The Matrix Template Library
User Manual

Jeremy G. Siek, Andrew Lumsdaine and Lie-Quan Lee

February 2, 2004

Contents

List of Tables

List of Figures

Part I

Introduction to Generic Programming

Chapter 1

Traits Classes

One of the most important techniques used in generic programming is the traits class, which was introduced by Nathan Meyers in XX. The traits class technique may seem strange and somewhat daunting when first encountered (granted the syntax looks a bit strange) but the essense of the idea is simple, and it is essential to learn how to use traits classes, for they appear over and over again in generic libraries such as the STL, and are also used heavily here in the MTL. Here we give a short motivation and tutorial for traits classes via an extended example.

A traits class is basically a way of finding out information about a type that you otherwise would not know anything about. For instance, suppose I want to write a generic `sum()` function:

```
template <class Vector>
X sum(const Vector& v, int n)
{
  X total;
  for (int i = 0; i < n; ++i)
    total += v[i];
  return total;
}
```

From the point of view of this template function, not much is know about the template type `Vector`. For instance, I don't know what kind of elements are inside the vector. But I need to know this in order to declare the local variable `total`, which should be the same type as the elements of `Vector` (the `X` there right now is just a bogus placeholder that needs to be replaced by something else!).

1.1 Typedefs Nested in Classes

One way to access information out of a type is to use the scope operator "::" to get at typedefs that are nested inside the class. Imagine I've created a vector class that looks like this:

```
class my_vector {
public:
  typedef double value_type; // the type for elements in the vector
  double& operator[](int i) { ... };
  ...
};
```

Now I can access the type of the elements by writing `my_vector::value_type`. Here's the generic `sum()` function again, with the `X`'s replaced:

```
template <class Vector>
typename Vector::value_type sum(const Vector& v, int n)
{
  typename Vector::value_type total = 0;
  for (int i = 0; i < n; ++i)
    total += v[i];
  return total;
}
```

The use of the keyword `typename` deserves some explaining. Due to some quirks in C++, nested typedefs and nested members can cause ambiguity from the compiler's point of view: when parsing a template function the compiler may not know whether the thing on the right hand side of the scope operator is a type or an object. The `typename` keyword is used to clear up this ambiguity. The rule of thumb is, whenever you use the scope operator to access a nested typedef, and when the type on the left hand side of the scope operator somehow depends on a template argument, then use the `typename` keyword. If the type on the left hand side does not depend on a template argument, then do not use `typename`. In the above `sum()` function we use `typename` since the left hand side, `Vector`, is a template argument. In contrast, `typename` is not used below since the type `std::vector<int>` does not depend on any template arguments (it is not even in a template function!).

```
int main(int, char*[]) {
  std::vector<int>::value_type x;
  return 0;
}
```

Getting back to the `sum()` function, the technique of using a nested typedef works as long as `Vector` is a class type that has such a nested typedef. But what if I want to use the generic `sum()` function with a builtin type such as `double*` that couldn't possibly have any typedefs? Or what if we want to use `sum()` with a vector class from a third party who didn't provide the necessary typedef? The `operator[]` works with `double*` and our imaginary third party vector, so it would be a shame to miss out on a chance for re-use just because of the issue of accessing the value type. Below shows the situation we want to make possible, where `sum()` can be reused with types such as `double*`.

```
double* x = ...;
int n = ...;

sum(x, n);
```

1.2 Template Specialization

The solution to this problem is the traits class technique. To understand how traits classes work, we first need to review some facts about template classes and something called specialization. We start with a simple (though perhaps gratuitous) example of a template class:

```
template <class T>
class hello_world {
public:
  void say_hi() { cout << "Hello world!" << endl; }
};
```

One interesting thing about C++ templates is that they can be specialized. That is, a special version of the class can be explicitly created for a particular case of the template arguments (in this case `T`). So we can write this:

```
template <>
class hello_world<int> {
public:
  void say_hi() { cout << "I'm special!" << endl; }
};

template <>
class hello_world<double*> {
public:
  void say_hi() { cout << "I'm special too!" << endl; }
};
```

Now can you guess what happens when I do the following?

```
hello_world<char> h1;
hello_world<long> h2;
hello_world<int> h3;
hello_world<double*> h4;

h1.say_hi();
h2.say_hi();
h3.say_hi();
h4.say_hi();
```

Here's what the output will look like:

```
Hello world!
Hello world!
I'm special!
I'm special too!
```

In this example, template specialization allowed us to pick different versions of the `say_hi()` member function based on some type (`char, int, long, double*`). The specializations of `hello_world` created a mapping between a type and a version of `say_hi()`. The original templated version of `hello_world` acted like a default if none of the specializations covered the type.

1.3 Definition of a Traits Class

In general, we can use specialization to create mappings from types to other nested types, member functions, or constants. It might be helpful to think of the template class as a function that executes at compile-time, whose input is the template parameters and whose output is the things nested in the class. A *traits class* is a class who's sole purpose is to define such a mapping.

Getting back to the generic `sum()` function, here's an example traits class, templated on the `Vector` type, that allows us to get the element, or `value_type` of the vector. For the default case, we'll assume the vector is a class with a nested typedef like `my_vector`:

```
template <class Vec>
struct vector_traits {
  typedef typename Vec::value_type value_type;
};
```

But now we can also handle the case when the `Vector` template argument is something else like a `double*`:

```
template <>
struct vector_traits<double*> {
  typedef double value_type;
};
```

or even some third party class, say `johns_int_vector`:

```
template <>
struct vector_traits<johns_int_vector> {
  typedef int value_type;
};
```

1.4 Partial Specialization

Now one might get bored of writing a traits class for every pointer type, or perhaps the third party class is templated. The solution to this is the long-winded term *partial specialization*. Here's what you can write:

```
template <class T>
struct vector_traits<T*> {
  typedef T value_type;
};
template <class T>
struct vector_traits< johns_vector<T> > {
  typedef T value_type;
};
```

Your C++ compiler will attempt a pattern match between the template argument provided at the "call" to the traits class, and all the specializations defined, picking the specialization that is the closest match. The above partial specialization for `T*` will match whenever the type is a pointer, though the previous complete specializations for `double*` would match first for that particular pointer type.

The most well known use of a traits class is the `iterator_traits` class used in the Standard Template Library, which provides access to things like the `value_type` and `iterator_category` that is associated with an Iterator. MTL also uses traits classes, such as `matrix_traits`. Typically a traits class is used with with a particular concept or family of concepts. The `iterator_traits` class is used with the family of Iterator concepts. The `matrix_traits` class is used with the familty of MTL Matrix concepts. The traits class is what provides access to the *associated types* of a concept. We will explain concepts, associated types, etc. in detail in Chapter 2.

1.5 External Polymorphism and Tags

A technique that often goes hand in hand with traits classes is *external polymorphism*, which is a way of using function overloading to dispatch based on properties of a type. A good example of this is the implementation of the `std::advance()` function in the STL, which increments an iterator `n` times. Depending on the kind of iterator, there are different optimizations that can be applied in the implementation. If the iterator is random access (can jump forward and backward arbitrary distances), then the `advance()` function can simply be implemented with `i += n`, and is very efficient: constant time. If the iterator is bidirectional, then it makes sense for `n` to be negative, we can decrement the iterator `n` times. The relation between external polymorphism and traits classes is that the property to be exploited (in this case the `iterator_category`) is accessed through a traits class. The main `advance()` function uses the `iterator_traits` class to get the `iterator_category`. It then makes a call the the overloaded `__advance()` function. The appropriate `__advance()` is selected by the compiler based on whatever type the `iterator_category` resolves to, either `input_iterator_tag`, `bidirectional_iterator_tag`, or `random_access_iterator_tag`. A *tag* is simply a class whose only purpose is to convey some property for use in external polymorphism.

```
struct input_iterator_tag { };
struct bidirectional_iterator_tag { };
struct random_access_iterator_tag { };

template <class InputIterator, class Distance>
void __advance(InputIterator& i, Distance n, input_iterator_tag) {
  while (n--) ++i;
}

template <class BidirectipuronalIterator, class Distance>
void __advance(BidirectionalIterator& i, Distance n,
               bidirectional_iterator_tag) {
```

```
  if (n >= 0)
    while (n--) ++i;
  else
    while (n++) --i;
}

template <class RandomAccessIterator, class Distance>
void __advance(RandomAccessIterator& i, Distance n,
               random_access_iterator_tag) {
  i += n;
}

template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n) {
  typedef typename iterator_traits<InputIterator>::iterator_category Cat;
  __advance(i, n, Cat());
}
```

Chapter 2

Concepts and Models

Here we define the basic terminology of generic programming, much of which was introduced in [1]. In the context of generic programming, the term *concept* is used to describe the collection of requirements that a template argument must meet for the template function or templated class to compile and operate properly. The requirements are described as a set of valid expressions, associated types, invariants, and complexity guarantees. A type that meets the set of requirements is said to *model* the concept.

2.1 Requirements

Valid Expressions are C++ expressions which must compile successfully for the objects involved in the expression to be considered *models* of the concept.

Associated Types are types that are related to the modelling type in one of two ways. They are either provided by typedefs nested within class definition for the type, or they are accessed through a traits class, such as iterator traits.

Invariants are typically run-time characteristics of the objects that must always be true, that is, the functions involving the objects must preserve these characteristics.

Complexity Guarantees are maximum limits on how long the execution of one of the valid expressions will take, or how much of various resources its computation will use.

2.2 Example: InputIterator

Examples of concept definitions can be found in the C++ Standard, many of which deal with the requirements for iterators. The InputIterator[1] concept is one of these. The following expressions are valid if the object `i` is an instance of some type that models InputIterator.

```
++i
i++
*i
```

The `std::iterator_traits` class provides access to the associated types of an iterator type. In the following example we find out what type of object is pointed to by some iterator type (call it `X`) via the `value_type` of the traits class.

```
typename iterator_traits<X>::value_type t;
t = *i;
```

As for complexity guarantees, all of the InputIterator operations are required to be constant time. Some examples of types that satisfy the requirements for InputIterator are `double*`, `std::list<int>::iterator`, and `std::istream_iterator<char>`.

2.3 Concepts vs. Abstract Base Classes

In many respects a concept is similar to an abstract base class (or interface): it defines a set of types that are interchangeable from the point of view of some algorithm (or collection of algorithms). Also, much in the way abstract base classes can inherit from (extend) other base classes, a concept can *refine* or add requirements to another concept.

However a concept is much looser than an abstract base class: there is no inheritance requirement and the valid expressions offer more freedom than the requirement for member functions with exactly matching signatures. Also dispatch based on type is not required to be via a virtual function (though it still can be). Also, for expressions that involve multiple types, the function overload resolution can depend on both types, which avoids the *binary method* [3] problem associated with inheritance-based polymorphism.

2.4 Multi-type Concepts and Modules

Most concepts describe expressions that involve interaction between two or more types. Often one of the types is the "main" type and the other types can be derived from the "main" type via typdefs or traits class (they are associated types). We talk of the "main" type as the one that models the concept. This is the case with the iterators and containers of the STL. In our example above, the dereference expression `*i` required by InputIterator returns a second type,

[1] We always use the **bold sans serif** font for concept names.

namely the `value_type` associated with the iterator type (the "main" type in this case).

However, for some concepts there is not a "main" type. There are multiple types involved, none of which can be derived from the others. We call a set of types that together model a concept a *module*. For example, later in this chapter we will be defining the concept of a VectorSpace which consists of some vector type, a scalar type, and a multiplication operator between the two. Since one can multiply a complex number by a `float`, the module {`complex<float>,float`} is a model of VectorSpace. It is also true that the module {`complex<float>,double`} is a model of VectorSpace, so we see that there is not necessarily a one-to-one relationship between the vector type and the scalar type, as there was between the iterator type and value type discussed above. In the descriptions of multi-type concepts we will list the *participating types* instead of associated types. No traits classes or nested typedefs are specified, as there is not a one-to-one mapping between the participating types. The participating types are typically readily available by other means.

2.5 Concept Checking

The C++ language does not provide direct support for ensuring that template arguments meet the requirements demanded of them by the generic algorithm or template class. This means that if the user makes an error, the resulting compiler error will point to somewhere deep inside the implementation of the generic algorithm, giving an error that may not be easy to match with the cause.

Together with the SGI STL team we have developed a method for forcing the compiler to give better error messages. The idea is to exercise all the requirements placed on the template arguments at the very beginning of the generic algorithm. We have created some macros and a methodology for how to do this.

Suppose we wish to add concept checks to the STL `copy()` algorithm, which has the following prototype.

```
template <class InIter, class OutIter>
OutIter copy(InIter first, InIter last, OutIter result);
```

We will need to make sure the `InIter` is a model of InputIterator and that `OutIter` is a model of OutputIterator. The first step is to create the code that will excercise the expressions associated with each of these concepts. The following is the concept checking class for OutputIterator.

```
template <class T>
struct OutputIterator {
  CLASS_REQUIRES(T, Assignable);
  void constraints() {
    (void)*i;          // require dereference operator
    ++i;               // require preincrement operator
    i++;               // require postincrement operator
    *i++ = *j;         // require postincrement and assignment
```

```
  }
  T i, j;
};
```

Once the concept checking classes are complete, one simple needs to invoke them at the beginning of the generic algorithm using our `REQUIRE` macro. Here's what the STL `copy()` algorithm looks like with the concept checks inserted.

```
template <class InIter, class OutIter>
OutIter copy(InIter first, InIter last, OutIter result)
{
  REQUIRE(OutIter, OutputIterator);
  REQUIRE(InIter, InputIterator);
  return copy_aux(first, last, result, VALUE_TYPE(first));
}
```

Looking back at the `OutputIterator` concept checking class you might wonder why we used the `CLASS_REQUIRES` macro instead of just `REQUIRE`. The reason for this is that different tricks are needed to force compilation of the checking code when invoking the macro from inside a class definition instead of a function.

Sometimes there is more than one type involved in a concept. This means that the corresponding concept checking class will have more than one template argument. The `VectorSpace` concept checker is an example of one of this, it involves an AbelianGroup and a Field.

```
template <class G, class F>
struct VectorSpace {
  CLASS_REQUIRES2(G, F, R_Module);
  CLASS_REQUIRES(F, Field);
  void constraints() {
    y = x / a;
    y /= a;
  }
  G x, y, z;
  F a;
};
```

When invoking a concept checker with more than one type it is necessary to append the number of type arguments to the macro name. Here is an example of using a multi-type concept checker.

```
template <class Vector, class Real>
void foo(Vector& x, Real a) {
  REQUIRE2(Vector, Real, VectorSpace);
  ...
}
```

For the most part the user of MTL does not need to know how to create concept checks and insert them in generic algorithms, however it is good to know what they are and how to use them. This will make the error messages easier to understand. Also if you are unsure about whether you can use a certain MTL class with a particular algorithm, or whether some custom-made class of your own is compatible, a good first check is to invoke the appropriate concept checker. Here's a quick example program that one could write to see if the types `std::complex<double>` and `float` together satisfy the requirements for VectorSpace.

```
#include <complex>
#include <mtl/linear_algebra_concepts.h>

int main(int,char*[])
{
  using mtl::VectorSpace;
  REQUIRE2(std::complex<double>, float, VectorSpace);
  return 0;
}
```

In addition, if you create generic algorithms of your own then we highly encourage you to create, publish and use concept checks.

Part II

Introduction to Numerical Linear Algebra

Part III

Tutorials

Chapter 3

Gaussian Elimination

```
template <class Matrix>
void gaussian_elimination(Matrix& A)
{
  typename matrix_traits<Matrix>::size_type
    m = A.nrows(), n = A.ncols(), i, k;
  typename matrix_traits<Matrix>::value_type s;

  for (k = 0; k < std::min(m-1,n-1); ++k) {
    if (A(k,k) != zero(s))
      for (i = k + 1; i < m; ++i) {
        s = A(i,k) / A(k,k);
        A(i,all) -= s * A(k,all);
      }
  }
}

template <class Matrix, class Vector>
void gauss_elim_with_partial_pivoting(Matrix& A, Vector& pivots)
{
  typename matrix_traits<Matrix>::size_type
    m = A.nrows(), n = A.ncols(), pivot, i, k;
  typename matrix_traits<Matrix>::value_type s;

  for (k = 0; k < std::min(m-1,n-1); ++k) {
    pivot = k + max_abs_index( A(range(k,m),k) );
    if (pivot != k)
      swap(A(pivot,all), A(k,all));
    pivots[k] = pivot;
    if (A(k,k) != zero(s))
      for (i = k + 1; i < m; ++i) {
        s = A(i,k) / A(k,k);
        A(i,all) -= s * A(k,all);
      }
  }
```

```
}
```

Chapter 4

Pointwise LU Factorization

This section shows how one could implement the usual pointwise LU factorization. The next section will describe an implementation of the blocked LU factorization. First, a quick review of LU factorization. It is basically gaussian elimination, where a general matrix is transformed into a lower triangular matrix and an upper triangular matrix. The purpose of this transformation is to solve a system of equations, and it is easy to solve a system once it is in triangular form (using forward or backward substitution). So if we start with the equation $Ax = b$, using LU factorization this becomes $LUx = b$. We can then solve the system in two simple steps: first we solve $Ly = b$ (where y has replaced Ux), and then we solve $Ux = y$. For more background on LU factorization see [6] or [10].

The algorithm for LU factorization is given in Figure 4.1 and the graphical representation of the algorithm is given in Figure 4.2, as it would look part way through the computation. The black square represents the current pivot element. The horizontal shaded rectangle is a row from the upper triangle of the matrix. The vertical shaded rectangle is a column from the lower triangle of the matrix. The L and U labeled regions are the portions of the matrix that have already been updated by the algorithm. The algorithm has not yet reached the region labeled A'.

for $i = 1 \ldots min(M-1, N-1)$
  find maximum element in the column section $A(i+1:M, i)$
  swap the row of maximum element with row $A(i,:)$
  scale column section $A(i+1:M, i)$ by $1/A(i,i)$
  let $A' = A(i+1:M, i+1:N)$
  $A' \leftarrow A' + L(:,i)U(i,:)$ (rank one update)

Figure 4.1: LU factorization pseudo-code.

We will implement the LU factorization as a template function that takes a matrix input-output argument and a vector output-argument to record the

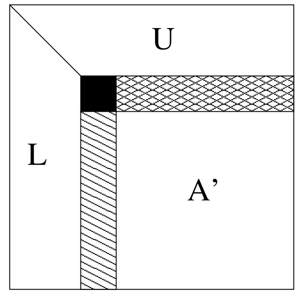

Figure 4.2: Diagram for LU factorization.

pivots. The return type in an integer that will be zero if the algorithm is successful (the matrix is non-singular), otherwise the matrix is singular and $U(i, i)$ is zero with i given as the return value. The return type is the `size_type` which is associated with the matrix using the `matrix_traits` class. For most cases one could get away with using `int` instead, but when writing a generic algorithm it is best to use this more portable method[1].

Inside the algorithm we will use many of the matrix operations specified in the SubdividableMatrix concept, so we insert a `REQUIRE` clause to ensure that the user does not do something like try to call the function with a sparse matrix (which is not a model of SubdividableMatrix). In addition, we require the `PVector` type to really be a Vector[2].

```
template <class Matrix, class PVector>
typename matrix_traits<Matrix>::size_type
lu_factor(Matrix& A, PVector& pivots)
{
  REQUIRE(Matrix, SubdividableMatrix);
  REQUIRE(PVector, Vector);
  ...
}
```

To make the indexing as simple as possible, we can create matrix objects to make explicit the upper and lower triangular views of the original matrix A. First the types of the views must be obtained. This is done using the `triangle_view` traits class. We also create a `typedef` for the sub-matrix type, which will be used later. The `U` and `L` matrix objects are then created and the matrix `A` is passed to their constructors. The `L` and `U` objects are just handles, so their creation is inexpensive (constant time complexity).

```
typedef typename triangle_view<Matrix, unit_upper>::type Unit_Upper;
typedef typename triangle_view<Matrix, unit_lower>::type Unit_Lower;
typedef typename submatrix<Matrix>::type SubMatrix;
Unit_Upper U(A);
```

[1]Often times the biggest headache in porting large codes is changing integer types. This problem can be mitigated by the correct use of traits classes and the associated `size_type`

[2]The require clauses are not really necessary and can be left out. Their purpose is to make error messages more understandable when the user incorrectly applies a generic algorithm.

```
Unit_Lower L(A);
```

Next we need to declare some index variables, again using the `matrix_traits` class to get the correct type.

```
typedef typename matrix_traits<Matrix>::size_type SizeT;
typedef typename matrix_traits<Matrix>::value_type T;
SizeT i, ip, M = A.nrows(), N = A.ncols(), info = 0;
```

The first operation in the LU factorization is to find the maximum element in the column, which will tell us how to pivot. The expression `A(i,range(i,M))` returns a subsection of the `i`th column, from the `i`th row to the bottom. The range describes a half-open interval which does not include the element `A(i,M)` (which would be out-of-bounds). The sub-column object is a full-fledge MTL Vector, and can be used with any of the MTL vector operations. The same is true for sub-rows and sub-matrices. We then apply the `abs()` function to create an expression object, which will apply `abs()` to each element of the sub-column during the evaluation of the `max_index()` function.

```
ip = max_index(abs(A(i,range(i,M))));
```

The next operation in the LU factorization is to swap the current row with the row that has the maximum element.

```
if (ip != i)
  swap((A(i,all), A(ip,all));
```

The third operation in the LU factorization is to scale the column under the pivot by $1/A(i,i)$. The use of the `identity()` function needs some explaining. Since we are writing a generic algorithm, we do not know the element type for the matrix, and therefore can not be sure that an integer constant `1` is convertible to the element type. The solution is to use the generic `identity()` function provided by MTL which returns a 1 of the appropriate type.

```
L(all,i) *= identity(T()) / A(i,i);
```

The final operation in the LU factorization is to update the trailing sub-matrix according to $A' \leftarrow A' + L(:,i)U(i,:)$.

```
SubMatrix Aprime = A(range(i+1, M), range(i+1, N));
Aprime -= L(all,i) * U(i,all);
```

The complete LU factorization implementation is given in Figure 4.3.

```
template <class Matrix, class PivotVector>
typename matrix_traits<Matrix>::size_type
lu_factor(Matrix& A, PivotVector& pivots)
{
  typedef typename triangle_view<Matrix, unit_upper>::type Unit_Upper;
  typedef typename triangle_view<Matrix, unit_lower>::type Unit_Lower;
  typedef typename submatrix<Matrix>::type SubMatrix;
  typedef typename matrix_traits<Matrix>::size_type SizeT;
  typedef typename matrix_traits<Matrix>::value_type T;

  REQUIRE(Matrix, SubdividableMatrix);
  REQUIRE(Unit_Lower, SubdividableMatrix);
  REQUIRE2(Unit_Lower, T, VectorSpace);
  REQUIRE2(SubMatrix, Ring);
  REQUIRE(PivotVector, Vector);

  Unit_Upper U(A);
  Unit_Lower L(A);
  int info = 0;
  SizeT i, ip, M = A.nrows(), N = A.ncols();

  for (i = 0; i < min(M - 1, N - 1); ++i) {
    ip = max_index(abs(A(i,range(i,M)))); /* find pivot */
    pivots[i] = ip + 1;
    if ( A(ip, i) != zero(T()) ) {        /* make sure pivot isn't zero */
      if (ip != i)
        swap((A(i,all), A(ip,all));       /* swap the rows i and ip */
      L(all,i) *= identity(T()) / A(i,i); /* update column under the pivot */
    } else {
      info = i + 1;
      break;
    }
    SubMatrix Aprime = A(range(i+1, M), range(i+1, N));
    Aprime -= L(all,i) * U(i,all);        /* update the submatrix */
  }
  pivots[i] = i + 1;
  return info;
}
```

Figure 4.3: Complete MTL version of pointwise LU factorization.

Chapter 5

Blocked LU Factorization

The execution time of many linear algebra operations on modern computer architectures can be decreased dramatically through blocking to increase cache utilization [4, 5]. In algorithms where repeated matrix-vector operations are done (such as the rank-one-update of the LU factorization), it is beneficial to convert the algorithm to use matrix-matrix operations to introduce more opportunities for blocking.

Here we give an example of how to MTL to reformulate the LU factorization algorithm to use more matrix-matrix operations [**?**]. First the matrix A is split into four submatrices, using a blocking factor of r. A_{11} is $r \times r$ and A_{12} is $r \times n - r$ where $dim(A) = n \times n$.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Then $A = LU$ is formulated in terms of the blocks.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

From this the following equations are derived for the submatrices of A. The matrix on the right shows the values that should occupy A after one step of the blocked LU factorization.

$$\begin{aligned} A_{11} &= L_{11}U_{11} \\ A_{12} &= L_{11}U_{12} \\ A_{21} &= L_{21}U_{11} \\ A_{22} &= L_{21}U_{12} + L_{22}U_{22} \end{aligned} \qquad \begin{bmatrix} L_{11} \setminus U_{11} & U_{12} \\ L_{21} & \tilde{A}_{22} \end{bmatrix}$$

L_{11}, U_{11}, and L_{21} are updated by applying the pointwise LU factorization to the combined region of A_{11} and A_{21}. U_{12} is then updated with a triangular solve applied to A_{12}. Finally $\tilde{A}_{22}$ is calculated with a matrix product of L_{21} and U_{12}. The algorithm is then applied recursively to $\tilde{A}_{22}$.

