



# Sparse Matrix Implementation in Octave

D. Bateman†, A. Adler‡

† Centre de Recherche, Motorola, Les Algorithmes, Commune de St Aubin, 91193 Gif-Sur-Yvette, FRANCE

‡ School of Information Technology and Engineering (SITE), University of Ottawa, 161 Louis Pasteur Ottawa, Ontario, Canada, K1N 6N5

April 2006

# Sparse matrices?

## ➪ Why sparse matrices?

- ✔ Many classes of problems result in matrices with a large number of zeros,
- ✔ A sparse matrix is a special class of matrix that allows only the non-zero terms to be stored
- ✔ Reduction in the storage requirements for sparse matrices
- ✔ Significant speed improvement as many calculations involving zero elements are neglected

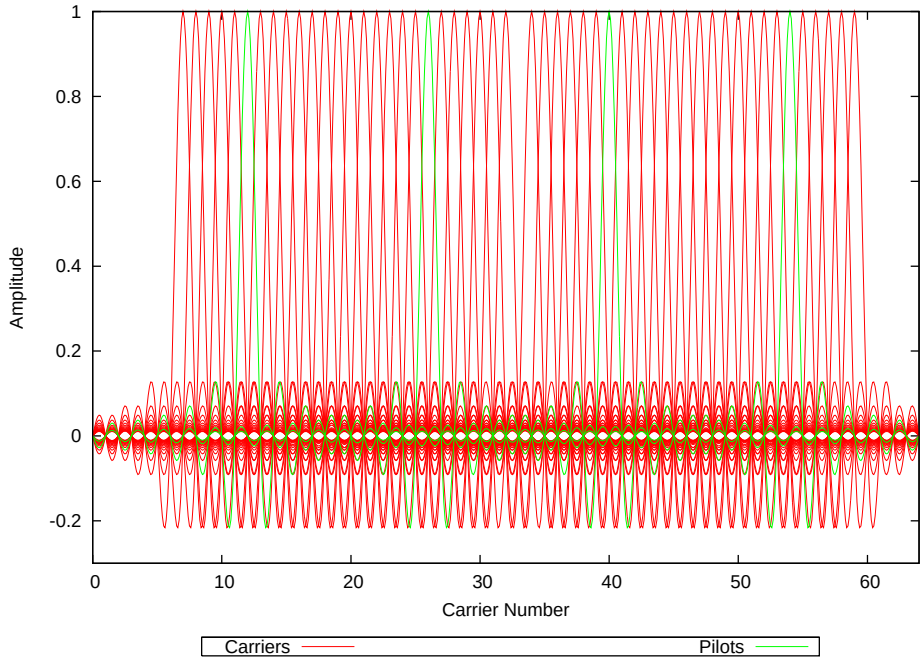
## ➪ Sparse matrices prior to Octave 2.9

- ✔ Fake sparse implementation that allows sparse keywords while treating the matrices as full
- ✔ Sparse matrix implmentation in octave-forge with SuperLU solvers
  - ✘ Missing several key features. Notably sparse indexing. That is

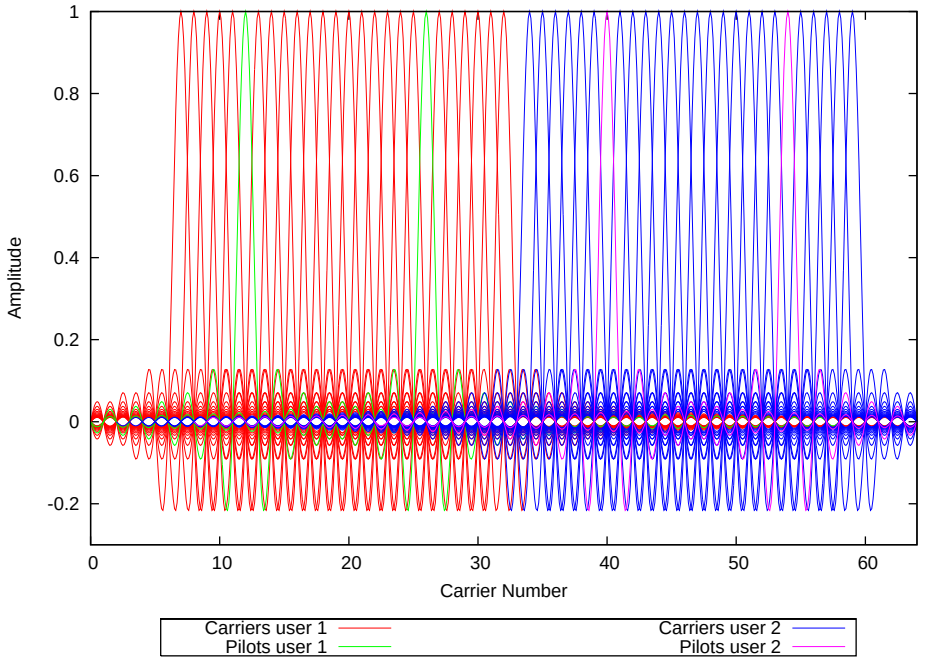
```
a = sparse(...);   ## Sparse Matrix
a(1) = 1;          ## Full Matrix
```

  - ✘ Large use of C constructs such as malloc/free preventing its inclusion in octave.

