Gmm++ user documentation

Release 5.3

Yves Renard

December 21, 2018

Contents


Gmm++

CHAPTER 1

Introduction

Gmm++ provides some basic types of sparse and dense matrices and vectors. It provides some generic operations on them (copy, addition, multiplication, sub-vector and sub-matrices, solvers ...). The syntax of *Gmm++* is very close to MTL and ITL (see http://www.osl.iu.edu/research/mtl/). Especially, the code for most of the iterative solvers has been imported from ITL. The performance of *Gmm++* is also close to the one of MTL, sometimes better. The difference is that basically *Gmm++* has been written to be able to interface other libraries and gives an access to sub matrices and sub vectors in all cases. Also some optimizations has been made for matrix-matrix multiplication, usage of reference has been somewhat cleared, const qualifier usage is clarified, and we hope it is somewhat easier to use.

CHAPTER 2

Installation

Since we use standard GNU tools, the installation of the *Gmm++* library is somewhat standard.

Note that if you use *GetFEM++*, you do not have to install *Gmm++* since *GetFEM++* is provided with its own version of *Gmm++*.

Moreover, as *Gmm++* is a template library, no compilation is needed to install it. If the *Gmm++* archive is on your current directory you can unpack it and enter inside the directory of the distribution with the commands:

```
gunzip -c gmm-x.xx.tar.gz | tar xvf -
cd  gmm-x.xx
```

Then you you have to run the configure script just typing:

```
./configure
```

or if you want to set the prefix directory where to install the library you can use the `--prefix` option (the default prefix directory is `/usr/local`):

```
./configure --prefix=\textit{dest_dir}
```

then start the installation with:

```
make install
```

You can also check if your configuration is correct with:

```
make check
```

which compiles random tests.

If you want to use a different compiler than the one chosen automatically by the `./configure` script, just specify its name on the command line:

```
./configure CXX=mycompiler
```

More specific instructions can be found in the `README*` files of the distribution.

Now, to use *Gmm++* in you programs, the simpler manner is to include the file `gmm/gmm.h` which includes all the template library. If the compilation time is too important, the minimum to be included is contained is the file `gmm/gmm\_kernel.h` (vectors and matrix types, blas, sub vector and sub matrices).

DO NOT FORGET to catch errors messages. See the corresponding section.

CHAPTER 3

Matrix and Vector type provided by *Gmm++*

The convention is that any vector or matrix type (except if it is a reference) can be instantiated with the constructors:

```
Vector V(n);        // build a vector of size n.
Matrix M(n, m);     // build a matrix with n rows and m columns.
```

No other constructor is used inside *Gmm++* and you should not use any other if you want your code to be compatible with any matrix and vector type.

It is assumed that each vector type interfaced with *Gmm++* allows to access to a component with the following syntax:

```
a = V[i];    // read the ith component of V.
V[i] = b;    // write the ith component of V.
```

The write access being available if the vector is not a constant reference. For a matrix:

```
a = M(i, j); // read the component at row i and column j of M.
M(i, j) = b; //  write the component at row i and column j of M.
```

Again the write access is available if the matrix is not a const reference. Generally, especially for sparse matrices, this access is not very efficient. Linear algebra procedures access to the components of the vectors and matrices via iterators. (see section *Deeper inside Gmm++*)

It is also not recommended (at all) to use the original copy operator for vectors or matrices. Generally, it will not do the appropriate job. instead, you have to use the method:

```
gmm::copy(V, W);  //  W <-- V
```

which works for all correctly interfaced matrix and vector type, even if `V` is not of the same type as `W` (`V` could be sparse and `W` dense for instance).

in *Gmm++*, a vector is not a (n by 1) matrix, it is a one dimensional object. If you need to use a vector as a (n by 1) column matrix or a (1 by n) row matrix, you can do it with:

```
gmm::row_vector(V) // gives a reference on V considered as
                   // a (1 by n) row matrix
gmm::col_vector(V) // gives a reference on V considered as
                   // a (n by 1) col matrix
```

In the following, the template parameter `T` will represent a scalar type like `double` or `std::complex<double>`.

dense vectors

Gmm++ interfaces `std::vector<T>` so you can use it as your basic dense vector type. If you need to interface another type of dense vector you can see in `gmm/gmm_interface.h` some examples.

sparse vectors

Gmm++ provides two types of sparse vectors: `gmm::wsvector<T>` and `gmm::rsvector<T>`. `gmm::wsvector<T>` is optimized for write operations and `gmm::rsvector<T>` is optimized for read operations. It should be appropriate to use `gmm::wsvector<T>` for assembling procedures and then to copy the vector in a `gmm::rsvector<T>` for the solvers. Those two vector types can be used to create row major or column major matrices (see section *generic row and column matrices*).

skyline vectors

The type `gmm::slvector<T>` defines a skyline vector, in the sense that only an interval of this vector is stored. With this type of vector you can build skyline matrices as `gmm::row_matrix< gmm::slvector<T> >` (see next section *generic row and column matrices*).

generic row and column matrices

Gmm++ provides the two following types of matrices: `gmm::row_matrix<VECT>` and `gmm::col_matrix<VECT>` where `VECT` should be a valid (i.e. interfaced) vector type. Those two type of matrices store an array of `VECT` so the memory is not contiguous. Initializations are:

```
gmm::row_matrix< std::vector<double> > M1(10, 10);  // dense row matrix
gmm::col_matrix< gmm::wsvector<double> > M2(5, 20); // sparse column matrix
```

Of course `gmm::row_matrix<VECT>` is a row matrix and it is impossible to access to a particular column of this matrix.

`gmm::mat_nrows(M)` gives the number of rows of a matrix and `gmm::mat_ncols(M)` the number of columns.

dense matrices

It is recommended to use the type:

```
gmm::dense_matrix<T>
```

to represent a dense matrix type because it is compatible with the Fortran format (column major) and some operations are interfaced with blas and Lapack (see section *Interface with BLAS, LAPACK or ATLAS*). It is considered as a column and row matrix (column preferred) which means that you can access both to the columns and rows.