In the implementation of block LU factorization, MTL can be used to create a partitioning of the matrix into submatrices. The use of the submatrix objects throughout the algorithm removes the redundant indexing that a programmer would typically have to do to specify the regions for each submatrix for each operation. The code below shows how the partitioning is performed that corresponds to Figure 5.1 with the matrix objects `A_0`, `A_1`, and `A_2`. The partitioning for Figure 5.2 is created with matrix objects `A_11`, `A_12`, `A_21`, and `A_22`.

```
SubMatrix
  A_0 = A(range(j,M), range(0,j)),
  A_1 = A(range(j,M), range(j,j+jb)),
  A_2 = A(range(j,M), range(j+jb,N)),
  A_11 = A(range(j,j+jb), range(j,j+jb)),
  A_12 = A(range(j,j+jb), range(j+jb,N)),
  A_21 = A(range(j+jb,M), range(j,j+jb)),
  A_22 = A(range(j+jb,M), range(j+jb,N));
triangle_view<SubMatrix, unit_lower>::type L_11(A_11);
```

Figure 5.1 depicts the block factorization part way through the computation. The matrix is divided up for the pointwise factorization step. The region including A_{11} and A_{21} is labeled A_1. Since there is pivoting involved, the rows in the regions labeled A_0 and A_2 must be swapped according to the pivots used in A_1.

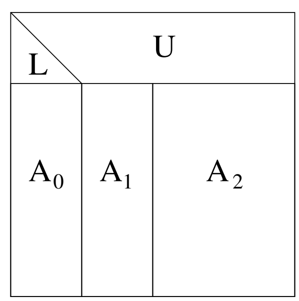

Figure 5.1: Pointwise step in block LU factorization.

The implementation of this step in MTL is very concise. The `A_1` submatrix object is passed to the `lu_factorize()` algorithm. The `multi_swap()` function is then used on `A_0` and `A_2` to pivot their rows to match `A_1`.

```
PivotVector sub_pivots(jb);
S ret = lu_factor(A_1, sub_pivots);
if (ret != 0)
  return ret + j;
for (S i = j; i < min(M, j + jb); ++i)
  pivots[i] = sub_pivots[i-j] + j;
```

```
if (j > 0)
  permute(A_0, sub_pivots, left_size());
if (j + jb < M) {
  permute(A_2, sub_pivots, left_side());
```

Once A_1 has been factored, the A_{12} and A_{22} regions must be updated. The submatrices involved are depicted in Figure 5.2. The A_{12} region needs to be updated with a triangular solve.

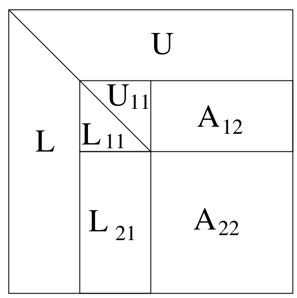

Figure 5.2: Update steps in block LU factorization.

To apply the `tri_solve()` algorithm to L_{11} and A_{12}, we merely call the MTL routine and pass in the `L_11` and `A_12` matrix objects.

```
tri_solve(L_11, A_12, left_side());
```

The last step is to calculate $\tilde{A}_{22}$ with a matrix-matrix multiply according to $\tilde{A}_{22} \leftarrow A_{22} - L_{21}U_{12}$. The `A_12` and `A_21` matrix objects are used to implement then operation. They have been overwritten in the previous steps with U_{12} and L_{21}.

```
A_22 -= A_21 * A_12;
```

The complete version of the MTL block LU factorization algorithm is given in Figure 5.3.

```
template <class Matrix, class PivotVector>
typename Matrix::size_type
block_lu(Matrix& A, PivotVector& pivots)
{
  typedef typename Matrix::value_type T;
  typedef typename Matrix::size_type S;
  typedef typename submatrix_view<Matrix>::type SubMatrix;
  const S BF = LU_BF; // blocking factor
  const S M = A.nrows();
  const S N = A.ncols();

  if (min(M, N) <= BF || BF == 1)
    return lu_factor(A, pivots);

  for (S j = 0; j < min(M, N); j += BF) {
    S jb = min(min(M, N) - j, BF);
    SubMatrix
      A_0 = A(range(j,M), range(0,j)),
      A_1 = A(range(j,M), range(j,j+jb)),
      A_2 = A(range(j,M), range(j+jb,N)),
      A_11 = A(range(j,j+jb), range(j,j+jb)),
      A_12 = A(range(j,j+jb), range(j+jb,N)),
      A_21 = A(range(j+jb,M), range(j,j+jb)),
      A_22 = A(range(j+jb,M), range(j+jb,N));
    triangle_view<SubMatrix, unit_lower>::type L_11(A_11);

    PivotVector sub_pivots(jb);
    S ret = lu_factor(A_1, sub_pivots);
    if (ret != 0)
      return ret + j;
    for (S i = j; i < min(M, j + jb); ++i)
      pivots[i] = sub_pivots[i-j] + j;
    if (j > 0)
      permute(A_0, sub_pivots, left_size());
    if (j + jb < M) {
      permute(A_2, sub_pivots, left_side());
      tri_solve(L_11, A_12, left_side());
      A_22 -= A_21 * A_12;
    }
  }
}
```

Figure 5.3: MTL version of block LU factorization.

Chapter 6

Preconditioned GMRES(m)

One important use for MTL is for the rapid construction of numerical libraries. Although not necessary, a generic approach can be used when developing these libraries as well, resulting in reusable scientific software at a higher level. To demonstrate how one might use MTL for a non-trivial high-level library, we show the complete implementation of the preconditioned GMRES(m) algorithm [9] in Figure 6.1 (taken from our Iterative Template Library).

The basic algorithmic steps (corresponding to the GMRES algorithm as given in [9]) are given in the comments and the calls to MTL in the body of the algorithm should be fairly clear. Some of the other code may seem somewhat impenetrable at first glance, so we'll take a quick walk-through the more difficult statements.

The algorithm parameterizes GMRES in some important ways, as shown in the `template` statement on lines 1 and 2. The matrix and vector types are parameterized, so that any matrix type can be used. In particular, matrices having any *element type* can be used — e.g., real or complex. In fact, matrices without explicit elements at all (matrix-free operators [2]) can be used. All that is required is for the `mult()` algorithm be suitably defined. For MTL matrices, the generic MTL `mult()` will generally suffice. For non-MTL matrices, or matrix-free operators, a suitably overloaded `mult()` must be provided.

There are also two other type parameterizations of interest, the `Preconditioner` and the `Iteration`. Similar to the parameterization of the matrix, the preconditioner type is parameterized so that arbitrary preconditioners can be used. It is only required that the preconditioner be callable with the `solve()` algorithm. The `Iteration` type parameter allows the user to control the stopping criterion for the algorithm. (Pre-defined stopping criteria are included as part of ITL.)

The `using namespace mtl` statement on line 6 allows us to access MTL functions (which are all declared within the `mtl` namespace) without explicitly using the `mtl::` scope. The use of a namespace helps to prevent name clashes with other libraries and user code.

On line 7, we use the internally defined `typedef` for the `value_type` to determine the type of the individual elements of the matrix. All MTL matrix and

```
template <class Matrix, class Vector, class VectorB,
          class Preconditioner, class Iteration>
int gmres(const Matrix &A, Vector &x, const VectorB &b,
          const Preconditioner &Minv, int m, Iteration& outer,
          Norm norm, InnerProduct dot)
{
  typedef typename Matrix::value_type T;
  typedef mtl::matrix<T, rectangle<>,
     dense<>, column_major>::type InternalMatrix;
  typedef itl_traits<Vector>::internal_vector InternalVec;

  REQUIRE4(InternalVec, T, Norm, InnerProduct, HilbertSpace);
  REQUIRE4(Matrix, Vector, VectorB, T, LinearOperator);
  REQUIRE4(Preconditioner, InternalVec, InternalVec, T, LinearOperator);

  InternalMatrix H(m+1, m), V(size(x), m+1);
  InternalVec  s(m+1), w, r, u;
  std::vector< givens_rotation<T> > rotations(m+1);

  w = b - A * x;
  r = Minv * w;
  typename Iteration::real beta = std::abs(norm(r));

  while (! outer.finished(beta)) {         // Outer iteration
    V[0] = r /beta;
    s = zero(s[0]);
    s[0] = beta;
    int j = 0;
    Iteration inner(outer.normb(), m, outer.tol());

    do {                                   // Inner iteration
      u = A * V[j];
      w = Minv * u;
      for (int i = 0; i <= j; i++) {
        H(i,j) = dot(w, V[i]);
        w -= V[i] * H(i,j);
      }
      H(j+1,j) = norm(w);
      V[j+1] = w / H(j+1,j);

      // QR triangularization of H
      for (int i = 0; i < j; i++)
        rotations[i].apply(H(i,j), H(i+1,j));

      rotations[j] = givens_rotation<T>(H(j,j), H(j+1,j));
      rotations[j].apply(H(j,j), H(j+1,j));
      rotations[j].apply(s[i], s[i+1]);

      ++inner, ++outer, ++j;
    } while (! inner.finished(std::abs(s[j])));

    // Form the approximate solution
    tri_solve(tri_view<upper>()(H(range(0, j), range(0, j))), s);
    x += V(range(0,x.size()), range(0,j)) * s;

    // Restart
    w = b - A * x;
    r = Minv * w;
    beta = std::abs(norm(r));
  }
  return outer.error_code();
}
```

Figure 6.1: The Iterative Template Library (ITL) implementation of the preconditioned GMRES(m) algorithm. This algorithm computes an approximate solution to $Ax = b$ preconditioned with M. The restart value is specified by the parameter m.

vector classes have an accessible type member called `value_type` that specifies the type of the element data. By using this internal type, rather than a fixed type, we can make the algorithm generic with respect to element type.

Finally, although the entire algorithm fits on a single page, it is not a toy implementation — it is both high-quality and high-performance. This is where the power of reusable software components can be appreciated. Now that there exists a generic GMRES(m) that has been implemented, tested, and debugged, programmers wanting to use GMRES are forever spared the work of implementation, testing, and debugging GMRES themselves. Both time and reliability are gained.

Part IV

Reference Manual

Chapter 7

Concepts

7.1 Algebraic Concepts

One of the beautiful aspects of linear algebra is that in many respects matrices and vectors act like numbers, for instance you can add and multiply them. Of course, matrices and vectors don't act exactly like numbers, there are some important differences and restrictions on the operations. And since many algorithms operate on matrices at this abstraction level (without looking "inside" the matrices) it is important to formulate a precise description of this abstraction level. Fortunately, mathematicians [7, 8, 11] have already developed a very precise definition for a *linear algebra* which we will use, merely adapting it to C++ syntax. The section following this one will describe the interface for looking "inside" the matrix and vector data structures.

The definition of a *linear algebra* is somewhat complex and relies on a family of algebraic concepts which we also define here. If you are not particularly interested in this mathematical structure, you can skip forward to the description of the LinearAlgebra concept in Section 7.1.12, which summarizes all of the operations on matrices and vectors. Later, when you encounter algorithms that use the other concepts defined here, you can return to this section for reference.

With the following concept definitions we attempt to map the purely mathematical algebraic concepts into the realm of C++ components. Due to many practical considerations the mapping is not perfect, and the mathematical concepts will be stretched and bent here and there. The overall purpose here is to define useful interfaces and terminology for use in the precise documentation of algorithms in C++. This is not a theoretical exercise for determining how closely we can model the mathematical concepts in C++.

In our formulation of these C++ concepts we have left out the mathematical concepts that do not aid in defining interfaces for real C++ components. However, much of the fine granularity present in the mathematical concepts has been retained. This granularity is useful when documenting algorithms, as it makes it easier to choose a concept that closely matches the minimal requirements for each parameter to an algorithm. For instance, most of the algorithms in ITL only use the subset of matrix operations contained in the LinearOperator concept. Figure 7.1 gives an overview of the algebraic concepts, with the arrows representing the refinement relationship.

Equality

Stating whether two objects are "equal" is somewhat of a sticky subject. To begin with, we will want to talk about whether objects are equal even if there is not an `operator==` defined for that type (it is not EqualityComparable). Also, when dealing with floating point numbers we want to talk about an equality that is much looser than the bit-level equality that is given by `operator==`. To this end we use the symbol $=_\epsilon$ to mean $a =_\epsilon b$ iff $|a - b| < \epsilon$ where ϵ is some appropriate small number for the situation (like machine epsilon). If the number is not LessThanComparable or it does not make sense to take the absolute value, then $=_\epsilon$ means that during computation, if the value on the left-hand-side was

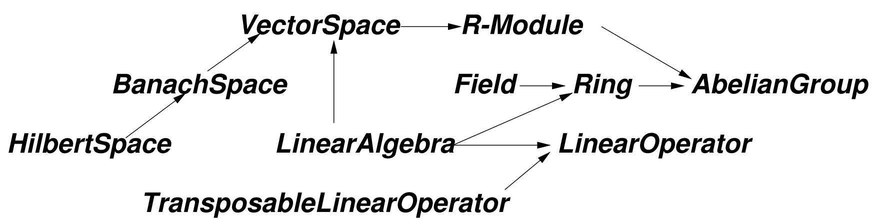

Figure 7.1: Refinement of the algebraic concepts.

substituted with the value on the right-hand-side, the difference in the resulting behaviour of the program would not be large enough to care about.

7.1.1 AbelianGroup

A *group* is a set of elements and an operator over the set that obeys the associative law and has an *identity* element. If the operator is commutative it is called an *Abelian* group, and if the notation used for the operator is + then it is an additive group. The concept AbelianGroup we define here is an additive Abelian group .

Refinement of

Assignable

Notation

`X`	is a type that is a model of AbelianGroup.
`a,b,c`	are objects of type `X`.

Valid Expressions

- Addition
  `a + b`
  Return Type: `X` or a type convertible to `X` that is also a model of AbelianGroup.
  Semantics: See below for the invariants.

- Addition Assignment
  `a += b`
  Return Type: `X&`
  Semantics: Equivalent to `a = a + b`.

- Additive Inverse
  `-a`
  Return Type: `X` or a type convertible to `X` that is also a model of AbelianGroup.
  Semantics: See below.

- Subtraction
  `a - b`
  Return Type: `X` or a type convertible to `X` that is also a model of AbelianGroup.
  Semantics: Equivalent to `a + -b`.

- Subtraction Assignment
  `a -= b`
  Return Type: `X&`
  Semantics: Equivalent to `a = a + -b`.

- Zero Element (Additive Identity)
  `zero(a)`
  Return Type: `X`
  Semantics: This function returns a zero element of the same type as the argument `a`. `a` is not changed, the purpose of the argument is merely to carry type information and also size information in the case of vectors and matrices. See below for the algebraic properties of the zero element.

Invariants

- Associativity
  `(a + b) + c` $=_\epsilon$ `a + (b + c)`

- Definition of the Identity Element
  `a + zero(a)` $=_\epsilon$ `a`

- Definition of Additive Inverse
  `a + -a` $=_\epsilon$ `zero(a)`

- Commutativity
  `a + b` $=_\epsilon$ `b + a`

Models

- `int`

- `float`

- `complex<double>`
- `vector<int>::type`
- `matrix<float>::type`

Constraints Checking Class

```
template <class X>
struct AbelianGroup {
  void constraints() {
    c = a + b;
    b += a;
    b = -a;
    c = a - b;
    b -= a;
    b = zero(a);
  }
  X a, b, c;
};
```

7.1.2 Ring

A Ring adds the notion of a second operation, namely multiplication, to the concept of a AbelianGroup. The multiplication obeys the law of associativity and distributes with addition. Adding a unity element (the multiplicative identity) to a *ring* give a *ring-with-unity*. For simplicity we will include the requirement for a unity element in the Ring concept. Another variant of the Ring concept adds the requirement that multiplication be commutative. We will refer to this concept as a CommutativeRing

Refinement of

AbelianGroup

Notation

`X`	is a type that is a model of Ring.
`a,b`	are objects of type `X`.

Requirements

- Multiplication
  `a * b`
  Return Type: convertible to type `X`. — `X` or a type convertible to `X`.

- Multiplicative Identity Element
  `identity(a)`
  Return Type: `X`
  Semantics: Returns the appropriate identity element for the type of `a`. The argument `a` is not changed, its purpose is to carry type information and also size information in the case when `a` is a matrix. The algebraic properties of the identity element are listed below.

Invariants

- Associativity of Multiplication
  `a * (b * c)` $=_\epsilon$ `(a * b) * c`
- Distributivity
  `a*(b + c)` $=_\epsilon$ `a*b + a*c`
  `(b + c)*a` $=_\epsilon$ `b*a + c*a`
- Definition of Multiplicative Identity
  `a * identity(a)` $=_\epsilon$ `a`
- Commutativity (for a CommutativeRing)
  `a * b` $=_\epsilon$ `b * a`

Models

- `int`
- `float`
- `complex<double>`
- `matrix<float>::type`

Constraints Checking Class

```
template <class X>
struct Ring {
  CLASS_REQUIRES(X, AbelianGroup);
  void constraints() {
    c = a * b;
    b = identity(a);
  }
  X a, b, c;
};
```

7.1.3 Field

A Field adds the notion that there is always a solution for the equations

$$\begin{aligned} ax &= b \\ ya &= b \quad \forall a \neq 0. \end{aligned}$$

This means that the set is closed under division, so we can add the division operator to the requirements.

Refinement of

Ring, EqualityComparable, and LessThanComparable

Notation

`X`	is a type that is a model of Field.
`a,b`	are objects of type `X`.

Valid Expressions

- Division
  `a / b`
  Return Type: `X` or a type convertible to `X`.
- Division Assignment
  `a /= b`
  Return Type: `X&`

Invariants

- Definition of Multplicative Inverse
  if `a * x` $=_\epsilon$ `b` then `x` $=_\epsilon$ b/a.

Models

- `float`
- `double`
- `complex<double>`

Constraints Checking Class

```
template <class X>
struct Field {
  CLASS_REQUIRES(X, Ring);
  CLASS_REQUIRES(X, EqualityComparable);
  CLASS_REQUIRES(X, LessThanComparable);
```

```
  void constraints() {
    c = a / b;
    b /= a;
  }
  X a, b, c;
};
```

7.1.4 R-Module

The R-Module concept defines multiplication (or "scaling") between an Abelian-Group and a Ring, where the result of the multiplication is an object of the group type. Also the multiplication must be associative and it must distribute with the addition operator of the AbelianGroup.

Refinement of

AbelianGroup

Notation

`G`	is a type that is a model of AbelianGroup.
`R`	is a type that is a model of Ring.
`a, b`	are objects of type `R`
`x`	is an object of type `X`

Participating Types

- Vector Type
  The type that plays the role of the vector (type `G`) and that is a model of AbelianGroup.
- Scalar Type
  The type that plays the role of the scalar (type `R`) and that is a model of Ring.

Valid Expressions

- Right Scalar Multiplication
  `x * a`
  Return Type: `G` or a type convertible to `G`.
- Left Scalar Multiplication
  `a * x`
  Return Type: `G` or a type convertible to `G` which also with the scalar type satisfies R-Module.
- Scalar Multiplication Assignment
  `x *= a`
  Return Type: `G&`

Invariants

- Distributive
  (a + b)*x $=_\epsilon$ a*x + b*x
  a*(x + y) $=_\epsilon$ a*x + a*y

- Associative
  a * (b * x) $=_\epsilon$ (a * b) * x

- Identity
  identity(a) * x $=_\epsilon$ x

Models

- { vector<int>::type, int }

- { matrix<int>::type, int }

Constraints Checking Class

```
template <class G, class R>
struct R_Module {
  CLASS_REQUIRES(G, AbelianGroup);
  CLASS_REQUIRES(R, Ring);
  void constraints() {
    y *= a;
    y = x * a;
    y = a * x;
  }
  G x, y;
  R a;
};
```

7.1.5 VectorSpace

The VectorSpace concept is a refinement of the R-Module concept, adding the addition requirement that the scalar type be a model of Field instead of Ring. With this we are able to define vector division by a scalar. A VectorSpace is also called a F-module.

Refinement of

R-Module

Notation

`G`	is a type that is a model of AbelianGroup.
`F`	is a type that is a model of Field.
`x`	is an object of type `G`
`a`	is an object of type `F`

Participating Types

- Vector Type
  The type that plays the role of the vector (type `G`) and that is a model of AbelianGroup.
- Scalar Type
  The type that plays the role of the scalar (type `F`) and that is a model of Field.

Valid Expressions

- Scalar Division
  `x / a`
  Return Type: `G` or a type convertible to `G`.
- Scalar Division Assignment
  `x /= a`
  Return Type: `G&`

Models

- { `vector<float>::type`, `float` }
- { `matrix<double>::type`, `double` }
- { `std::valarray<float>`, `float` }
- { `std::complex<double>`, `float` }

Constraints Checking Class

```
template <class G, class F>
struct VectorSpace {
  CLASS_REQUIRES2(G, F, R_Module);
  CLASS_REQUIRES(F, Field);
  void constraints() {
    y = x / a;
    y /= a;
  }
  G x, y;
  F a;
};
```

7.1.6 FiniteVectorSpace, FiniteBanachSpace, FiniteHilbertSpace

A finite-dimensional vector space has a basis consisting of finite number of vectors $x_1, \ldots, x_n$ where n is the *dimension* of the vector space. Any vector in such a space can be expressed in terms of n coordinates with respect to the basis. With this in mind, the concept FiniteVectorSpace requires a method of access for the coordinates of a vector and access to the dimension, namely the `operator[]` and a `size()` function. The behaviour of these operations and the associated traits information is described in BasicVector.

The next two vector space concepts, BanachSpace and HilbertSpace, also have finite-dimensional variants which we will call FiniteBanachSpace and FiniteHilbertSpace.

Refinement of

VectorSpace and BasicVector

Constraints Checking Class

```
template <class G, class F>
struct FiniteVectorSpace {
  CLASS_REQUIRES2(G, F, VectorSpace);
  CLASS_REQUIRES(G, BasicVector);
};

template <class G, class F, class Norm>
struct FiniteBanachSpace {
  CLASS_REQUIRES3(G, F, Norm, BanachSpace);
  CLASS_REQUIRES(G, BasicVector);
};

template <class G, class F, class Norm, class InnerProduct>
struct FiniteHilbertSpace {
  CLASS_REQUIRES4(G, F, Norm, InnerProduct, HilbertSpace);
  CLASS_REQUIRES(G, BasicVector);
};
```

7.1.7 BanachSpace

A BanachSpace is basically a VectorSpace composed with a definition for a *norm* function. More technically, a BanachSpace is a complete normed vector space [8]. Since one may want to use the same generic algorithm on different spaces with different norms, we specify access to the norm function through a function object which is passed to algorithms or classes that operate on a BanachSpaces[1].

[1]The MTL and ITL algorithms provide the 2-norm as a default.

Refinement of

VectorSpace

Notation

`{G,F}`	is a module that models VectorSpace.
`Norm`	is a functor type as defined below.
`x,y`	is an object of type `X`.
`a,b`	is an object of type `F`.
`r`	is an object of type `magnitude<F>::type`.
`norm`	is an object of type `Norm`.

Participating Types

- Norm Functor Type
  The `Norm` type is a function object that can be applied to vector type `X` and whose return type is `magnitude<F>::type`. In addition, the norm function satisfies the invariants listed below.

Associated Types

- Magnitude Type
  `magnitude<F>::type`
  The return type of `abs()` applied to the scalar type `F`. Typically this is some real number type.

Valid Expressions

- Norm Functor Application
  `norm(x)`
  Return Type: `magnitude<F>::type`
  Semantics: See below.

- Absolute Value
  `abs(a)`
  Return Type: `magnitude<F>::type`
  Semantics: The distance between `a` and zero.

Invariants

The `norm` functor must obey the following invariants.

- `norm(x) >= zero(r)`

- `norm(x) == zero(r) iff x == zero(x)`

- Homogeneity
  `norm(a*x)` $=_\epsilon$ `abs(a) * norm(x)`

- Triangle Inequality
  `norm(x + y) <= norm(x) + norm(y)`

The `abs()` function must obey a similar set of invariants.

- `abs(a) >= zero(r)`
- `abs(a) == zero(r) iff a == zero(a)`
- Homogeneity
  `abs(a*b)` $=_\epsilon$ `abs(a) * abs(b)`
- Triangle Inequality
  `abs(a + b) <= abs(a) + abs(b)`

Constraints Checking Class

```
template <class G, class F, class Norm>
struct BanachSpace
{
  CLASS_REQUIRES2(G, F, VectorSpace);
  void constraints() {
    r = norm(x);
    r = abs(a);
  }
  G x;
  F a;
  Norm norm;
  typename magnitude<F>::type r;
};
```

7.1.8 HilbertSpace

A HilbertSpace is basically a VectorSpace composed with an *inner product* function. The inner product is also known as dot product or scalar product. Similarly to the norm of the BanachSpace, the inner product must be provided as a function object[2].

Refinement of

BanachSpace

Notation

`{G,F}`	is a module that models BanachSpace.
`InnerProduct`	is a functor as defined below.
`x,y,z`	are objects of type `X`.
`a,b`	are objects of type `F`.
`dot`	is an object of type `InnerProduct`.

[2]MTL and ITL algorithms use `mtl::dot_functor` as a default for the function object.

Participating Types

- Inner Product Type
  The `InnerProduct` type is a function object that can be applied to two vectors of type `X` and whose return type is the scalar type `F`. In addition, the inner product satisfies the invariants listed below.

Valid Expression

- Inner Product Functor Application
  `dot(x,y)`
  Return Type: `F`
  Semantics: See below.
  Preconditions: `size(x) == size(y)` if they are finite

- Scalar Conjugate
  `conj(a)`
  Return Type: `F`
  Semantics:

- Vector Conjugate
  `conj(x)`
  Return Type: convertible to `X`
  Semantics:

Invariants

The `dot` functor obeys the following invariants.

