SPARSEM
COLLECTION OF SPARSE MATRIX MODULES FOR FORTRAN 90 USEFUL IN ANIMAL BREEDING PROBLEMS

Ignacy Misztal, University of Georgia

9/4/97 - 5/25/2007

Introduction

Traditionally, programming in animal breeding is done in 2 stages: in a matrix language and in a regular programming language. Programs in a matrix language such as IML SAS, Matlab, Mathematica or APL are reasonably simple and useful for creating examples but inefficient for large problems. Programs in a regular programming language such as Fortran or C/C++ are much more efficient but could take much longer to write and require substantial training.

Matrix languages are easy to deal with matrices partly because usually only one format is usually supported: dense rectangular. Operations on such matrices are easy to specify and program, but large matrices require large memory and long running time. Also, memory and computations are equal whether matrices are sparse (contain very few nonzero elements) or not. In animal breeding, many matrices are sparse. If that sparsity is taken into account, the memory requirements and computations can decrease dramatically. Unfortunately, there is more than one format for storing sparse matrices, and some computations are fast with one format and but not with another one. Also, the storage formats and operations are considerably more complicated than dense rectangular matrices. A library to handle multiple matrix formats and multiple operations would contain many subroutines, each with a long list of arguments. Such a library would involve considerable learning, and many details associated with the library would create many opportunities for making a mistake.

One matrix package, Matlab, has some forms of sparse-matrix storage and operations included.

Modern programming languages with “object-oriented” features, such as C++ or Fortran 90, have abilities to create classes/modules, where many implementation details on specific data structures can be hidden. A technique called overloading allows single function/subroutine to work with different formats of its arguments. Therefore, the number of details to remember can be drastically reduced. Subsequently, programming can be done much easier and quicker.

SPARSEM is a module for Fortran 90 that enables programming common sparse matrix operations almost as easily as with dense matrices. It supports two dense matrix formats, useful for testing, and two sparse matrix formats. Changing a program from dense- to sparse-matrix format using DENSEM can be as simple as changing one declaration line. SPARSEM incorporates an interface to FSPAK, which enables efficient sparse matrix factorization, solving, sparse inversion and calculation of determinant on matrices much larger than possible with dense matrix structures.

Matrix formats

Four matrix formats are available.

DENSEM - dense square matrix.

DENSE_SYMM -dense symmetric upper-stored.
It has approximately only half memory requirements of the dense square matrix.

SPARSE_HASHM - sparse triple accessed by hash algorithm.
This is a very efficient format for set-up and for iterative-solving of sparse matrices.

SPARSE_IJA - Sparse IJA.
This is a memory-efficient format for sparse matrices used by sparse matrix packages. Format IJA cannot easily be set up directly but can be derived by conversion from the hash format.

For more information on all these formats see Duff et al, George and Liu, or my class notes.

A popular format that is not included here is linked list. That format is reasonably efficient for creating and computing with sparse matrices if the number of nonzero elements per row is not too high and the matrix is not too large. However, the combination of hash plus ija is generally more efficient.

Matrix operations

The following subroutines/functions are supported. All real scalars and vectors are single precision unless indicated otherwise.

Operation	**Description**	**Comments**
call init(x)	Initialize x	Required by standard but usually not necessary because on most systems pointers are initialized automatically
call zerom(x,n)	Allocate storage for x as an n*n matrix and zero it	If x was set before, it is reallocated[1]
call reset(x)	Deallocates storage	
call addm(a,i,j,x)	Add to matrix: x(i,j)=x(i,j)+a	Does not work on SPARSE_IJA
call setm(a,i,j,x)	sets element of matrix: x(i,j)=a	Does not work on SPARSE_IJA