However, matrix types as `gmm::row_matrix< std::vector<double> >` or `gmm::col_matrix< std::vector<double> >` represent also some dense matrices.

sparse matrices

Similarly, `gmm::row_matrix< gmm::wsvector<double> >` or `gmm::col_matrix< gmm::rsvector<double> >` represents some sparse matrices, but *Gmm++* provides also two types of classical sparse matrix types:

```
gmm::csr_matrix<T>
gmm::csc_matrix<T>
```

The type `gmm::csr_matrix<T>` represents a compressed sparse row matrix and `gmm::csc_matrix<T>` a compressed sparse column matrix. The particularity of these two types of matrices is to be read only, in the sense that it is not possible to access at a particular component to write on it (the operation is too expansive). The only write operation permitted is `gmm::copy`. The right way to use these matrices is first to execute the write operations on another type of matrix like `gmm::row_matrix< gmm::wsvector<double> >` then to do a copy:

```
gmm::row_matrix< gmm::wsvector<double> > M1;
...
assembly operation on M1
...
M1(i,j) = b;
...
gmm::csc_matrix<double> M2;
gmm::clean(M1, 1E-12);
gmm::copy(M1, M2);
```

Matrices `gmm::csr_matrix<T>` and `gmm::csc_matrix<T>` have the advantage to have a standard format (interfacable with Fortran code) and to have a compact format (contiguous in memory). To be able to be compatible with Fortran programs a second template parameter exists on these type, you can declare:

```
gmm::csc_matrix<double, 1> M1;
gmm::csr_matrix<double, 1> M2;
```

The `1` means that a shift will be done on all the indices.

CHAPTER 4

Input and output with Harwell-Boeing and Matrix Market formats

Including the file `gmm/gmm_inoutput.h` you will be able to load and save matrices with Harwell-Boeing and Matrix Market formats. Concerning the Harwell-Boeing format, only the type `gmm::csc_matrix<double>` and `gmm::csc_matrix<std::complex<double> >` has been interfaced, so you can execute:

```
Harwell_Boeing_save("filename", A) // save the matrix A .
Harwell_Boeing_load("filename", A) // load the matrix A.
```

If `A` is not a `gmm::csc_matrix<double>` or a `gmm::csc_matrix<std::complex<double> >` a copy is made.

Concerning the Matrix Market format, it is possible to save a `gmm::csc_matrix<double>` or a `gmm::csc_matrix<std::complex<double> >` and to load a `gmm::row_matrix<VECT>` or a `gmm::col_matrix<VECT>`:

```
MatrixMarket_save("filename", A) // save a csc_matrix.
MatrixMarket_load("filename", A) // load a row_matrix or a col_matrix
```

CHAPTER 5

sub-vectors and sub-matrices

It is possible to obtain any sub-vector or sub-matrix of a fully interfaced object. There are four types of sub indices:

```
gmm::sub_interval(first, length);
```

represents an interval whose first index is `first` and length is `length` (for instance `gmm::sub_interval(10, 3);` represents the indices `{10, 11, 12}`).

```
gmm::sub_slice(first, length, step);
```

represents also an interval in which one index over `step` is taken. (for instance `gmm::sub_slice(10, 3, 2);` represents the indices `{10, 12, 14}`)

```
gmm::sub_index(CONT c);
```

represents the sub-index which is the collection of index contained in the container `c`. For instance:

```
std::vector<size_t> c(3);
c[0] = 1; c[1] = 3; c[2] = 16;
gmm::sub_index(c);
```

represents the indices `{1, 3, 16}`.

VERY IMPORTANT : the container `c` has to be *sorted* from the smaller index to the greater one (i.e. with increasing order) and no repetition is allowed.

For unsorted index such as permutation, a special type of sub index is defined:

```
gmm::unsorted_sub_index(CONT c);
```

Some algorithms are a little bit slower with unsorted sub indices.

Now `gmm::sub_vector(V, subi)` gives a reference to a sub-vector:

```
gmm::vsvector<double> V(10);
V[5] = 3.0;
std::cout << gmm::sub_vector(V, gmm::sub_interval(2, 3)) << std::endl;
```

prints to the standard output `V[2], V[3]` and `V[4]`.

`gmm::sub_matrix(V, subi1, subi2)` gives a reference to a sub-matrix. For instance:

```
gmm::col_matrix< gmm::wsvector<double> > M(5, 20);
M(3, 2) = 5.0;
```

```
std::cout << gmm::sub_matrix(M, gmm::sub_interval(2, 3), gmm::sub_interval(2, 3))
          << std::endl;
```

prints to the output a sub-matrix. If the two sub-indices are equal, it is possible to omit the second. For instance:

```
gmm::col_matrix< gmm::wsvector<double> > M(5, 20);
M(3, 2) = 5.0;
std::cout << gmm::sub_matrix(V, gmm::sub_interval(2, 3)) << std::endl;
```

The reference on sub_matrix is writable if the corresponding matrix is writable (so you can copy on a sub_matrix, add sub-matrices ...).

row and column of a matrix

`gmm::mat_row(M, i)` gives a (possibly writable) reference to the row `i` of matrix `M`, and `gmm::mat_col(M, i)` gives a (possibly writable) reference to the column `i`. It is not possible to access to the rows if `M` is a column matrix and to the columns if it is a row matrix. It is possible to use `gmm::mat_const_row(M, i)` and `gmm::mat_const_col(M, i)` to have constant references.

CHAPTER 6

Miscellaneous methods

```
gmm::vect_size(V); // gives the size of the vector V.

gmm::resize(V, n); // Change the size of the vector V.
                   // Preserve the min(n, vect_size(V)) first components.
                   // Do not work for references.
gmm::resize(M, m, n); // Change the dimensions of matrix M.
                      // Preserve the
                      // min(m, mat_nrows(M)) x min(n, mat_ncols(M))
                      // first components. Do not work for references.
gmm::reshape(M, m, n);  // returns the m-by-n matrix whose elements
                        // are taken columnwise from M.
                        // An error results if M does not have m*m
                        // elements. Works only with dense_matrix<T> for
                        // the moment.

gmm::nnz(V); // gives the number of stored components of the vector V.
gmm::nnz(M); // gives the total number of stored components of the matrix M.

gmm::mat_nrows(M) // gives the number of rows of a matrix M.
gmm::mat_ncols(M) // gives the number of columns of a matrix M.

gmm::write(o, V); // print the vector V to the output stream o.
gmm::write(o, M); // print the matrix M to the output stream o.
```

Most of the time it is more convenient to use:

```
std::cout << gmm::vref(V) << std::endl;
std::cout << M << std::endl;

gmm::clear(V); // set to zero all the components of the vector V;
gmm::clear(M); // set to zero all the components of the matrix M;

gmm::clean(V, 1E-10); // set to zero all the components of the vector V
                      // whose modulus is less or equal to 1E-10
gmm::clean(M, 1E-10); // idem for a matrix M.

gmm::fill_random(V); // fill a dense vector V with random number
                     //  between -1 and 1
gmm::fill_random(V, cfill); // fill a dense or sparse vector with random
```

```
                    // numbers. cfill should be between 0.0 qnd 1.0 and
                    // represent the ratio of filled components.
gmm::fill_random(M); // fill a dense matrix M with random number
gmm::fill_random(M, cfill); // fill a dense or sparse matrix M with random
                    // numbers.
```