- `dot(x,x) > zero(a)` for all `x != zero(x)`
- `dot(x,x) == zero(a)` if `x == zero(x)`
- `dot(x,y) == dot(y,conj(x))`
- `dot(a*x + b*y, z)` $=_\epsilon$ `a*dot(x,z) + b*dot(y,z)`
- `dot(x, a*y + b*z)` $=_\epsilon$ `conj(a)*dot(x,y) + conj(b)*dot(x,z)`
- `norm(x)` $=_\epsilon$ `sqrt(dot(x,x))`

The scalar `conj()` function must obey the following laws.

- `conj(a * b)` $=_\epsilon$ `conj(a) * conj(b)`
- `conj(a + b)` $=_\epsilon$ `conj(a) + conj(b)`

The vector `conj()` function must obey similar laws.

- `conj(x * y)` $=_\epsilon$ `conj(x) * conj(y)`
- `conj(x + y)` $=_\epsilon$ `conj(x) + conj(y)`

Constraints Checking Class

```
template <class G, class F, class Norm, class InnerProduct>
struct HilbertSpace
{
  CLASS_REQUIRES3(G, F, Norm, BanachSpace);
  void constraints() {
    a = dot(x, y);
    a = conj(a);
    y = conj(x);
  }
  G x, y;
  F a;
  InnerProduct dot;
};
```

7.1.9 LinearOperator

A linear operator is a function from one vector space to another. Also known as a linear transformation. The concept LinearOperator consists of an operator and two vector types, a domain and range vector type, and a scalar type.

Notation

`Op`	is the operator type.
`F`	is the scalar type which must model Field.
`{VX,F}`	is a module that models VectorSpace.
`{VY,F}`	is a module that models VectorSpace.
`A`	is an object of type `Op`
`x,w`	are objects of type `VX`
`a`	is an object of type `F`

Participating Types

- Operator Type
  The type of the operator function `Op` which can be applied to the domain vector space.

- Domain Vector Space
  A type that models VectorSpace which is the argument to the LinearOperator. An access method (via a traits class) is not provided since a given linear operator type may be applicable to many different vector types.

- Range Vector Space
  A type that models VectorSpace (or at least is convertible to such a type).

Associated Types

- Size Type
  `matrix_traits<G>::size_type`
  The Integral type that is the return type for the `nrows()` and `ncols()` functions.

Valid Expressions

- Operator Application
  `A * x`

Return Type:	a type that models the range vector space (or is convertible to it).

Invariates

- Linearity
  `A * (x + w)` $=_\epsilon$ `A * x + A * y`.
  `A * (x * a)` $=_\epsilon$ `(A * x) * a`

Constraints Checking Class

```
template <class Op, class VX, class VY, class F>
struct LinearOperator {
  CLASS_REQUIRES2(VX, F, VectorSpace);
  CLASS_REQUIRES2(VY, F, VectorSpace);
  void constraints() {
    y = A * x;
  }
  Op A;
  VX x;
  VY y;
};
```

7.1.10 FiniteLinearOperator

A FiniteLinearOperator is a LinearOperator the operates over finite vector spaces.

Notation

`Op`	is the operator type.
`F`	is the scalar type which must model Field.
`{VX,F}`	is a module that models VectorSpace.
`{VY,F}`	is a module that models VectorSpace.
`A`	is an object of type `Op`
`x`	is an object of type `VX`

Associated Types

- Size Type
  `matrix_traits<Op>::size_type`
  The Integral type that is the return type for the `nrows()` and `ncols()` functions.

Valid Expressions

- Operator Application
  `A * x`

Return Type:	a type that models the range vector space (or is convertible to it).
Preconditions:	`ncols(A) == size(x)`
Semantics:	The returned vector will have size equal to `nrows(A)`.

- Number of Rows
  `nrows(A)`

Return Type:	`matrix_traits<X>::size_type`
Semantics:	The dimension of the resulting vector space.

- Number of Columns
  `ncols(A)`

Return Type:	`matrix_traits<X>::size_type`
Semantics:	The dimension of the input vector space.

Constraints Checking Class

```
template <class Op, class VX, class VY, class F>
struct FiniteLinearOperator {
  CLASS_REQUIRES2(Op, VX, VY, F, LinearOperator);
  void constraints() {
    m = nrows(A);
    n = ncols(A);
  }
  Op A;
  typename matrix_traits<Op>::size_type m, n;
};
```

7.1.11 TransposableLinearOperator

A TransposableLinearOperator is simply a LinearOperator for which the transpose of the linear operator can also be applied.

Refinement of

LinearOperator

Notation

`X`	is a type models TransposableLinearOperator.
`V`	is a type that models VectorSpace.
`A`	is an object of type `X`
`y`	is an object of type `V`

Associated Types

In addition to the associated types inherited from LinearOperator we have:

- Transpose Linear Operator Type
  `transpose_view<X>::type`
  The type returned by `trans(A)`, which is a transposed view into the matrix.

Valid Expressions

- Transposed Operator Application (Transposed Matrix-Vector Multiplication)
  `trans(A) * y`

Return Type:	some type that models VectorSpace or is convertible to one.
Preconditions:	`nrows(A) == size(y)`
Semantics:	The resulting vector (if it is finite) will have size equal to `ncols(A)`.

- Transposed Operator Application (Left Matrix-Vector Multiplication)
  `y * A`

Return Type:	The return type will be a model of VectorSpace (or at least convertible to one)
Preconditions:	`nrows(A) == size(y)`
Semantics:	The resulting vector (if it is finite) will have size equal to `ncols(A)`.

Constraints Checking Class

```
template <class X, class VX, class VY, class F>
struct TransposableLinearOperator {
  CLASS_REQUIRES4(X, VX, VY, F, LinearOperator);
  void constraints() {
    typedef typename transpose_view<X>::type Transpose;
    Transpose AT = trans(A);
    y = AT * x;
    y = x * A;
  }
  X A;
  VX x;
  VY y;
};
```

7.1.12 LinearAlgebra

The LinearAlgebra adds several operations to the LinearOperator concept. With the scalar multiplication and addition operators the LinearOperator forms a VectorSpace, and with multiplication the LinearOperator forms a Ring. The multiplication operator associated with this Ring is the composition of linear operators, which in the context of MTL is matrix multiplication. One of the requirements for a Ring is that it be closed under multiplication. Matrix multiplication is only closed for square matrices. In the general case of rectangular matrices none of the matrices involved in the multiplication are over the same vector space (for $C = AB$, A maps $R^m \rightarrow R^k$, B maps $R^k \rightarrow R^n$, and C maps $R^m \rightarrow R^n$). We gloss over this distinction and include rectangular matrices in the LinearAlgebra concept.

The LinearAlgebra concept does not introduce any new requirements, it is just a composition of the algebraic concepts discussed in this chapter. However, to help clarify the definition of this rather large concept, here we list the complete set of requirements. The usual definition for the computation of these operations is listed with each operator merely as a reminder to the reader and does not require that the operation be implemented in that manner. We use t to denote the temporary vector or matrix resulting from some of the operations [3].

Refinement

LinearOperator, VectorSpace, and Ring

Notation

`Matrix`	is a type that models TransposableLinearOperator.
`Vector`	is a type that models VectorSpace.
`A,B`	is an object of type `Matrix`.
`r, c`	are objects of type `Matrix`, which in this case is a VectorMatrix (a row or column vector).
`x,y`	are objects of type `Vector`.
`a`	is an object of type `Scalar`.

Participating Types

- The Matrix Type
- The Vector Type
- The Scalar Type

[3] Due to the expression template optimization that MTL performs, in many cases the temporary is not actually formed

Valid Expressions

- Vector Addition ($t_i = x_i + y_i$)
  `x + y`
  Preconditions: `size(x) == size(x)` if `x` is finite-dimensional.

- Vector Addition Assignment ($x_i = x_i + y_i$)
  `x += y`
  Preconditions: `size(x) == size(y)`

- Vector Additive Inverse ($t_i = -x_i$)
  `-x`

- Vector Subtraction ($t_i = x_i - y_i$)
  `x - y`
  Preconditions: `size(x) == size(y)`

- Vector Subtraction Assignment ($x_i = x_i - y_i$)
  `x -= y`
  Preconditions: `size(x) == size(x)`

- Zero Vector ($t_i = 0$)
  `zero(x)`
  Semantics: This function returns a zero vector of the same type as the argument `x`. `x` is not changed, the purpose of the argument is merely to carry type and size information.

- Vector-Scalar Multiplication ($t_i = x_i\alpha$)
  `x * a`
  Return Type: `Vector` or a type convertible to `Vector` that is a model of VectorSpace.

- Scalar-Vector Multiplication ($t_i = \alpha x_i$)
  `a * x`
  Return Type: `Vector` or a type convertible to `Vector`. that is a model of VectorSpace.

- Vector-Scalar Multiplication Assignment ($x_i = x_i\alpha$)
  `x *= a`
  Return Type: `Vector&`

- Matrix Addition ($t_{ij} = a_{ij} + b_{ij}$)
  `A + B`
  Preconditions: `nrows(A) == nrows(B) && ncols(A) == ncols(B)`

- Matrix Addition Assignment ($b_{ij} = b_{ij} + a_{ij}$)
  `B += A`
  Preconditions: `nrows(A) == nrows(B) && ncols(A) == ncols(B)`

- Matrix Additive Inverse ($t_{ij} = -a_{ij}$)
  `-A`

- Matrix Subtraction ($t_{ij} = a_{ij} - b_{ij}$)
  `A - B`
  Preconditions: `nrows(A) == nrows(B) && ncols(A) == ncols(B)`

- Matrix Subtraction Assignment ($b_{ij} = b_{ij} - a_{ij}$)
  `B -= A`
  Return Type: `Matrix&`
  Preconditions: `nrows(A) == nrows(B) && ncols(A) == ncols(B)`

- Zero Matrix ($t_{ij} = 0$)
  `zero(A)`
  Return Type: `Matrix`
  Semantics: This function returns a zero matrix of the same type as the argument `A`. `A` is not changed, the purpose of the argument is merely to carry type and size information.

- Matrix-Scalar Multiplication ($t_{ij} = a_{ij}\alpha$)
  `A * a`

- Scalar-Matrix Multiplication ($t_{ij} = \alpha a_{ij}$)
  `a * A`

- Matrix Scalar Multiplication Assignment ($a_{ij} = a_{ij}\alpha$)
  `A *= a`
  Return Type: `Matrix&`

- Matrix-Vector Multiplication ($t_i = \sum_i a_{ij} x_j$)
  `A * x`
  Preconditions: `ncols(A) == size(x)`
  Semantics: The return type convertible to a model of `Vector`, and the size is equal to `nrows(A)`.

- Transposed Matrix-Vector Multiplication ($t_j = \sum_j a_{ij} y_i$)
  `trans(A) * y`
  Preconditions: `nrows(A) == size(y)`
  Semantics: The return type convertible to a model of Vector, and the size is equal to `ncols(A)`.

- Left Matrix-Vector Multiplication ($t_j = \sum_j y_i a_{ij}$)
  `y * A`
  Preconditions: `nrows(A) == size(y)`
  Semantics: The return type convertible to a model of Vector, and the size is equal to `ncols(A)`.

- Inner Product ($\sum_i r_i c_i$)
  `r * c`
  Return Type: `Scalar`
  Preconditions: `size(r) == size(c)`

- Outer Product ($t_{ij} = c_i r_j$)
  `c * r`
  Semantics: The return type is a model of Matrix with dimensions `(size(c),size(r))`. The matrix defaults to be row-major, and is always dense.

- Matrix Multiplication ($t_{ij} = \sum_k a_{ik} b_{kj}$)
  `A * B`
  Preconditions: `ncols(A) == nrows(B)`
  Semantics: The return type is a model of Matrix with dimensions `(nrows(A),ncols(B))`.

- Identity Matrix ($t_{ii} = 1$)
  `identity(A)`
  Return Type: `Matrix`
  Semantics: Returns the identity matrix for the same type as `A`. The argument `A` is not changed, its purpose is to carry type and size information for the creation of the identity matrix.

- Matrix Transpose ($t_{ij} = a_{ji}$)
  `trans(A)`
  Return Type: `transpose_view<Matrix>::type`
  Semantics: Returns a transposed view of the matrix, where `A(i,j)` appears to be `A(j,i)`.

- Row and Column Vector Transpose
  `trans(r)` and
  `trans(c)`
  Return Type: `transpose_view<Matrix>::type`
  Semantics: If `Matrix` is a row it becomes a column and vice-versa.

Constraints Checking Class

```
template <class Op, class VX, class VY, class F>
struct LinearAlgebra
{
  CLASS_REQUIRES4(Op, VX, VY, F, LinearOperator);
  CLASS_REQUIRES2(Op, F, VectorSpace);
  CLASS_REQUIRES(Op, Ring);
};
```

7.2 Collection Concepts

7.2.1 Collection

A Collection is a *concept* similar to the STL Container concept. A Collection provides iterators for accessing a range of elements and provides information about the number of elements in the Collection. However, a Collection has fewer requirements than a Container. The motivation for the Collection concept is that there are many useful Container-like types that do not meet the full requirements of Container, and many algorithms that can be written with this reduced set of requirements. To summarize the reduction in requirements:

- It is not required to "own" its elements: the lifetime of an element in a Collection does not have to match the lifetime of the Collection object, though the lifetime of the element should cover the lifetime of the Collection object.
- The semantics of copying a Collection object is not defined (it could be a deep or shallow copy or not even support copying).
- The associated reference type of a Collection does not have to be a real C++ reference.

Because of the reduced requirements, some care must be taken when writing code that is meant to be generic for all Collection types. In particular, a Collection object should be passed by-reference since assumptions can not be made about the behaviour of the copy constructor.

Associated types

- Value type
  `X::value_type`
  The type of the object stored in a Collection. If the Collection is *mutable* then the value type must be Assignable. Otherwise the value type must be CopyConstructible.
- Iterator type
  `X::iterator`
  The type of iterator used to iterate through a Collection's elements. The iterator's value type is expected to be the Collection's value type. A conversion from the iterator type to the const iterator type must exist. The iterator type must be an InputIterator.
- Const iterator type
  `X::const_iterator`
  A type of iterator that may be used to examine, but not to modify, a Collection's elements.

- Reference type
  `X::reference`
  A type that behaves like a reference to the Collection's value type. [4]

- Const reference type
  `X::const_reference`
  A type that behaves like a const reference to the Collection's value type.

- Pointer type
  `X::pointer`
  A type that behaves as a pointer to the Collection's value type.

- Distance type
  `X::difference_type`
  A signed integral type used to represent the distance between two of the Collection's iterators. This type must be the same as the iterator's distance type.

- Size type
  `X::size_type`
  An unsigned integral type that can represent any nonnegative value of the Collection's distance type.

Notation

`X`	A type that is a model of Collection.
`a`, `b`	Object of type `X`.
`T`	The value type of `X`.

Valid expressions

- Beginning of range
  `a.begin()`

Return Type:	`iterator` if `a` is mutable, `const_iterator` otherwise
Semantics:	Returns an iterator pointing to the first element in the Collection.
Postcondition:	`a.begin()` is either dereferenceable or past-the-end. It is past-the-end if and only if `a.size() == 0`.

[4]The reference type does not have to be a real C++ reference. The requirements of the reference type depend on the context within which the Collection is being used. Specifically it depends on the requirements the context places on the value type of the Collection. The reference type of the Collection must meet the same requirements as the value type. In addition, the reference objects must be equivalent to the value type objects in the Collection (which is trivially true if they are the same object). Also, in a mutable Collection, an assignment to the reference object must result in an assignment to the object in the Collection (again, which is trivially true if they are the same object, but non-trivial if the reference type is a proxy class).

- End of range
  `a.end()`
  Return Type: `iterator` if `a` is mutable, `const_iterator` otherwise
  Semantics: Returns an iterator pointing one past the last element in the Collection.
  Postcondition: `a.end()` is past-the-end.

- Size
  `a.size()`
  Return Type: `size_type`
  Semantics Returns the size of the Collection, i.e., the number of elements.
  Poscondition: `a.size() >= 0`

- Maximum size
  `a.max_size()`
  Return Type: `size_type`
  Semantics: Returns the largest size that this Collection can ever have.
  Postcondition: `a.max_size() >= 0 && a.max_size() >= a.size()`

- Empty Collection
  `a.empty()`
  Return Type: Convertible to `bool`
  Semantics: Equivalent to `a.size() == 0`. (But possibly faster.)

- Swap
  `a.swap(b)`
  Return Type: `void`
  Semantics: Equivalent to `swap(a,b)`

Complexity guarantees

`begin()` and `end()` are amortized constant time. `size()` is at most linear in the Collection's size. `empty()` is amortized constant time. `swap()` is at most linear in the size of the two collections.

Invariants

- Valid range
  For any Collection `a`, `[a.begin(), a.end())` is a valid range.

- Range size
  `a.size()` is equal to the distance from `a.begin()` to `a.end()`.

- Completeness
  An algorithm that iterates through the range `[a.begin(), a.end())` will pass through every element of `a`.

Models

- `boost::array`
- `boost::array_ptr`
- `std::vector<bool>`
- `mtl::vector<T>::type`
- `mtl::matrix_traits<Matrix>::OneD` where `Matrix` is some type that models the MTL Matrix concept.

7.2.2 ForwardCollection

The elements are arranged in some order that does not change spontaneously from one iteration to the next. As a result, a ForwardCollection is EqualityComparable and LessThanComparable. In addition, the iterator type of a ForwardCollection is a MultiPassInputIterator which is just an InputIterator with the added requirements that the iterator can be used to make multiple passes through a range, and that if `it1 == it2` and `it1` is dereferenceable then `++it1 == ++it2`. The ForwardCollection also has a `front()` method.

Refinement of

Collection

Valid Expressions

- Front
  `a.front()`
  Return Type: `reference` if `a` is mutable, `const_reference` otherwise.
  Semantics: Equivalent to `*(a.first())`.

7.2.3 ReversibleCollection

The container provides access to iterators that traverse in both directions (forward and reverse). The iterator type must meet all of the requirements of BidirectionalIterator except that the reference type does not have to be a real C++ reference. The ReversibleCollection adds the following requirements to those of ForwardCollection.

Refinement of

ForwardCollection

Valid Expressions

- Beginning of range
  `a.rbegin()`
  Return Type: `reverse_iterator` if `a` is mutable, `const_reverse_iterator` otherwise.
  Semantics: Equivalent to `reverse_iterator(a.end())`.

- End of range
  `a.rend()`
  Return Type: `reverse_iterator` if `a` is mutable, `const_reverse_iterator` otherwise.
  Semantics: Equivalent to `X::reverse_iterator(a.begin())`.

- Back
  `a.back()`
  Return Type: `reference` if `a` is mutable, ¡br¿ `const_reference` otherwise.
  Semantics: Equivalent to `*(--a.end())`.

7.2.4 SequentialCollection

Refinement of ReversibleCollection. The elements are arranged in a strict linear order. No extra methods are required.

7.2.5 RandomAccessCollection

The iterators of a RandomAccessCollection satisfy all of the requirements of RandomAccessIterator except that the reference type does not have to be a real C++ reference. In addition, a RandomAccessCollection provides an element access operator.

Refinement of

SequentialCollection

Valid Expressions

- Element Access
  `a[n]`
  Return Type: `reference` if `a` is mutable, `const_reference` otherwise.
  Semantics: Returns the nth element of the collection.
  Precondition: `n` must be convertible to `size_type` and `0 <= n && n < a.size()`.

7.3 Iterator Concepts

7.3.1 IndexedIterator

Refinement of

ForwardIterator

Notation

`X`	is a type that is a model of IndexedIterator.
`i`	is an object of type `X`.

Associated Types

- Size Type
  `X::size_type`
  The return type of the `index()` member function.

Valid Expressions

- Element Index
  `i.index()`

Return Type:	`X::size_type`
Semantics:	Returns the index associated with the element currently pointed to by the iterator `i`.

7.3.2 MatrixIterator

Refinement of

IndexedIterator

Notation

`X`	is a type that is a model of MatrixIterator.
`i`	is an object of type `X`.

Valid Expressions

- Row Index
  `i.row()`

Return Type:	`difference_type`
Semantics:	Returns the row index associated with the element currently pointed to by the iterator `i`.

- Column Index
  `i.column()`

Return Type:	`difference_type`
Semantics:	Returns the column index associated with the element currently pointed to by the iterator `i`.

7.3.3 IndexValuePairIterator

Valid Expressions

- Index
  `index(*i)`
  Return Type: `difference_type`
  Semantics: Returns the index associated with the element currently pointed to by the iterator `i`.

- Value
  `value(*i)`
  Return Type: `value_type`
  Semantics: Returns the value associated with the element currently pointed to by the iterator `i`.

7.4 Vector Concepts

7.4.1 BasicVector

A BasicVector provides a mapping from a set of indices to the associated elements. The indices do not have to form a contiguous range though the indices must fall between zero and `size(x)`. An access to `x[i]` where `i >= size(x)` is considered out-of-bounds. We add a few useful traits such as whether the vector is sparse or dense and if the vector has static size, what that size is.

Associated Types

- Linear Algebra Category
  `linalg_category<X>::type`
  For vectors this is `vector_tag` unless the vector is also a matrix (such as a row or column) in which case this is `matrix_tag`.

- Value Type
  `vector_traits<X>::value_type`

- Reference Type
  `vector_traits<X>::reference`

- Const Reference Type
  `vector_traits<X>::const_reference`

- Pointer Type
  `vector_traits<X>::pointer`

- Const Pointer Type
  `vector_traits<X>::const_pointer`

- Size Type
  `vector_traits<X>::size_type`

- Sparsity
  `vector_traits<X>::sparsity`
  A Vector can be either sparse (`sparse_tag` or dense (`dense_tag`).

- Static Size
  `vector_traits<X>::static_size`
  If the vector has static size (size determined at compile-time) then `static_size` gives the length of the vector. Otherwise `static_size` has the value `dynamic_sized`.

Notation

`X`	is a type that is a model of BasicVector.
`x`	is an object of type `X`.
`i`	is an object of type `vector_traits<X>::size_type`.

Valid expressions

- Element Access
  `x[i]`
  Return Type: `vector_traits<X>::reference` if `x` is mutable, `vector_traits<X>::const_reference` otherwise
  Semantics: returns the element with index `i`.

- Size
  `size(x)`
  Return Type: `vector_traits<X>::size_type`
  Semantics: Returns the extent of the index set for the vector. For sparse vectors `size(x)` will typically be much larger the number of stored elements.

Complexity Guarantees

Unlike RandomAccessContainer, the BasicVector concept does not guarantee amortized constant time for element access (`operator[]`) since that would rule out sparse vectors. Element access is only guaranteed to be linear time in the number of non-zeroes in the vector. The most efficient method and also the preferred method for accessing elements of sparse vectors is to use an iterator which is introduced in the Vector concept.

Models

- `std::valarray<double>`
- `std::vector<double>`
- `mtl::vector<double>::type`

Constraints Checking Class

```
template <class X>
struct BasicVector
{
  typedef typename linalg_category<X>::type category;
  typedef typename vector_traits<X>::sparsity sparsity;
  enum { CATEGORY = linalg_category<X>::RET,
         SPARSITY = vector_traits<X>::SPARSITY,
         SIZE = vector_traits<X>::SIZE };
  typedef typename vector_traits<X>::size_type size_type;
  typedef typename vector_traits<X>::value_type value_type;
  typedef typename vector_traits<X>::reference reference;
  typedef typename vector_traits<X>::const_reference const_reference;
  typedef typename vector_traits<X>::pointer pointer;
  typedef typename vector_traits<X>::const_pointer const_pointer;

  void constraints() {
```

```
    reference r = x[n];
    const_constraints(x);
  }
  void const_constraints(const X& x) {
    const_reference r = x[n];
    n = size(x);
  }
  X x;
  size_type n;
};
```

7.4.2 Vector

This describes the MTL Vector concept, which is not to be confused with the `std::vector<T,Alloc>` class or the family of `mtl::vector<T,Storage,Orien>::type` classes. All of the vector classes in MTL model this Vector concept. That is, they fulfill the requirements (member functions and associated types) described here. The MTL Vector concept is a refinement of ForwardCollection (not Container), and BasicVector which adds the property that each element of a Vector has a unique corresponding index. The index can be used to access the corresponding element through the vector's bracket operator (i.e., `x[i]`). The elements do not have to be sorted by their index, and the indices do not necessarily have to start at zero (though they often are sorted and start at zero).

Refinement of

ForwardCollection and BasicVector

Invariants

The invariant `x[i] == *(x.begin() + i)` that applies to RandomAccessContainer does not apply to Vector, since the `x[i]` is defined for Vector to return the element with the `i`th index with is not required to be the element at position `i` of the container.

