# The Design of a Templated C++ Small Vector Class for Numerical Computing

Patrick J. Moran*

Advanced Management Technology, Incorporated
NASA Ames Research Center, M/S T27A-2
Moffett Field, CA, 94035, USA

NAS Technical Report NAS-00-013

June 20, 2000

**Abstract**

We describe the design and implementation of a templated C++ class for vectors. The vector class is templated both for vector length and vector component type; the vector length is fixed at template instantiation time. The vector implementation is such that for a vector of `N` components of type `T`, the total number of bytes required by the vector is equal to `N * sizeof(T)`, where `sizeof` is the built-in C operator. The property of having a size no bigger than that required by the components themselves is key in many numerical computing applications, where one may allocate very large arrays of small, fixed-length vectors.

In addition to the design trade-offs motivating our fixed-length vector design choice, we review some of the C++ template features essential to an efficient, succinct implementation. In particular, we highlight some of the standard C++ features, such as partial template specialization, that are not supported by all compilers currently. This report provides an inventory listing the relevant support currently provided by some key compilers, as well as test code one can use to verify compiler capabilities.

## 1 Introduction

Vectors are a commonly occurring type of object in many software efforts. Their prevalence and utility led the designers of the Standard Template Library (STL) [MS96] to make vectors one of the basic classes in their library. This library in turn was adopted by the C++ standards committee, thus the STL vector class is now a standard part of C++ [ISO98]. STL vectors are quite general and flexible. They can be dynamically

*pmoran@nas.nasa.gov

grown and shrunk, and adapted to implement other collection types such as stacks and lists. The fact that STL vector instances can dynamically grow to arbitrary size implies that their implementation contains some indirection, typically a pointer to a buffer that is dynamically allocated to the appropriate size. This in turn implies that the total amount of memory required by an STL vector is greater than the memory required to store the vector elements. At minimum there is the extra memory for the pointer to the data buffer. Typically, there is also the memory required for internal bookkeeping values, such as an integer storing the currently allocated vector length. When using a small number of vector objects, this overhead is typically unimportant. When allocating large arrays of small vectors, the overhead can be significant. Scenarios in numerical computing where one allocates arrays consisting of millions of small, fixed-length vectors occur frequently, thus one obvious optimization is to develop a vector class that trades off dynamically changeable length for a smaller memory footprint.

Even with a fixed-length vector class, there are design trade-offs to be made in order to keep the memory requirement of each instance equal to that of the components alone. In particular, we choose to not make the vector classes polymorphic. In C++, polymorphism is supported via `virtual` functions. Each instance of a class with any virtual functions contains a pointer to a virtual function table, thus the size of each instance includes the size of a pointer. Note that the virtual functions could be defined in our vector class, or they could be inherited if our class inherits from another class. One scenario where this could happen occurs if all objects are required to inherit from a common base class. The base class typically declares interface common to all objects, for instance, serialization routines for network communication and persistence. Having such common functionality for all objects is handy, but it comes at a memory cost that we cannot afford for our vector design.

The template vector class we describe here is not the first templated small vector design to address numerical computing needs. The second edition of the Field Encapsulation Library (FEL) [MHE00, MH00] contained vector classes that were used extensively throughout the library. FEL vectors were templated by component type, but not by length. One could not compose matrices as vectors of vectors (see Section 2.4 below). The Blitz++ [Bli] library is another effort towards high-performance numeric computing using C++ and templates. Blitz++ contains a class called `TinyVector` that is similar to the class we describe in that it is templated by component type and length. The interface of `TinyVector` contains some features our vector class does not have, such as iterator definitions, but lacks some of the basic features that we will need, such as definitions for `operator+`.

The next section presents the criteria we used in making our vector class design choices, followed by a discussion covering the salient features of our implementation. Because the template standards for C++ have been finalized relatively recently, and because support is nontrivial to implement, one still cannot assume that every compiler will handle templated code correctly. Following our implementation discussion we provide an inventory of the template features our design requires, along with the compliance of some key compilers. We also provide some example numbers showing some of the performance implications of our design.

# 2 Design Criteria

There are four primary criteria guiding our vector design, which we present not necessarily in priority order.