y=getm(i,j,x)	find element of matrix: y=x(i,j)	real(4) function; returns lower-diagonal elements of upper-stored matrix
x=y	Conversion between formats	Conversion from sparse to dense formats may require too much storage
call printm(x)	Prints x as square matrix	print(x,'internal') prints sparse matrices in internal format
call solve_iterm(x,rs,sol)	Solves: x sol=rs iteratively by SOR	
call default_iter (conv,maxround,relax, zerosol)	Changes default iteration parameters	All parameters are optional; default values are: conv(ergence criterion)=1e-10, max round(s)=1000, relax(ation factor)=1.0, zerosol(utions ar beginning of iteration) = .true.
x=block(y,i1,i2,j1,j2)	Selects block from y: x=y(i1:i1,j1:j2)	does not work on dense_symm format; may not work with unsymmetric blocks from symmetric matrices.
q=quadrf(u,x,v)	q=u'Xv	real(8) function; does not work on dense_symm format
tr=trace(x,y)	Self explanatory	real(8) function; x and y must be in same formats; works on densem and sparse_ija formats only;
tr=traceblock(x,y,i1,i2,j1, j2)	tr=trace(xy(i1:i2,j1:j 2))	Works as a block-trace combination; produces correct results when blocks of y are nonsymmetric.

[1] The hash matrix is allocated for a default number of *elements*. If the default is too small, the hash matrix is enlarged automatically. To change the default p elements, use call zerom(x,n,p). One matrix element in hash format takes 12 bytes, and for efficient operation there should be at least 10% more nonzero elements available than used.

Future work: x=solve_iterm(x,rs),x=zerom(n)

All operations assume that the densem type is general while all the other types are upper-stored.

Operations tr, quadf work with both upper- and full-stored matrices but the block operation works literally, i.e., selecting a lower block would return an empty matrix and selecting an upper block would return only an upper-stored matrix. This could be a source of incompatibility between densem and other formats that use the block operation without taking its limitations into consideration. Potential problems can be noticed in examples by printing matrices of interest.

Storage type

Matrices in the hash or ija format are half-stored by default. To change the storage type to full, add the option 'f' to the addm subroutine:

```
call addm(a,i,j,x,'f')
```

The subsequent conversion to the ija format will also be full-stored. For conversion from half-stored hash matrix to full-stored ija, please see a documentation for the GIBBS module.

The printing and other functions/subroutines have been designed for half-stored hash and ija matrices. Results may not be correct with full-stored matrices.

Numerical accuracy

Module KINDS defines precision r4 to be equivalent to real*4, and r8 to be equivalent to r8. Precision rh can be set up to r4 or r8 dependent on whether memory or precision is more important.

Formats DENSEM, DENSE_SYMM, and SPARSE_IJA use precision r8. Format SPARSE_HASHM uses precision rh. Whenever the precision of numbers in SPARSEM functions/subroutines is not specified, it is of type rh. Setting rh to r4 is useful when memory usuage needs to be reduced, e.g., for large BLUP programs. Setting rh to r8 is necessary when numerical accuracy is important, e.g., in variance component programs, and is usually a safer choice.

Diagnostics

Printing of some diagnostic messages depends on the value of an integer variable sparsem_msg. The value of 3 means maximum diagnostic messages while the value of 0 means no diagnostic messages. The default is 2. This variable can be set in any part of the application program using the module SPARSEM.

FSPAK90

FSPAK is a sparse matrix package written in F77 that performs operations on sparse matrices in format SPARSE_IJA. Operations include solving a system of linear equations by factorization, calculating a (log)determinant or finding a sparse inverse of a matrix. A sparse inverse is such a matrix that contains inverse values only for those elements that were nonzero in the original matrix. For sparse matrices, FSPAK is very efficient computationally. FSPAK90 is a F90 interface written to simplify the use of FSPAK.