Models

- `mtl::vector<double>::type`

Constraints Checking Class

```
template <class X>
struct Vector
{
  CLASS_REQUIRES(X, ForwardCollection);
  CLASS_REQUIRES(X, BasicVector);
};
```

7.4.3 SparseVector

The SparseVector concept adds several operations that are typically needed for sparse vectors. The number of non-zeroes (number of stored elements) in the vector can be accessed with the `nnz(x)` expression. The index of an element in the sparse vector can be obtained from the iterator, using the `index()` member function. For example, to print out the elements of the vector as index-value pairs one can write this:

```
template <class SparseVector>
void print_sparse_vector(const SparseVector& x) {
  typename SparseVector::const_iterator i;
  for (i = x.begin(); i != x.end(); ++i)
    cout << "(" << i.index() << "," << *i << ") ";
}
```

If one wishes to examine only the indices of the sparse vector, the `nz_struct(x)` function can be used to obtain a view of the element indices.

```
template <class SparseVector>
void print_vector_indices(const SparseVector& x) {
  typedef typename nonzero_structure<SparseVector>::type NzStruct;
  typename NzStruct::const_iterator i;
  NzStruct x_nz = nz_struct(x);
  for (i = x_nz.begin(); i != x_nz.end(); ++i)
    cout << *i << " ";
}
```

Constraints Checking Class

```
template <class X>
struct SparseVector
{
  CLASS_REQUIRES(X, Vector);
  typedef typename X::iterator iterator;
  typedef typename X::const_iterator const_iterator;
  CLASS_REQUIRES(iterator, IndexedIterator);
  CLASS_REQUIRES(const_iterator, IndexedIterator);
  typedef typename nonzero_structure<X>::type NonZeroStruct;
  CLASS_REQUIRES(NonZeroStruct, Collection);
  typedef typename X::size_type size_type;

  void constraints() {
    n = nnz(x);
    z = nonzero_structure(x);
  }
  X x;
  size_type n;
  NonZeroStruct z;
};
```

Refinement of

Vector

Associated Types

- Iterator
  `X::iterator`
  The iterator must be a model of IndexedIterator.
- Non-Zero Structure Type
  `nonzero_structure<X>::type`

Valid expressions

- Number of Non-Zeroes
  `nnz(x)`

Return Type:	`X::size_type`
Semantics:	returns the number of stored elements in the vector. Note that if an element stored in the vector happens to be zero it is still counted towards the `nnz`. For dense vectors, `nnz(x) == size(x)`. For sparse vectors `nnz(x)` is typically much smaller than `size(x)`.

- Non-Zero Structure Access
  `nz_struct(x)`

Return Type:	`nz_struct<X>::type`
Semantics:	Returns a Collection that consists of the indices of the elements in the vector `x`. The `size()` of the collection is `nnz(x)`.

Invariants

`x[i] == *iter` if and only if `iter.index() == i`.

Complexity Guarantees

The `nnz()` an `nz_struct()` functions are amortized constant time.

Models

- `mtl::vector<float, compressed<> >::type`
- `mtl::vector<float, sparse_pair<> >::type`
- `mtl::vector<float, tree<> >::type`

7.4.4 SubdividableVector

Constraints Checking Class

```
template <class X>
struct SubdividableVector
{
  CLASS_REQUIRES(X, Vector);
  void constraints() {
    sub_x = x[range(s,f)];
  }
  typedef typename X::size_type size_type;
  size_type s, f;
  X x;
  typename subvector<X>::type sub_x;
};
```

Refinement of

Vector

Associated Types

- Sub-Vector Type
  `subvector<X>::type`
  The type used to represent sub-vector "views" of the vector.

Notation

`X`	is a type that is a model of Vector.
`x`	is an object of type `X`.
`r`	is an object of type `std::pair<size_type,size_type>`.

Valid expressions

- Sub-Vector Access
  `x[r]`

Return Type:	`subvector<X>::type`
Semantics:	returns a sub-vector "view" of `a`. The region is defined by the half-open interval `[r.first,r.second)`. That is, the region includes `x[r.first]` but not `x[r.second]`. The `range()` funciton can be used for conveniently creating the `r` argument from a pair of indices.

Complexity Guarantees

- The sub-range access is only guaranteed to be linear time (but is typically constant time for dense vectors).

7.4.5 ResizableVector

Constraints Checking Class

```
template <class X>
  struct ResizableVector
  {
    CLASS_REQUIRES(X, Vector);
    typedef typename X::size_type size_type;
    void constraints() {
      x.resize(n);
    }
    X x;
    size_type n;
  };
```

Refinement of

Vector

Valid expressions

- Resize Vector
  `x.resize(n)`

Return Type:	`void`
Semantics:	Changes the size of the vector to `n`. New elements will be initialized to zero.

Complexity Guarantees

- The `resize` function is guaranteed to be linear in the size of the vector.

7.5 Matrix Concepts

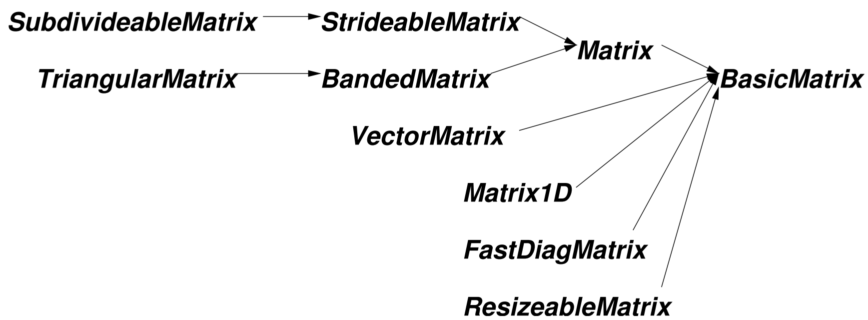

Figure 7.2: Refinement of the matrix concepts.

7.5.1 BasicMatrix

The BasicMatrix concept defines the associated types and operations that are common to all MTL matrix types. A *matrix* consists of *elements*, each of which has a row and column index. The expression `A(i,j)` returns the element with row index `i` and column index `j`. For most matrix algorithms, one needs to traverse through all the of elements of a matrix. One could use `A(i,j)` to do this, but for some matrix types this is inefficient (e.g., sparse matrices) or can be confusing (e.g., banded matrices). The iterators provided by the Matrix and Matrix1D concepts provide a better method for efficiently traversing a matrix, and the SubdividableMatrix concept defines a nice interface for accessing slices and sub-matrices.

There are several traits classes used with the BasicMatrix concept: `linalg_traits`, and `matrix_traits`. The `linalg_traits` class is shared with the MTL vector and scalar concepts. For each of the tags defined in these traits classes there is also a numerical constant defined, for example `matrix_traits<X>::shape` and `matrix_traits<X>::SHAPE`. Both the tag and the constant are provided because the type tag is needed to implement external polymorphism (dispatching based to tag categories), while the constant is more convenient to use in compile-time (template meta-programming) logic.

Associated Types

The `matrix_traits` class provides access to the associated types of a BasicMatrix, though the `linalg_traits` class can also be used for some of the associated types.

- Category
  `linalg_category<X>::type` and `linalg_category<X>::RET`
  For matrices this is always `matrix_tag` and `MATRIX`.

- Element Type
  matrix_traits<X>::element_type
  The type of the elements contained in the matrix. The choice of naming this element_type over value_type is to help differentiate the element type from the 1-D type which for most MTL matrices is given by X::value_type (as they are a 2-D Collection).

- Reference
  matrix_traits<X>::reference The mutable reference type associated with the element type.

- Const Reference
  matrix_traits<X>::const_reference The constant (immutable) reference type associated with the element type.

- Size Type
  matrix_traits<X>::size_type
  The type used for expressing matrix indices and dimensions.

- Sparsity
  matrix_traits<X>::sparsity
  Access whether the matrix is dense or sparse (dense_tag orsparse_tag).

- Dimension
  matrix_traits<X>::dimension
  Access the dimension of the matrix (oned_tag or twod_tag).

- Static Number of Rows
  matrix_traits<X>::static_nrows
  If the matrix is static (size determined at compile-time) then static_nrows gives the number of rows in the matrix. Otherwise static_nrows is dynamic_sized.

- Static Number of Columns
  matrix_traits<X>::static_ncols
  If the matrix is static (size determined at compile-time) then static_ncols gives the number of rows in the matrix. Otherwise static_ncols is dynamic_sized.

- Shape
  matrix_traits<X>::shape
  The general layout of where the non-zeroes appear in the matrix. The possible types are rectangle_tag, banded_tag, triangle_tag, symmetric_tag, hermitian_tag, or diagonal_tag.

- Orientation
  matrix_traits<X>::orientation
  The natural order of traversal of the matrix, either row_tag, column_tag, or diagonal_tag, or no_orientation_tag.

Notation

`X`	is a type that is a model of BasicMatrix.
`A`	is an object of type `X`.
`i,j`	are objects of type `matrix_traits<X>::size_type`.

Valid Expressions

- Element Access
  `A(i,j)`
  Return Type: `reference` if `A` is mutable, `const_reference` otherwise
  Semantics: Returns the element at row `i` and column `j`.

- Number of Rows
  `nrows(A)`
  Return Type: `size_type`

- Number of Columns
  `ncols(A)`
  Return Type: `size_type`

- Number of Non-Zeroes
  `nnz(A)`
  Return Type: `size_type`
  Semantics: Returns the number of stored elements in the matrix. Note that if an element that is stored in the matrix happens to be zero it is still counted towards the `nnz`. For dense matrices `nnz(A) == nrows(A) * ncols(A)`. For sparse and banded matrices `nnz(A)` is typically much small than `nrows(A) * ncols(A)`.

Complexity Guarantees

- Element access is only guaranteed to be $O(mn)$. The time complexity for this operation varies widely from matrix type to matrix type. For dense matrices it is constant, while for coordinate scheme sparse matrices it is on averange $mn/2$. See the documentation for each matrix type for the specific time complexity.

- The `nrows()`, `ncols()`, and `nnz()` members are all constant time.

7.5.2 VectorMatrix

A VectorMatrix is basically a vector that is also a matrix. Examples of this include a row or column section of a matrix or a free-standing vector which is being used to represent a *row matrix* or *column matrix* (a matrix consisting of a single row or column). MTL vectors are by default considered to be a *column matrix* and have `column_tag` for their `orientation`. For the most part, the properties of the VectorMatrix derive from being a Vector, though the VectorMatrix also fulfills the requirements for BasicMatrix. The `iterator` and `const_iterator` types of the VectorMatrix must model MatrixIterator.

Refinement of

Vector and BasicMatrix

Associated Types

VectorMatrix inherits its associated types from Vector and BasicMatrix.

Valid Expressions

- Transposed View of the Vector
  `trans(x)`

Return Type:	`transpose_view<X>::type`
Semantics:	Creates a transposed view of the vector. If the vector was a row it is now a column and vice-versa.
Complexity:	Constant time.

Models

- `matrix<double>::type::value_type` (a row of a dense matrix)
- `vector<double>::type`

7.5.3 Matrix1D

A Matrix1D is a matrix that provides an iterator type that traverses all of the elements of the matrix in one pass. Dereferencing the iterator gives an element of the matrix, so `std::iterator_traits<X::iterator>::value_type` is the same type as `matrix_traits<X>::value_type`. The iterator type must be a model of MatrixIterator.

Refinement of

BasicMatrix and Collection

7.5.4 Matrix

The main interface to the Matrix concept is basically that of a two-dimensional Collection (which is very similar to the STL Container concept). It has a `begin()` and `end()` method which provides access to 2-D iterators. The 2-D iterators dereference to give 1-D sections of the matrix, which also have `begin()` and `end()` methods for access to the 1-D iterators. The 1-D iterators dereference to give matrix elements. The 1-D iterators (which model MatrixIterator) provide access to the row and column index of each element, through the `row()` and `column()` methods. These iterators provide an efficient traversal method for any

MTL matrix type, and usually correspond to the "natural" traversal order given by how the matrix is stored in memory. For matrices with special structure, such as banded or sparse matrices, only the *stored* elements are traversed.

If the 1-D iterators traverse down each row, then we call the matrix *row-oriented*. If the traversal goes down each column, we call the matrix *column-oriented*. Some MTL matrices are even *diagonally-oriented*. The 1-D sections of a matrix can also be accessed via the bracket operator `A[i]` (which gives the `i`th 1-D section). The type of the 1-D section is given by `X::value_type` (where `X` is some matrix type) which is a model of VectorMatrix. The element type of the matrix is accessed through `matrix_traits<X>::value_type` as described in BasicMatrix.

Example

Print out the elements of a matrix.

```
template <class Matrix>
void print_matrix(const Matrix& A)
{
  typedef typename matrix_traits<A>::const_iterator Iter2D;
  typedef typename A::OneD::const_iterator Iter1D;

  for (Iter2D i = A.begin(); i != A.end(); ++i)
   for (Iter1D j = (*i).begin(); j != (*i).end(); ++j)
     cout << "(" << j.row() << "," << j.column() << ") =" << *j << endl;
}
```

Refinement of

BasicMatrix and RandomAccessCollection

Notation

`X`	is a type that is a model of Matrix.
`A`	is an object of type `X`.
`i,j`	are objects of type `matrix_traits<X>::size_type`.

Associated Types

Matrix inherits the associated types from BasicMatrix and RandomAccessCollection.

Valid Expressions

Most of the valid expressions for Matrix are inherited from the concepts it refines, but we restate them here for convenience of reference. The three new expressions, `trans(A)`, `abs(A)`, and `conj(A)` are described first.

- Transposed View of a Matrix
  `trans(A)`
  Return Type: `transpose_view<X>::type`
  Semantics: Creates a transposed view of the matrix. Access to an element at `A(i,j)` will retrieve the element at `A(j,i)`.
  Complexity: Constant time.

- Absolute Value
  `abs(A)`
  Return Type: a Matrix with element type `magnitude<T>::type` (where `T` is `matrix_traits<X>::value_type`).
  Semantics: applies `abs()` to each element of the matrix.

- Conjugate
  `conj(A)`
  Return Type: convertible to `X`
  Semantics: applies `conj()` to each element of the matrix.

- Element Access
  `A(i,j)`
  Return Type: `matrix_traits<X>::reference` if `A` is mutable, `matrix_traits<X>::const_reference` otherwise
  Semantics: Returns the element at row `i` and column `j`.

- Number of Rows
  `nrows(A)`
  Return Type: `X::size_type`

- Number of Columns
  `ncols(A)`
  Return Type: `X::size_type`

- Number of Non-Zeroes
  `nnz(A)`
  Return Type: `X::size_type`
  Semantics: Returns the number of stored elements in the matrix. Note that if an element that is stored in the matrix happens to be zero it is still counted towards the `nnz`. For dense matrices `nnz(A) == nrows(A) * ncols(A)`. For sparse and banded matrices `nnz(A)` is typically much small than `nrows(A) * ncols(A)`.

- 2-D iterator access to beginning of range
  `A.begin()`
  Return Type: `X::iterator` if `A` is mutable, `X::const_iterator` otherwise.
  Semantics: returns an iterator pointing to the first 1D section of the matrix.

- 2-D iterator access to end of range
  `A.end()`
  Return Type: `X::iterator` if `A` is mutable, `X::const_iterator` otherwise.
  Semantics: returns an iterator pointing after the last 1D section of the matrix.

- OneD Access
  `A[i]`
  Return Type: `X::reference` if `A` is mutable, `X::const_reference` otherwise
  Semantics: returns the `ith` one-dimensional section of the matrix (e.g., for a row-oriented matrix this returns the `ith` row).

- Size
  `A.size()`,
  `size(A)`
  Return Type: `X::size_type`
  Semantics Returns the number of 1D sections (typically rows or columns) in the matrix.
  Invariants: `A.size() == A.end() - A.begin()`
  Postcondition: `A.size() >= 0`

- Maximum size
  `A.max_size()`
  Return Type: `X::size_type`
  Semantics: Returns the largest size that this Matrix can ever have.
  Postcondition: `a.max_size() >= 0 && a.max_size() >= a.size()`

- Empty Matrix
  `A.empty()`
  Return Type: Convertible to `bool`
  Semantics: Equivalent to `a.size() == 0`.

- Swap
  `A.swap(B)`
  Return Type: `void`
  Semantics: Equivalent to `swap(A,B)`

Complexity Guarantees

- Element access time via `A(i,j)` is guaranteed to be $O(m + n)$.
- All iterator access methods and iterator operations are constant time.
- The 1-D access though operator bracket is constant time.

Examples

A transposed view of a small matrix. The transposed view is column-oriented, which means that `trans(A)[0]` is a column (whereas `A[0]` is a row).

```
matrix<int>::type A(2,2);
A(0,0) = 1; A(0,1) = 2;
A(1,0) = 3; A(1,1) = 4;
cout << "A =" << endl << A << endl;
cout << "A[0] = " << A[0] << endl;
cout << "trans(A) =" << endl << trans(A) << endl;
cout << "trans(A)[0] = " << trans(A)[0] << endl;
```

The output is:

```
A =
[1 2;
 3 4]
A[0] = [1 2]
trans(A) =
[1 3;
 2 4]
trans(A)[0] = [1 2]
```

7.5.5 BandedMatrix

A BandedMatrix provides an interface that restricts access to a region or *band* of a matrix, which is all elements a_{ij} where $i - j < l$ and $j - i < u$, l being the number of *sub* diagonals (below the main diagonal) and u the number of *super* diagonals (above the main diagonal). So the shape of the band is described by the pair (l, u) and the bandwidth is $l + u + 1$. A BandedMatrix is typically used when a matrix constains mostly zero elements, and all of the non-zeros fall within the band.

Elements outside of the band are considered out-of-bounds, and attempts to access those elements (via `A(i,j)` or `A[i][j]`) will raise an exception. The `begin()` and `end()` methods of the matrix's 1-D sections are modified so that the iterators traverse only the band of the matrix. If the BandedMatrix is also a SubdividableMatrix, then it is only valid to ask for sub-matrix, sub-row and sub-column regions that are entirely within the band. However, if `all` is specified, then this is taken to mean all of the region within the band.

Refinement of

Matrix

Requirements

- Number of Sub-Diagonals
  `nsub(A)`
  Return Type: `size_type`
  Semantics: returns the number of sub-diagonals in the bandwidth.

- Number of Super-Diagonals
  `nsuper(A)`
  Return Type: `size_type`
  Semantics: returns the number of super-diagonals in the bandwidth.

7.5.6 TriangularMatrix

A TriangularMatrix provides an interface that restricts access to either the upper or lower triangle of the matrix. A TriangularMatrix is a kind of BandedMatrix, where the bandwith is $(m-1, 0)$ for a lower triangular matrix, and $(0, n-1)$ for an upper triangular matrix.

One common case is for a triangular matrix to have all ones on the main diagonal, in which case the it is called a *unit diagonal* matrix. A TriangularMatrix that is declared unit diagonal restricts access from the main diagonal (no need to access those elements since they are always one). Some MTL algorithms are more efficient when handling triangular matrices that are declared unit diagonal.

Refinement of

BandedMatrix

Requirements

- Is Upper Triangular?
  `is_upper(A)`
  Return Type: `bool`
  Semantics: Returns `true` if the upper triangular region of the matrix contains the non-zeroes. In the case of a symmetric matrix, this returns `true` if the elements of the upper triangle are the ones that are actually stored and accessed.

- Is Lower Triangular?
  `is_lower(A)`
  Return Type: `bool`
  Semantics: Returns `true` if the lower triangular region of the matrix contains the non-zeroes. In the case of a symmetric matrix, this returns `true` if the elements of the lower triangle are the ones that are actually stored and accessed.

- Is Unit Diagonal?
  `is_unit_diag(A)`
  Return Type: `bool`
  Semantics: Returns `true` if the diagonal of the matrix is not stored, and can be assumed to be all ones. This allows some algorithms to be more efficient.

7.5.7 StrideableMatrix

A dense matrix is typically stored in a row-major or column-major fashion in memory. By default MTL provides a row-oriented and column-oriented interface respectively for accessing these matrices. Sometimes, however, it is necessary to access the columns of row-major matrix, or the rows of a column-major matrix. The StrideableMatrix concept defines the interface for creating a new matrix object that provides this kind of strided-view of the original matrix.

Requirements

- Row Oriented View Type
  `row_view<X>::type`
  The type of the object returned by `rows(A)`. This type must is a model of Matrix.

- Column Oriented View Type
  `column_view<X>::type`
  The type of the object returned by `columns(A)`. This type must is a model of Matrix.

- Row Oriented View Access Function
  `rows(A)`
  Return Type: `row_view<X>::type`
  Semantics: Create a row-oriented "view" of a matrix.
  Complexity: Constant time.

- Column Oriented View Access Function
  `columns(A)`
  Return Type: `column_view<X>::type`
  Semantics: Create a column-oriented "view" of a matrix.
  Complexity: Constant time.

Example

Inspecting a row and column oriented view of the same matrix.

```
typedef matrix<int>::type Matrix;
Matrix A(2,2);
A(0,0) = 1; A(0,1) = 2;
A(1,0) = 3; A(1,1) = 4;
row_view<Matrix>::type Ar = rows(A);
```

```
column_view<Matrix>::type Ac = columns(A);
cout << "A =" << endl << A << endl;
cout << "rows(A)[0] = " << Ar[0] << endl;
cout << "columns(A)[0] = " << Ac[0] << endl;
```

The output is:

```
A =
[1 2;
 3 4]
rows(A)[0] = [1 2]
columns(A)[0] = [1 3]
```

7.5.8 SubdividableMatrix

A SubdividableMatrix provides methods for accessing sub-matrices, sub-rows, and sub-columns of the matrix. The syntax is similar to that of MATLAB, though the ":" which is used in MATLAB to specify all of a row or all of a column has been replaced by `all` (since ":" is not a legal C++ identifier). Unlike MATLAB (and most array libraries) the ranges are specified with half-open intervals instead of closed intervals (e.g, `A(range(0,2),(0,3))` is a 2×3 matrix, not 3×4) [5].

Note that the origin for the row and column indices for sub-sections of a matrix is always reset to $(0, 0)$.

Refinement of

StrideableMatrix

Requirements

`X`	is a type that is a model of Matrix.
`A`	is an object of type `X`.
`i,j`	are objects of type `size_type`.
`r1,r2`	are objects of type `std::pair<size_type,size_type>`.
`all`	is the only value of the enumerated type `all_index_e`.

- Sub-Matrix Type
  `submatrix<X>::type`
  The type for sub-matrix views into the matrix.

- Sub-Row Type
  `subrow<X>::type`
  The type for sub-row views into the matrix.

[5] Specifying ranges with half-open intervals is consistent with C++ Standard iterators, and with C-style loop indexing conventions.

- Sub-Column Type
  `subcolumn<X>::type`
  The type for sub-column views into the matrix.

- Column Access
  `A(all,j)`
  Return Type: `subcolumn<X>::type`
  Semantics: Return the `j`th column.

- Sub-Column Access
  `A(r1,j)`
  Return Type: `subcolumn<X>::type`
  Semantics: Return a sub-section of the `j`th column, with row indices in the range `[r1.first,r1.second)`.

- Row Access
  `A(i,all)`
  Return Type: `subrow<X>::type`
  Semantics: Return the `i`th row.

- Sub-Row Access
  `A(i,r2)`
  Return Type: `subrow<X>::type`
  Semantics: Return a sub-section of the `i`th row, with row indices in the range `[r2.first, r2.second)`.

- Sub-Matrix Access
  `A(r1,all)`
  Return Type: `submatrix<X>::type`
  Semantics: Return the sub-matrix whose top-left element is `(r1.first,0)` and bottom-right element is `(r1.second-1,N-1)`.

- Sub-Matrix Access
  `A(all,r2)`
  Return Type: `submatrix<X>::type`
  Semantics: Return the sub-matrix whose top-left element is `(0,r2.first)` and bottom-right element is `(M-1,r2.second-1)`.

- Sub-Matrix Access
  `A(r1,r2)`
  Return Type: `submatrix<X>::type`
  Semantics: Return the sub-matrix whose top-left element is `(r1.first,r2.first)` and bottom-right element is `(r1.second-1,r2.second-1)`.

- Range Creation
  `range(b,e)`
  Return Type: `std::pair<size_type,size_type>`
  Semantics: A helper function for specifying ranges.

Example

A textbook implementation of gaussian elimination with partial pivoting.

```
template <class Matrix, class Vector>
void gaussian_elimination(Matrix& A, Vector& pivots)
{
  CLASS_REQUIRES(Matrix, SubdividableMatrix);
  typename matrix_traits<Matrix>::size_type
    m = A.nrows(), n = A.ncols(), pivot, i, k;
  typename matrix_traits<Matrix>::value_type s;

  for (k = 0; k < std::min(m-1,n-1); ++k) {
    pivot = k + max_abs_index( A(range(k,m),k) );
    if (pivot != k)
      swap(A(pivot,all), A(k,all));
    pivots[k] = pivot;
    if (A(k,k) != zero(s))
      for (i = k + 1; i < m; ++i) {
        s = A(i,k) / A(k,k);
        A(i,all) -= s * A(k,all);
      }
    }
}
```

7.5.9 ResizeableMatrix

A ResizeableMatrix can grow or shrink using the `resize()` method. When growing, the new elements are initialized to zero (or to the default value for the element type) and the old elements of the matrix remain unchanged. The time complexity for this operation can vary quite a bit from matrix type to matrix type. The `array<>` based matrices can grow and shrink quickly in the 2-D dimension.

Refinement of

BasicMatrix

Requirements

`X`	is a type that is a model of Matrix.
`A`	is an object of type `X`.
`m,n`	are objects of type `matrix_traits<X>::size_type`.

- Resize Matrix Dimensions
  `A.resize(m, n)`
  Return Type: `void`
  Semantics: Changes the dimensions of the matrix to $m \times n$.

7.5.10 FastDiagMatrix

A FastDiagMatrix provides an interface for fast traversal of the main diagonal of the matrix. The `diag(A)` function returns a Collection object which provides a view to the diagonal elements (a_{ii} $\forall i = 0 \ldots \min(m, n)$) of the matrix.

Refinement of

BasicMatrix

Requirements

`X`	is a type that is a model of FastDiagMatrix.
`A`	is an object of type `X`.

- The Main Diagonal View Type
  `diagonal_view<X>::type`
  This calculates the type of the object returned by `diag(A)`, which is required to be a model of Collection.

- Main Diagonal Access Function
  `diag(A)`
  Return Type: `diagonal_view<X>::type`
  Complexity: Constant time.