# Why am I using sparse matrices




Typical 802.11a carrier layout




Extension of 802.11a carrier layout to OFDMA

- ✔ OFDMA is an extension of OFDM where the carriers of OFDM are shared between users
- ✔ Sparse block diagonal matrices arise naturally from OFDMA systems
- ✔ Some systems propose upto 4096 carriers with only a small fraction used by a single user

# Storage of sparse matrices

- ✔ Given a-priori knowledge of problems to be solve, choose sparse matrix storage to reduce stirage and complexity
- ✔ Octave use column-major storage of matrices, and so compressed column format is a natural choice of sparse storage

$$\begin{matrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 4 \end{matrix}$$

$$\begin{aligned} (1,1) &= 1 \\ (1,2) &= 2 \\ (2,4) &= 3 \\ (3,4) &= 4 \end{aligned}$$

$$\begin{aligned} cidx &= [0,1,2,2,4] \\ ridx &= [0,0,1,2] \\ data &= [1,2,3,4] \end{aligned}$$

- ✔ Octave forces the elements of each column to be stored in increasing order of their row index

# Creating Sparse Matrices

## ➪ Ways of creating Sparse matrices

- ✔ Returned from a function (eg *sprand*, *speye*, *spdiags*, etc
- ✔ Constructed from matrix or vectors (eg *sparse*, *spconvert*)
- ✔ Created, and then filled
- ✔ Return from user oct-file

## ➪ What to look out for

Recommended way to create a sparse matrix is from three vectors with the *sparse* function. There is a memory reallocation at each step of a sparse assignment (vectorize!)

```
k = 5; nz = r * k; s = spalloc (r, c, nz)
for j = 1:c
  idx = randperm (r);
  s (:, j) = [zeros(r - k, 1); rand(k, 1)] (idx);
endfor
```

# Sparse Functions

- Generate sparse matrices: *spalloc*, *spdiags*, *speye*, *sprand*, *sprandn*, *sprandsym*
- Sparse matrix conversion: *full*, *sparse*, *spconvert*, *spfind*
- Manipulate sparse matrices: *issparse*, *nnz*, *nonzeros*, *nzmax*, *spfun*, *spones*, *spy*,
- Graph Theory: *etree*, *etreeplot*, *gplot*, *treeplot*, treelayout
- Sparse matrix reordering: *ccolamd*, *colamd*, *colperm*, *csymamd*, *symamd*, *randperm*, *dmperm*, symrcm
- Linear algebra: *matrix_type*, *spchol*, *spcholinv*, *spchol2inv*, *spdet*, *spinv*, *spkron*, *splchol*, *splu*, *spqr*, condest, eigs, normest, sprank, svds, spaugment
- Iterative techniques: *luinc*, bicg, bicgstab, cholinc, cgs, gmres, lsqr, minres, pcg, pcr, qmr, symmlq
- Miscellaneous: *spparms*, *symbfact*, *spstats*, *spprod*, *spcumsum*, *spsum*, *spsumsq*, *spmin*, *spmax*, *spatan2*, *spdiag*

And all of the one argument mapper functions (eg sin, cos, etc)

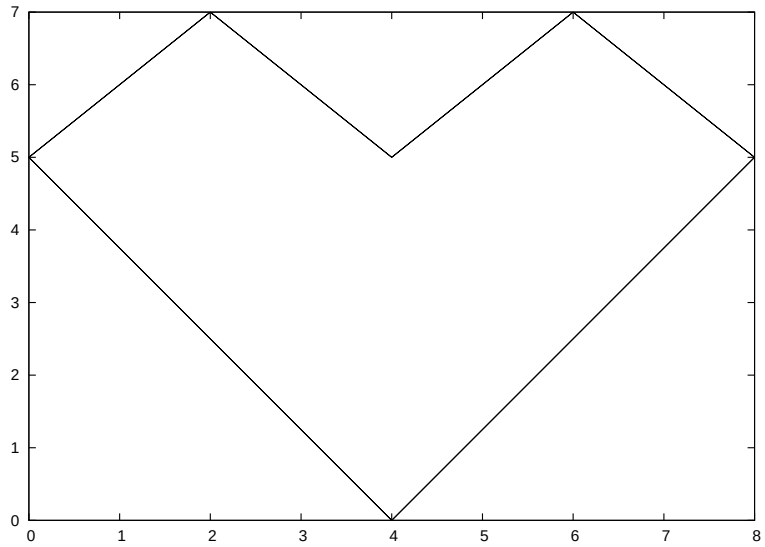
# Sparse Return Types

- ✔ In general the return type of sparse functions and operators is adapted to minimize the memory requirements to store the matrix.
- ✔ Functions that have a high probability of always returning a sparse matrix, will return one (eg. *speye(3) + 0*)
- ✔ Any sparse matrix that takes more memory than the full version will be promoted to a full matrix when returned to the user.