CHAPTER 7

Basic linear algebra operations

The same choice has been made as in MTL to provide basic operations as functions not as operators. The advantages are that it is clearer to see where are the linear algebra operations in the program and the programming of optimized basic linear operations is greatly simplified.

scale and scaled

`gmm::scale` is used to multiply a vector or a matrix with a scalar factor:

```
gmm::scale(V, 10.0);  // V * 10.0 ---> V
```

If one not needs to multiply the vector but wants to use the multiplied vector in an expression `gmm::scaled` gives a reference to a multiplied vector. This is only a reference, no operation is made until this reference is used somewhere. For instance:

```
std::cout << gmm::scaled(V, 10.0) << std::endl;
```

print to the standard output the vector `V` multiplied by `10.0` without changing `V`.

transposition

`gmm::transposed(M)` gives a possibily modifiable reference on the transposed matrix of `M`.

imaginary and real part

For a complex matrix `M` or a complex vector `V`, `gmm::real_part(M)`, `gmm::real_part(V)`, `gmm::imag_part(M)` or `gmm::imag_part(V)` give a possibily modifiable reference on the real or imaginary part of the matrix or vector (for instance `gmm::clear(gmm::imag_part(M))` will set to zero the imaginary part of a matrix `M`). These functions cannot be applied to real matrices or vectors.

conjugate

For a matrix `M` or a vector `V`, `gmm::conjugated(M)` and `gmm::conjugated(V)` give a constant reference on the conjugated vector or matrix. Of course, for a real vectors this has no effect (and no cost at all). Note : `gmm::conjugated(M)` transposes the matrix `M` so that this

is the hermitian conjugate of `M`. If you need only the conjugate of each component you have to use both transposition and conjugate with `gmm::conjugated(gmm::transposed(M))` or equivalently `gmm::transposed(gmm::conjugated(M))`.

add

addition of vectors or matrices. It is alway possible to mix different type of vector or matrices in the operations. The following operations are valid:

```
std::vector<double> V1(10);
gmm::wsvector<double> V2(10);
gmm::clear(V1);
...
gmm::add(V1, V2); // V1 + V2 --> V2
cout << gmm::vref(V2);

gmm::add(V1, gmm::scaled(V2, -2.0), V2); // V1 - 2.0 * V2 --> V2
cout << gmm::vref(V2);

gmm::row_matrix< std::vector<double> > M1(10, 10);
gmm::col_matrix< gmm::wsvector<double> > M2(1000, 1000);

// M1 + (sub matrix of M2) ---> (sub matrix of M2)
gmm::add(M1, gmm::sub_matrix(M2, gmm::sub_interval(4,10)));
```

IMPORTANT : all the vectors have to have the same size, no resize will be automatically done. If a vector has not the good size, an error will be thrown.

mult

Matrix-vector or matrix-matrix multiplication. Again, all the matrices and vectors have to have the good size. The following operations are valid:

```
std::vector<double> V1(10);
gmm::wsvector<double> V2(10);
...
gmm::row_matrix< std::vector<double> > M1(10, 10);
...

gmm::mult(M1, V2, V1);  // M1 * V2 --> V1

gmm::mult(M1, V2, V2, V1);  // M1 * V2 + V2 --> V1

gmm::mult_add(M1, V2, V1);  // M1 * V2 + V1 --> V1

gmm::mult(M1, gmm::scaled(V2, -1.0), V2, V1);  // M1 * (-V2) + V2 --> V1

gmm::col_matrix< gmm::wsvector<double> > M2(10, 10);
gmm::col_matrix< gmm::vsvector<double> > M3(10, 10);
...

gmm::mult(M1, M2, M3); // M1 * M2 ---> M3

gmm::mult(gmm::sub_matrix(M1, sub_interval(0, 3)),
```

```
          gmm::sub_matrix(M2, sub_interval(4, 3)),
          gmm::sub_matrix(M3, sub_interval(2, 3)));
```

norms

```
gmm::vect_norm1(V)  // sum of the modulus of the components of vector V.
gmm::vect_norm2(V)  // Euclidean norm of vector V.
gmm::vect_dist2(V1, V2)  // Euclidean distance between V1 and V2.
gmm::vect_norminf(V)    // infinity norm of vector V.
gmm::mat_euclidean_norm(M) // Euclidean norm of matrix ``M``
                           // (called also Frobenius norm).
gmm::mat_maxnorm(M) // Max norm (defined as max(|m_ij|; i,j ))
gmm::mat_norm1(M)   // max(sum(|m_ij|, i), j)
gmm::mat_norminf(M) // max(sum(|m_ij|, j), i)
```

trace

`gmm::mat_trace(M)` gives the trace of matrix `M`.

scalar product

for vectors only, `gmm::vect_sp(V1, V2)` gives the scalar product between `V1` and `V2`. For complex vectors, this do not conjugate `V1`, you can use `gmm::vect_sp(V1, gmm::conjugated(V2))` or `gmm::vect_hp(V1, V2)` which is equivalent.

CHAPTER 8

Solving triangular systems

If M is a triangular matrix (upper or lower) and X a vector containing the right hand side, the following procedures solve the system $x \leftarrow M^{-1}x$. The vector X contains the result:

```
gmm::upper_tri_solve(M, X, false) // Solving an upper triangular system
gmm::upper_tri_solve(M, X, true)  // Solving an upper triangular system
                                  // assuming there is 1 on the diagonal
gmm::lower_tri_solve(M, X, false) // Solving a lower triangular system
gmm::lower_tri_solve(M, X, true)  // Solving a lower triangular system
                                  // assuming there is 1 on the diagonal
```

components which are lower the diagonal are ignored by `gmm::upper_tri_solve` and components which are upper the diagonal are ignored by `gmm::lower_tri_solve`.