Example

Calculate the trace of a matrix, which is the sum of the elements on the main diagonal.

```
namespace mtl {
  template <class FastDiagMatrix>
  typename matrix_traits<FastDiagMatrix>::value_type
  trace(const FastDiagMatrix& A)
  {
    return sum(diag(A));
  }
}
```

Chapter 8

Arithmetic Types and Classes

Chapter 9

Object Model and Memory Management

9.1 Object Model

The MTL object-model defines what happens when you construct, copy, and assign one vector or matrix object to another. Most MTL vector and matrix classes behave like "handles" to the underlying data, and use shallow-copy semantics. For example:

```
mtl::vector<double>::type x(5, 1.0), y, z(5);
y = x;
mtl::copy(x, z);
y[2] = 3.0;
cout << x << endl;
cout << z << endl;
```

The output is:

```
[1,1,3,1,1]
[1,1,1,1,1]
```

The shallow copy semantics means that for most MTL objects, copy and assignment is a fast constant time operation.

The shallow-copy semantics does not apply to the stack allocated static-sized vector and matrix objects. The reason for this is that it is impossible to implement stack-allocated objects with shallow-copy semantics in C++. When passing MTL objects to functions, if you plan to use both stack-allocated and heap-allocated vectors it is best to pass-by-reference.

9.2 Memory Management

In MTL there are three memory management categories for vector and matrix objects:

- stack-allocated
- external memory (the MTL object is just a view to pre-existing memory, created via a pointer to that memory)
- heap-allocated (the normal case)

For the first two categories, MTL does no memory management, as none is needed. For heap-allocated objects, MTL automatically keeps a reference-count of the underlying data objects, and frees the memory when the reference count reaches zero. A know limitation of reference counting is that if there are cycles in the graph of reference-counted pointers then the cycles will not be deallocated properly. This is not a concern for vector and matrix objects, which typically do not contain other vectors and matrices, and when they do, it is in a tree structure (for sub-matrices) and does not form cycles.

Chapter 10

Vector Classes:

`vector<T,Storage,Orien>::type`

In MTL there are a fair number of vector types, so to make the selection process easier this vector *type generator* is provided. The `mtl::vector` [1] class is for selecting the type of vector that you want to use. This section describes the choices that are possible for the template parameters of the `mtl::vector` type generator. For most common uses the vector type given by the default arguments is the correct one, so you can just declare a vector with the type `mtl::vector<T>::type`. For example,

```
// create a vector of double precision numbers with size 10
mtl::vector<double>::type x(10);

// assign a value into the vector
x[4] = 2.14159;
```

The sections following this one describe the actual vector classes in more detail.

Template Parameters

`T`	The value type, the type of object stored in the vector.
`Storage`	Selects the way in which the elements are stored in memory. Possible choices are `dense`, `static_size`, `sparse_pair`, `compressed`, and `tree`. See below for a description of the storage types. **Default:** `dense<>`
`Orien`	The orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Storage Type Selection

The first decision to make when choosing a storage type is whether you want a sparse or dense vector. Second you must decide whether you want new memory allocated for the vector elements (internal memory), or if you are just creating a vector "view" to pre-existing memory (external memory). If memory will be allocated you can go with the default allocator, or provide a custom allocator. In addition, for dense vectors you can choose stack-allocated memory, in which case the vector will have static (constant) size. For external memory vectors, use `external` for the `Allocator` template argument and then supply a pointer to the memory in vector object's constructor.

For static-sized and external memory vectors, MTL performs no memory management, as none is needed. For heap-allocated vectors MTL automatically keeps a reference count of the element data. See Section 9.1 for a discussion of how MTL objects use shallow-copy semantics, and keep reference counts. The

[1] It is a shame that the name `vector` was choosen for the dynamic array class in the C++ standard. If you are using both the `std::vector` class and MTL at the same time, it is not advisable to do `using mtl::vector;` or `using namespace mtl;`.

following is a list of the valid choices for the `Storage` template parameter. The storage types are described in more detail in the following sections.

- `dense<Allocator>` A general purpose vector, typically with heap allocated memory.
- `static_size<N>`
  A vector who's size is constant and memory is stack-allocated.
- `sparse_pair<Allocator>`
  This is a sparse vector in which index-value pairs are stored in an array.
- `compressed<Idx,Allocator,Start>`
  This is a sparse vector in which the indices and element values are stored in parallel arrays.

Model of

Vector

Members

The MTL vector classes meet the requirements for the Vector concept, and therefore has the member functions and associated type defined in Vector and in the concepts that Vector refines, which include Linalg and ReversibleCollection. The constructors for each vector type are described in the following sections.

10.1 Dense Vectors

10.1.1 vector<T, dense<Allocator>, Orien>::type

This is the main vector class. The vector elements are stored contiguously on the heap, and the iterators are random-access. Element access with `operator[]` is constant time.

Example

```
// This is a dumb example, replace it! -JGS
typedef std::complex<float> C;
typedef mtl::vector< C, dense<> >::type Vec;
Vec x(100);
int cnt = 0;
for (Vec::iterator i = x.begin(); i != x.end(); ++i)
  *i = C(cnt, ++cnt);
cout << x[40] << endl;
```

The output is:

```
(40,41)
```

Template Parameters

`T`	is the vector's value type, the type of object stored in the vector.
`Allocator`	is the allocator used to obtain memory for storing the elements of the vector. In addition to C++ standard conforming allocators (models of the Allocator concept), you can choose `external` which creates an MTL vector "view" to pre-existing memory. **Default:** `std::allocator<T>`
`Orien`	is the orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Model of

ResizableVector, SubdividableVector, RandomAccessCollection, and VectorSpace.

Members

In addition to the member function required by ResizableVector, SubdividableVector and RandomAccessCollection, the following members are defined. We use `self` as a placeholder for the actual class name.

```
self(const Allocator& a = Allocator())
```

This is the default constructor, which creates a vector of size zero.

```
self(size_type n, const T& init = T(),
     const Allocator& a = Allocator())
```

Creates a vector of size `n` and initialize all the elements to the value `init`[2].

```
self(const self& x)
```

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`. The reference count of the data object is incremented.

```
~self()
```

The destructor. The reference count of the underlying vector data is decremented.

```
self& operator=(const self& x)
```

The assignment operator. Like the copy constructor, this makes a shallow copy and increments the reference count of the data object.

10.1.2 `vector<T, dense<external,N>, Orien>::type`

Use this class to create a vector "view" to pre-existing memory. This is useful when interfacing with legacy code or to other programming languages. This MTL vector claims no responsibility for the lifetime of the underlying memory, and some care must be taken to ensure that the memory lasts longer than any use of this vector object.

Example

```
extern "C" float sum(float* xp, int n)
{
  typedef mtl::vector<float, dense<external> >::type Vec;
  Vec x(xp, n);
  return mtl::sum(x);
}
```

(JGS comment: Should there be a template argument to specify if it is a const pointer and therefore must be a const vector?)

Template Parameters

`T`	is the value type, the type of object stored in the vector.
`N`	specifies the static size of the vector. A value of `dynamic_size` denotes a vector with size determined at run-time. **Default:** `dynamic_size`
`Orien`	is the orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Model of

SubdividableVector, RandomAccessCollection, and VectorSpace.

Members

In addition to the member function required by SubdividableVector and RandomAccessCollection, the following members are defined. We use `self` as a placeholder for the actual class name.

`self()`

> This is the default constructor, which creates an empty vector handle. You will need to assign another vector to this handle before it will be useful.

`self(T* data, size_type n)`

> Construct a view to the existing memory given by the `data` pointer.

`self(const self& x)`

> The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`.

`self& operator=(const self& x)`

> The assignment operator. Like the copy constructor, this make a shallow copy.

`~self()`

> The destructor. This function does nothing.

10.1.3 vector<T, static_size<N>, Orien>::type

This is a dense vector that is stack-allocated. It is especially advisable to use this vector type when the vector size is small (less then 20), since it avoids the overhead of heap-allocation. The vector size must be constant, and specified in the template argument. Unlike most MTL vectors, this vector is not initialized to some value (zero by default) but left uninitialized. Also this vector type has deep copy semantics instead of shallow copy semantics as is usual for MTL objects (see Section 9.1 for details).

Example

```
const int N = 3;
typedef mtl::vector<float, static_size<N> >::type Vec;
Vec x; // there is only a default constructor
// but you can also use intializer list syntax
Vec y = { 4.0, 1.2, 5.6 }; // not true anymore! -JGS
x = y; // copy and assignment is deep for stack-allocated vectors
x[0] = 3.0;
assert(x[0] != y[0]);
```

Template Parameters

`T`	is the value type, the type of object stored in the vector.
`N`	specifies the static size of the vector.
`Orien`	is the orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Model of

SubdividableVector, RandomAccessCollection and VectorSpace.

Members

This vector implements the member functions and associated types defined in the SubdividableVector, RandomAccessCollection, and VectorSpace concepts (and the concepts they refine).

`self()`

Default constructor.

10.2 Sparse Vectors

A sparse vector provides an efficient method for storing vectors when most of the elements are zero. The sparse vectors only store the non-zero elements and the corresponding index (location) of each element. The MTL sparse vectors have the same Vector interface as the dense vectors. They provide the usual iterators for traversal and the `operator[]` for element access. The MTL sparse vectors also provide the additional functionality specified in the SparseVector concept described in Section 7.4.3.

MTL provides three sparse vector storage formats: compressed, sparse pair, and tree. The compressed format uses a Fortran-style parallel array, one array for the indices and one array for the values, as shown in Figure 10.1. The sparse-pair format uses a single array, where each element of the array is an index-value pair, as shown in Figure 10.2. The tree format stores the elements as index-value pairs in a `std::set`, which is a red-black tree that provides the advantage of fast random insertion. All three formats keep the elements in sorted order according to their index.

The time complexity for element access using `operator[]` is not constant for sparse vectors, but is logarithmic in the number of non-zeroes. In addition, assignment to elements through `operator[]` may cause allocation and data movement, since if the accessed element was not yet explicitly stored (previously zero) then space must be made in the appropriate place in the sparse vector. Due to this extra overhead, it is not advisable to use a sparse vector in situations where the elements are dense, or as the output argument for operations that result in dense vectors, such as matrix-vector multiplication.

Indices	1	3	8	11	22	24
Values	1.3	5	3.1	4	0	2.1

Figure 10.1: The Compressed Sparse Vector Format.

(1,1.3)	(3,5)	(8,3.1)	(11,4)	(22,0)	(24,2.1)

Figure 10.2: The Sparse Pair Vector Format.

The stored indices are by default zero-based, but this can be changed to one-based with the `IdxStart` template parameter. Note that this only changes the way in which the indices are stored, the vector will still appear to be zero-based. The purpose of allowing one-based internal storage is to accomodate data sharing with Fortran codes.

10.2.1 `vector<T, compressed<Idx,Alloc,IdxStart>, Orien>::type`

This class implements a sparse vector using two parallel arrays, one for the elements values and one for the indices. The interface provided by this class is similar to all the other MTL vectors, and is described in the ResizableVector and SequentialCollection concepts. The memory for the elements is obtained via `Alloc` which must be an Allocator.

Example

Calculate the infinity norm of a sparse vector.

```
typedef mtl::vector<float, compressed<> >::type Vec;
const int n = 10;
Vec x(n), y(n);

x[3] = 2.5;
x[8] = 0.1;
x[5] = 3.1;

float inf_norm = mtl::infinity_norm(x);
cout << inf_norm << endl;
```

The output is:

```
3.1
```

Template Parameters

`T`	is the sparse vector's value type, which is the element type of the value array.
`Idx`	is the index type (and `size_type`), which is the element type of the index array. **Default:** `int`
`Alloc`	is the allocator type, which must model Allocator. **Default:** `std::allocator<T>`
`IdxStart`	The counting convention for the indices, either C style `index_from_zero` or Fortran style `index_from_one`. **Default:** `index_from_zero`
`Orien`	is the orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Model of

SparseVector, ResizableVector, SequentialCollection, and VectorSpace.

Complexity

Element assignment through `operator[]` is linear in the number of non-zeroes, since data movement may occur. For better performance insert the elements all at once using the constructor for `IndexValuePairIterator`, which is $O(nnz \log nnz)$ for the insertion of all the elements. Element access via `operator[]` is $O(\log nnz)$, and iterator traversal (`operator++`) is constant time as usual.

Members

In addition to the member functions required by ResizableVector and SequentialCollection, the following members are defined. We use `self` as a placeholder for the actual class name.

```
self(const Alloc& a = Alloc())
```

This is the default constructor, which creates a vector of size zero.

```
self(size_type n, const Alloc& a = Alloc())
```

Creates a vector of size `n` but with `nnz() == 0`. The vector will allow assignment to element indices in the range `[0, n)`.

```
template <class IndexValuePairIterator>
self(IndexValuePairIterator first, IndexValuePairIterator last,
     size_type n)
```

Construct a sparse vector from any iterators that dereferences to give index-value pairs, where the functions `value()` and `index()` provide access to the value and index. `value(*first)` must be convertible to `T` and `index(*first)` must be convertible to `Idx`. The iterators must model InputIterator, though the construction is more efficient when the iterators model RandomAccessIterator. The indices should fall in the range `[0, n)`.

```
self(const self& x)
```

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`. The reference count of the data object is incremented.

```
~self()
```

The destructor. The reference count of the underlying vector data is decremented.

```
self& operator=(const self& x)
```

The assignment operator. Like the copy constructor, this makes a shallow copy and increments the reference count of the data object.

10.2.2 `vector<T, compressed<Idx,external,IdxStart>, Orien>::type`

This class can be used to create an MTL "view" to a pre-existing sparse vector in memory. This is useful when interfacing with legacy code or other programming languages. This MTL vector claims no responsibility for the lifetime of the underlying memory, and some care must be taken to ensure that the memory lasts longer than any use of this vector object. Make sure to choose the `T` and `Idx` template argument to match the pointer type for the values and indices array. Note that for this class, element assignment through `operator[]` is only valid for non-zero elements, and will result in a "no-op" (nothing done) for zero elements. This is because this vector type does not control the memory.

Example

```
extern "C"
float sparse_one_norm(float* values, int* indices, int n, int nnz)
{
  typedef mtl::vector<float, compressed<int, external> >::type Vec;
  Vec x(values, indices, n, nnz);
  return mtl::one_norm(x);
}
```

Complexity

Element access via `operator[]` is $O(\log nnz)$, and iterator traversal (`operator++`) is constant time as usual.

Template Parameters

See the description for `vector<T,compressed<Idx,Alloc,IdxStart>,Orien>::type`.

Members

`self()`

> This is the default constructor, which creates an empty vector handle. You will need to assign another vector to this handle before it will be useful.

`self(T* values, Idx* indices, size_type n, size_type nnz)`

> Construct a view to the existing memory given by the `values` and `indices` pointers, with `nnz` non-zeroes (which should match the length of the two arrays) and with indices in the range `[0, n)`.

`self(const self& x)`

> The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`.

`self& operator=(const self& x)}`

The assignment operator. Like the copy constructor, this make a shallow copy.

`~self()`

The destructor. This function does nothing.

10.2.3 `vector<T, sparse_pair<Idx,Alloc>, Orien>::type`

This class implements a sparse vector as a sorted array of index-value pairs. The interface provided by this class is similar to all the other MTL vectors, and is described in the ResizableVector and SequentialCollection concepts. The memory for the elements is obtained via `Alloc` which must be an Allocator.

Model of

SequentialCollection, ResizableVector and MatrixExpression.

Complexity

Element assignment through `operator[]` is linear in the number of non-zeroes, since data movement may occur. For better performance insert the elements all at once using the constructor for `IndexValuePairIterator`, which is $O(nnz \log nnz)$ for the insertion of all the elements. Element access via `operator[]` is $O(\log nnz)$, and iterator traversal (`operator++`) is constant time as usual.

Template Parameters

`T`	is the sparse vector's value type.
`Idx`	is the index type (and `size_type`). **Default:** `int`
`Alloc`	is the allocator type, which must model Allocator. **Default:** `std::allocator<T>`
`Orien`	is the orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Members

In addition to the member functions required by ResizableVector and SequentialCollection, the following members are defined. We use `self` as a placeholder for the actual class name.

`self(const Alloc& a = Alloc())`

This is the default constructor, which creates a vector of size zero.

```
self(size_type n, const Alloc& a = Alloc())
```

Creates a vector of size `n` but with `nnz() == 0`. The vector will allow assignment to element indices in the range `[0, n)`.

```
template <class IndexValuePairIterator>
self(IndexValuePairIterator first, IndexValuePairIterator last,
     size_type n)
```

Construct a sparse vector from any iterators that dereference to give index-value pairs, where the function `value()` and `index()` provide access to the index and value. `value(*first)` must be convertible to `T` and `index(*first)` must be convertible to `size_type`. The iterators must model InputIterator, though the construction is more efficient when the iterators model RandomAccessIterator. The indices should fall in the range `[0, n)`.

```
self(const self& x)
```

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`. The reference count of the data object is incremented.

```
~self()
```

The destructor. The reference count of the underlying vector data is decremented.

```
self& operator=(const self& x)
```

The assignment operator. Like the copy constructor, this makes a shallow copy and increments the reference count of the data object.

10.2.4 `vector<T, sparse_pair<Idx,external>, Orien>::type`

This class can be used to create an MTL "view" to a pre-existing sparse vector in memory. This MTL vector claims no responsibility for the lifetime of the underlying memory, and some care must be taken to ensure that the memory lasts longer than any use of this vector object. The pointer to the underlying array must be of type `std::pair<T,Idx>*`. Note that for this class, element assignment through `operator[]` is only valid for non-zero elements, and will result in a "no-op" (nothing done) for zero elements. This is because this vector type does not control the memory.

Complexity

Element access via `operator[]` is $O(\log nnz)$, and iterator traversal (`operator++`) is constant time as usual.

Template Parameters

`T`	is the sparse vector's value type.
`Idx`	is the index type (and `size_type`). **Default:** `int`
`Orien`	is the orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Members

`self()`

This is the default constructor, which creates an empty vector handle. You will need to assign another vector to this handle before it will be useful.

`self(std::pair<T,Idx>* data, size_type n, size_type nnz)`

Construct a view to the existing memory given by the `data` pointer, with `nnz` non-zeroes (which should match the length of the data array) and with indices in the range `[0, n)`.

`self(const self& x)`

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`.

`self& operator=(const self& x)`

The assignment operator. Like the copy constructor, this make a shallow copy.

`~self()`

The destructor. This function does nothing.

10.2.5 `vector<T, tree<Alloc>, Orien>::type`

This class implements a sparse vector using `std::set` with index-value pairs as the value type for the `std::set`. The interface provided by this class is similar to all the other MTL vectors, and is described in the ResizableVector and SequentialCollection concepts. The memory for the elements is obtained via `Alloc` which must be an Allocator.

Model of

SequentialCollection, ResizableVector and MatrixExpression.

Complexity

Element assignment through `operator[]` is logarithmic in the number of non-zeroes ($O(\log nnz)$). If you are inserting the elements all at once, it is typically more efficient to use one of the array-based sparse vectors and their constructor for `IndexValuePairIterator` (though this class also provides such a constructor). Element access via `operator[]` is $O(\log nnz)$, and iterator traversal (`operator++`) is constant time as usual.

Template Parameters

`T`	is the sparse vector's value type.
`Alloc`	is the allocator type, which must model Allocator. **Default:** `std::allocator<T>`
`Orien`	The orientation of the vector, either `row_major` or `column_major`. This argument is important only when vectors are used inside operator expressions. **Default:** `column_major`

Members

In addition to the member functions required by ResizableVector and SequentialCollection, the following members are defined. We use `self` as a placeholder for the actual class name.

```
self(const Alloc& a = Alloc())
```

This is the default constructor, which creates a vector of size zero.

```
self(size_type n, const Alloc& a = Alloc())
```

Creates a vector of size `n` but with `nnz() == 0`. The vector will allow assignment to element indices in the range `[0, n)`.

```
template <class IndexValuePairIterator>
self(IndexValuePairIterator first, IndexValuePairIterator last,
     size_type n)
```

Construct a sparse vector from any iterators that dereference to give index-value pairs, where the functions `value()` and `index()` must be defined for the pair type. `value(*first)` must be convertible to `T` and `index(*first)` must be convertible to `Idx`. The iterators must model InputIterator, though the construction is more efficient when the iterators model RandomAccessIterator. The indices should fall in the range `[0, n)`.

```
self(const self& x)}
```

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this vector and vector `x`. The reference count of the data object is incremented.

`~self()`

The destructor. The reference count of the underlying vector data is decremented.

`self& operator=(const self& x)`

The assignment operator. Like the copy constructor, this makes a shallow copy and increments the reference count of the data object.

Chapter 11

Matrix Classes:

`matrix<T,Shape,Storage,Orien>::type`

The MTL contains a large number of matrix types, ranging from your basic dense matrix, to banded, symmetric, and various sparse matrices. MTL also provides several memory allocation alternatives for the matrix data, including static-sized stack allocation (good for small matrix computations), heap allocation based on generalized STL-style Allocators (good for arbitrarily sized matrices), and simple pointer-wrappers for matrix data that originated from other sources (which we refer to as *external* data).

Due to the heavy parameterization of the MTlL matrix classes, it is somewhat complicated to deal with them directly. In an effort to simplify the interface we provide this `mtl::matrix` class which is a *type generator*. The user provides several template parameter choices (called *selectors*) and the `mtl::matrix` provides an inner `typedef` for the correct matrix type. Many of the template arguments have default values, so creating an MTL matrix can be as simple as in the first line of the following example.

Example

```
typedef mtl::matrix<double>::type Matrix; // select the matrix type
Matrix A(num_rows, num_colums);  // create a matrix object

// Fill in the matrix with random values, using
// STL-style iterators to access the matrix elements
for (Matrix::iterator i = A.begin(); i != A.end(); ++i)
  for (Matrix::OneD::iterator j = (*i).begin(); j != (*i).end(); ++j)
    *j = rand();
```

Template Parameters

`T`	The value type, the type of the elements in the matrix.
`Shape`	The shape of the matrix, described below.
`Storage`	The storage format for the placement of elements in memory.
`Orien`	The orientation of the matrix, either `row_major`, `column_major`, or `diagonal_major`.

11.0.6 Shape Selectors

The `Shape` template argument of the matrix generator specifies the layout of the non-zero elements of the matrix and whether the matrix is symmetric or Hermitian. Here we give an overview of the choices for matrix shape, and details are given in the following sections.

- `rectangle<>`
  A rectangular matrix is a general purpose matrix in which elements can appear in any position in the matrix, i.e., there can be any element a_{ij} where $0 \leq i \leq M$ and $0 \leq j \leq N$. Both dense and sparse matrices can fit into this category.

- `banded<>`
  A band shaped matrix is one in which non-zero matrix elements only appear within a certain distance to the main diagonal of the matrix. The bandwidth of a matrix is described by the number of diagonals in the band that are below the main diagonal, referred to as the *sub* diagonals, and the number of diagonals in the band above the main diagonal, referred to as the *super* diagonals. Figure 11.1 is an example of a matrix with a bandwidth of (1,2). There are several storage types that can be used to efficiently represent banded matrices, including the banded format (equivalent to the LAPACK banded matrix) and dense matrices with diagonal orientation. Accesses made to elements outside of the band are considered to be out-of-bounds.

1	2	3	0	0
4	5	6	7	0
0	8	9	10	11
0	0	12	13	14
0	0	0	15	16

Figure 11.1: Example of a banded matrix with bandwidth (1,2).

- `triangle<Uplo,Diag>`
  The triangle shape is a special case of the banded shape. A triangular matrix of shape (M, N) has a bandwidth $(M-1, 0)$ for a lower triangular matrix or $(0, N-1)$ for an upper triangular matrix. There is also the special case of when the main diagonal of the matrix is all ones, in which case the matrix is called unit diagonal.

- `symmetric<Uplo>`
  The symmetric shape is also a special case of the banded shape, with a twist. Since the matrix is symmetric ($a_{ij} = a_{ji}$) it is only necessary to store half of the matrix, either the elements above or below the diagonal. The *sub* and *super* of a symmetric matrix must be equal, as the number of rows and columns must also be equal.

- `hermitian<Uplo>`
  This is similar to a symmetric matrix, except that the elements must be complex numbers, and the elements above the diagonal are the complex conjugates of the elements below the diagonal ($a_{ij} = \overline{a_{ji}}$).

11.0.7 Storage Selectors

The matrix storage types are introduced here, and are described in more detail in the subsequent sections.

1	2	3	0	0
2	6	7	8	0
3	7	10	11	12
0	8	11	13	14
0	0	12	14	15

Figure 11.2: Example of a symmetric matrix with bandwidth (2,2).

- `dense<Allocator>`
  This is the most common way of storing matrices, and consists of one contiguous piece of memory that is divided up into rows or columns of equal length. The example in Figure 11.3 shows how a matrix can be mapped to linear memory in either a row-major or column-major fashion.

1	2	3
4	5	6
7	8	9

row major storage

1	2	3	4	5	6	7	8	9

column major storage

1	4	7	2	5	8	3	6	9

Figure 11.3: Example of the dense matrix storage format.

- `static_size<M,N>`
  This is similar to the `dense<>` matrix storage, except that the matrix is stack-allocated and the matrix dimension must be constants (fixed at compile-time). For operations on small matrices, it is often more efficient to use this matrix over the `dense<>` matrix.

- `banded<Allocator>`
  This storage format is equivalent to the banded storage used in the BLAS and LAPACK. The banded storage format maps the bands of the matrix to an array of dimension $(M, sub + super + 1)$ for row major and $(sub + super + 1, N)$ for column major matrices. For a row major matrix, the diagonals of the matrix fall into the columns of the array. For a column major matrix the diagonals fall into the rows of the array. The two-dimensional array is mapped to the linear memory space of a single chunk of memory. Figure 11.4 is an example banded matrix with the mapping to

the row-major and column-major 2D arrays. The X's represent memory locations that are not used.

