GammaLib

User Manual

Version 0.1pre
2 November 2006

Jürgen Knödlseder
Centre d'Etude Spatiale des Rayonnements
knodlseder@cesr.fr
http://www.cesr.fr./~jurgen/index.html

Note to the user

This software has been written to analyse data of the SPI telescope onboard INTEGRAL. Particular care has been taken in making the software user friendly and well documented. If you appreciated this effort, and if this software and User Manual were useful for your scientific work, the author would appreciate a corresponding acknowledgment in your published work.

Contents

1 Introduction

1.1 Motivation

Again another library for scientific calculus? Yes and no. `GammaLib` indeed provides a number of functionnalities that are also found in other libraries, but `GammaLib` goes beyond these libraries in that it provides tools that are rather specific to the analysis of high-energy astronomy data. The idea was to put everything together in one single place, limiting thus the dependencies.

It all started when I was preparing the software for the scientific exploitation of the data from the SPI telescope aboard INTEGRAL. Instead of writing a number of indepedent executables, I decided to write a set of powerful C++ classes that provide the required functionalities, such as data handling, response calculation, image convolution, model fitting, etc. Analysis executables were then simple clients of these C++ classes and basically only provided the user interface. The C++ classes were designed quite general, so that it was obvious to attempt their utilisation also for the analysis of other data. However, it turned out that the SPI classes were still too spezialised for this purpose, and a new, more general library was required to fullfil my goal.

I started the work when I needed sparse matrix capabilities for my SPI model fitting routines. With the ongoing INTEGRAL mission, the datasets became increasinly larger and the number of fit parameters required to model these data increased accordingly. The initial `GammaLib` thus had vector and matrix classes which it provided to the SPI tools library.

1.2 Using `GammaLib`

TBD: How to install, how to write code ..

2 `GammaLib` objects

2.1 Vectors

2.1.1 General

A vector is a one-dimensional array of successive `double` type values. Vectors are handled in `GammaLib` by `GVector` objects. On construction, the dimension of the vector has to be specified. In other words

```
GVector vector;                         // WRONG: constructor needs dimension
```

is not allowed. The minimum dimension of a vector is 1, i.e. there is no such thing like an empty vector:

```
GVector vector(0);                      // WRONG: empty vector not allowed
```

The correct allocation of a vector is done using

```
GVector vector(10);                     // Allocates a vector with 10 elements
```

On allocation, all elements of a vector are set to 0. Vectors may also be allocated by copying from another vector

```
GVector vector(10);                     // Allocates a vector with 10 elements
GVector another = vector;               // Allocates another vector with 10 elements
```

or by using

```
GVector vector = GVector(10);          // Allocates a vector with 10 elements
```

Vector elements are accessed using the `( )` operator:

```
GVector vector(10);                    // Allocates a vector with 10 elements
for (int i = 0; i < 10; ++i)
  vector(i) = (i+1)*10.0;              // Set elements 10, 20, ..., 100
for (int i = 0; i < 10; ++i)
  cout << vector(i) << endl;           // Dump all elements, one by row
```

The content of a vector may also be dumped using

```
cout << vector << endl;                // Dump entire vector
```

which in the above example will put the sequence

```
10 20 30 40 50 60 70 80 90 100
```

on the screen.

2.1.2 Vector arithmetics

Vectors can be very much handled like `double` type variables with the difference that operations are performed on each element of the vector. The complete list of fundamental vector operators is:

```
c = a + b;                             // Vector + Vector addition
c = a + s;                             // Vector + Scalar addition
c = s + b;                             // Scalar + Vector addition
c = a - b;                             // Vector - Vector subtraction
c = a - s;                             // Vector - Scalar subtraction
c = s - b;                             // Scalar - Vector subtraction
s = a * b;                             // Vector * Vector multiplication (dot product)
c = a * s;                             // Vector * Scalar multiplication
c = s * b;                             // Scalar * Vector multiplication
c = a / s;                             // Vector * Scalar division
```

where `a`, `b` and `c` are of type `GVector` and `s` is of type `double`. Note in particular the combination of `GVector` and `double` type objects in addition, subtraction, multiplication and division. In these cases the specified operation is applied to each of the vector elements. It is also obvious that only vector of identicial dimension can occur in vector operations. Dimension errors can be catched by the `try` - `catch` functionality:

```
try {
  GVector a(10);
  GVector b(11);
  GVector c = a + b;                   // WRONG: Vectors have incompatible dimensions
}
catch (GVector::vector_mismatch &e) {
  cout << e.what() << endl;            // Dimension exception is catched here
  throw;
}
```