# Information About Sparse Matrices

Basic functions giving information include, *issparse* which identifies a sparse matrix, *nnz* which returns the number of non-zero elements, *nzmax* which returns the storage for non-zero elements and *spstats* that gives some basic statistics.
Functions giving graphical information are *spy* giving the structure of the matrix, *etreeplot* giving the elimination tree and *gplot* the adjacency of nodes in the matrix. *gplot* is used as

```
A    =    sparse([2,6,1,3,2,4,3,5,4,6,1,5],
              [1,1,2,2,3,3,4,4,5,5,6,6],1,6,6);
xy   =    [0,4,8,6,4,2;5,0,5,7,5,7]';
gplot(A,xy)
```




The final function giving information of interest is *matrix_type* that identifies that type of matrix as it will be treated by the left- and right-division operators.

# Mathematical Considerations

## Treatment of zeros in sparse matrices

The basic premise of Octave's sparse matrix implementation is that sparse matrices should give equivalent results to calculations involving full matrix. Consider

```
s  = speye(4);   ## Sparse Matrix
a1 = s.^2;       ## No problems
a2 = s.^s;       ## Attention to 0.^0 terms (1)
a3 = s.^ - 2;    ## Attention of 0.^ - 2 terms (Inf)
a4 = s./2;       ## No problems
a5 = 2./s;       ## Attention to 2./0 terms (Inf)
a6 = s./s;       ## Attention to 0./0 terms (NaN)
```

This is in contrast to Matlab that returns different matrices for full and sparse versions for the above. Similar behavior to Matlab can be had by using

```
function z = f(x), z = 2 ./ x; endfunction; a5 = spfun (@f, s);
```

## The sign-bit of zero

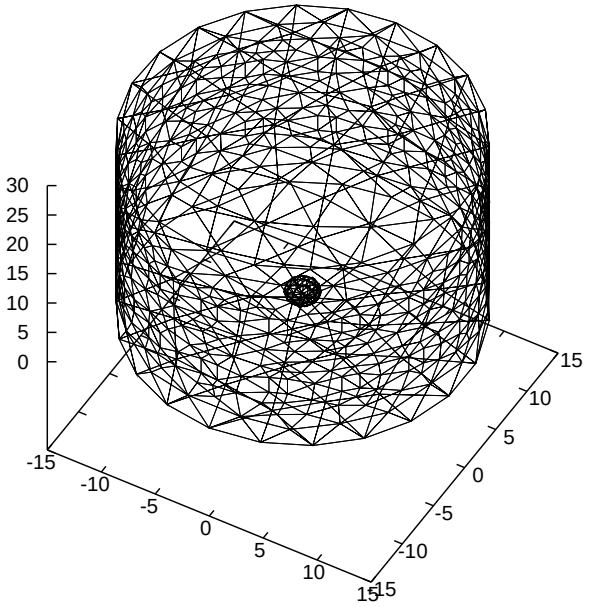
As sparse matrices do not store zero values, they do not store the sign bit of zero values. In certain cases [a] the sign bit is important.

```
a = 0 ./ [-1, 1; 1, -1];
b = 1 ./ a
   -Inf    Inf
    Inf   -Inf
c = 1 ./ sparse (a)
    Inf    Inf
    Inf    Inf
```

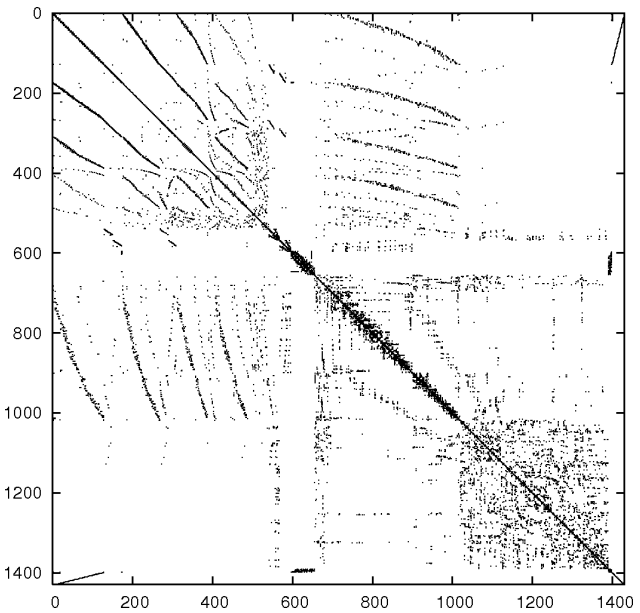
[a] W Kahan, "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", Chapter 7, "The State of the Art in Numerical Analysis", Oxford 1987

## Reordering for sparsity

- ✔ Functions/operators on a sparse matrix result in a sparse matrix with more non-zero values
- ✔ Reformulated with suitable permutations, such that the operator or function on the permuted problem is sparser than the original problems ($LU = PSQ$ versus $LU = S$)