1	2	3	0	0
4	5	6	7	0
0	8	9	10	11
0	0	12	13	14
0	0	0	15	16

original matrix

X	1	2	3
4	5	6	7
8	9	10	11
12	13	14	X
15	16	X	X

banded storage (row major)

X	X	3	7	11
X	2	6	10	14
1	5	9	13	16
4	8	12	15	X

banded storage (column major)

Figure 11.4: Example of the banded matrix storage format.

- `packed<Allocator>`
  This storage format is equivalent to the BLAS/LAPACK packed storage format. This format provides an efficient way to represent triangular, symmetric and Hermitian matrices. Either the elements in the upper or lower triangle of the matrix are stored. Each row (or column) is packed into a contiguous block of memory, one row starting immediately after the next. Figure 11.5 depicts and example of a matrix stored in this way.

1	2	3	4	5
0	6	7	8	9
0	0	10	11	12
0	0	0	13	14
0	0	0	0	15

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

Figure 11.5: Example of the packed matrix storage format.

- `compressed<Idx,Allocator,IdxStart>`
  This storage format is the traditional compressed row or compressed column format. The storage consists of three arrays, one array for all of the elements, one array consisting of the row or column index (row for column-major and column for row-major matrices), and one array consisting of pointers to the start of each row/column. Figure 11.6 is an example sparse matrix in compressed for format, with the stored indices starting from one (Fortran style). They can also be indexed from zero (C style).

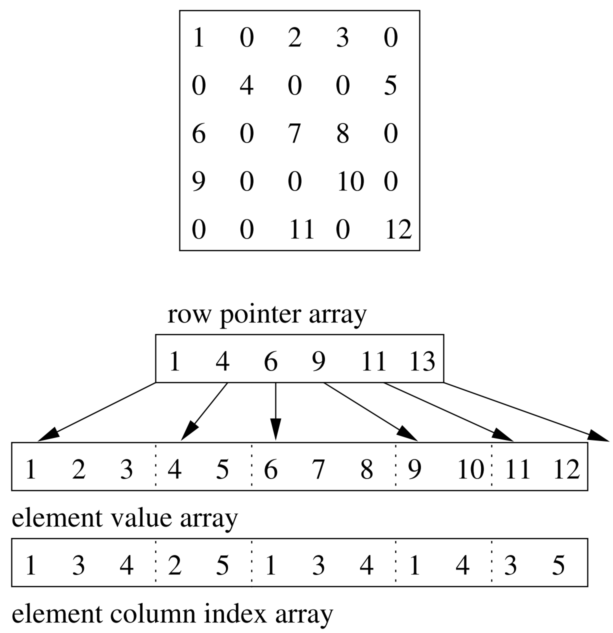

Figure 11.6: Example of the compressed column matrix storage format.

- `array<OneD>`
  This storage format gives an "array of pointers" style implementation of a matrix. Each row or column of the matrix is allocated separately. The type of vector used for the rows or columns is flexible, and one can choose from any of the 1-D storage types, which include `dense`, `compressed`, `sparse_pair`, `tree`, `linked_list`, and `set`. Figure 11.7 gives two examples of array storage types.

- `envelope<Allocator>`
  This storage scheme is for sparse symmetric matrices, where most of the non-zero elements fall near the main diagonal. The storage format is useful for certain factorizations since the fill-ins fall into areas already allocated. This scheme is different than most sparse matrices since the row containers are actually dense, similar to a banded matrix. Figure 11.8 gives an example of a matrix in the envelope storage scheme.

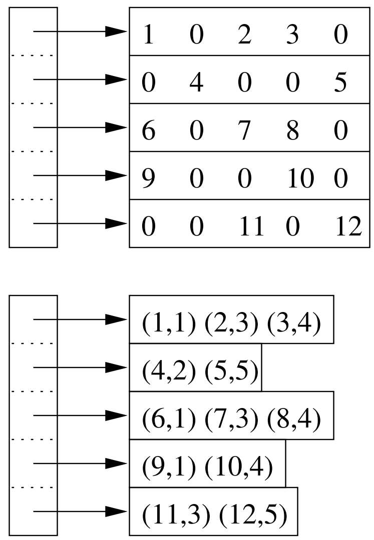

Figure 11.7: Example of the array matrix storage format with dense and with sparse pair OneD storage types.

11.0.8 Shape and Storage Combinations

Table 11.1 shows the valid combinations of `Shape` and `Storage` template parameters for the `mtl::matrix`. The X's mark the valid combinations.

(JGS comment: how should we handle the interface to blocked matrices? Implementation-wise there are two varieties of blocked matrices. A blocking imposed on a normal (dense) matrix, and a matrix which literally contains sub-matrices.)

Storage	**Shape**				
	`rectangle`	`banded`	`triangle`	`symmetric`	`hermitian`
`dense`	X	X	X	X	X
`static_size`	X				
`banded`		X	X	X	X
`packed`			X	X	X
`array`	X	X	X	X	X
`sparse_array`	X	X	X	X	X
`compressed`	X		X	X	X
`coordinate`	X			X	X
`envelope`				X	X

Table 11.1: Valid Shape and Storage Combinations.

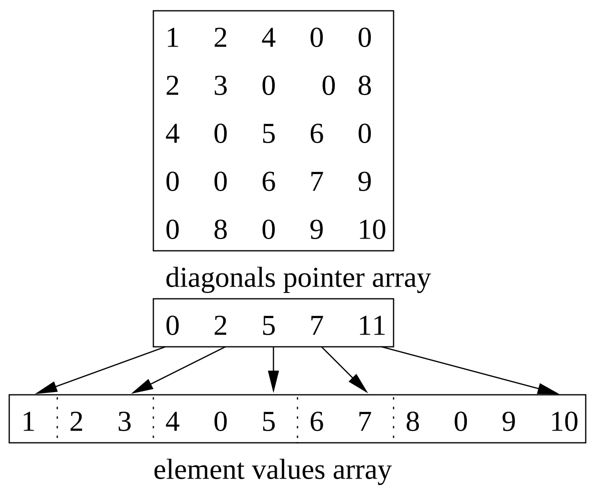

Figure 11.8: Example of the envelope matrix storage format.

11.1 Dense Matrices

The MTL dense matrices come in three flavors, the general purpose heap-allocated single block of data (in either row major or column major format), the "array of pointers" style of matrix, and the stack-allocated, constant size matrix. Also there is the "external" variant of the heap-allocated matrix, which has a constructor that takes a pointer to some pre-existing data, and which does no memory management.

11.1.1 `matrix<T, rectangle<>, dense<Alloc>, Orien>::type`

Example

```
typedef matrix<double, rectangle<>,
               dense<>, row_major>::type Matrix;
Matrix A(m, n); // construct a matrix object
// fill matrix ...

A.submatrix( ) = x * trans(y);
```

Template Parameters

`T`	is the matrix's value type, the type of object stored in the matrix.
`Alloc`	is the allocator used obtain memory, which must be a C++ standard compliant allocator. **Default:** `std::allocator<T>`
`Orien`	The orientation of the matrix, either `row_major` or `column_major`.

Model of

StrideableMatrix, ResizableMatrix, SubdividableMatrix, FastDiagMatrix and MatrixExpression.

Members

In addition to the member function required by StrideableMatrix, ResizableMatrix, SubdividableMatrix, FastDiagMatrix and LinearAlgebra the following members are defined. We use `self` as a placeholder for the actual class name.

`self(const Alloc& a = Alloc())`

This is the default constructor, which creates a matrix of size $(0, 0)$.

```
self(size_type m, size_type n,
     const T& init = T(),
     size_type m_origin = 0, size_type n_origin = 0,
     const Alloc& a = Alloc())
```

Creates a matrix of size `(m,n)` and initalizes all of the elements to the value of `init`[1]. The element indices will be `([m_origin,m_origin + m), [n_origin,n_origin + n))`.

```
self(const self& x)
```

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this matrix and matrix `x`. The reference count of the data object is incremented.

```
~self()
```

The destructor. The reference count of the underlying data object is decremented.

```
self& operator=(const self& x)
```

The assignment operator. Like the copy constructor, this makes a shallow copy and increments the reference count of the data object.

11.1.2 `matrix<T, rectangle<>, dense<external, M, N>, Orien>::type`

This matrix class is used to create a matrix "view" to pre-existing memory. This is useful when interfacing with legacy code or to other programming languages. This matrix type claims no responsibility for the lifetime of the underlying memory, and some care must be taken to ensure that the memory lasts longer than any use of the matrix object.

Template Parameters

`T`	is the value type, the type of object stored in the matrix.
`M`	specifies the number of rows for a static sized matrix. A value of `dynamic_size` denotes a matrix with size determined at run-time. **Default:** `dynamic_size`
`N`	specifies the number of columns for a static sized matrix. A value of `dynamic_size` denotes a matrix with size determined at run-time. **Default:** `dynamic_size`
`Orien`	The orientation of the matrix, either `row_major` `column_major`.

Model of

StrideableMatrix, ResizableMatrix, SubdividableMatrix, FastDiagMatrix and LinearAlgebra.

Members

In addition to the member function required by StrideableMatrix, ResizableMatrix, SubdividableMatrix, FastDiagMatrix and LinearAlgebra the following members are defined. We use `self` as a placeholder for the actual class name.

`self()`

> This is the default constructor, which creates an empty matrix. This matrix object is unusable until assigned to some matrix with data.

```
self(T* data, size_type m, size_type n, size_type ld,
     size_type m_origin = 0, size_type n_origin = 0)
```

> Creates a matrix view to the existing memory given by the `data` pointer. The matrix will be of size `(m,n)`, with a leading dimension of `ld`. The element indices will be `([m_origin,m_origin + m), [n_origin,n_origin + n))`.

`self(const self& x)`

> The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this matrix and matrix `x`.

`~self()`

> The destructor, which does nothing.

`self& operator=(const self& x)`

> The assignment operator. Like the copy constructor, this makes a shallow copy so that this matrix and matrix `x` share the same data.

11.1.3 `matrix<T, rectangle<>, static_size<M, N>, Orien>::type`

This is a stack-allocated matrix. For operations on small matrices, it is more efficient to use this matrix type than the general dense matrix since it avoids the overhead of heap-allocation. The size of the matrix is constant, and specified as a template argument. Unlike most MTL matrices, the elements are not initialized to some value (zero by default) but left unitialized (unless explicitly initialized via initializer list syntax as in the following example). This matrix type has deep copy semantics instead of the shallow copy semantics that is usual for MTL objects (see Section 9.1 for details).

Example

```
const int m = 2, n = 2;
typedef mtl::matrix<int, rectangle<>,
  static_size<m,n> >::type Matrix;
Matrix A = { 1, 1,
             1, 1 };
Matrix B = { 3, 3,
             3, 3 };
Matrix C;

mtl::add(A, B, C);

for (int i = 0; i < m; ++i)
  for (int j = 0; j < n; ++j)
    assert(C(i,j) == 4);
```

Template Parameters

T	is the value type, the type of object stored in the matrix.
M	specifies the number of rows.
N	specifies the number of columns.
Orien	is the orientation of the matrix, either row_major or column_major.

Model of

StrideableMatrix, SubdividableMatrix, FastDiagMatrix and LinearAlgebra.

Members

This matrix implements the member function required by StrideableMatrix, SubdividableMatrix, FastDiagMatrix and MatrixExression.

11.1.4 matrix<T, rectangle<>, array< dense<Alloc> >, Orien>::type

This matrix type provides an “array of pointers” style of matrix implementation. Each row (or column) of the matrix is allocated separately. This gives the advantage the rows can be swapped in constant time. In addition, growing the matrix along the major dimension is more efficient than for the normal dense matrix. Unlike the other dense matrix types, this matrix is not a SubdividableMatrix, StrideableMatrix, or a FastDiagMatrix.

Example

Swapping the rows of a matrix in constant time.

```
typedef matrix< double,
                rectangle<>,
                array< dense<> >,
                row_major>::type Matrix;
Matrix B(m, n);
Matrix::Row tmp = B[2];
B[2] = B[3];
B[3] = tmp;
```

Template Parameters

`T`	is the matrix's value type, the type of object stored in the matrix.
`Alloc`	is the allocator used obtain memory, which must be a C++ standard compliant allocator. **Default:** `std::allocator<T>`
`Orien`	The orientation of the matrix, either `row_major` or `column_major`.

Members

This matrix implements the member functions required by Matrix and LinearAlgebra. In addition this class defines the following constructions and destructor.

`self(const Alloc& a = Alloc())`

This is the default constructor, which creates a matrix of size $(0, 0)$.

```
self(size_type m, size_type n,
     const T& init = T(),
     size_type m_origin = 0, size_type n_origin = 0,
     const Alloc& a = Alloc())
```

Creates a matrix of size `(m,n)` and initalizes all of the elements to the value of `init`[2]. The element indices will be `([m_origin,m_origin + m), [n_origin,n_origin + n))`.

`self(const self& x)`

The copy constructor. This performs a shallow copy, which means that the underlying data will be shared between this matrix and matrix `x`. The reference count of the data object is incremented.

`~self()`

The destructor. The reference count of the underlying data object is decremented.

`self& operator=(const self& x)`

The assignment operator. Like the copy constructor, this makes a shallow copy and increments the reference count of the data object.

11.2 Sparse Matrices

11.2.1 matrix<T, Shape, compressed<Idx,Alloc,IdxStart>, Orien>::type

Template Parameters

T	is the value type, the type of object stored in the matrix.
Shape	is the shape of the matrix, either rectangle<>, symmetric<Uplo>, hermitian<Uplo>, or triangle<Uplo,-Diag>.
Idx	is the index type (also the size_type) of the matrix, which is the element type of the index array. **Default:** int
Alloc	is the allocator type, which must be a model of Allocator or external. **Default:** std::allocator<T>
IdxStart	specifies the counting conversion for the indices, either C style index_from_zero or Fortran style index_from_one. **Default:** index_from_zero
Orien	is the orientation of the matrix, either row_major or column_major.

Members

```
self(Idx m, Idx n, Idx nnz = max(m,n) * 10)
```

Create a sparse matrix with m rows and n columns. The nnz argument is a hint for how many nonzeroes will be in the matrix.

```
template <class ElementIterator>
self(ElementIterator first, ElementIterator last,
     Idx m, Idx n, Idx nnz)
```

11.2.2 matrix<T, Shape, compressed<Idx,external,IdxStart>, Orien>::type

Members

```
self(Idx m, Idx n, Idx nnz, T* val, Idx* indx, Idx* ptrs)
```

This creates a sparse matrix "view" to pre-existing memory, which must be in the traditional compressed row/column matrix format given by the three arrays `val`, `ptrs`, and `inds`, the number of rows and columns, `m` and `n` and the number of nonzeros `nnz`. The `val` and `inds` arrays should be of length `nnz`, and the `ptrs` array should be either length `m + 1` for a row major matrix or `n + 1` for a column oriented matrix. The `m_origin` and `n_origin` parameters specify the coordinate system for the indices of the matrix.

11.2.3 `matrix<T, Shape, array<SparseOneD>, Orien>::type`

Template Parameters

`T`	is the value type, the type of object stored in the matrix.
`Shape`	is the shape of the matrix, either `rectangle<>`, `symmetric<Uplo>`, `hermitian<Uplo>`, or `triangle<Uplo,Diag>`.
`SparseOneD`	is the type used for the one-dimensional segments (row or columns) of the matrix. The choices are `compressed<Idx,Alloc,IdxStart>`, `sparse_pair<Alloc>`, and `tree<Alloc>`.
`Orien`	is the orientation of the matrix, either `row_major` or `column_major`.

11.2.4 `matrix<T, Shape, coordinate<Alloc>, Orien>::type`

11.3 Banded Matrices

11.3.1 matrix<T,banded<>,banded<Allocator>,Orien>::type

11.3.2 matrix<T,banded<>,banded<external>,Orien>::type

11.3.3 matrix<T,banded<>,dense<Allocator>,Orien>::type

11.3.4 matrix<T,banded<>,dense<external>,Orien>::type

11.4 Triangular Matrices

11.4.1 matrix<T, triangle<Uplo,Diag>, Storage, Orien>::type

Template Parameters

- `Uplo` specifies whether the non-zeroes fall in the upper or lower triangle of the matrix. The valid arguments for this parameter are `upper`, `lower` and `dynamic_uplo`, which specifies that the choice between upper and lower is postponed until run-time.
- `Diag` specifes whether the matrix is guaranteed to be unit diagonal. Some algorithms are specialized to be more efficient for unit diagonal matrices. The valid choices for this parameter are `unit_diag`, `non_unit_diag` and `dynamic_diag`.

11.4.2 matrix<T, triangle<Uplo,Diag>, dense<Allocator>,Orien>::type

11.4.3 matrix<T, triangle<Uplo,Diag>, dense<external>, Orien>::type

11.4.4 matrix<T, triangle<Uplo,Diag>, banded<Allocator>,Orien>::type

11.4.5 matrix<T, triangle<Uplo,Diag>, banded<external>, Orien>::type

11.4.6 matrix<T, triangle<Uplo,Diag>, packed<Allocator>, Orien>::type

11.4.7 matrix<T, triangle<Uplo,Diag>, packed<external>, Orien>::type

11.5 Symmetric Matrices

11.5.1 matrix<T, symmetric<Uplo>, Storage, Orien>::type

Template Parameters

Uplo specifies whether the matrix elements of the upper or lower triangle are the ones stored. The valid arguments for this parameter are upper, lower and dynamic_uplo, which specifies that the choice between upper and lower is postponed until run-time.

11.5.2 matrix<T, symmetric<Uplo>, dense<Allocator>,Orien>::type

11.5.3 matrix<T, symmetric<Uplo>, dense<external>, Orien>::type

11.5.4 matrix<T, symmetric<Uplo>, banded<Allocator>,Orien>::type

11.5.5 matrix<T, symmetric<Uplo>, banded<external>, Orien>::type

11.5.6 matrix<T, symmetric<Uplo>, packed<Allocator>, Orien>::type

11.5.7 matrix<T, symmetric<Uplo>, packed<external>, Orien>::type

11.6 Hermitian Matrices

11.6.1 `matrix<T, hermitian<Uplo>, Storage, Orien>::type`

Template Parameters

`Uplo` specifies whether the matrix elements of the upper or lower triangle are the ones stored. The valid arguments for this parameter are `upper`, `lower` and `dynamic_uplo`, which specifies that the choice between upper and lower is postponed until run-time.

11.6.2 `matrix<T, hermitian<Uplo>, dense<Allocator>,Orien>::type`

11.6.3 `matrix<T, hermitian<Uplo>, dense<external>, Orien>::type`

11.6.4 `matrix<T, hermitian<Uplo>, banded<Allocator>,Orien>::type`

11.6.5 `matrix<T, hermitian<Uplo>, banded<external>, Orien>::type`

11.6.6 `matrix<T, hermitian<Uplo>, packed<Allocator>, Orien>::type`

11.6.7 `matrix<T, hermitian<Uplo>, packed<external>, Orien>::type`

11.7 Diagonal Matrices

11.7.1 matrix<T, banded<>, dense<Allocator>, diagonal_major>::type

11.7.2 matrix<T, banded<>, dense<external>, diagonal_major>::type

11.7.3 matrix<T, banded<>, array<OneD>, diagonal_major>::type

Chapter 12

Adaptors and Helper Functions

Chapter 13

Overloaded Operators

Chapter 14

Vector Operations

The time complexity of the MTL vector operations is linear in the size of the vector, or in the number of non-zeroes for sparse vectors.

14.1 Vector Data Movement Operations

14.1.1 set

$x_i \leftarrow \alpha$

```
template <class Vector, class T>
void set(Vector& x, const T& alpha)
```

This operation assigns the value of `alpha` to each element of vector `x`. For a sparse vector `alpha` is assigned to only the non-zero, explicitly *stored* elements of the vector.

Equivalent MXTL Expression

```
x = alpha
```

Requirements on Types

- `Vector` must be a model of the Vector concept.
- `T` must be convertible to the value type of `Vector`.

Complexity

Linear. n stores are performed into vector `x`.

Example

```
mtl::vector<int>::type x(10);

mtl::set(x, 2);

for (int i = 0; i < 10; ++i)
  assert(x[i] == 2);
```

14.1.2 copy

$y \leftarrow x$

```
template <class VectorX, class VectorY>
void copy(const VectorX& x, VectorY& y)
```

This operation copies all of the elements of the vector `x` into the vector `y`. The two vectors must be the same size. After the copy is performed the vectors will be element-wise equal, that is `x[i] == y[i]` for `i=0...n-1`. When `y` is a sparse vector, the old sparse structure of `y` is thrown away along with the old values, and replaced by the new values and indices of vector `x`. When `x` is a sparse vector and `y` is a dense vector (copying from sparse to dense), the non-zero elements of `y` are copied into the appropriate positions in `x` and the other elements of `x` are set to zero. If you do not want the other elements of `x` set to zero use the `scatter()` function instead. It is not recommended to copy a dense vector into a sparse vector, since the sparse vector formats are inefficient for storing and manipulating a full vector.

Requirements on Types

- `VectorX` and `VectorY` must be models of the Vector concept.
- The value type of `VectorX` must be convertible to the value type of `VectorY`.

Preconditions

- `x.size() == y.size()`

Postconditions

- `x[i] == y[i]` for `i=0...n-1`.

Complexity

Linear. If either vector is dense, then there are N assignments. If both vectors are sparse then there are nnz assignments. If the target vector is sparse then some memory allocation and or deallocation may occur.

Example

Copy a compressed sparse vector into another sparse vector.

```
typedef mtl::vector<int, compressed<> >::type CompVec;
CompVec x(15), y(15);
for (int i = 0; i < x.size(); i += 3)
  x[i] = i;

mtl::copy(x, y);

for (int i = 0; i < x.size(); ++i)
  assert(x[i] == y[i]);
```

14.1.3 swap

$x \leftrightarrow y$

```
template <class VectorX, class VectorY>
void swap(VectorX& x, VectorY& y)
```

This function swaps the elements of vector `x` with the elements of vector `y`.

Requirements on Types

- `VectorX` and `VectorY` must be models of the Vector concept.
- The value type of `VectorX` must be convertible to the value type of `VectorY`, and vice versa.

Preconditions

- `x.size() == y.size()`

Complexity

Linear. $2n$ loads and stores are performed.

Example

```
typedef mtl::vector<int>::type Vec;
Vec x(10), y(5);

mtl::set(x, 1);
mtl::set(y, 2);

swap(strided(x, 2), y);
cout << x << endl;
cout << y << endl;
```

The output is:

```
[2,1,2,1,2,1,2,1,2,1]
[1,1,1,1,1]
```

14.1.4 gather

$y \leftarrow x|_y$

```
template <class DenseVector, class SparseVector>
void gather(const DenseVector& x, SparseVector& y)
```

This function copies the elements of dense vector `x` into sparse vector `y` according to the non-zero structure of vector `y`.

Equivalent MXTL Expression

`y << x`

Preconditions

- `x.size() == y.size()`.

Requirements on Types

- `DenseVector` and `SparseVector` must be models of the Vector concept.
- The value type of `DenseVector` must be convertible to the value type of `SparseVector`.

Complexity

Linear. nnz loads and stores are performed.

Example

```
typedef mtl::vector<int, compressed<external> >::type SparseVec;
typedef mtl::vector<int>::type DenseVec;

const int n = 9, nnz = 3;

int y_values[] = { 0, 0, 0 };
int y_indices[] = { 2, 5, 7 };

SparseVec y(y_values, y_indices, n, nnz);
DenseVec x(n);

for (int i = 0; i < n; ++i)
  x[i] = 2*i;

mtl::gather(x, y);
cout << y << endl;
```

The output is:

```
[(4,2),(10,5),(14,7)]
```

14.1.5 scatter

$y|_x \leftarrow x$

```
template <class SparseVector, class DenseVector>
void scatter(const SparseVector& x, DenseVector& y)
```

This function copies the non-zero elements of the sparse vector `x` into the dense vector `y`. The elements of `y` that correspond to zeroes in `x` are left unchanged.

Equivalent MXTL Expression

```
x >> y
```

Preconditions

- `x.size() == y.size()`.

Requirements on Types

- `DenseVector` and `SparseVector` must be models of the Vector concept.
- The value type of `SparseVector` must be convertible to the value type of `DenseVector`.

Complexity

Linear. *nnz* loads and stores are performed.

Example

```
typedef mtl::vector<int, compressed<external> >::type SparseVec;
typedef mtl::vector<int>::type DenseVec;

const int n = 9, nnz = 3;

int x_values[] = { 4, 4, 4 };
int x_indices[] = { 2, 5, 7 };

SparseVec x(x_values, x_indices, n, nnz);
DenseVec y(n);

y = 1;

mtl::scatter(x, y);
cout << y << endl;
```

The output is:

```
[1,1,4,1,1,4,1,4,1]
```

14.2 Vector Reduction Operations

14.2.1 dot (inner product)

$r \leftarrow r + x \cdot y$

```
template <class VectorX, class VectorY, class T>
T dot(const VectorX& x, VectorY& y, T r)
```

$r \leftarrow x \cdot y$

```
template <class VectorX, class VectorY>
T dot(const VectorX& x, VectorY& y)
```

In the second version `T` is defined by

```
XT = typename VectorX::value_type
YT = typedef typename VectorY::value_type
T = typename binary_op_trait<mult_tag, XT, YT>::result_type
```

This operation computes the inner product of two vectors, that is it returns the value of `r` which is calculated by `r = r + x[i] * y[i]` for `i=0...n-1`. In version 2 of the algorithm the initial value for `r` is obtained via `binary_op_trait<add_tag,T,T>::identity()`.