Further vector operations are

```
c = a;                              // Vector assignment
c = s;                              // Scalar assignment
s = c(index);                       // Vector element access
c += a;                             // c = c + a;
c -= a;                             // c = c - a;
c += s;                             // c = c + s;
c -= s;                             // c = c - s;
c *= s;                             // c = c * s;
c /= s;                             // c = c / s;
c = -a;                             // Vector negation
```

Finally, the comparison operators

```
int equal   = (a == b);             // True if all elements equal
int unequal = (a != b);             // True if at least one elements unequal
```

allow to compare all elements of a vector. If all elements are identical, the `==` operator returns true, otherwise false. If at least one element differs, the `!=` operator returns true, is all elements are identical it returns false.

In addition to the operators, the following mathematical functions can be applied to vectors:

```
acos       atan       exp        sin        tanh
acosh      atanh      fabs       sinh
asin       cos        log        sqrt
asinh      cosh       log10      tan
```

Again, these functions should be understood to be applied element wise. They all take a vector as argument and produce a vector as result. For example

```
c = sin(a);
```

attributes the sine of each element of vector `a` to vector `c`. Additional implemented functions are

```
c = cross(a, b);                    // Vector cross product (for 3d only)
s = norm(a);                        // Vector norm |a|
s = min(a);                         // Minimum element of vector
s = max(a);                         // Maximum element of vector
s = sum(a);                         // Sum of vector elements
```

Finally, a small number of vector methods have been implemented:

```
int n = a.size();                   // Returns dimension of vector
int n = a.non_zeros();              // Returns number of non-zero elements in vector
```

2.2 Matrixes

2.2.1 General

A matrix is a two-dimensional array of `double` type values, arranged in rows and columns. Matrixes are handled in `GammaLib` by `GMatrix` objects and the derived classes `GSymMatrix` and `GSparseMatrix` (see section 2.2.2). On construction, the dimension of the matrix has to be specified:

```
GMatrix matrix(10,20);                  // Allocates 10 rows and 20 columns
```

Similar to vectors, there is no such thing as a matrix without dimensions in `GammaLib`.

2.2.2 Matrix storage classes

In the most general case, the `rows` and `columns` of a matrix are stored in a continuous array of `rows` × `columns` memory locations. This storage type is referred to as a *full matrix*, and is implemented by the class `GMatrix`. Operations on full matrixes are in general relatively fast, but memory requirements may be important to hold all the elements. In general matrixes are stored by `GammaLib` column-wise (or in column-major format). For example, the matrix

```
  1  2  3  4  5
  6  7  8  9 10
 11 12 13 14 15
```

is stored in memory as

```
 |  1  6 11 |  2  7 12 |  3  8 13 |  4  9 14 |  5 10 15 |
```

Many physical or mathematical problems treat with a subclass of matrixes that is symmetric, i.e. for which the element `(row,col)` is identical to the element `(col,row)`. In this case, the duplicated elements need not to be stored. The derived class `GSymMatrix` implements such a storage type. `GSymMatrix` stores the lower-left triangle of the matrix in column-major format. For illustration, the matrix

```
  1  2  3  4
  2  5  6  7
  3  6  8  9
  4  7  9 10
```

is stored in memory as

```
 |  1  2  3  4 |  5  6  7 |  8  9 | 10 |
```

This divides the storage requirements to hold the matrix elements by almost a factor of two.

Finally, quite often one has to deal with matrixes that contain a large number of zeros. Such matrixes are called *sparse matrixes*. If only the non-zero elements of a sparse matrix are stored the memory requirements are considerably reduced. This goes however at the expense of matrix element access, which has become now more complex. In particular, filling efficiently a sparse matrix is a non-trivial problem (see section 2.2.7). Sparse matrix storage is implemented in `GammaLib` by the derived class `GSparseMatrix`. A `GSparseMatrix` object contains three one-dimensional arrays to store the matrix elements: a `double` type array that contains in continuous column-major order all non-zero elements, an `int` type array that contains for each non-zero element the row number of its location, and an `int` type array that contains the storage location of the first non-zero element for each matrix column. To illustrate this storage format, the matrix

```
  1  0  0  7
  2  5  0  0
  3  0  6  0
  4  0  0  8
```

is stored in memory as

```
| 1  2  3  4 |  5 |  6 |  7  8 |  Matrix elements
| 0  1  2  3 |  1 |  2 |  0  3 |  Row indices for all elements
| 0          |  4 |  5 |  6    |  Storage location of first element of each column
```

This example is of course not very economic, since the total number of Bytes used to store the matrix is $8 \times 8 + (8 + 4) \times 4 = 112$ Bytes, while a full 4×4 matrix is stored in $(4 \times 4) \times 8 = 128$ Bytes (recall: a `double` type values takes 8 Bytes, an `int` type value takes 4 Bytes). For realistic large systems, however, the gain in memory space can be dramatical.