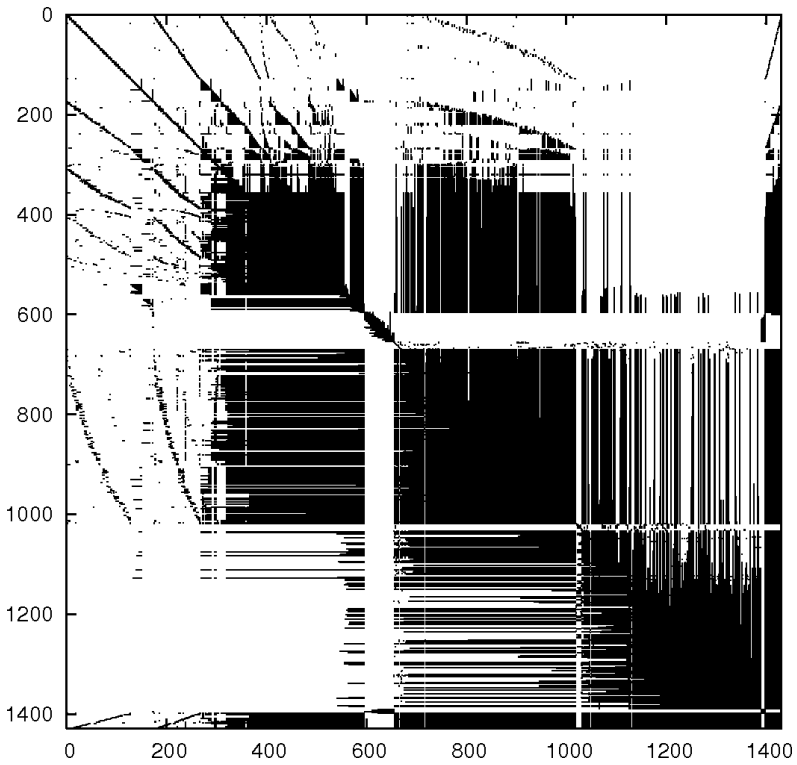



Geometry of FEM model of phantom ball [a]

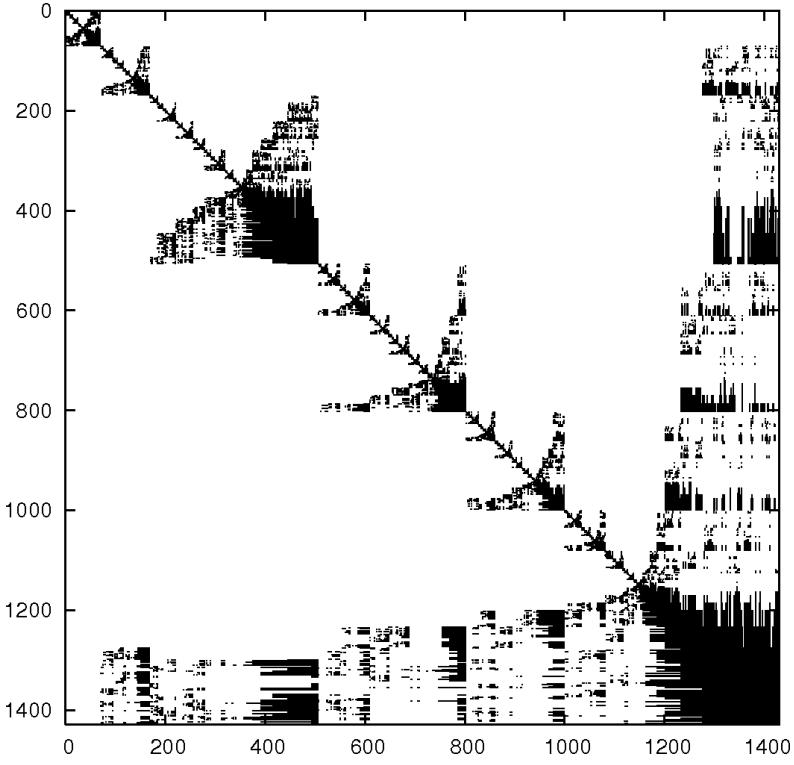



Structure of derived spare matrix

[a] http://eidors3d.sourceforge.net




Structure of LU factorization




Structure of permuted LU factorization

LU factorization of unpermuted matrix has 212044 non-zero terms while LU factorization of permuted matrix has 143491 terms.

# Linear Algebra

1. if the matrix is diagonal, solve directly and goto 8
2. If the matrix is a permuted diagonal, solve directly taking into account the permutations. Go to 8
3. If the matrix is square, banded and if the band density is less than that given by *spparms ("bandden")* continue, else go to 4.
   (a) If the matrix is tridiagonal and the right-hand side is not sparse continue, else go to 3b.
      i. If the matrix is hermitian, with a positive real diagonal, attempt Cholesky factorization using *Lapack* xPTSV.
      ii. If the above failed, or the matrix is not hermitian, use Gaussian elimination with pivoting using *Lapack* xGTSV, and go to 8.
   (b) If the matrix is hermitian with a positive real diagonal, attempt a Cholesky factorization using *Lapack* xPBTRF.
   (c) if the above failed or the matrix is not hermitian with a positive real diagonal use Gaussian elimination with pivoting using *Lapack* xGBTRF, and go to 8.