Requirements on Types

- `VectorX` and `VectorY` must be models of the Vector concept.
- The value type of `VectorX` and the value type of `VectorY` must be Multipliable.
- The result type and type `T` of the multiply must be Addable.

Preconditions

- `x.size() == y.size()`

Complexity

Linear. The algorithm performs n multiplications and additions and $2n$ loads if both vectors are dense. The algorithm performs nnz multiplications and additions and $2nnz$ loads if at least one vector is sparse.

Example

Calculate the dot product of two orthogonal vectors.

```
typedef mtl::vector<int, static_size<3> >::type Vec;
Vec x = { 1, 2, 3 };
Vec y = { 3, 0, -1 };

int r = mtl::dot(x, y);
assert(r == 0);
```

14.2.2 one_norm

$r \leftarrow \parallel x \parallel_1$ or equivalently $r \leftarrow \sum_i |x_i|$

```
template <class Vector>
T one_norm(const Vector& x)

VT = typename Vector::value_type
T = typename magnitude<VT>::type
```

This algorithm calculates the one norm of a vector, which is the sum of the absolute values of the elements.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be Addable.
- The `abs()` function must be defined for the value type of `Vector`.

Complexity

Linear. n loads, additions, and `abs()` are performed (nnz for sparse vectors)..

Example

Calculate the one norm of a complex vector.

```
typedef std::complex<double> Z;
typedef mtl::vector<Z>::type Vec;

Vec x(10); // create a vector size 10

x = Z(2.0, 1.0)); // fill vector will complex number (2,1)

double r = mtl::one_norm(x);
cout << r << endl;
```

The output is:

```
22.3607
```

14.2.3 two_norm (euclidean norm)

$r \leftarrow \| x \|_2$ or equivalently $r \leftarrow \sqrt{\sum_i x_i^2}$

```
template <class Vector>
T two_norm(const Vector& x)

T = typename Vector::value_type
```

This operation calculates the two norm (or euclidean norm) of a vector, which is calculated by taking the square root of the sum of the squares of all the elements.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be Addable and Multipliable.
- The `sqrt()` function must be defined for the value type of `Vector`.

Complexity

Linear. n loads, multiplies and adds are performed. The `sqrt()` function is invoked once (nnz for sparse vectors).

Example

Calculate the two norm of a vector

```
typedef mtl::vector<double>::type Vec;
Vec x(10);
x = 2.0;

double r = mtl::two_norm(x);
cout << r << endl;
```

The output is:

```
6.32456
```

14.2.4 infinity_norm

$r \leftarrow \| x \|_\infty$ or equivalently $r \leftarrow \max_i |x_i|$

```
template <class Vector>
T infinity_norm(const Vector& x)
```

```
VT = typename Vector::value_type
T = typename magnitude<VT>::type
```

This algorithm computes the infinity norm of a vector, which is equal to the maximum absolute value of any element in the vector.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The `abs()` function must be defined for the value type of `Vector`.
- The result type of the `abs()` operation must be LessThanComparable.

Complexity

Linear. n loads, `abs()`, and comparisons are performed (nnz for sparse vectors).

Example

Find the infinity norm of a complex vector.

```
typedef std::complex<double> Z;
typedef mtl::vector<Z>::type Vec;

Vec x(10); // create a vector size 10

for (int i = 0; i < x.size(); ++i)
  x = Z(2*(i+1), i+1));

double r = mtl::infinity_norm(x);
cout << r << endl;
```

The output is:

```
UNDER CONSTRUCTION
```

14.2.5 sum

$r \leftarrow \sum_i x_i$

```
template <class Vector>
T sum(const Vector& x)

T = typename Vector::value_type
```

This operation calculates the sum of the elements in the vector. The initial value of `r` is obtained via `binary_op_trait<add_tag,T,T>::identity()`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be a model of Addable.

Complexity

Linear. The algorithm performs N loads and additions.

Example

The sum of an arithmetic series, $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$.

```
typedef mtl::vector<int>::type Vec;
const int N = 10;
Vec x(N);
for (int i = 0; i < N; ++i)
  x[i] = i + 1;

int r = mtl::sum(x);

assert(r == N * (N + 1) / 2);
```

14.2.6 sum_squares

$r \leftarrow \sum_i x_i^2$

```
template <class Vector>
T sum_squares(const Vector& x)

T = typename Vector::value_type
```

This operation calculate the sum of the squares of the elements in the vector. The initial value of `r` is obtained via `binary_op_trait<add_tag,T,T>::identity()`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be a model of Addable and Multipliable.

Complexity

Linear. The algorithm performs N loads, additions, and multiplications.

Example

```
UNDER CONSTRUCTION
```

14.2.7 `max`

$r \leftarrow max_i(x_i)$

```
template <class Vector>
T max(const Vector& x)

T = typename Vector::value_type
```

This function returns the maximum value of any of the elements of vector `x`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be LessThanComparable. For example, complex numbers are *not* LessThanComparable.

Complexity

Linear.

Example

```
typedef mtl::vector<int, static_size<4> >::type Vec;
Vec x = { 3, 1, 7, 4 };

int r = mtl::max(x);
assert(r == 7);
```

14.2.8 `min`

$r \leftarrow min_i(x_i)$

```
template <class Vector>
T min(const Vector& x)

T = typename Vector::value_type
```

This function returns the maximum value of any of the elements of vector `x`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be LessThanComparable. For example, complex numbers are *not* LessThanComparable.

Complexity

Linear.

Example

```
typedef mtl::vector<int, static_size<4> >::type Vec;
Vec x = { 3, 1, 7, 4 };

int r = mtl::min(x);
assert(r == 1);
```

14.2.9 max_index

$k \leftarrow arg\ max_i(x_i)$

```
template <class Vector>
S max_index(const Vector& x)

S = typename Vector::size_type
```

This function finds the index (location) of the maximum element in vector `x`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be LessThanComparable.

Complexity

Linear.

Example

```
using boost tie;
typedef mtl::vector<double, static_size<5> >::type Vec;
Vec x = { 5.0, -7.0, -4.0, 6.0 , 0.0 };
int k;

k = mtl::max_index(x);
assert(k == 3);

k = mtl::max_index(abs(x));
assert(k == 1);
```

14.2.10 max_with_index

$(k, x_k) \leftarrow arg\ max_i(x_i)$

```
template <class Vector>
std::pair<S,T> max_with_index(const Vector& x)

S = typename Vector::size_type
T = typename Vector::value_type
```

This function finds the location (index) and value of the maximum element in vector `x`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be LessThanComparable.

Complexity

Linear.

Example

```
using boost tie;
typedef mtl::vector<double, static_size<5> >::type Vec;
Vec x = { 5.0, -7.0, -4.0, 6.0 , 0.0 };
int k;
double xk;

tie(k,xk) = mtl::max_with_index(x);
assert(k == 3 && xk == 6.0);

tie(k,xk) = mtl::max_with_index(abs(x));
assert(k == 1 && xk == 7.0);
```

14.2.11 `min_index`

$(k, x_k) \leftarrow arg\ min_i(x_i)$

```
template <class Vector>
std::pair<S,T> min_index(const Vector& x)

S = typename Vector::size_type
T = typename Vector::value_type
```

This function finds the location (index) and value of the minimum element in vector `x`.

Requirements on Types

- `Vector` must be a model of the Vector concept.
- The value type of `Vector` must be LessThanComparable.

Complexity

Linear.

Example

```
using boost::tie;
typedef mtl::vector<double, static_size<5> >::type Vec;
Vec x = { 5.0, -7.0, -4.0, 6.0 , 0.0 };
int k;
double xk;

tie(k,xk) = mtl::min_index(x);
assert(k == 1 && xk == -7.0);

tie(k,xk) = mtl::min_index(abs(x));
assert(k == 4 && xk == 0.0);
```

14.3 Vector Arithmetic Operations

14.3.1 scale

$x \leftarrow \alpha x$

```
template <class Vector, class T>
void scale(Vector& x, const T& alpha)
```

This operation multiplies each element of vector `x` by the scalar `alpha`.

Equivalent MXTL Expression

```
x *= alpha
```

Requirements on Types

- The type `Vector` must be a model of the Vector concept.
- The value type of `Vector` and type `T` must be Multipliable.

Complexity

Linear. The algorithm performs n loads, stores and multiplications if the vector is dense, and nnz if the vector is sparse.

Example

Perform one of the steps in a Gaussian Elimination by scaling a row of the matrix.

```
typedef mtl::matrix<double, rectangle<>,
                    static_size<3,3>, row_major>::type Mat;

Mat A = { 5.0, 5.5, 6.0,
          2.5, 3.0, 3.5,
          1.0, 1.5, 2.0 };

double scal = A(0,0) / A(1,0);
mtl::scale(A[1], scal);
assert(A(0,0) == A(1,0));
```

14.3.2 add

$y \leftarrow x + y$

```
template <class VectorX, class VectorY>
void add(const VectorX& x, VectorY& y)
```

$w \leftarrow x + y$

```
template <class VectorX, class VectorY, class VectorW>
void add(const VectorX& x, const VectorY& y, VectorW& w)
```

$w \leftarrow x + y + z$

```
template <class VectorX, class VectorY, class VectorZ, class VectorW>
void add(const VectorX& x, const VectorY& y, const VectorZ& z, VectorW& w)
```

These algorithms perform element-wise addition of vectors, and place the results either in vector `y` or in the output vector `w`.

Equivalent MXTL Expressions

```
add(x, y)          y += x
add(x, y, w)       w = x + y
add(x, y, z, w)    w = x + y + z
```

Preconditions

- For all versions: `x.size() == y.size()`.
- For version 2 and 3, `x.size() == w.size()`.
- For version 3, `x.size() == z.size()`.

Requirements on Types

- `VectorX`, `VectorY`, `VectorZ` and `VectorW` must be models of the Vector concept.
- The value types of `VectorX`, `VectorY`, and `VectorZ` must be Addable.
- The result type of the addition must be convertible to the value type of the output vector, either `VectorY` or `VectorW`.

Complexity

Linear. The algorithm performs $2n$ loads and n additions and stores. If both vectors are sparse, only $O(nnz)$ operations are performed. In addition, if the output vector is sparse some allocation and or deallocation may occur.

Examples

Add two vectors into a third.

```
typedef mtl::vector<int, static_size<3> >::type Vec;
Vec x = { 1, 1, 1 };
Vec y = { 3, 3, 3 };
Vec w;

mtl::add(x, y, w);
cout << w << endl;
```

The output is:

```
[4,4,4]
```

Perform one of the steps in a Gaussian Elimination by subtracting one row of the matrix from another.

```
typedef mtl::matrix<double, rectangle<>,
                    static_size<3,3>, row_major>::type Mat;

Mat A = { 5.0, 5.5, 6.0,
          5.0, 6.0, 7.0,
          1.0, 1.5, 2.0 };

double scal = A(0,0) / A(1,0);
mtl::add(mtl::scaled(A[0],-1), A[1]);
assert(A(1,0) == 0.0);
```

14.3.3 `iadd`

$y \leftarrow y + x|_y$

```
template <class VectorX, class VectorY>
void iadd(const VectorX& x, VectorY& y)
```

This algorithm performs incomplete addition for sparse vectors. This means that only the elements of `x` that correspond to non-zero elements in `y` are added. The non-zero structure of vector `y` is left unchanged.

Preconditions

- `x.size() == y.size()`.

Requirements on Types

- `VectorX` and `VectorY` must be models of the Vector concept.
- The value types of `VectorX` and `VectorY` must be Addable.

Complexity

Linear. The algorithm performs $2nnz$ loads and nnz additions and stores.

Example

Add two sparse vectors, with respect to the sparsity structure of the destination vector.

```
typedef std::pair<int,double> P;

const int n = 6, x_nnz = 3, y_nnz = 4;

P xp[] = { P(1,11.0), P(3,1.0), P(5,4.0) };
P yp[] = { P(0,4.0), P(3,2.0), P(4,6.0), P(5,3.0) };

typedef mtl::vector<int, sparse_pair<external> >::type Vec;
Vec x(n, x_nnz, xp), y(n, y_nnz, yp);

mtl::iadd(x, y);
cout << y << endl;
```

The output is:

```
[(0,4),(3,3),(4,6),(5,7)]
```

14.3.4 `ele_mult`

$w \leftarrow x \otimes y$

```
template <class VectorX, class VectorY, class VectorW>
void ele_mult(const VectorX& x, const VectorY& y, VectorW& w)
```

Element-wise multiply vector `x` and `y`, placing the result in vector `z`.

Preconditions

- `x.size() == y.size() && x.size() == w.size()`.

Requirements on Types

- `VectorX`, `VectorY`, and `VectorW` must be models of the Vector concept.
- The value types of `VectorX` and `VectorY` must be Multipliable.
- The result type of the multiply must be convertible to the value type of `VectorW`.

Complexity

Linear. The algorithm performs $2n$ loads and n multiplies and stores if both vectors are dense. If either input vector is sparse then the algorithm performs $2nnz$ loads and nnz multiplies, and either n or nnz stores depending on whether the output vector is dense or sparse. Also, if the output vector is sparse some allocation and or deallocation may occur.

Example

```
typedef mtl::vector<int, static_size<3> >::type Vec;
Vec x = { 2, 2, 2 };
Vec y = { 3, 3, 3 };
Vec w;

mtl::ele_mult(x, y, w);
cout << w << endl;
```

The output is:

```
[6,6,6]
```

14.3.5 ele_div

$w \leftarrow x \oslash y$

```
template <class VectorX, class VectorY, class VectorW>
void ele_div(const VectorX& x, const VectorY& y, VectorW& w)
```

Element-wise divide vector `x` and `y`, placing the result in vector `z`. Note that if vector `y` is sparse a divide by zero error will surely occur. If vector `x` is sparse, then `INF` will typically be assigned to many of the elements of `w`.

Preconditions

- `x.size() == y.size() && x.size() == w.size()`.

Requirements on Types

- `VectorX`, `VectorY`, and `VectorW` must be models of the Vector concept.
- The value types of `VectorX` and `VectorY` must be Dividable.
- The result type of the divide must be convertible to the value type of `VectorW`.

Complexity

Linear. The algorithm performs $2n$ loads and n divides and stores if both input vectors are dense. If vector `x` is sparse then the algorithm performs $2nnz$ loads, nnz multiplies, and n stores.

Example

```
typedef mtl::vector<int, static_size<3> >::type Vec;
Vec x = { 6, 6, 6 };
Vec y = { 2, 2, 2 };
Vec w;

mtl::ele_div(x, y, w);
cout << w << endl;
```

The output is:

```
[3,3,3]
```

Chapter 15

Matrix Operations

15.1 Matrix Data Movement Operations

15.1.1 set

$a_{ij} \leftarrow \alpha$

```
template <class Matrix, class T>
void set(Matrix& A, const T& alpha)
```

This operation assigns the value of `alpha` to each *stored* element of the matrix `A`. For some matrices (sparse, banded, unit diagonal, etc.), the post-condition `A(i,j) == alpha` for all `i=0...m-1,j=0...n-1` will *not* hold, as the non-stored elements will still have their assumed value (typically zero or one).

Equivalent MXTL Expression

```
A = alpha
```

Requirements on Types

- `Matrix` must be a model of the Matrix concept.
- The type `T` must be convertible to the value type of `Matrix`.

Complexity

The algorithm will perform mn stores for a dense matrix, and nnz stores for a sparse matrix.

Example

```
mtl::matrix<double>::type A(2, 2);

mtl::set(A, 7.3);
cout << A << endl;
```

The output is:

```
[[7.3,7.3],
 [7.3,7.3]]
```

15.1.2 copy

$B \leftarrow A$

```
template <class MatrixA, class MatrixB>
void copy(const MatrixA& A, MatrixB& B)
```

This operation copies the elements of matrix `A` into matrix `B`. After the copy, the following post-condition holds: `A(i,j) == B(i,j)` for all `i=0...m-1,j=0...n-1`. The copy function is useful for converting from one matrix format to another. For instance, during the initialization phase of some algorithm it may be more efficient to use a sparse matrix that supports fast insertion, and then in the computation phase change to another sparse matrix format that supports fast traversal. When using this routine on banded or diagonal matrices some care must be taken to avoid the situation where elements of matrix `A` map to positions in matrix `B` that are out-of-bounds.

Preconditions

- `A.nrows() == B.nrows() && A.ncols() == B.ncols()`

Postconditions

- `A(i,j) == B(i,j)` for all `i=0...m-1,j=0...n-1` (that is if the value types of the matrices are EqualityComparable).

Requirements on Types

- `MatrixA` and `MatrixB` must be models of the Matrix concept.
- `typeid(MatrixA::orientation) == typeid(MatrixB::orientation)`
- The value type of `MatrixA` must be convertible to the value type of `MatrixB`.

Complexity

The algorithm will perform mn stores for a dense matrix, and nnz stores for a sparse matrix.

Example

```
typedef mtl::matrix<double, rectangle<>,
          array<tree>, row_major>::type TreeMat;
typedef mtl::matrix<double, rectangle<>,
          compressed<>, row_major>::type CompMat;

const int m = 5, n = 5;
TreeMat A(m,n);
CompMat B(m,n);

// insert some values at random locations
for (int i = 0; i < m*2; ++i)
  A(rand() % 5, rand() % 5) = i;

mtl::copy(A, B);
```

```
assert(A == B);
```

15.1.3 swap

$A \leftrightarrow B$

```
template <class MatrixA, class MatrixB>
void swap(MatrixA& A, MatrixB& B)
```

This functions exchanges the elements of matrix `A` with the elements of matrix `B`. The routine assumes the two matrices are the same orientation (e.g., both row major) and that the non-zero structure of the two matrices is identical. Anotherwords, for sparse matrices only the *values* are swapped, and not the indices.

Preconditions

- `A.nrows() == B.nrows() && A.ncols() == B.ncols()`
- The non-zero structure of `A` and `B` must match.

Requirements on Types

- `MatrixA` and `MatrixB` must be models of the Matrix concept.
- `typeid(MatrixA::orientation) == typeid(MatrixB::orientation)`
- `typeid(MatrixA::sparsity) == typeid(MatrixB::sparsity)`
- The value type of `MatrixA` must be convertible to the value type of `MatrixB` and vice versa.

Complexity

The algorithm will perform $2mn$ loads and stores for dense matrices, and $2nnz$ loads and stores for sparse matrices.

Example

```
typedef mtl::matrix<double, banded<>,
                    banded<>, row_major>::type BandedMatrix;
typedef mtl::matrix<double, banded<>,
                    packed<>, row_major>::type PackedMatrix;

const int m = 4, n = 3, kl = 2, ku = 1;

BandedMatrix A(m, n, kl, ku);
PackedMatrix B(m, n, kl, ku);
```

```
A = 2.5;
B = 1.4;

mtl::swap(A, B);

// The elements in the non-zero band of A should be == 1.4
BandedMatrix::iterator Ai = A.begin()
for (; Ai != A.end(); ++Ai) {
  BandedMatrix::Row::iterator Aij = (*Ai).begin();
  for (; Aij != (*Ai).end(); ++Aij)
     assert(*Aij == 1.4);
}
```

15.1.4 transpose

$A \leftarrow A^T$ or equivalently $a_{ij} \leftrightarrow a_{ji}$

```
template <class Matrix>
void transpose(Matrix& A)
```

$B \leftarrow A^T$ or equivalently $b_{ji} \leftarrow a_{ij}$

```
template <class MatrixA, class MatrixB>
void transpose(const MatrixA& A, MatrixB& B)
```

This function moves each element of the matrix to the mirror-image of its location, across the main diagonal of the matrix. The element at `(i,j)` will be moved to `(j,i)`. Version 1 of the algorithm transposes the matrix in place, while version 2 places the result in matrix `B`.

Preconditions

- For version 2, `A.nrows() == B.ncols() && A.ncols() == B.nrows()`
- Allow what kind of sparse/dense/banded/orientation combinations? JGS

Requirements on Types

- `MatrixA` and `MatrixB` must be models of the Matrix concept.
- The value type of `MatrixA` must be convertible to the value type of `MatrixB` and vice versa.
- For version 1, if the matrix is static sized, it must also be square for the transpose operation to work correctly.

Complexity

The algorithm performs $O(mn)$ loads and stores.

Examples

Transpose a matrix in place.

```
typedef mtl::matrix<int, rectangle<>,
                    static_size<3,3>, row_major>::type Matrix;

Matrix A = { 1, 2, 3,
             4, 5, 6,
             7, 8, 9 };
mtl::transpose(A);
cout << A << endl;
```

The output is:

```
[[1,4,7],
 [2,5,8],
 [3,6,9]]
```

Assign the transpose to a second matrix.

```
mtl::matrix<int>::type A(3,2), B(2,3);

for (int i = 0; i < 3; ++i)
  for (int j = 0; j < 2; ++j)
    A(i,j) = i * 2 + j + 1;

mtl::transpose(A, B);
cout << A << endl << endl;
cout << B << endl;
```

The output is:

```
[[1,2,3],
 [4,5,6]]

[[1,4],
 [2,5],
 [3,6]]
```

15.2 Matrix Norms

15.2.1 one_norm

$r \leftarrow \| A \|_1$ or equivalently $r \leftarrow \max_j \sum_i |a_{ij}|$

```
template <class Matrix>
T one_norm(const Matrix& A)

VT = typename Matrix::value_type
T = typename unary_op_trait<abs_tag,VT>::result_type
```

This function returns the one norm of a matrix, which is the maximum of the column sums.

Requirements on Types

- `Matrix` must be a model of the Matrix concept.
- The `abs()` function must be defined for the value type of `Matrix`.
- The result type of the `abs()` function must be Addable and LessThanComparable.

Complexity

$O(mn)$ for dense matrices and $O(nnz)$ for sparse.

Example

```
UNDER CONSTRUCTION
```

15.2.2 frobenius_norm

$r \leftarrow \| A \|_F$ or equivalently $r \leftarrow \sqrt{\sum_{ij} |a_{ij}|^2}$

```
template <class Matrix>
T frobenius_norm(const Matrix& A)

VT = typename Matrix::value_type
T = typename unary_op_trait<abs_tag,VT>::result_type
```

The fobenius norm is calculated by taking the square root of the sum of the squares of all the elements in the matrix.

Requirements on Types

- `Matrix` must be a model of the Matrix concept.
- The value type of `Matrix` must be Addable.
- The `abs()` function must be defined for the value type of `Matrix`.
- The `sqrt()` function must be defined for the value type of `Matrix`.

Complexity

$O(mn)$ for dense matrices and $O(nnz)$ for sparse.

Example

```
UNDER CONSTRUCTION
```

15.2.3 infinity_norm

$r \leftarrow \parallel A \parallel_\infty$ or equivalently $r \leftarrow \max_i \sum_j |a_{ij}|$

```
template <class Matrix>
T infinity_norm(const Matrix& A)

VT = typename Matrix::value_type
T = typename unary_op_trait<abs_tag,VT>::result_type
```

The infinity norm of a matrix is the maximum row sum.

Requirements on Types

- `Matrix` must be a model of the Matrix concept.
- The `abs()` function must be defined for the value type of `Matrix`.
- The result type of the `abs()` function must be Addable and LessThanComparable.

Complexity

$O(mn)$ for dense matrices and $O(nnz)$ for sparse.

Example

```
UNDER CONSTRUCTION
```

15.3 Element-wise Arithmetic Operations

15.3.1 scale

$A \leftarrow \alpha A$

```
template <class Matrix, class T>
void scale(Matrix& A, const T& alpha)
```

This function multiplies each element in matrix `A` by `alpha`. For matrices with special structure (sparse, banded, etc.) only the stored elements are scaled.

Requirements on Types

- `Matrix` must be a model of the Matrix concept.
- The value type of `Matrix` and type `T` must be Multipliable.

Complexity

The algorithm performs $O(mn)$ loads, stores, and multiplies for dense matrices and $O(nnz)$ for sparse.

Example

Multiply the elements of a sparse matrix by 2.

```
typedef mtl::matrix<double, rectangle<>,
   compressed<>, row_major>::type Matrix;

Matrix A(3,3);

A(0,1) = 2.5;
A(1,0) = 0.2;
A(1,2) = 1.4;
A(2,0) = 4.1;

mtl::scale(A, 2.0);
cout << A << endl;
```

The output is:

```
[[(1,5.0)],
 [(0,0.4),(2,2.8)],
 [(0,8.2)]]
```

15.3.2 add

$B \leftarrow A + B$

```
template <class MatrixA, class MatrixB>
void add(const MatrixA& A, MatrixB& B)
```

$C \leftarrow A + B$

```
template <class MatrixA, class MatrixB, class MatrixC>
void add(const MatrixA& A, const MatrixB& B, MatrixC& C)
```

This function adds the elements of the two matrices, and assigns the result to matrix `B` in version 1 or to matrix `C` in version 2. If the destination matrix is banded, the other matrices must have the same shape (bandwidth) so as to avoid assignment to out-of-bounds elements. If the destination matrix is sparse, then some allocation and or deallocation may occur.

Equivalent MXTL Expressions

```
add(A, B)      B += A
add(A, B, C)   C = A + B
```

Preconditions

- For version 1, `A.nrows() == B.nrows() && A.ncols() == B.ncols()`.
- For version 2, `A.nrows() == B.nrows() && A.ncols() == B.ncols() && A.nrows() == C.nrows() && A.ncols() == C.ncols()`.

Requirements on Types

- `MatrixA`, `MatrixB`, and `MatrixC` must be models of the Matrix concept.
- `typeid(MatrixA::orientation) == typeid(MatrixB::orientation)`.
- For version 2, `typeid(MatrixA::orientation) == typeid(MatrixC::orientation)`.
- The value type of `MatrixA` and `MatrixB` must be Addable.
- The result type of the addition must be convertible to the value type of the output matrix.