## 2.1 Memory Layout

In the previous section we described the need for a vector class where the memory requirement of each instance is equal to that of the vector components alone. In general, we are interested not only in the amount of memory required, but also in how the data components are layed out in memory. This interest comes from the desire to be able to share data arrays with applications that may not be written using our vector class, or perhaps not written in C++ at all. One of the leading candidates for sharing would be numerical field solvers. Such applications typically operate with very large numeric arrays. In the cases where an application has the components for each vector instance contiguous in memory, we can share the buffer and treat it as an array of our vector instances. By sharing we can avoid the memory and time consumed by a copy and reorganize step, a step that can be quite costly and possibly not even feasible when working with problems close to the capacity of a system.

## 2.2 Templated Vector Length

A second criterion for our vector class is that it be templated by length. The motivation for this requirement is the desire to develop objects that are templated by the dimensionality of their base space and the space they are embedded in. For example, we anticipate developing classes for regular meshes embedded in $d$-dimensional space. Such objects naturally have a need for vectors that are also $d$-dimensional, both for internal implementation and for the class interface.

## 2.3 Performance

While we want to have the generality of $d$-dimensional vectors with a variety of component types, we do not want to achieve this at the expense of performance. Furthermore, we hope to maintain a level of performance such that library users will feel comfortable using our vector objects rather than hand coding their own vector math.

## 2.4 Matrices Via Composition

The fourth criterion is probably the least obvious of the four. We want to support constructing matrices as vectors of vectors. For example, a 3 by 4 matrix of `float`s could be declared as `FM_vector<3,FM_vector<4,float> >`. The syntax for such declarations is admittedly a bit awkward — clearly our motivation is not syntax. We can hide most of the syntax unattractiveness via `typedef` statements. The motivation for the design comes from anticipated uses with differential-operator fields. Such fields require that we compute various partial terms expressing change in field value with respect to coordinate axes. Given a scalar field, a differential-operator field may

return the partials in a vector, for example in response to a gradient query. Imagine that the field type is templated, so that the same implementation could be used for any scalar type. Now imagine if we were to try to use the same differential-operator mechanism with a vector field. The result of a gradient query would be a vector of vectors containing all the terms needed for the second order tensor, albeit transposed from the usual layout. Thus, the same templated code could be useful for both scalar and vector fields. But this reuse breaks down at compile time if we cannot instantiate vectors of vectors. In general, we have found composition to be a powerful technique for defining new objects in terms of other objects (as for example in FEL2 [MHE00, MH00]). We anticipate that support for vector composition could lead to some interesting capabilities here as well.

As we will see in the following sections, supporting vector of vectors composition does have some sometimes subtle implications. In short, we must be careful to account for cases where the vector component type `T` is not a scalar when we define member and friend functions for our vector class. We address the specific issues as they arise in the following sections.

# 3 Implementation

## 3.1 The Generic Vector Class

We call our vector class `FM_vector`[1], where `FM` stands for "Field Model". Eventually the `FM_vector` class may be used in support of a larger field model library. The class provides the basic math operators that one would expect of vectors, though the set of operators provided in the current implementation is by no means exhaustive.

The beginning of the `FM_vector` class looks like:

```
template <int N, typename T>
class FM_vector
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    for (int i = 0; i < N; i++)
      d[i] = dat[i];
  }
```

[1] A classic problem when writing a library that will be used in application development is avoiding namespace collision problems. A *namespace collision* occurs when multiple libraries use the same identifier name, for example, `vector`. The traditional solution to this problem is for each library to add some standard prefix to all the names it introduces, for example, prefixing `vector` with `FM_` to get `FM_vector`. Recent C++ offers a second option: the `namespace` mechanism ([ISO98] §7.3). We have chosen the traditional solution here because the name we want to use, `vector`, is likely to already occur in the environment (via the standard library). Typing `FM_vector` is just as convenient as typing `FM::vector`, and is always unambiguous.

```
template <typename S>
explicit FM_vector(const FM_vector<N,S>& dat)
{
  for (int i = 0; i < N; i++)
    d[i] = (T) dat[i];
}
```