4. If the matrix is upper or lower triangular perform a sparse forward or backward substitution, and go to 8
5. If the matrix is a upper triangular matrix with column permutations or lower triangular matrix with row permutations, perform a sparse forward or backward substitution, and go to 8
6. If the matrix is square hermitian with a real positive diagonal, attempt a sparse Cholesky factorization using CHOLMOD.
7. If the sparse Cholesky factorization failed or the matrix is not hermitian, and the matrix is square, perform LU factorization using UMFPACK.
8. If the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, find a minimum norm solution using CXSPARSE.

# Benchmarking Addition and Multiplication

Perform simple benchmarking of sparse addition and multiplication operator with the code

```
nrun = 5; tmin = 1; time = 0; n = 0;
while (time < tmin || n < nrun)
  clear a, b;
  a = sprand (order, order, density);
  t = cputime ();
  b = a OP a;
  time = time + cputime () - t;
  n = n + 1;
end
time = time / n;
```

where OP is * or +, and compare against Matlab R14sp2.

| Order | Den-sity | Execution Time for Operator (sec) | | | | Order | Den-sity | Execution Time for Operator (sec) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Matlab | | Octave | | | | Matlab | | Octave | |
| | | + | * | + | * | | | + | * | + | * |
| 500 | 1e-02 | 0.00049 | 0.00250 | 0.00039 | 0.00170 | 5000 | 1e-03 | 0.00526 | 0.03000 | 0.00257 | 0.01896 |
| 500 | 1e-03 | 0.00008 | 0.00009 | 0.00022 | 0.00026 | 5000 | 1e-04 | 0.00076 | 0.00083 | 0.00049 | 0.00074 |
| 500 | 1e-04 | 0.00005 | 0.00007 | 0.00020 | 0.00024 | 5000 | 1e-05 | 0.00051 | 0.00051 | 0.00031 | 0.00043 |
| 500 | 1e-05 | 0.00004 | 0.00007 | 0.00021 | 0.00015 | 5000 | 1e-06 | 0.00048 | 0.00055 | 0.00028 | 0.00026 |
| 500 | 1e-06 | 0.00006 | 0.00007 | 0.00020 | 0.00021 | 10000 | 1e-02 | 0.22200 | 24.2700 | 0.10878 | 6.55060 |
| 1000 | 1e-02 | 0.00179 | 0.02273 | 0.00092 | 0.00990 | 10000 | 1e-03 | 0.02000 | 0.30000 | 0.01022 | 0.18597 |
| 1000 | 1e-03 | 0.00021 | 0.00027 | 0.00029 | 0.00042 | 10000 | 1e-04 | 0.00201 | 0.00269 | 0.00120 | 0.00252 |
| 1000 | 1e-04 | 0.00011 | 0.00013 | 0.00023 | 0.00026 | 10000 | 1e-05 | 0.00094 | 0.00094 | 0.00047 | 0.00074 |
| 1000 | 1e-05 | 0.00012 | 0.00011 | 0.00028 | 0.00023 | 10000 | 1e-06 | 0.00110 | 0.00098 | 0.00039 | 0.00055 |
| 1000 | 1e-06 | 0.00012 | 0.00010 | 0.00021 | 0.00022 | 20000 | 1e-03 | 0.08286 | 2.65000 | 0.04374 | 1.71874 |
| 2000 | 1e-02 | 0.00714 | 0.23000 | 0.00412 | 0.07049 | 20000 | 1e-04 | 0.00944 | 0.01923 | 0.00490 | 0.01500 |
| 2000 | 1e-03 | 0.00058 | 0.00165 | 0.00055 | 0.00135 | 20000 | 1e-05 | 0.00250 | 0.00258 | 0.00092 | 0.00149 |
| 2000 | 1e-04 | 0.00032 | 0.00026 | 0.00026 | 0.00033 | 20000 | 1e-06 | 0.00189 | 0.00161 | 0.00058 | 0.00121 |
| 2000 | 1e-05 | 0.00019 | 0.00020 | 0.00022 | 0.00026 | 50000 | 1e-04 | 0.05500 | 0.39400 | 0.02794 | 0.28076 |
| 2000 | 1e-06 | 0.00018 | 0.00018 | 0.00024 | 0.00023 | 50000 | 1e-05 | 0.00823 | 0.00877 | 0.00406 | 0.00767 |
| 5000 | 1e-02 | 0.05100 | 3.63200 | 0.02652 | 0.95326 | 50000 | 1e-06 | 0.00543 | 0.00610 | 0.00154 | 0.00332 |

Benchmark of basic operators on Matlab R14sp2 against Octave 2.9.5, on a Pentium 4M 1.6GHz machine with 1GB of memory.