Complexity

The algorithm performs $O(mn)$ loads, stores, and additions for dense matrices, and $O(nnz)$ for sparse.

Example

```
const int m = 2, n = 2;
typedef mtl::matrix<int, rectangle<>,
  static_size<m,n> >::type Matrix;
Matrix A = { 1, 1,
             1, 1 };
Matrix B = { 3, 3,
             3, 3 };
Matrix C;

mtl::add(A, B, C);

for (int i = 0; i < m; ++i)
  for (int j = 0; j < n; ++j)
    assert(C(i,j) == 4);
```

15.3.3 iadd

$B \leftarrow B + A|_B$

```
template <class SparseMatrixA, class SparseMatrixB>
void iadd(const SparseMatrixA& A, SparseMatrixB& B)
```

This function performs an incomplete addition of the sparse matrix `A` to sparse matrix `B` according to the non-zero structure of `B`. Elements of `A` are added to `B` only if they match the indices of existing non-zeroes in `B`. The non-zero structure of `B` is not changed.

Preconditions

- `A.nrows() == B.nrows() && A.ncols() == B.ncols()`.

Requirements on Types

- `SparseMatrixA` and `SparseMatrixB` must be models of the Matrix concept.
- The value type of `SparseMatrixA` and `SparseMatrixB` must be Addable.
- `typeid(SparseMatrixA::orientation) == typeid(SparseMatrixB::orientation)`.

Complexity

The algorithm performs $O(nnz)$ loads, stores, and additions.

Example

```
UNDER CONSTRUCTION
```

15.3.4 ele_mult

$B \leftarrow B \otimes A$ or equivalently $b_{ij} \leftarrow b_{ij} a_{ij}$

```
template <class MatrixA, class MatrixB>
void ele_mult(const MatrixA& A, MatrixB& B)
```

This function performs element-wise multiplication of two matrices.

Preconditions

- `A.nrows() == B.nrows() && A.ncols() == B.ncols()`.

Requirements on Types

- `MatrixA` and `MatrixB` must be models of the Matrix concept.
- The value type of `MatrixA` and `MatrixB` must be Multipliable.
- `typeid(MatrixA::orientation) == typeid(MatrixB::orientation)`.

Complexity

The algorithm performs $O(mn)$ loads, stores, and additions if bother matrices are dense, and $O(nnz)$ if at least one matrix is sparse.

Example

```
UNDER CONSTRUCTION
```

15.3.5 ele_div

$B \leftarrow B \oslash A$ or equivalently $b_{ij} \leftarrow b_{ij} / a_{ij}$

```
template <class MatrixA, class MatrixB>
void ele_div(const MatrixA& A, MatrixB& B)
```

This function performs element-wise division of two matrices. This function can not be used with matrices of special structure (sparse, banded, etc.).

Preconditions

- `A.nrows() == B.nrows() && A.ncols() == B.ncols()`.

Requirements on Types

- `MatrixA` and `MatrixB` must be models of the Matrix concept.
- The value type of `MatrixA` and `MatrixB` must be Multipliable.
- `typeid(MatrixA::orientation) == typeid(MatrixB::orientation)`.

Complexity

The algorithm performs $O(mn)$ loads, stores, and additions if bother matrices are dense, and $O(nnz)$ if at least one matrix is sparse.

Example

```
UNDER CONSTRUCTION
```

15.4 Rank Updates (Outer Products)

The result matrix must be a full matrix (not sparse or banded). If the matrix is symmetric or hermitian then the vector `x` and `y` must be equal to maintain the symmetry of the resultant matrix.

15.4.1 `rank_one_update`

$A \leftarrow A + xy^T$ or equivalently $a_{ij} \leftarrow a_{ij} + x_i y_j$

```
template <class Matrix, class VectorX, class VectorY>
void rank_one_update(Matrix& A, const VectorX& x, const VectorY& y)
```

The rank one update function takes the outer product of two vectors and assigns the result to matrix `A`.

Equivalent MXTL Expression

```
A += x * trans(y)
```

15.4.2 `rank_one_conj`

$A \leftarrow A + xy^H$ or equivalently $a_{ij} \leftarrow a_{ij} + x_i \overline{y_j}$

```
template <class Matrix, class VectorX, class VectorY>
void rank_one_conj(Matrix& A, const VectorX& x, const VectorY& y)
```

The rank one update function takes the outer product of vector `x` and the conjugate of vector `y` and assigns the result to matrix `A`.

Equivalent MXTL Expression

```
A += x * conj(trans(y))
```

15.4.3 `rank_two_update`

$A \leftarrow A + xy^T + yx^T$ or equivalently $a_{ij} \leftarrow a_{ij} + x_i y_j + y_i x_j$

```
template <class Matrix, class VectorX, class VectorY>
void rank_two_update(Matrix& A, const VectorX& x, const VectorY& y)
```

Equivalent MXTL Expression

```
A += x * trans(y) + y * trans(x)
```

15.4.4 `rank_two_conj`

$A \leftarrow A + xy^H + yx^H$ or equivalently $a_{ij} \leftarrow a_{ij} + x_i \overline{y_j} + y_i \overline{x_j}$

```
template <class Matrix, class VectorX, class VectorY>
void rank_two_conj(Matrix& A, const VectorX& x, const VectorY& y)
```

Equivalent MXTL Expression

```
A += x * conj(trans(y)) + y * conj(trans(x))
```

15.5 Trianglular Solves (Forward & Backward Substitution)

15.5.1 tri_solve

$x \leftarrow T^{-1}x$

```
template <class Matrix, class Vector>
void tri_solve(const Matrix& T, Vector& x)
```

$B \leftarrow T^{-1}B$

```
template <class MatrixT, class MatrixB>
void tri_solve(const MatrixT& T, MatrixB& B, right_side)
```

$B \leftarrow BT^{-1}$

```
template <class MatrixT, class MatrixB>
void tri_solve(const MatrixT& T, MatrixB& B, left_side)
```

The triangular solve routines perform forward (for lower triangular matrices) or backward (for upper triangular matrices) substitution. The routines assume that the elements on the main diagonal are non-zero and do not check. This is because `tri_solve()` is typically used after the matrix has been factored, and the factoring routine will ensure that the main diagonal is non-zero. In addition, checking for zeroes would cause a small performance degradation.

Examples

Perform forward substitution on a lower-triangular matrix.

```
typedef mtl::matrix<double, triangle<lower>,
                    packed<>, row_major>::type Matrix;
typedef mtl::vector<double, static_size<3> > Vector;
const int N = 3;

Matrix A(N);
A(0,0) = 1;
A(1,0) = 2;  A(1,1) = 4;
A(2,0) = 3;  A(2,1) = 5; A(2,2) = 7;

Vector b = { 7, 46, 124};

mtl::tri_solve(A, b);

// The solution should be
Vector x = { 7, 46, 124 };

assert(b == x);
```

Solve the triangular system for multiple right-hand-sides.

```
typedef mtl::matrix<double>::type Matrix;
typedef mtl::matrix<double, triangle<upper>,
  packed<> >::type TriMatrix;

const int m = 3, n = 2;
TriMatrix U(m,m);
Matrix B(m,n);

U(0,0) = 1;  U(0,1) = 2;  U(0,2) = 4;
             U(1,1) = 3;  U(1,2) = 5;
                          U(2,2) = 6;

B(0,0) = 7;  B(0,1) = 34;
B(1,0) = 8;  B(1,1) = 42;
B(2,0) = 6;  B(2,1) = 36;

cout << U << endl << endl;
cout << B << endl << endl;

mtl::tri_solve(U, B, left_side());

cout << B << endl;
```

The output is:

```
[[1,2,4],
 [0,3,5],
 [0,0,6]]

[[7,34],
 [8,42],
 [6,36]]

[[1,2],
 [1,4],
 [1,6]]
```

15.6 Matrix-Vector Multiplication

15.6.1 mult

$y \leftarrow Ax$

```
template <class Matrix, class VectorX, class VectorY>
void mult(const Matrix& A, const VectorX& x, Vector& y)
```

Equivalent MXTL Expression

```
y = A * x
```

Example

```
UNDER CONSTRUCTION
```

15.6.2 mult_add

$y \leftarrow Ax + y$

```
template <class Matrix, class VectorX, class VectorY>
void mult_add(const Matrix& A, const VectorX& x, Vector& y)
```

$z \leftarrow Ax + y$

```
template <class Matrix, class VectorX,
          class VectorY, class VectorZ>
void mult_add(const Matrix& A, const VectorX& x,
              const VectorY& y, VectorZ& z)
```

Equivalent MXTL Expression

```
mult_add(A,x,y)        y += A * x
mult_add(A,x,y,z)      z = A * x + y
```

Example

```
  typedef mtl::matrix<double, banded<>,
    banded<>, row_major>::type Matrix;
  typedef mtl::vector<double>::type Vector;
  const int m = 5, n = 5, kl = 0, ku = 2;
  Matrix A(m, n, kl, ku);
  Vector y(m);
  Vector x(n);

  int val = 1;
  Matrix::iterator ri = A.begin();
  while (ri != A.end()) {
```

```
    Matrix::Row::iterator i = (*ri).begin();
    while (i != (*ri).end())
      *i++ = val++;
    ++ri;
  }
  for (int i = 0; i < n; ++i)
    x[i] = i + 1;
  mtl::set(y, 1);

  cout << A << endl << endl;
  cout << x << endl << endl;
  cout << y << endl << endl;
  mtl::mult_add(A, scaled(x, 2), scaled(y, 3), y);
  cout << y << endl;
```

The output is:

```
[[1,1,1,],
 [2,2,2,],
 [3,3,3,],
 [4,4,],
 [5,]]

[1,2,3,4,5,]

[1,1,1,1,1,]

[15,39,75,75,53,]
```

15.7 Matrix-Matrix Multiplication

15.7.1 mult

$C \leftarrow AB$

```
template <class MatrixA, class MatrixB, class MatrixC>
void mult(const MatrixA& A, const MatrixB& B, MatrixC& C)
```

15.7.2 mult_add

$C \leftarrow C + AB$

```
template <class MatrixA, class MatrixB, class MatrixC>
void mult_add(const MatrixA& A, const MatrixB& B, MatrixC& C)
```

15.7.3 lrdiag_mult

$C \leftarrow D_L A D_R$

```
template <class MatrixA, class MatrixDl,
          class MatrixDr, class MatrixC>
void lrdiag_mult(const MatrixA& A, const MatrixDr& Dr,
                 const MatrixDl& Dl, MatrixC& C)
```

15.7.4 lrdiag_mult_add

$C \leftarrow C + D_L A D_R$

```
template <class MatrixA, class MatrixDl,
          class MatrixDr, class MatrixC>
void lrdiag_mult(const MatrixA& A, const MatrixDr& Dr,
                 const MatrixDl& Dl, MatrixC& C)
```

15.7.5 babt_mult

$C \leftarrow BAB^T$

```
template <class MatrixA, class MatrixB, class MatrixC>
void babt_mult(const MatrixA& A, const MatrixB& B, MatrixC& C)
```

15.8 Miscellaneous Matrix Operations

15.8.1 trace

$r \leftarrow \sum_i a_{ii}$

```
template <class FastDiagMatrix>
T trace(const FastDiagMatrix& A)

T = typename FastDiagMatrix::value_type
```

This function returns the sum of the elements in the main diagonal of the matrix.

Requirements on Types

- The matrix must be a model of the FastDiagMatrix concept, which means there must be a `diag(A)` function available to provide access to the main diagonal.

Complexity

Linear. It will perform $\max(m, n)$ additions and loads.

Example

The implementation of this function is trivial:

```
namespace mtl {
  template <class FastDiagMatrix>
  typename FastDiagMatrix::value_type
  trace(const FastDiagMatrix& A)
  {
    return sum(diag(A));
  }
}
```

Chapter 16

Miscellaneous Operations

16.1 Basic Transformations

16.1.1 givens

```
template <class T, class Real>
void givens(T a, T b, Real& c, T& s, T& r);
```

This function generates a *Givens rotation* which can be used to selectively zero elements in a vector. More specifically, this function computes `c`, `s`, and `r` such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}.$$

Requirements on Types

`T`	is the element type, which can be real or complex.
`Real`	is a real number type.

Complexity

Constant time.

Example

```
const int N = 5;
double dx[] = { 1, 2, 3, 4, 5 };
double dy[] = { 2, 4, 8, 16, 32};
vector<double, dense<external> >::type x(dx, N), y(dy, N);

cout << x << endl << y << endl;

double a = x[N-1];
double b = y[N-1];
double c, s, r;
givens(a, b, c, s, r);

givens_apply(c, s, x, y);

cout << x << endl << y << endl;
```

The output is:

```
[1 2 3 4 5 ]
[2 4 8 16 32 ]
[2.1304 4.2608 8.36723 16.4257 32.3883 ]
[-0.679258 -1.35852 -1.72902 -1.48202 0 ]
```

16.1.2 givens_apply

```
template <class T, class Real, class VectorX, class VectorY>
void givens_apply(Real c, T s, VectorX& x, VectorY& y);
```

This function applies the givens rotation generated by `givens()` and stored in `c` and `s` to the vectors `x` and `y`.

Requirements on Types

`T`	is the element type, which can be real or complex.
`VectorX`	is a type that models Vector. `VectorX::value_type` should be the same type as `T`.
`VectorY`	is a type that models Vector. `VectorY::value_type` should be the same type as `T`.
`Real`	is a real number type.

Preconditions

- `size(x) == size(y)`

Complexity

Linear.

16.1.3 house

```
template <class Vector, class T, class Real>
void house(T alpha, Vector& x, T& tau, Real& beta);
```

The Householder transformation (reflection) is used to zero out (annihilate) components of a vector (typically in rows or columns of a matrix) . This is useful in algorithms like QR factorization. The Householder transformation is described by the Householder matrix H such that Hx yields a vector with all zeroes except $x[0]$.

$$H \begin{bmatrix} \alpha \\ x[1 \ldots n-1] \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix} \quad , \quad HH^T = I$$

The Householder matrix is stored in the Householder vector v and then generated implicitly during application since

$$\begin{aligned} H = I - \tau v v^T \quad &, \quad \tau = \frac{2}{v^T v} \\ HA = (I - \tau v v^T)A = A - v w^T \quad &, \quad w = \tau A^T v \\ AH = A(I - \tau v v^T) = A - w v^T \quad &, \quad w = \tau A v \end{aligned}$$

This routines takes as input the vector x in two peices: `alpha` which is from $x[0]$ and `x` which is $x[1 \ldots n-1]$. The output is `tau`, `beta`, and `x` which has been overwritten by the vector $v[1 \ldots n-1]$. Before applying the transform (with either `house_apply_left()` or `house_apply_right()`) you need to set `x[0]` to `1`. The algorithm used to implement this function is the same as that of LAPACK's `xLARFG` routine.

Requirements on Types

`T`	is the element type, which can be real or complex.
`Vector`	is a type that models Vector.
`Real`	is a real number type.

Complexity

Linear.

Examples

Example of generating a Householder vector and applying it to the original vector, annihilating all the elements but the first.

```
double x_[] = { 3, 1, 5, 1 };
vector<double, dense<external> >::type x(x_,4);
vector<double>::type v(4);

v = x;
house(x[0], v[range(1,4)], tau, beta); // generate Householder transform

v[0] = 1.0;
house_apply_left(v, tau, x);           // x = H * x

cout << x << endl;
```

The output is:

```
[-6 0 0 0]
```

The Householder bidiagonalization, Algorithm 5.4.2 from [6]. The somewhat cumbersome interface for `house` is necessitated by the desire to make this algorithm work in-place. The matrix `A` is overwritten by the bidiagonal result and is also overwritten by the Householder vectors used to get there.

```
template <class Matrix>
void bidiagonalize(Matrix& A)
{
  typedef typename matrix_traits<Matrix>::element_type T;
  typename Matrix::size_type M = A.nrows(), N = A.ncols(), j;
  T tau, alpha;
  typename magnitude<T>::type beta;
  for (j = 0; j < N; ++j) {
```

```
      alpha = A(j,j);
      house( alpha, A(range(j+1,M),j), tau, beta );
      A(j,j) = one(alpha);
      house_apply_left( A(range(j,M),j), tau,
                        A(range(j,M),range(j,N)) );
      A(j,j) = beta;
      if (j <= N - 2) {
        alpha = A(j,j+1);
        house( alpha, A(j,range(j+2,N)), tau, beta );
        A(j,j+1) = one(alpha);
        house_apply_right( trans(A(j,range(j+1,N))), tau,
                           A(range(j,M),range(j+1,N)) );
        A(j,j+1) = beta;
      }
    }
  }
```

16.1.4 house_apply_left

$A \leftarrow HA$

```
template <class Vector, class T, class Matrix>
void house_apply_left(const Vector& v, T tau, Matrix& A);
```

This function applies the Householder transformation stored in `v` (which can be created with the `house()` function) to the Matrix `A`. The implementation of the application consists of a matrix vector multplication and a rank-1 update.

$$HA = (I - \tau v v^T)A = A - v w^T \quad , \quad w = \tau A^T v$$

See the documentation for `house()` for more details and and examples.

Preconditions

- `size(v) == nrows(A)`

Complexity

$O(mn)$

16.1.5 house_apply_right

$A \leftarrow AH$

```
template <class Vector, class T, class Matrix>
void house_apply_right(const Vector& v, T tau, Matrix& A);
```

This function applies the Householder transformation stored in `v` (which can be created with the `house()` function) to the Matrix `A`. The implementation of the application consists of a matrix vector multplication and a rank-1 update.

$$AH = A(I - \tau v v^T) = A - w v^T \quad , \quad w = \tau A v$$

See the documentation for `house()` for more details and and examples.

Complexity

$O(mn)$

Preconditions

- `size(v) == ncols(A)`

Appendix A

Concept Checks for STL

A.1 STL Basic Concept Checks

A.1.1 Assignable

```
template <class T>
struct Assignable {
  void constraints() {
    a = a;              // require assignment operator
    T c(a);             // require copy constructor
    const_constraints(a);
    ignore_unused_variable_warning(c);
  }
  void const_constraints(const T& b) {
    a = b;              // const required for argument to assignment
    T c(b);             // const required for argument to copy constructor
    ignore_unused_variable_warning(c);
  }
  T a;
};
```

A.1.2 DefaultConstructible

```
template <class T>
struct DefaultConstructible {
  void constraints() {
    T a;            // require default constructor
  }
};
```

A.1.3 CopyConstructible

```
template <class T>
struct CopyConstructible {
```

```
  void constraints() {
    T a(b);             // require copy constructor
    T* ptr = &a;        // require address of operator
    const_constraints(a);
  }
  void const_constraints(const T& a) {
    T c(a);             // require const copy constructor
    const T* ptr = &a; // require const address of operator
  }
  T b;
};
```

A.1.4 EqualityComparable

```
template <class T>
struct EqualityComparable {
  void constraints() {
    r = a == b;       // require equality operator
    r = a != b;       // require inequality operator
  }
  T a, b;
  bool r;
};
```

A.1.5 LessThanComparable

```
template <class T>
struct LessThanComparable {
  void constraints() {
    // require comparison operators
    r = a < b || a > b || a <= b || a >= b;
  }
  T a, b;
  bool r;
};
```

A.1.6 Generator

```
template <class Func, class Ret>
struct Generator {
  void constraints() {
    r = f();         // require operator() member function
  }
  Func f;
  Ret r;
};
```

A.1.7 UnaryFunction

```
template <class Func, class Ret, class Arg>
struct UnaryFunction {
  void constraints() {
    r = f(arg);              // require operator()
  }
  Func f;
  Arg arg;
  Ret r;
};
```

A.1.8 BinaryFunction

```
template <class Func, class Ret, class First, class Second>
struct BinaryFunction {
  void constraints() {
    r = f(first, second); // require operator()
  }
  Func f;
  First first;
  Second second;
  Ret r;
};
```

A.1.9 UnaryPredicate

```
template <class Func, class Arg>
struct UnaryPredicate {
  void constraints() {
    r = f(arg);          // require operator() returning bool
  }
  Func f;
  Arg arg;
  bool r;
};
```

A.1.10 BinaryPredicate

```
template <class _Func, class _First, class _Second>
struct BinaryPredicate {
  void constraints() {
    r = f(a, b);        // require operator() returning bool
  }
  Func f;
  First a;
  Second b;
  bool r;
};
```

A.2 STL Iterator Concept Checks

A.2.1 TrivialIterator

```
template <class T>
struct TrivialIterator {
  CLASS_REQUIRES(T, Assignable);
  CLASS_REQUIRES(T, DefaultConstructible);
  CLASS_REQUIRES(T, EqualityComparable);

  void constraints() {
    typedef typename std::iterator_traits<T>::value_type V;
    (void)*i;           // require dereference operator
  }
  T i;
};
```

A.2.2 Mutable-TrivialIterator

```
template <class T>
struct Mutable_TrivialIterator {
  CLASS_REQUIRES(T, TrivialIterator);
  void constraints() {
    *i = *j;            // require dereference and assignment
  }
  T i, j;
};
```

A.2.3 InputIterator

```
template <class T>
struct InputIterator {
  CLASS_REQUIRES(T, TrivialIterator);
  void constraints() {
    // require iterator_traits typedef's
    typedef typename std::iterator_traits<T>::difference_type D;
    typedef typename std::iterator_traits<T>::reference R;
    typedef typename std::iterator_traits<T>::pointer P;
    typedef typename std::iterator_traits<T>::iterator_category C;
    REQUIRE2(typename std::iterator_traits<T>::iterator_category,
             std::input_iterator_tag, Convertible);
    ++i;                // require preincrement operator
    i++;                // require postincrement operator
  }
  T i;
};
```

A.2.4 OutputIterator

```
template <class T>
```

```
struct OutputIterator {
  CLASS_REQUIRES(T, Assignable);
  void constraints() {
    (void)*i;           // require dereference operator
    ++i;                // require preincrement operator
    i++;                // require postincrement operator
    *i++ = *j;          // require postincrement and assignment
  }
  T i, j;
};
```

A.2.5 ForwardIterator

```
template <class T>
struct ForwardIterator {
  CLASS_REQUIRES(T, InputIterator);
  void constraints() {
    REQUIRE2(typename std::iterator_traits<T>::iterator_category,
             std::forward_iterator_tag, Convertible);
  }
};
```

A.2.6 Mutable-ForwardIterator

```
template <class T>
struct Mutable_ForwardIterator {
  CLASS_REQUIRES(T, ForwardIterator);
  CLASS_REQUIRES(T, OutputIterator);
  void constraints() { }
};
```

A.2.7 BidirectionalIterator

```
template <class T>
struct BidirectionalIterator {
  CLASS_REQUIRES(T, ForwardIterator);
  void constraints() {
    REQUIRE2(typename std::iterator_traits<T>::iterator_category,
             std::bidirectional_iterator_tag, Convertible);
    --i;                // require predecrement operator
    i--;                // require postdecrement operator
  }
  T i;
};
```

A.2.8 Mutable-BidirectionalIterator

```
template <class T>
struct Mutable_BidirectionalIterator {
```

```
  CLASS_REQUIRES(T, BidirectionalIterator);
  CLASS_REQUIRES(T, Mutable_ForwardIterator);
  void constraints() {
    *i-- = *i;                  // require postdecrement and assignment
  }
  T i;
};
```

A.2.9 RandomAccessIterator

```
template <class T>
struct RandomAccessIterator {
  CLASS_REQUIRES(T, BidirectionalIterator);
  CLASS_REQUIRES(T, LessThanComparable);

  void constraints() {
    REQUIRE2(typename std::iterator_traits<T>::iterator_category,
      std::random_access_iterator_tag, Convertible);

    typedef typename std::iterator_traits<T>::reference R;

    i += n;             // require assignment addition operator
    i = i + n; i = n + i; // require addition with difference type
    i -= n;             // require assignment subtraction operator
    i = i - n;                  // require subtraction with difference type
    n = i - j;                  // require difference operator
    (void)i[n];                 // require element access operator
  }
  T a, b;
  T i, j;
  typename std::iterator_traits<T>::difference_type n;
};
```

A.2.10 Mutable-RandomAccessIterator

```
template <class T>
struct Mutable_RandomAccessIterator {
  CLASS_REQUIRES(T, RandomAccessIterator);
  CLASS_REQUIRES(T, Mutable_BidirectionalIterator);
  void constraints() {
    i[n] = *i;                  // require element access and assignment
  }
  T i;
  typename std::iterator_traits<T>::difference_type n;
};
```

Bibliography

[1] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[2] P. N. Brown and A. C. Hindmarsh. Matrix-free methods for stiff systems of ODE's. *SIAM J. Numer. Anal.*, 23(3):610–638, June 1986.

[3] K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[4] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In D. C. Sorensen, J. Dongarra, P. Messina, and R. G. Voigt, editors, *Proceedings of the 4th Conference on Parallel Processing for Scientific Computing*, pages 400–405, Philadelphia, PA, USA, Dec. 1989. SIAM Publishers.

[5] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95)*, pages 239–245, Los Alamitos, CA, USA, Apr. 1995. IEEE Computer Society Press.

[6] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, Maryland, 1983.

[7] P. R. Halmos. *Finite-Dimensional Vector Spaces*. The University Series in Undergraduate Mathematics. van Nostrand Company, 2 edition, 1958.

[8] A. N. Michel and C. J. Herget. *Applied Algebra and Functional Analysis*. Dover, 1981.

[9] Y. Saad and M. Schultz. GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7(3):856–869, July 1986.

[10] G. Strang. *Linear Algebra and Its Applications*. Harcourt, Brace, Jovanovich, San Diego, third edition, 1988.

[11] B. L. van der Waerden. *Algebra*. Frederick Ungar Publishing, 1970.