CHAPTER 9

Dense LU decomposition

The following procedures are available in the file gmm/gmm_dense_lu.h for dense real and complex matrices (gmm::dense_matrix<T>, gmm::row_matrix< std::vector<T> > and gmm::col_matrix< std::vector<T> >):

```
gmm::lu_factor(M, ipvt) : compute the LU factorization of M in M. ipvt should be
                     an gmm::lapack_ipvt (of size gmm::mat_nrows(M))
                     which will contain the indices of the pivots.

gmm::lu_solve(LU, ipvt, x, b) : solve the system LUx = b. LU is the LU
                           factorization which has to be computed first.

gmm::lu_solve(M, x, b) : solve the system Mx=b calling the lu factorization on
                    a copy of M.

gmm::lu_solve_transposed(LU, ipvt, x, b) : solve the system transposed(LU)x = b.
                                      LU is the LU factorization which
                                      has to be computed first.

gmm::lu_inverse(LU, ipvt, A) : compute the inverse of LU in A. LU is the LU
                          factorization which has to be computed first

gmm::lu_inverse(A) : invert A calling the LU factorization and the latter
                procedure.

gmm::lu_det(LU, ipvt) : compute the determinant of LU. LU is the LU
                   factorization which has to be computed first

gmm::lu_det(A) : compute the determinant of A calling the LU factorization
             and the latter function.
```

CHAPTER 10

Dense QR factorisation, eigenvalues and eigenvectors

The following procedures are available in the file gmm/gmm_dense_qr.h for dense real and complex matrices:

```
gmm::qr_factor(M, Q, R) // compute the QR factorization of M in Q and R
                        // (Householder version)

implicit_qr_algorithm(M, eigval, double tol = 1E-16) // compute the
   // eigenvalues of M using the implicit QR factorisation (Householder and
   // Francis QR step version). eigval should be a vector of appropriate size
   // in which the eigenvalues will be computed. If the matrix have
   // complex eigenvalues, please use a complex vector.

implicit_qr_algorithm(M, eigval, shvect, double tol = 1E-16) // idem,
   // compute additionally the schur vectors in the matrix shvect.

symmetric_qr_algorithm(M, eigval, double tol = 1E-16) // idem for symmetric
   // real and hermitian complex matrices (based on Wilkinson QR step)

symmetric_qr_algorithm(M, eigval, eigvect, double tol = 1E-16) // idem,
   // compute additionally the eigenvectors in the matrix eigvect.
```

Remark: The computation of eigenvectors for non hermitian matrices is not yet implemented. You can use for the moment the functions geev_interface_left and geev_interface_right from the LAPACK interface (see gmm/gmm_lapack_interface.h. These LAPACK functions compute right and left eigen vectors.

The following function defined in the file gmm/gmm_condition_number.h:

```
gmm::condition_number(M)
```

compute the condition number of a matrix M. This function uses a dense QR algorithm and thus is only usable for dense matrices.

CHAPTER 11

Iterative solvers

Most of the solvers provided in *Gmm++* come form ITL with slight modifications (gmres has been optimized and adapted for complex matrices). Include the file `gmm/gmm_iter_solvers.h` to use them.

iterations

The iteration object of *Gmm++* is a modification of the one in ITL. This is not a template type as in ITL.

The simplest initialization is:

```
gmm::iteration iter(2.0E-10);
```

where `2.0E-10` is the (relative) residual to be obtained to have the convergence. Some possibilities:

```
iter.set_noisy(n) // n = 0 : no output
                  // n = 1 : output of iterations on the standard output
                  // n = 2 : output of iterations and sub-iterations
                  //         on the standard output
                  // ...
iter.get_iteration() // after a computation, gives the number of
                     // iterations made.
iter.converged()     // true if the method converged.
iter.set_maxiter(n)  // Set the maximum of iterations.
                     // A solver stops if the maximum of iteration is
                     // reached, iter.converged() is then false.
```

Linear solvers

Here is the list of available linear solvers:

```
gmm::row_matrix< std::vector<double> > A(10, 10);  // The matrix
std::vector<double> B(10); // Right hand side
std::vector<double> X(10); // Unknown
gmm::identity_matrix PS;   // Optional scalar product for cg
gmm::identity_matrix PR;   // Optional preconditioner
...
gmm::iteration iter(10E-9);// Iteration object with the max residu
size_t restart = 50;       // restart parameter for GMRES
```

```
gmm::cg(A, X, B, PS, PR, iter); // Conjugate gradient

gmm::bicgstab(A, X, B, PR, iter); // BICGSTAB BiConjugate Gradient Stabilized

gmm::gmres(A, X, B, PR, restart, iter) // GMRES generalized minimum residual

gmm::qmr(A, X, B, PR, iter) // Quasi-Minimal Residual method.

gmm::least_squares_cg(A, X, B, iter) // unpreconditionned least square CG.
```

The solver `gmm::constrained_cg(A, C, X, B, PS, PR, iter);` solve a system with linear constaints, `C` is a matrix which represents the constraints. But it is still experimental.

(Version 1.7) The solver `gmm::bfgs(F, GRAD, X, restart, iter)` is a BFGS quasi-Newton algorithm with a Wolfe line search for large scale problems. It minimizes the function `F` without constraints, be given its gradient `GRAD`. `restart` is the max number of stored update vectors.

Preconditioners

The following preconditioners, to be used with linear solvers, are available:

```
gmm::identity_matrix P;   // No preconditioner

gmm::diagonal_precond<matrix_type> P(SM); // diagonal preconditioner

gmm::mr_approx_inverse_precond<matrix_type> P(SM, 10, 10E-17);
                                            // preconditioner based on MR
                                            // iterations

gmm::ildlt_precond<matrix_type> P(SM); // incomplete (level 0) ldlt
                                       // preconditioner. Fast to be
                                       // computed but less efficient than
                                       // gmm::ildltt_precond.

// incomplete ldlt with k fill-in and threshold preconditioner.
// Efficient but could be costly.
gmm::ildltt_precond<matrix_type> P(SM, k, threshold);

gmm::ilu_precond<matrix_type> P(SM);  // incomplete (level 0) ilu
                                      // preconditioner. Very fast to be
                                      // computed but less efficient than
                                      // gmm::ilut_precond.


// incomplete LU with k fill-in and threshold preconditioner.
// Efficient but could be costly.
gmm::ilut_precond<matrix_type> P(SM, k, threshold);

// incomplete LU with k fill-in, threshold and column pivoting preconditioner.
// Try it when ilut encounter too small pivots.
gmm::ilutp_precond<matrix_type> P(SM, k, threshold);
```

Except `ildltt\_precond`, all these precontionners come from ITL. `ilut_precond` has been optimized and simplified and `cholesky_precond` has been corrected and transformed in an incomplete LDLT preconditionner for stability reasons (similarly, we add `choleskyt_precond` which is in fact an incomplete LDLT with thresh-

old preconditionner). Of course, `ildlt\_precond` and `ildltt_precond` are designed for symmetric real or hermitian complex matrices to be use principaly with cg.

Additive Schwarz method

The additive Schwarz method is a decomposition domain method allowing the resolution of huge linear systems (see [SCHADD] for the principle of the method).