# Benchmarking Left-Division

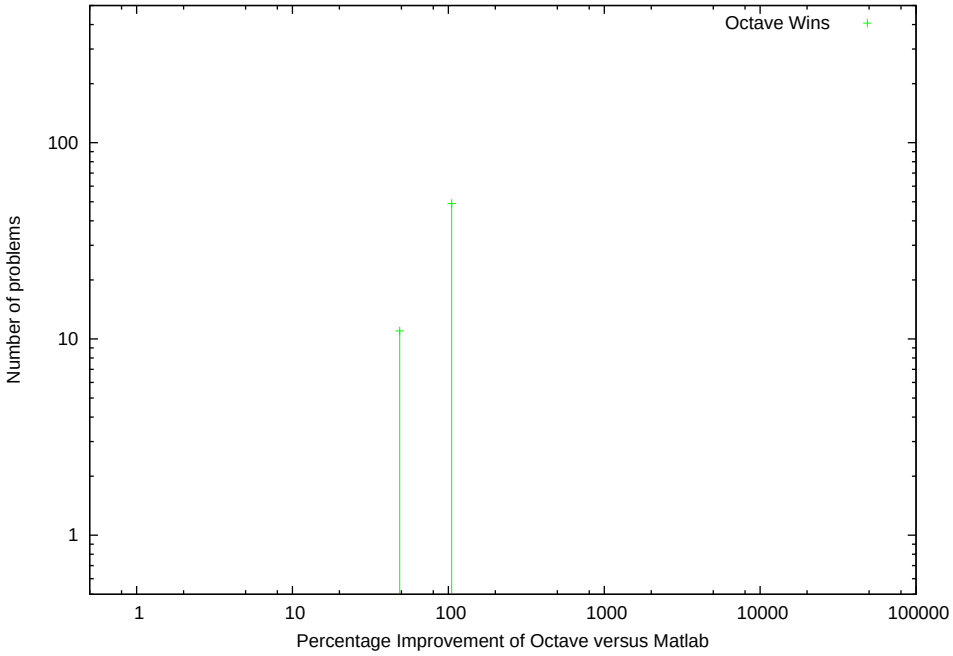
Perform simple benchmarking of sparse left-division operator agianst Matlab, with the code

```
nrun = 5; tmin = 1; time = 0; n = 0;
while (time < tmin || n < nrun)
  clear a, b; load test.mat;
  x = ones(order,1);
  t = cputime (); b = a \ x; time = time + cputime () - t;
  n = n + 1;
end
time = time / n;
```

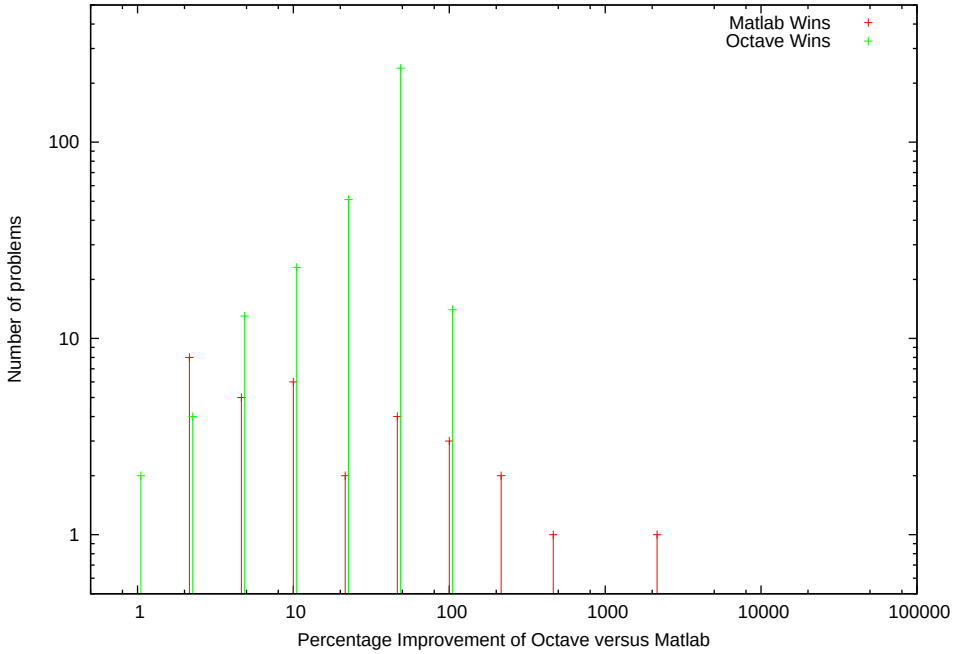
489 matrices from the University of Florida sparse matrix collection (http://www.cise.ufl.edu/research/sparse/matrices) with the following criteria

- Has real or complex data available, and not just the structure,
- Has between 10,000 and 1,000,000 non-zero element,
- Has equal number of rows and columns,
- The solution did not require more than 1GB of memory, to avoid issues with memory.