The usage of the `GMatrix`, `GSymMatrix` and `GSparseMatrix` classes is analoguous in that they implement basically all functions and methods in an identical way. So from the semantics the user has not to worry about the storage class. However, matrix element access speeds are not identical for all storage types, and if performance is an issue (as it certainly always will be), the user has to consider matrix access more carefully (see section 2.2.7).

Matrix allocation is performed using the constructors:

```
GMatrix       A(10,20);                // Full 10 x 20 matrix
GSymMatrix    B(10,10);                // Symmetric 10 x 10 matrix
GSparseMatrix C(1000,10000);           // Sparse 1000 x 10000 matrix

GMatrix       A(0,0);                  // WRONG: empty matrix not allowed
GSymMatrix    B(20,22);                // WRONG: symmetric matrix requested
```

In the constructor, the first argument specifies the number of rows, the second the number of columns: `A(row,column)`. A symmetric matrix needs of course an equal number of rows and columns. And an empty matrix is not allowed. All matrix elements are initialised to 0 by the matrix allocation.

Matrix elements are accessed by the `A(row,col)` function, where `row` and `col` start from 0 for the first row or column and run up to the number of rows or columns minus 1:

```
for (int row = 0; row < n_rows; ++row) {
  for (int col = 0; col < n_cols; ++col)
    A(row,col) = (row+col)/2.0;        // Set value of matrix element
}
...
double sum2 = 0.0;
for (int row = 0; row < n_rows; ++row) {
  for (int col = 0; col < n_cols; ++col)
    sum2 *= A(row,col) * A(row,col);   // Get value of matrix element
}
```

The content of a matrix can be visualised using

```
cout << A << endl;                     // Dump matrix
```

2.2.3 Matrix arithmetics

The following description of matrix arithmetics applies to all storage classes (see section 2.2.2). The following matrix operators have been implemented in `GammaLib`:

```
C = A + B;                             // Matrix Matrix addition
C = A - B;                             // Matrix Matrix subtraction
C = A * B;                             // Matrix Matrix multiplication
```

```
C = A * v;                              // Matrix Vector multiplication
C = A * s;                              // Matrix Scalar multiplication
C = s * A;                              // Scalar Matrix multiplication
C = A / s;                              // Matrix Scalar division
C = -A;                                 // Negation
A += B;                                 // Matrix inplace addition
A -= B;                                 // Matrix inplace subtraction
A *= B;                                 // Matrix inplace multiplications
A *= s;                                 // Matrix inplace scalar multiplication
A /= s;                                 // Matrix inplace scalar division
```

The comparison operators

```
int equal   = (A == B);                 // True if all elements equal
int unequal = (A != B);                 // True if at least one elements unequal
```

allow to compare all elements of a matrix. If all elements are identical, the `==` operator returns true, otherwise false. If at least one element differs, the `!=` operator returns true, is all elements are identical it returns false.

2.2.4 Matrix methods and functions

A number of methods has been implemented to manipulate matrixes. The method

```
A.clear();                              // Set all elements to 0
```

sets all elements to 0. The methods

```
int rows = A.rows();                    // Returns number of rows in matrix
int cols = A.cols();                    // Returns number of columns in matrix
```

provide access to the matrix dimensions, the methods

```
double sum = A.sum();                   // Sum of all elements in matrix
double min = A.min();                   // Returns minimum element of matrix
double max = A.max();                   // Returns maximum element of matrix
```

inform about some matrix properties. The methods

```
GVector v_row    = A.extract_row(row); // Puts row in vector
GVector v_column = A.extract_col(col); // Puts column in vector
```

extract entire rows and columns from a matrix. Extraction of lower or upper triangle parts of a matrix into another is performed using

```
B = A.extract_lower_triangle();         // B holds lower triangle
B = A.extract_upper_triangle();         // B holds upper triangle
```

`B` is of the same storage class as `A`, except for the case that `A` is a `GSymMatrix` object. In this case, `B` will be a full matrix of type `GMatrix`.

The methods

```
A.insert_col(v_col,col);               // Puts vector in column
A.add_col(v_col,col);                  // Add vector to column
```

inserts or adds the elements of a vector into a matrix column. Note that no row insertion routines have been implemented (so far) since they would be less efficient (recall that all matrix types are stored in column-major format).