For the moment, the method is not parallelized (this should be done ...). The call is the following:

```
gmm::sequential_additive_schwarz(A, u, f, P, vB, iter, local_solver, global_solver)
```

`A` is the matrix of the linear system. `u` is the unknown vector. `f` is the right hand side. `P` is an eventual preconditioner for the local solver. `vB` is a vector of rectangular sparse matrices (`of type const std::vector<vBMatrix>`, where `vBMatrix` is a sparse matrix type), each of these matrices is of size $N \times N_i$ where N is the size of `A` and N_i the number of variables in the i^{th} sub-domain ; each column of the matrix is a base vector of the sub-space representing the i^{th} sub-domain. `iter` is an iteration object. `local_solver` has to be chosen in the list `gmm::using_gmres()`, `gmm::using_bicgstab()`, `gmm::using_cg()`, `gmm::using_qmr()` and `gmm::using_superlu()` if SuperLu is installed. `global_solver` has to be chosen in the list `gmm::using_gmres()`, `gmm::using_bicgstab()`, `gmm::using_cg()`, `gmm::using_qmr()`.

The test program `schwarz_additive.C` is the directory `tests` of GetFEM++ is an example of the resolution with the additive Schwarz method of an elastostatic problem with the use of coarse mesh to make a better preconditioning (i.e. one of the sub-domains represents in fact a coarser mesh).

In the case of multiple solves with the same linear system, it is possible to store the preconditioners or the LU factorisations to save computation time.

A (too) simple program in `gmm/gmm_domain_decomp.h` allows to build a regular domain decomposition with a certain ratio of overlap. It directly produces the vector of matrices `vB` for the additive Schwarz method.

Range basis function

The function `gmm\_range\_basis(B, columns, EPS=1e-12)` defined in `gmm/gmm\_range\_basis.h` allows to select from the columns of a sparse matrix `B` a basis of the range of this matrix. The result is returned in `columns` which should be of type `std::set<size_type>` and which contains the indices of the selected columns.

The algorithm is specially designed to select independent constraints from a large matrix with linearly dependent columns.

There is four step in the implemented algorithm

- Elimination of null columns.
- Selection of a set of already orthogonal columns.
- Elimination of locally dependent columns by a blockwise Gram-Schmidt algorithm.
- Computation of vectors of the remaining null space by a global restarted Lanczos algorithm and deduction of some columns to be eliminated.

The algorithm is efficient if after the local Gram-Schmidt algorithm it remains a low dimension null space. The implemented restarted Lanczos algorithm find the null space vectors one by one.

The Global restarted Lanczos algorithm may be improved or replaced by a block Lanczos method (see [ca-re-so1994] for instance), a block Wiedelann method (in order to be parallelized) or simply the computation of more than one vector of the null space at each iteration.

CHAPTER 12

Catch errors

Errors used in *Gmm++* are defined in the file `gmm/gmm\_except.h`. In order to make easier the error catching all errors derive from the type `std::logic\_error` defined in the file ‘‘ stdexcept‘‘ of the S.T.L.

A standard procedure, `GMM\_STANDARD\_CATCH\_ERROR`, is defined in `gmm/gmm\_except.h`. This procedure catches all errors and print the error message when an error occurs. It can be used in the main procedure of the program as follows:

```
int main(void) \{
  try \{
    ... main program ...
      \}
   GMM\_STANDARD\_CATCH\_ERROR;
\}
```

It is highly recommended to catch the errors at least in the main function, because if you do not so, you will not be able to see error messages.

CHAPTER 13

Interface with BLAS, LAPACK or ATLAS

For better performance on dense matrices, it is possible to interface some operations of the type `gmm::dense_matrix<T>` with `LAPACK` (http://www.netlib.org/lapack/) or `ATLAS` (http://math-atlas.sourceforge.net/), for `T = float, double, std::complex<float>` or `std::complex<double>`. In fact, concerning `ATLAS` no specific interface has been made untill now, so the fortran interface of `ATLAS` should be used.

to use this interface you have first to define `GMM_USES_LAPACK` before including *Gmm++* files:

```
\#define GMM_USES_LAPACK
\#include <gmm/gmm.h>

... your code
```

or specify -DGMM_USES_LAPACK on the command line of your compiler. Of course, you have also to link `LAPACK` or `ATLAS` libraries. For example on a standard linux configuration and g++ compiler the adding libraries to link `LAPACK` are:

```
g++ ...  -llapack -lblas -lgfortanbegin -lgfortran
```

and to link `ATLAS`:

```
g++ ... /usr/lib/atlas/liblapack.a /usr/lib/atlas/libblas.a -latlas -lgfortranbegin
 ↪  -lgfortran
```

The library `libgfortranbegin` and `libgfortran` are specific to g++ compiler and may vary for other compilers.

Ask your system administrator if this configuration does not work.