The metric for the benchmark is $(OctaveTime - MatlabTime)/MatlabTime$.

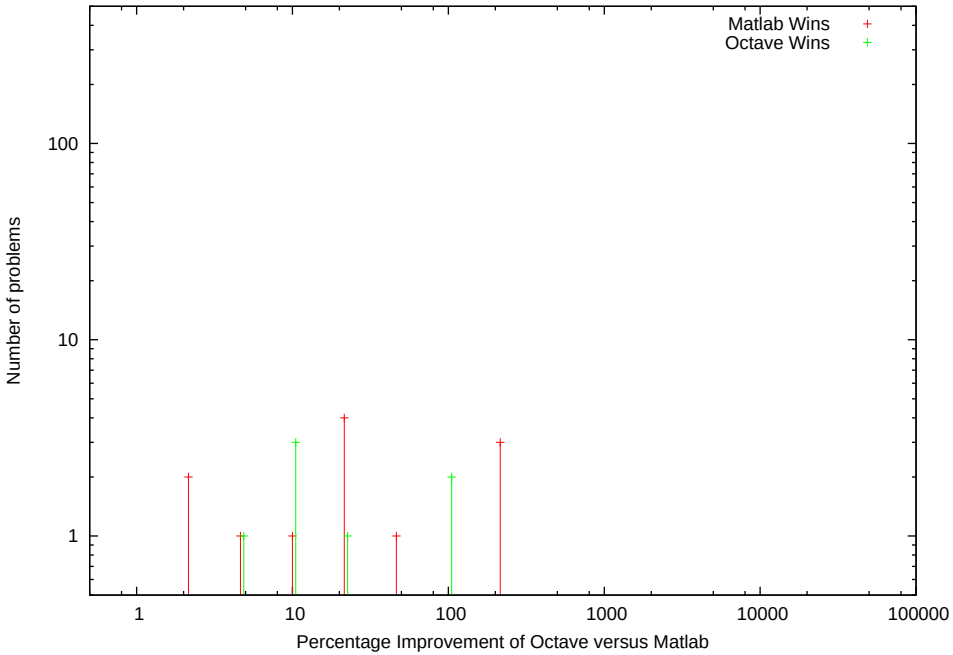



Benchmark of left-division for problems using Cholesky solver

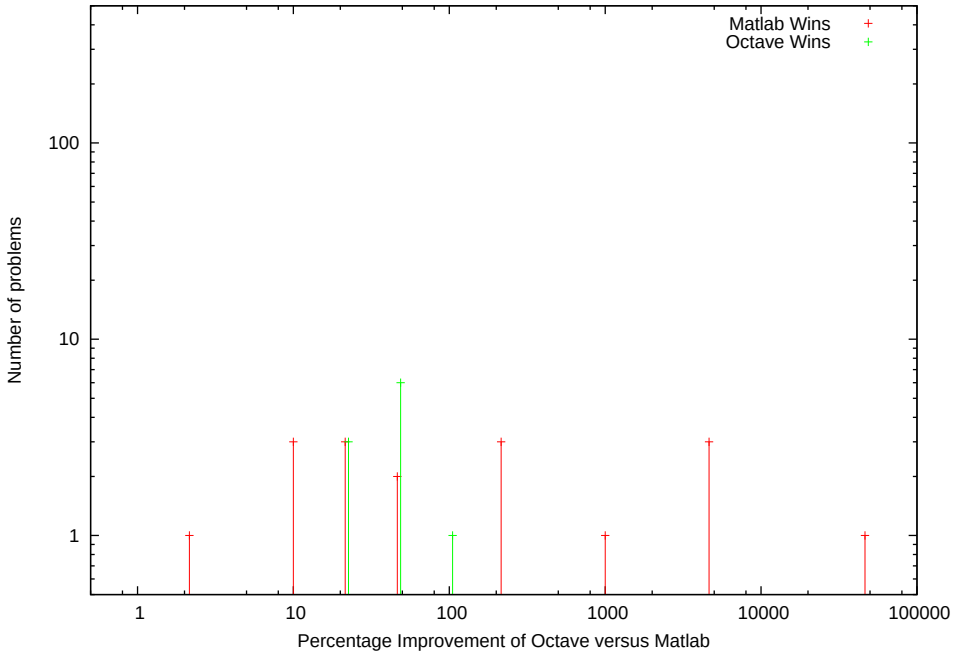



Benchmark of left-division for problems using LU solver

Octave is significantly better performance in the majority of cases. There are some outlying cases for the LU solvers that are probably due to lucky choices of reordering. Better Cholesky performance of Octave is due to CHOLMOD (used without METIS)




Benchmark of left-division for problems using banded solvers




Benchmark of left-division for problems using QR solver

Octave has inferior performance for QR problems due to the use of Householder reflections where Matlab uses Given's rotations. Insufficient data and run times for banded problems for valid comparison.