A complete call to FSPAK90 is:

```
call fspak90(operation,ija,rs,sol,det,msglev,maxmem,rank)
```

where

operation=	"factorize"	- calculate sparse factorization
	"invert"	- calculate sparse inverse
	"solve"	- solve a system of equation
	"reset"	- reset the storage
	"det"	- calculate determinant
	"stat"	- print statistics
	"fact_mult"	- multiplication by Cholesky factor of the reordered

matrix
"inv_fact_mult" - solve the system formed by the Cholesky factor of the reordered matrix

ija = matrix in SPARSE_IJA form
rs = real (r4) or (r8) vector of right hand side,
sol = real (r4) or (r8), identical to precision of rs, vector of solutions
det = real (r8) determinant or log-determinant
msglev= message level from 0 (minimum) to 3 (maximum); default=0
maxmem=maximum memory available in the system; default=infinite
rank=rank of matrix

All the arguments of fspak90 except "operation" and "ija" are optional except when they are needed in a specific "operation". Thus, rs and sol are needed for solving and det for "det" or "ldet".

Examples

To solve:

```
call fspak90('solve',ija,rs,sol)
```

for both rs and sol either in single or double precision; all. preceding steps are done automatically.

To solve using double precision right hand side and solutions:

```
call fspak90('solve',ija,rs8=rs,sol8=sol)
```

To sparse invert:

```
call fspak90('invert',ija)
```

To obtain the determinant d:

```
call fspak90('det',ija,det=d)
```

To obtain the log determinant ld:

```
call fspak90('ldet',ija,det=ld)
```

To obtain rank r with any operation:

```
call fspak90(.....,rank=r)
```

To force new factorization, when the input matrix has changed:

```
call fspak90('factor',ija)
```

To deallocate the internal memory:

```
call fspak90('reset')
```

To limit memory to a maximum od maxmem, e.g., 20,000k, with any operation

```
call fspak90(..............,maxmem=20000)
```

Note that only relevant arguments for each step need to be included in calling FSPAK90. Reordering is performed the first time when FSPAK90 is called. Subsequent factorization except after the option "reset" will reuse the ordering. Subsequent solves will reuse the factorization.

Additionally:

To sample y from N(0,A) where x~N(0,1)

```
call fspak90('fact_mult',A,rs8=x,sol8=y)
```

To sample y from N(0,A^{-1}) where x~N(0,1)

```
call fspak90('inv_fact_mult',A,rs8=x,sol8=y)
```

For details of the last operations, see Appendix B

Additional subroutines and functions

Function

y=mult(A,x)

y=mult(x,A)

implements the matrix by vector multiplication for all matrix formats except dense_symm, and for double precision x and y.

Subroutine

call multmatscal(A,x)

implements A=A*x for all matrix formats except dense_symm, and for double precision x.

Hints on using SPARSEM

Initially all the matrices can be implemented in DENSEM format. After the program works well with an example, convert all data structures for potentially large matrices to sparse formats and verify that same results are obtained.

Compiling

Matrix types and functions subroutine are defined in module sparsem. Subroutine fspak90 is in module sparseop. Program xx.f90 can be compiled as

```
f90 -Maa xx.f90 aa/sparsem.a
```

where aa is the directory containing the modules and the library, and M is the option to include module directory.

Beginning in May, 1999, SPARSEM is part of a programming package that includes BLUPF90, REMLF90, GIBSF90 etc. Compilation for several Unix environments is automated by makefiles. To find details, read Readme and Installation files in the package distributions. To create application with SPARSEM and possibly other modules, create a subdirectory in the main directory of the package, and adapt a makefile from the existing directory, e.g., blup.

Sample Programs

Dense matrix solution program

```
program test_sparse_structures
use sparsem; use kinds
type (densem)::x
integer,parameter ::n=5
```

```
integer :: i,j
real (rh):: rs(n),sol(n),val

call init(x)
call zerom(x,n)

! set up a sample matrix
do i=1,n
   rs(i)=n+1-i
   val=10.0*i/i
   call addm(val,i,i,x)
   do j=i+1,n
      val=10.0*i/j
      call addm(val,i,j,x); call addm(val,j,i,x)
   enddo
enddo

print*,'rs: ',rs
print*,'matrix' ; call printm(y)
call solve_iterm(y,rs,sol)    !solve iteratively
print*,'sol: ',sol
end
```