Conversion from one storage type to another is performed using

```
B = A.convert_to_full();               // Converts A -> GMatrix
B = A.convert_to_sym();                // Converts A -> GSymMatrix
B = A.convert_to_sparse();             // Converts A -> GSparseMatrix
```

Note that `convert_to_sym()` can only be applied to a matrix that is indeed symmetric.

The transpose of a matrix can be obtained by using one of

```
A.transpose();                         // Transpose method
B = transpose(A);                      // Transpose function
```

The absolute value of a matrix is provided by

```
B = fabs(A);                           // B = |A|
```

2.2.5 Matrix factorisations

A general tool of numeric matrix calculs is factorisation.

Solve linear equation `Ax = b`. Inverse a matrix (by solving successively `Ax = e`, where `e` are the unit vectors for all dimensions).

For symmetric and positive definite matrices the most efficient factorisation is the Cholesky decomposition. The following code fragment illustrates the usage:

```
GMatrix A(n_rows, n_cols};
GVector x(n_rows};
GVector b(n_rows};
...
A.cholesky_decompose();                // Perform Cholesky factorisation
x = A.cholesky_solver(b);              // Solve Ax=b for x
```

Note that once the function `A.cholesky_decompose()` has been applied, the original matrix content has been replaced by its Cholesky decomposition. Since the Cholesky decomposition can be performed inplace (i.e. without the allocation of additional memory to hold the result), the matrix replacement is most memory economic. In case that the original matrix should be kept, one may either copy it before into another `GMatrix` object or use the function

```
GMatrix L = cholesky_decompose(A);
x = L.cholesky_solver(b);
```

A symmetric and positif definite matrix can be inverted using the Cholesky decomposition using

```
A.cholesky_invert();                   // Inverse matrix using Cholesky fact.
```

Alternatively, the function

```
GMatrix A_inv = cholesky_invert(A);
```

may be used.

The Cholesky decomposition, solver and inversion routines may also be applied to matrices that contain rows or columns that are filled by zeros. In this case the functions provide the option to (logically) compress the matrices by skipping the zero rows and columns during the calculation.

For compressed matrix Cholesky factorisation, only the non-zero rows and columns have to be symmetric and positive definite. In particular, the full matrix may even be non-symmetric.

2.2.6 Sparse matrixes

The only exception that does not work is

```
GSparseMatrix A(10,10);
A(0,0) = A(1,1) = A(2,2) = 1.0;        // WRONG: Cannot assign multiple at once
```

In this case the value `1.0` is only assigned to the last element, i.e. `A(2,2)`, the other elements will remain `0`. This feature has to do with the way how the compiler translates the code and how `GammaLib` implements sparse matrix filling. `GSparseMatrix` provides a pointer for a new element to be filled. Since there is only one such *fill pointer*, only one element can be filled at once in a statement. **So it is strongly advised to avoid multiple matrix element assignment in a single row.** Better write the above code like

```
GSparseMatrix A;
A(0,0) = 1.0;
A(1,1) = 1.0;
A(2,2) = 1.0;
```

This way, element assignment works fine.

Inverting a sparse matrix produces in general a full matrix, so the inversion function should be used with caution. Note that a full matrix that is stored in sparse format takes roughly twice the memory than a normal `GMatrix` object. If nevertheless the inverse of a sparse matrix should be examined, it is recommended to perform the analysis column-wise:

```
GSparseMatrix A(rows,cols);            // Allocate sparse matrix
GVector       unit(rows);              // Allocate vector
...
A.cholesky_decompose();                // Factorise matrix

// Column-wise solving the matrix equation
for (int col = 0; col < cols; ++col) {
  unit(col) = 1.0;                     // Set unit vector
  GVector x = cholesky_solver(unit);   // Get column x of inverse
  ...
  unit(col) = 0.0;                     // Clear unit vector for next round
}
```

2.2.7 Filling sparse matrixes

The filling of a sparse matrix is a tricky issue since the storage of the elements depends on their distribution in the matrix. If one would know beforehand this distribution, sparse matrix filling would be easy and fast.

In general, however, the distribution is not known a priori, and matrix filling may become a quite time consuming task.

If a matrix has to be filled element by element, the access through the operator

```
m(row,col) = value;
```

may be mandatory. In principle, if a new element is inserted into a matrix a new memory cell has to be allocated for this element, and other elements may be moved. Memory allocation is quite time consuming, and to reduce the overhead, `GSparseMatrix` can be configured to allocate memory in bunches. By default, each time more matrix memory is needed, `GSparseMatrix` allocates 512 cells at once (or 6144 Bytes since each element requires a `double` and a `int` storage location). If this amount of memory is not adequat one may change this value by using

```
m.set_mem_block(size);
```

where `size` is the number of matrix elements that should be allocated at once (corresponding to a total memory of $12 \times$ `size` Bytes).

