTER

TWELVE

INDICES AND TABLES

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 Scacable 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. T. A. Davis, *Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method*, Mathematical Software, **30**(2), pp. 196. 2004.

[DD99] T. A. Davis and I. S. Duff, *A combined unifrontal/multifrontal method for unsymmetric s* Transactions on Mathematical Software, **25**(1), pp. 1-19, 1999. T. A. Davis and I. S. Duff, *An unsymm* Journal on Matrix Analysis and Application

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

INDEX

A

C

D

E

F