The following operations are interfaced:

```
vect_norm2(std::vector<T>)

vect_sp(std::vector<T>, std::vector<T>)
vect_sp(scaled(std::vector<T>), std::vector<T>)
vect_sp(std::vector<T>, scaled(std::vector<T>))
vect_sp(scaled(std::vector<T>), scaled(std::vector<T>))

vect_hp(std::vector<T>, std::vector<T>)
vect_hp(scaled(std::vector<T>), std::vector<T>)
vect_hp(std::vector<T>, scaled(std::vector<T>))
vect_hp(scaled(std::vector<T>), scaled(std::vector<T>))
```

```
add(std::vector<T>, std::vector<T>)
add(scaled(std::vector<T>, a), std::vector<T>)

mult(dense_matrix<T>, dense_matrix<T>, dense_matrix<T>)
mult(transposed(dense_matrix<T>), dense_matrix<T>, dense_matrix<T>)
mult(dense_matrix<T>, transposed(dense_matrix<T>), dense_matrix<T>)
mult(transposed(dense_matrix<T>), transposed(dense_matrix<T>),
     dense_matrix<T>)
mult(conjugated(dense_matrix<T>), dense_matrix<T>, dense_matrix<T>)
mult(dense_matrix<T>, conjugated(dense_matrix<T>), dense_matrix<T>)
mult(conjugated(dense_matrix<T>), conjugated(dense_matrix<T>),
     dense_matrix<T>)

mult(dense_matrix<T>, std::vector<T>, std::vector<T>)
mult(transposed(dense_matrix<T>), std::vector<T>, std::vector<T>)
mult(conjugated(dense_matrix<T>), std::vector<T>, std::vector<T>)
mult(dense_matrix<T>, scaled(std::vector<T>), std::vector<T>)
mult(transposed(dense_matrix<T>), scaled(std::vector<T>),
     std::vector<T>)
mult(conjugated(dense_matrix<T>), scaled(std::vector<T>),
     std::vector<T>)

mult_add(dense_matrix<T>, std::vector<T>, std::vector<T>)
mult_add(transposed(dense_matrix<T>), std::vector<T>, std::vector<T>)
mult_add(conjugated(dense_matrix<T>), std::vector<T>, std::vector<T>)
mult_add(dense_matrix<T>, scaled(std::vector<T>), std::vector<T>)
mult_add(transposed(dense_matrix<T>), scaled(std::vector<T>),
         std::vector<T>)
mult_add(conjugated(dense_matrix<T>), scaled(std::vector<T>),
         std::vector<T>)

mult(dense_matrix<T>, std::vector<T>, std::vector<T>, std::vector<T>)
mult(transposed(dense_matrix<T>), std::vector<T>, std::vector<T>,
     std::vector<T>)
mult(conjugated(dense_matrix<T>), std::vector<T>, std::vector<T>,
     std::vector<T>)
mult(dense_matrix<T>, scaled(std::vector<T>), std::vector<T>,
     std::vector<T>)
mult(transposed(dense_matrix<T>), scaled(std::vector<T>),
     std::vector<T>, std::vector<T>)
mult(conjugated(dense_matrix<T>), scaled(std::vector<T>),
     std::vector<T>, std::vector<T>)
mult(dense_matrix<T>, std::vector<T>, scaled(std::vector<T>),
     std::vector<T>)
mult(transposed(dense_matrix<T>), std::vector<T>,
     scaled(std::vector<T>), std::vector<T>)
mult(conjugated(dense_matrix<T>), std::vector<T>,
     scaled(std::vector<T>), std::vector<T>)
mult(dense_matrix<T>, scaled(std::vector<T>), scaled(std::vector<T>),
  std::vector<T>)
mult(transposed(dense_matrix<T>), scaled(std::vector<T>),
     scaled(std::vector<T>), std::vector<T>)
mult(conjugated(dense_matrix<T>), scaled(std::vector<T>),
     scaled(std::vector<T>), std::vector<T>)

lower_tri_solve(dense_matrix<T>, std::vector<T>, k, b)
upper_tri_solve(dense_matrix<T>, std::vector<T>, k, b)
lower_tri_solve(transposed(dense_matrix<T>), std::vector<T>, k, b)
```

```
upper_tri_solve(transposed(dense_matrix<T>), std::vector<T>, k, b)
lower_tri_solve(conjugated(dense_matrix<T>), std::vector<T>, k, b)
upper_tri_solve(conjugated(dense_matrix<T>), std::vector<T>, k, b)

lu_factor(dense_matrix<T>, std::vector<int>)
lu_solve(dense_matrix<T>, std::vector<T>, std::vector<T>)
lu_solve(dense_matrix<T>, std::vector<int>, std::vector<T>,
         std::vector<T>)
lu_solve_transposed(dense_matrix<T>, std::vector<int>, std::vector<T>,
         std::vector<T>)
lu_inverse(dense_matrix<T>)
lu_inverse(dense_matrix<T>, std::vector<int>, dense_matrix<T>)

qr_factor(dense_matrix<T>, dense_matrix<T>, dense_matrix<T>)

implicit_qr_algorithm(dense_matrix<T>, std::vector<T>)
implicit_qr_algorithm(dense_matrix<T>, std::vector<T>,
                      dense_matrix<T>)
implicit_qr_algorithm(dense_matrix<T>, std::vector<std::complex<T> >)
implicit_qr_algorithm(dense_matrix<T>, std::vector<std::complex<T> >,
                      dense_matrix<T>)
```

Of course, it is not difficult to interface another operation if needed.

The following interface does not correspond to an algorithm existing in *Gmm++*:

The interface to `gesvd` (singular value decomposition):

```
svd(dense_matrix<T> &X, dense_matrix<T> &U,
    dense_matrix<T> &Vt, std::vector<T> sigma);
svd(dense_matrix<std::complex<T> > &X, dense_matrix<std::complex<T> > &U,
    dense_matrix<std::complex<T> > &Vt, std::vector<T> sigma);
```

CHAPTER 14

Interface with SuperLU

It is possible to call SuperLU 3.0 (http://crd.lbl.gov/verb~xiaoye/SuperLU/) from *Gmm++*. The following function defined in the file `gmm/gmm_superlu_interface.h` is available:

```
SuperLU_solve(A, X, B, condest, permc_spec = 1)
```

solves the system `AX = B` where A is a sparse matrix of base type `float`, `double`, `std::complex<float>`, or `std::complex<double>`. `permc_spec` should be 0, 1 or 2 for respectively use the natural ordering, use minimum degree ordering on structure of `A'A` or use minimum degree ordering on structure of `A'+A` (1 is the default value), `condest` should be a reference on a double, it returns an estimate of the condition number of the matrix `A`.

To use these functions, you need to install SuperLU and compile your code with the additional options:

```
g++ ...  -DGMM_USES_SUPERLU (dir_of_superlu)/superlu.a -lblas -I(dir_of_superlu)
```

Some other functionalities of SuperLU can be interfaced.

CHAPTER 15

How to use *Gmm++* with QD type (double-double and quad-double)

The QD library (see http://www.cs.berkeley.edu/verb~yozo or http://www.nersc.gov/verb~dhb/mpdist/mpdist.html) is an efficient library for double-double (32 decimal digits) and quad-double (approx. 64 decimal digits). Once you installed this library on your system you have to link your program with QD library (with -lqd). In your program, include the header files of QD with:

```
#include <qd/dd.h>
#include <qd/qd.h>
#include <qd/fpu.h>
```

Then the two type `dd_real` and `qd_real` will be usable with *Gmm++*. You will also be able to use `std::complex<dd_real>` and `std::complex<qdreal>`

IMPORTANT : do not forget to initialize QD before using it with the following call:

```
unsigned int old_cw;
fpu_fix_start(&old_cw);
```

This disables the 80 bits precision of x86 processors which conflicts with QD. Once you finished to use QD you can reactivate it with:

```
fpu_fix_end(&old_cw);
```

(see the QD documentation for more details).

CHAPTER 16

First steps with *Gmm++*

How can I invert a matrix ?

It is not possible in *Gmm++* to invert all kind of matrices. For the moment, the only mean to invert a matrix is to use the dense LU decomposition (thus, only for dense matrices). An example:

```
gmm::dense_matrix<double> M(3, 3), M2(3,3), M3(3,3);
gmm::copy(gmm::identity_matrix(), M);  // M = Id.
gmm::scale(M, 2.0);                    // M = 2 * Id.
M(1,2) = 1.0;

gmm::copy(M, M2);

gmm::lu_inverse(M);

gmm::mult(M, M2, M3);

std::cout << M << " times " << M2 << " is equal to " << M3 << endl;
```

see the section corresponding to dense LU decomposition for more details. The type `gmm::dense_matrix<double>` can be replaced by `gmm::row_matrix< std::vector<double> >` or `gmm::col_matrix< std::vector<double> >`.

How can I solve a linear system ?