Template parameter `N` specifies the vector length, `T` specifies the component type. The class has three constructors: a default constructor (i.e., without any arguments), a constructor taking an array of initialization values, and a constructor taking a same-length `FM_vector` of type `S` components as an argument for initialization. Note that we use the keyword `explicit` for the third constructor. Without this keyword, the compiler may implicitly use the third constructor to convert a vector of one type of component to a vector of another. Sometimes this is just what we want: for example, to promote a vector of `float`s to a vector of `double`s. Unfortunately, at other times such conversions many lead to ambiguity problems for the compiler. Given a particular expression, the compiler may determine that there is more than one combination of conversions and functions that it can use to match. Compilers typically treat this as an error. By using the `explicit` keyword, we support type conversions, but only if the user explicitly casts from one type to another. See also for example Meyers ([Mey96] Item 5) for a discussion of why one should be wary of implicit conversion functions.

A typical example of a mathematical operator, `operator-`, looks like:

```
friend FM_vector<N,T>
operator-(const FM_vector<N,T>& lhs,
          const FM_vector<N,T>& rhs)
{
  T tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] - rhs[i];
  return FM_vector<N,T>(tmp);
}
```

The basic design pattern for `operator-` is followed by other similar operators as well. Each function contains a temporary array that gets filled in with result values, and that array is used to initialize the object constructed in the `return` statement. An alternative style would be to declare an `FM_vector<N,T>` instance at the beginning of the routine, fill in each element, then return it, with no constructor in the final statement. Note that the function is still meaningful even if the component type `T` is a vector type, i.e., the same template defines subtraction of like-sized matrices.

In designing our vector class, we have to be careful to choose between providing function declarations and function definitions inside the class definition. A *function declaration* specifies a function signature, in other words, a function name, argument types and return type. A *function definition* is like a declaration, except that the body of the function, defining its implementation, is also specified. The distinction is important because definitions and declarations are treated differently when it comes to template instantiation. Functions defined within a template class definition will be instantiated

with each type that the class is instantiated with, even if they are never used ([ISO98] §14.5.3 **temp.friend**, item 5). This policy impacts us when we consider cases where we will instantiate vectors of vectors. For example, we would run into problems if we provided `FM_dot` as a friend function definition within our generic vector class. When the vector class is instantiated with a vector component type, the compiler in turn would attempt to instantiate each friend function defined in the class. Even though we never attempt to dot one matrix with another, the compiler would attempt to do the instantiation. This instantiation would fail when the compiler attempts to instantiate `operator*` with corresponding vector components. We do not define `operator*` with two vector of scalars arguments (see the following section).

Within our vector class we define a relatively minimal set of `operator`s, a set constrained by what still makes sense when we compose vectors: `[]`, `==`, (unary) `-`, (binary) `-`, `+`, and `*`. Outside of the class we also provide `operator`s `!=` and `<<`. There are other operators that make sense, even with the "must work with vector of vectors" constraint. We have found the current set of functions sufficient for now. We also provide `friend` declarations for `FM_dot` and `FM_cross`, with the function definitions following outside the class.

## 3.2 Type Traits and `operator*`

Unlike `operator`s `+` and `-`, `operator*` has natural semantics for cases where the left-hand and right-hand arguments are not of the same dimension. For the present we will limit ourselves to defining scalar times an `FM_vector` object, and the commutative pair. As in the previous section, we have to be careful to consider the case where the component type is itself a vector. We want the scalar type to match the scalar type within the `FM_vector` argument, so we cannot simply hard-code the scalar argument to be a specific type, e.g., `double`. Simply using the template component type `T` works if we have a vector of scalars, but breaks down if we have a vector of vectors. The solution is a technique known as *type traits* [Mye95]. The idea is to use the C++ templated class and specialization mechanism to provide specific information about the instantiation types that we use elsewhere. In our case we are interested in determining the scalar element type of the template argument `T` used in an `FM_vector` instantiation. If our class is instantiated with `T` as a scalar, then the element type would be the same type. If `T` corresponds to a `FM_vector` of some type, then we want the trait mechanism to recursively obtain the element type of the vector that `T` corresponds to. We can express this in C++ as:

```
template <typename T>
struct FM_traits
{
  typedef T element_type;
};
```

```
template <int N, typename T>
struct FM_traits<FM_vector<N,T> >
{
  typedef typename FM_traits<T>::element_type element_type;
};
```