# Using Sparse Matrices in Oct-files

There are three basic sparse class and their correspondng octave_value classes

| | | |
|---|---|---|
| SparseMatrix | ⇒ | octave_sparse_matrix |
| SparseComplexMatrix | ⇒ | octave_sparse_complex_matrix |
| SparseBoolMatrix | ⇒ | octave_sparse_bool_matrix |

The three classes are based on the class Sparse<T> that is very similar to Array<T>. The sparse classes therefore have very similar usage to the existing Matrix classes.

## ➪ Differences between the Array and Sparse Classes

- ✔ The method *nelem* returns the storage allocated, rather than the product of the dimensions.
- ✔ Two additional methods *nnz* and *numel* that return the actual number of non-zero elements in the matrix and the product of the dimensions. *numel* can overflow and should be avoided.
- ✔ *fortran_vec* is replaced by *cidx*, *ridx* and *data* methods. User must respect CCS format.
- ✔ Extract data from octave_value with *sparse_matrix_value*, etc

## The *elem* and () operator

Care should be taken with the *elem* method and () operator. Used with zero elements on non-const values causes the element to be created, with a consquent memory reallocation.

Don't use the () operator like this

```
SparseMatrix sm;
...

for (int j = 0; j < nc; j++)
  for (int i = 0; i < nr; i++)
    std::cerr << " (" << i << ","
              << j << "): " << sm(i,j)
              << std::endl;
```

instead take a const copy first, like this

```
SparseMatrix sm;
...
const SparseMatrix tmp (sm);
for (int j = 0; j < nc; j++)
  for (int i = 0; i < nr; i++)
    std::cerr << " (" << i << ","
              << j << "): " << tmp(i,j)
              << std::endl;
```

The const copy is low-cost as only the container of the matrix is copied, not the data itself.

# Creating Sparse Matrices in Oct-Files

There are two basic methods of creating sparse matrices in oct-files. If the triplets of row, column and value are known then the sparse matrix can be created directly

```
int nz = 4, nr = 3, nc = 4;
ColumnVector ridx (nz);
ColumnVector cidx (nz);
ColumnVector data (nz);

ridx(0) = 0; ridx(1) = 0; ridx(2) = 1; ridx(3) = 2;
cidx(0) = 0; cidx(1) = 1; cidx(2) = 3; cidx(3) = 3;
data(0) = 1; data(1) = 2; data(2) = 3; data(3) = 4;

SparseMatrix sm(data, ridx, cidx, nr, nc);
```

A copy of the matrix existing in ColumnVector's which requires additional memory.

We can create the sparse matrix directly, by preallocating the sparse matrix then filling it. Points to note

- ✔ The size of the sparse matrix will be automatically enlarged by the *elem* method and () operator if needed
- ✔ Rather than increase the capacity of the sparse matrix element by element, the capacity can be increased with the *change_capacity* method.
- ✔ If additional storage for the sparse matrix is allocated when returning the matrix, it can be removed with the *maybe_compress* method.
- ✔ Rather than use the *elem* method or () operator, it is often better to use the *ridx*, *cidx* and *data* methods to fill in the sparse matrix directly in compressed column format.

A generic example of an efficient means of creating a sparse matrix in an oct-file is

```
octave_value arg;
...
// Assume we know the max no nz
int nz = 6, nr = 3, nc = 4;
SparseMatrix sm (nr, nc, nz);
Matrix m = arg.matrix_value ();

int ii = 0;
sm.cidx (0) = 0;
for (int j = 1; j < nc; j++)
  {
    for (int i = 0; i < nr; i++)
      {
        double tmp = foo (m(i,j));
        if (tmp != 0.)
          {
            if (ii == nz)
              {
                // Add 2 elements
                nz += 2;
                sm.change_capacity
                  (nz);
              }
            sm.data(ii) = tmp;
            sm.ridx(ii) = i;
            ii++;
          }
      }
    sm.cidx(j+1) = ii;
  }
// Don't know a-priori the
// final no of nz.
sm.maybe_compress ();
```

# Miscellaneous Tidbits

- Sparse logical indexing. There is no sparse logical indexing in the idx_vector class. This means

```
a = sprandn (1e6,1e6,1e-6) ; a (a < 1) = 0;
```

will cause a memory overflow

- Sparse indexed values that are strict permutations or permuted sub-blocks are faster than generic indexed assignments.

```
n = 1e4;
p = randperm(n);
a= sprandn(n,n,5/n);
t = cputime(); b = a(p,:); cputime() - t
      ans = 0.011998
p(1) = p(2);
t = cputime(); c = a(p,:); cputime() - t
      ans = 9.7325
```

# Conclusion

- ✔ Full implementation of sparse type for Octave including all indexed and assignment operators, mapper functions, etc
- ✔ Optimized poly-morphic solvers for left- and right-division operators.
- ✔ Many optimizations already made to the code to allow competitive performance compared to Matlab.
- ✔ Presented benchmarks for the addition, multiplication and left-dividion operators that demonstrates that Octave is competitive with Matlab.
- ✔ Presented use of the sparse types in real life examples and in oct-files.
- ✔ Major missing features are the *eigs* and *svds* functions, and iterative solvers.