Alternatively, a matrix may be filled column-wise using the functions

```
m.insert_col(vector,col);            // Insert a vector in column
m.add_col(vector,col);               // Add content of a vector to column
```

While `insert_col` sets the values of column `col` (deleting thus any previously existing entries), `add_col` adds the content of `vector` to all elements of column `col`. Using these functions is considerably more rapid than filling individual values.

Still, if the matrix is big (i.e. severeal thousands of rows and columns), filling individual columns may still be slow. To speed-up dynamical matrix filling, an internal fill-stack has been implemented in `GSparseMatrix`. Instead of inserting values column-by-column, the columns are stored in a stack and filled into the matrix once the stack is full. This reduces the number of dynamic memory allocations to let the matrix grow as it is built. By default, the internal stack is disabled. The stack can be enabled and used as follows:

```
m.stack_init(size, entries);         // Initialise stack
...
m.add_col(vector,col);               // Add columns
...
m.stack_destroy();                   // Flush and destory stack
```

The method `stack_init` initialises a stack with a number of `size` elements and a maximum of `entries` columns. The larger the values `size` and `entries` are chosen, the more efficient the stack works. The total amount of memory of the stack can be estimated as $12 \times$ `size` $+ 8 \times$ `entries` Bytes. If a rough estimate of the total number of non-zero elements is available it is recommended to set `size` to this value. As a rule of thumb, `size` should be at least of the dimension of either the number of rows or the number of columns of the matrix (take the maximum of both). `entries` is best set to the number of columns of the matrix. If memory limits are an issue smaller values may be set, but if the values are too small, the speed increase may become negligible (or stack-filling may even become slower than normal filling).

Stack-filling only works with the method `add_col`. Note also that filling sub-sequently the same column leads to stack flushing. In the code

```
for (int col = 0; col < 100; ++col) {
  column      = 0.0;                 // Reset column
  column(col) = col;                 // Set column
```

```
   m.add_col(column,col);               // Add column
 }
```

stack flushing occurs in each loop, and consequently, the stack-filling approach will be not very efficient (it would probably be even slover than normal filling). If successive operations are to be performed on columns, it is better to perform them before adding. The code

```
 column = 0.0;                          // Reset column
 for (int col = 0; col < 100; ++col)
   column(col) = col;                   // Set column
 m.add_col(column,col);                 // Add column
```

would be far more efficient.

A avoidable overhead occurs for the case that the column to be added is sparse. The vector may contain many zeros, and `GSparseMatrix` has to filter them out. If the sparsity of the column is known, this overhead can be avoided by directly passing a compressed array to `add_col`:

```
 int     number = 5;                    // 5 elements in array
 double* values = new double[number];   // Allocate values
 int*    rows   = new int[number];      // Allocate row index
 ...
 m.stack_init(size, entries);           // Initialise stack
 ...
 for (int i = 0; i < number; ++i) {     // Initialise array
   values[i] = ...                      // ... set values
   rows[i]   = ...                      // ... set row indices
 }
 ...
 m.add_col(values,rows,number,col);     // Add array
 ...
 m.stack_destroy();                     // Flush and destory stack
 ...
 delete [] values;                      // Free array
 delete [] rows;
```

The method `add_col` calls the method `stack_push_column` for stack filling. `add_col` is more general than `stack_push_column` in that it decides which of stack- or direct filling is more adequate. In particular, `stack_push_column` may refuse pushing a column onto the stack if there is not enough space. In that case, `stack_push_column` returns a non-zero value that corresponds to the number of non-zero elements in the vector that should be added. However, it is recommended to not use `stack_push_column` and call instead `add_col`.

The method `stack_destroy` is used to flush and destroy the stack. After this call the stack memory is liberated. If the stack should be flushed without destroying it, the method `stack_flush` may be used:

```
 m.stack_init(size, entries);           // Initialise stack
 ...
 m.add_col(vector,col);                 // Add columns
 ...
 m.stack_flush();                       // Simply flush stack
```

Once flushed, the stack can be filled anew.

Note that stack flushing is not automatic! This means, if one trys to use a matrix for calculs without flushing, the calculs may be wrong. **If a stack is used for filling, always flush the stack before using the matrix.**

3 Code reference

3.1 `GVector`

3.2 `GMatrix`