You have more than one possibility to solve a linear system. If you have a dense matrix, the best may be to use the LU decomposition. An example:

```
gmm::dense_matrix<double> M(3, 3);
gmm::clear(M);                    // M = 0.
M(0,0) = M(1,1) = M(2,2) = 2.0; // M = 2 * Id.
M(1,2) = 1.0;

std::vector<double> X(3), B(3), Bagain(3);
B[0] = 1.0; B[1] = 2.0; B[2] = 3.0;  // B = [1 2 3]

gmm::lu_solve(M, X, B);

gmm::mult(M, X, Bagain);
```

```
std::cout << M << " times " << gmm::vref(X) << " is equal to " << gmm::vref(Bagain) <<
 ↪  endl;
```

If, now, you have a sparse system coming for example from a pde discretization, you have various iterative solvers, with or without preconditioners. This is an example with a precontionned GMRES:

```
int nbdof = 1000; // number of degrees of freedom.
gmm::row_matrix< gmm::rsvector<double> > M(nbdof, nbdof); // a sparse matrix
std::vector<double> X(nbdof), B(nbdof); // Unknown and left hand side.

... here the assembly of the pde discretization stiffness matrix ...
... and left hand side ...


// computation of a preconditioner (ILUT)
gmm::ilut_precond< gmm::row_matrix< gmm::rsvector<double> > > P(M, 10, 1e-4);

gmm::iteration iter(1E-8);  // defines an iteration object, with a max residu of 1E-8

gmm::gmres(M, X, B, P, 50, iter);  // execute the GMRES algorithm

std::cout << "The result " << gmm::vref(X) << endl;
```

How can I transform a vector into a matrix and reshape it ?

In *Gmm++*, a vector is not considered as a matrix. If you need to use a vector as a (1 by n) row matrix or (n by 1) column matrix in a computation, you have to use:

```
gmm::row_vector(V) // gives a reference on V considered as
                   // a (1 by n) row matrix
gmm::col_vector(V) // gives a reference on V considered as
                   // a (n by 1) col matrix
```

for instance, you can transform a vector into a dense matrix with:

```
std::vector<double> V(50);

// ... computation of V

gmm::dense_matrix<double> M(1, gmm::vect_size(V));
gmm::copy(gmm::row_vector(V), M);
```

Then you can also reshape matrix `M` with:

```
gmm::reshape(M, 10, 5);
```

What is the better way to resize a matrix ?

You can change the dimensions of a matrix, if it is not a reference, using:

```
gmm::resize(M, m, n);
```

This function respects the intersection between the original matrix and the resized matrix, and new components are set to zero. An important thing is that it is based on the resize method of `std::vector`, thus no memory free is done when the size of the new matrix is smaller than the original one.

If you do not need to keep old values of the components, or if you want to really free the surplus of memory, you can resize a matrix using `std::swap` as follows:

```
MATRIX_TYPE M(m1, n1);

... your code

{ MATRIX_TYPE(m2, n2) M2; std::swap(M, M2); } // resize matrix M.
```

Of course, this works also for a vector.

CHAPTER 17

Deeper inside *Gmm++*

The linalg_traits structure

The major principle of *Gmm++* is that each vector and matrix type has a corresponding structure (which is never instantiated) named `linalg_traits` containing all informations on it. For instance, the component `linalg_type` of this structure is set to `abstract_vector` or `abstract_matrix` if the corresponding type represent a vector or a matrix. If `V` is an interfaced type of vector and `M` an interface type of matrix, it is possible to access to this component with:

```
typename gmm::linalg_traits<V>::linalg_type ...  // should be abstract_vector
typename gmm::linalg_traits<M>::linalg_type ...  // should be abstract_matrix
```

The types `abstract_vector` and `abstract_matrix` are defined in `gmm/gmm_def.h`. They are void type allowing to specialize generic algorithms.

For a vector type, the following informations are available:

```
typename gmm::linalg_traits<V>::value_type     --> type of the components of the
                                                   vector
typename gmm::linalg_traits<V>::reference      --> type of reference on a component
typename gmm::linalg_traits<V>::is_reference   --> if the vector is a simple
                                                   reference or an instantiated vector
typename gmm::linalg_traits<V>::linalg_type    --> should be abstract_vector
typename gmm::linalg_traits<V>::index_sorted    --> linalg_true or linalg_false
typename gmm::linalg_traits<V>::const_iterator --> const iterator to iterate on the
                                                   components of the vector in
                                                   order to read them.
typename gmm::linalg_traits<V>::iterator       --> iterator to iterate on the
                                                   components of the vector in
                                                   order to read or write them.
typename gmm::linalg_traits<V>::storage_type   --> should be abstract_sparse,
                                                   abstract_skyline or
                                                   abstract_dense

typename gmm::linalg_traits<V>::origin_type    --> the type of vector itself
                                                   or the type of referenced
                                                   vector for a reference.

gmm::linalg_traits<V>::size(v)      --> a method which gives the size of the vector.
gmm::linalg_traits<V>::begin(v)     --> a method which gives an iterator on the
                                        beginning of the vector
gmm::linalg_traits<V>::end(v)       --> iterator on the end of the vector
```

```
gmm::linalg_traits<V>::origin(v)    --> gives a void pointer allowing to identify
                                        the vector
gmm::linalg_traits<V>::do_clear(v) --> make a clear on the vector

gmm::linalg_traits<V>::access(o, it, ite, i) --> return the ith component or a
                                          reference on the ith component. o is a
                                          pointer o type ``origin_type *'' or
                                          ``const origin_type *''.

gmm::linalg_traits<V>::clear(o, it, ite) --> clear the vector. o is a
                                          pointer o type ``origin_type *'' or
                                          ``const origin_type *''.
```