Triangular dense matrix iterative-solution program

```
......
type (dense_symm)::x
.......
```

(The rest of the program remains identical)

Sparse hash matrix iterative-solution program

```
......
type (sparse_hashm)::x
.......
```

Sparse IJA matrix iterative-solution program

Matrix in ija form cannot be set up directly but can be converted from hash form.

```
......
type (sparse_hashm)::x
type (sparse_ija)::y
...
y=x       !conversion
call reset(x)  ! Optional statement to release storage
print*,'rs: ',rs
print*,'matrix' ; call printm(y)
call solve_iterm(y,rs,sol)
```

```
print*,'sol: ',sol
end
```

Sparse IJA matrix finite-solution and inversion program with FSPAK90

```
...
use sparsem
use sparseop   !fspak90 is in module sparseop
.....
call fspak90('solve',y,rs,sol)
....
!now invert
call fspak90('invert',y)
call printm(y)
end
```

References

George, A. and Liu, J.W.H. (1981) Computer solution of large sparse positive definite systems. Prentice-Hall, Englewood Cliffs, N.J.

Appendix A

Definitions of structures (types)

```
type densem     !traditional dense square matrix
   integer :: n
   real(8) ,pointer::x(:,:)
end type densem


type dense_symm       !upper stored symmetric dense matrix
   integer ::n
   real(8) ,pointer::x(:)
end type dense_symm


type sparse_hashm
    integer:: n,&       ! for compatibility mainly
            nel,&       ! number of elements
            filled,&    ! number of filled elements
            status      ! 1 if ready to hash, 2 if in sorted
                        ! order
     real (rh) , pointer :: x(:,:)
end type sparse_hashm


type sparse_ija
     integer :: n,&      ! number of equations
                 nel     ! number of nonzeroes
     integer, pointer::ia(:),ja(:)      !will be ia(n+1),
ja(m)
     real (8), pointer::a(:)               !will be a(m)
end type
```

Accessing structures

Structures can be accessed within the application program using the "%" symbol. This is useful, e.g., when using Fortran 77 programs. The example below shows how to use a determinant program written in F77.

```
type (densem):: z
integer::i,j
real (rh)::value

call init(z)
call zerom(x,2)

! initialize z
do i=1,2
   do j=1,2
     value=i**j/10.
     call addm(value, i,j,z)
  enddo
enddo

print*, det(z%n,z%x)
end

function det(n,x)
!calculate determinant for a 2x2 matrix
integer n
real (r8):: x(n,n),det
!
det= x(1,1)*x(2,2)/x(1,2)/x(2,1)
end
```

Library

The following files are compiled into the library:

kind.f90	- definitions of precisions
sparse.f90	- type definitions + main subroutines,
sparse2.f	- supporting subroutines (in f77),
fspak.f90	- f90 interface to fspak
fspak.f	- main fspak subroutine (in f77),
fspaksub.f	- supporting fspak subroutines (in f77),
sparssub.f	- low-level subroutines from the book of George and Liu (in f77),
second.f	- timing subroutine specific to each computer (in f77).

Subroutines second() specific to other computers can be found in the FSPAK manual.

Appendix B

Multiplications and solving using factors

Let A be a matrix. Factorization produced by FSPAK is L:

$$A=P'LL'P$$

where P is a reordering matrix chosen to minimize the size of L:

$$PP'=P'P=I$$

Operation "fact_mult" mutiplies the factor by a vector:

$$y=P' L P x$$

Operation " inv_fact_mult" solves the system of equation:

$$P'L' Py=x$$

This is equivalent to:

$$y= P' (L'^{-1}) Px$$

Both operations were programmed by Juan Pablo Sanchez. The operations are useful for generation of large random samples from a multivariate normal distribution. They may be useful in Gibbs sampler algorithms when setting up and factorization of the system of equations in each round are feasible.