and for a matrix type:

```
typename gmm::linalg_traits<M>::value_type     --> type of the components of the
                                                   matrix
typename gmm::linalg_traits<M>::reference      --> type of reference on a component
typename gmm::linalg_traits<M>::is_reference   --> if the matrix is a simple
                                                   reference or an instantiated matrix
typename gmm::linalg_traits<M>::linalg_type    --> should be abstract_matrix
typename gmm::linalg_traits<M>::storage_type   --> should be abstract_sparse,
                                                   abstract_skyline or
                                                   abstract_dense
typename gmm::linalg_traits<M>::index_sorted    --> linalg_true or linalg_false
typename gmm::linalg_traits<M>::sub_orientation --> should be row_major, col_major
                                                    row_and_col or col_and_row.
typename gmm::linalg_traits<M>::sub_col_type      --> type of reference on a column
                                                     (if the matrix is not row_major)
typename gmm::linalg_traits<M>::const_sub_col_type --> type of const reference on a
                                                       column
typename gmm::linalg_traits<M>::col_iterator      --> iterator on the columns
typename gmm::linalg_traits<M>::const_col_iterator --> const iterator on the columns
typename gmm::linalg_traits<M>::sub_row_type      --> type of reference on a row
                                                     (if the matrix is not col_major)
typename gmm::linalg_traits<M>::const_sub_row_type --> type of const reference on a
                                                       row
typename gmm::linalg_traits<M>::const_row_iterator --> const iterator on the rows
typename gmm::linalg_traits<M>::row_iterator       --> iterator on the rows

typename gmm::linalg_traits<M>::origin_type    --> the type of vector itself
                                                   or the type of referenced
                                                   vector for a reference.


gmm::linalg_traits<M>::nrows(m)     --> methods which gives the number of rows of
                                        the matrix
gmm::linalg_traits<M>::ncols(m)     --> number of columns
gmm::linalg_traits<M>::row_begin(m) --> iterator on the first row (if not col_major)
gmm::linalg_traits<M>::row_end(m)   --> iterator on the end of the rows
gmm::linalg_traits<M>::col_begin(m) --> iterator on the first column
                                        (if not row_major)
gmm::linalg_traits<M>::col_end(m)   --> iterator on the end of the columns
gmm::linalg_traits<M>::row(it)      --> gives the reference on a row with an iterator
                                        (if not col_major)
gmm::linalg_traits<M>::col(it)      --> gives the reference on a column with an
                                        iterator  (if not row_major)
gmm::linalg_traits<M>::origin(m)    --> gives a void pointer allowing to identify
                                        the matrix
```

```
gmm::linalg_traits<M>::access(it,i) --> return the ith component or a reference
                                        on the ith component of the row or
                                        column pointed by it.
gmm::linalg_traits<M>::do_clear(m)  --> make a clear on the matrix
```

This is this structure you have to fill in to interface a new vector or matrix type. You can see some examples in `gmm/gmm_interface.h`. Most of the generic algorithms are in `gmm/gmm_blas.h`.

How to iterate on the components of a vector

Here is an example which accumulate the components of a vector. It is assumed that `V` is a vector type and `v` an instantiated vector:

```
typename gmm::linalg_traits<V>::value_type r(0); // scalar in which we accumulate
typename gmm::linalg_traits<V>::const_iterator it = vect_const_begin(v); // beginning
                                                                         // of v
typename gmm::linalg_traits<V>::const_iterator ite = vect_const_end(v); // end of v

for (; it != ite; ++it)  // loop on the components
  r += *it;              // accumulate the components
```

This piece of code will work with every kind of interfaced vector.

For sparse or skyline vectors, it is possible to obtain the index of the components pointed by the iterator with `it.index()`. Here is the example of the scalar product of two sparse or skyline vectors, assuming `V1` and `V2` are two vector types and `v1`, `v2` two corresponding instantiated vectors:

```
typename gmm::linalg_traits<V1>::const_iterator it1 = vect_const_begin(v1),
typename gmm::linalg_traits<V1>::const_iterator ite1 = vect_const_end(v1);
typename gmm::linalg_traits<V2>::const_iterator it2 = vect_const_begin(v2),
typename gmm::linalg_traits<V2>::const_iterator ite2 = vect_const_end(v2);
typename gmm::linalg_traits<V1>::value_type r(0); // it is assumed that V2 have a
                                                  // compatible value_type

while (it1 != ite1 && it2 != ite2) \{  // loops on the components
  if (it1.index() == it2.index()) \{
    res += (*it1) * (*it2));          // if the indices are equals accumulate
    ++it1;
    ++it2;
  \}
  else if (it1.index() < it2.index())
    ++it1;
  else
    ++it2;
\}
```

This algorithm use the fact that indices are increasing in a sparse vector. This code will not work for dense vectors because dense vector iterators do not have the method `it.index()`.

How to iterate on a matrix

You can iterate on the rows of a matrix if it is not a column major matrix and on the columns of a matrix if it is not a row major matrix (the type `gmm::dense_matrix<T>` has is sub orientation type as col_and_rox, so you can iterate on both rows and columns).

If you need not to be optimal, you can use a basic loop like that:

```
for (size_t i = 0; i < gmm::mat_nrows(m); ++i) \{
  typename gmm::linalg_traits<M>::const_sub_row_type row = mat_const_row(M, i);

  ...

  std::cout << "norm of row " << i << " : " << vect_norm2(row) << std::endl;
\}
```

But you can also use iterators, like that:

```
typename gmm::linalg_traits<M>::const_row_iterator it = mat_row_const_begin(m);
typename gmm::linalg_traits<M>::const_row_iterator ite = mat_row_const_end(m);

for (; it != ite; ++it) \{
  typename gmm::linalg_traits<M>::const_sub_row_type
    row = gmm::linalg_traits<M>::row(it);

  ...

  std::cout << "norm of row " << i << " : " << vect_norm2(row) << std::endl;
\}
```

How to make your algorithm working on all type of matrices

For this, you will generally have to specialize it. For instance, let us take a look at the code for `gmm::nnz` which count the number of stored components (in fact, the real `gmm::nnz` algorithm is specialized in most of the cases so that it does not count the components one by one):

```
template <class L> inline size_type nnz(const L& l) \{
  return nnz(l, typename linalg_traits<L>::linalg_type());
\}

template <class L> inline size_type nnz(const L& l, abstract_vector) \{
  typename linalg_traits<L>::const_iterator it = vect_const_begin(l);
  typename linalg_traits<L>::const_iterator ite = vect_const_end(l);
  size_type res(0);
  for (; it != ite; ++it) ++res;
  return res;
\}

template <class L> inline size_type nnz(const L& l, abstract_matrix) \{
  return nnz(l,  typename principal_orientation_type<typename
                 linalg_traits<L>::sub_orientation>::potype());
\}

template <class L> inline size_type nnz(const L& l, row_major) \{
  size_type res(0);
  for (size_type i = 0; i < mat_nrows(l); ++i)
    res += nnz(mat_const_row(l, i));
  return res;
\}

template <class L> inline size_type nnz(const L& l, col_major) \{
  size_type res(0);
```

```
  for (size_type i = 0; i < mat_ncols(l); ++i)
    res += nnz(mat_const_col(l, i));
  return res;
\}
```

The first function dispatch on the second or the third function respectively if the parameter is a vector or a matrix. The third function dispatch again on the fourth and the fifth function respectively if the matrix is row_major or column major. Of course, as the function are declared `inline`, at least the two dispatcher functions will not be implemented. Which means that this construction is not costly.

CHAPTER 18

How to disable verifications

On some type of matrices such as `gmm::dense_matrix` some verification are made on the range of indices. This could deteriorate the performance of your code but is satisfactory in the developpment stage. You can disable these verifications adding a `-dNDEBUG` to the compiler options.