We will revisit the use of template specializations in the next section. For `operator*`, usage of the traits mechanism looks like:

```
friend FM_vector<N,T>
operator*(typename FM_traits<T>::element_type lhs,
          const FM_vector<N,T>& rhs)
{
  T tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs * rhs[i];
  return FM_vector<N,T>(tmp);
}
```

The syntax for using traits is a bit verbose, and perhaps not particularly easy to use for those not familiar with templates. Fortunately, it is primarily just the `FM_vector` library developers who have to worry about getting the syntax right. One case where the user would have provide his or her own traits specialization, like that for `FM_vector` above, would be if he or she instantiated an `FM_vector` with some other non-scalar type.

## 3.3 Specializations

So far we have presented an overview of the generic `FM_vector` class implementation. The class provides basic vector functionality for vectors of essentially arbitrary length and component type. Unfortunately, we sacrifice a bit in performance in order to get such generality. For example, in the definition for `operator-` presented previously in Section 3.1, note that we require building up the result in a temporary array `tmp` before constructing the final result in the `return` statement. We would like to avoid use of the temporary when possible. We would also like to avoid the looping construct, since the overhead could be significant when the vector length `N` is very small. In effect we would like to unroll the loop. Fortunately, C++ provides a mechanism that enables us to provide these optimizations: specializations ([ISO98] §14.7, **temp.spec**). *Specializations* are template definitions where one or more template parameters are fixed, compared to the initial, most general template definition. If some, but not all the parameters are fixed, then we have a *partial specialization* ([ISO98] §14.5.4, **temp.class.spec**). Specializations can be provided for class templates or function templates, in our library we use both. For our `FM_vector` class, the most obvious opportunity for optimization is to provide specializations based on particular vector lengths. Using such specializations, we can unroll the loops. We can also provide operator definitions that use patterns more likely to be optimized by the compiler.

The key to our specialization optimizations is providing an additional constructor, one that takes the vector components as arguments. Within operator definitions, we can utilize this constructor to avoid the temporary array that we had to use previously. For example, within a partial specialization for length-3 vectors, we could provide a more optimized version of `operator-`:

```
friend FM_vector<3,T>
operator-(const FM_vector<3,T>& lhs,
          const FM_vector<3,T>& rhs)
{
  return FM_vector<3,T>(lhs.d[0] - rhs.d[0],
                        lhs.d[1] - rhs.d[1],
                        lhs.d[2] - rhs.d[2]);
}
```

In addition to eliminating the temporary array and loop, we have a pattern that facilitates what is known as a *named return value optimization* (see for example [Mey96], Item 20). This is a pattern that many compilers can exploit to avoid creating temporary objects, at least when one uses the higher optimization levels.

Note in the previous example that a partial specialization is precisely what we needed. The optimizations do not depend on the type `T`, they would apply regardless of whether `T` corresponded to `int`, `float`, or even `FM_vector<3,double>`. When providing specializations, there is in general a trade-off between the added performance one can achieve with type specific code on one hand, and the code maintenance implications on the other. Providing many specializations implies having a lot of nearly-the-same code to manage—which often implies headaches for library maintainers in the long term. Partial specializations help reduce this problem somewhat, since we can get by with fewer definitions, compared to using full specializations everywhere. Partial specializations, compared to total specializations, also free the author from having to completely anticipate every instantiation type that the user will care about. Unfortunately, not every compiler supports partial specializations. We consider several key compilers and their current compliance with the C++ template standard in the following section. Our implementation currently provides partial specializations for vectors of length 1, 2, 3, and 4.

# 4 Compiler Support

Table 1 lists the current state of affairs when it comes to features required by our templated vector design. The three compilers families that we are particularly interested in are Microsoft Visual C++, gnu g++, and SGI MIPSpro. The short summary, as one can see from the table, is that the three compilers all support the features we need, except that the Visual C++ compiler does not handle partial specializations. In general, proper template handling requires that the compiler have a relatively sophisticated pattern matching mechanism for deciding which implementation to use when it encounters the need for a particular instantiation. Our experience with the Visual C++ compiler has been that their pattern matching mechanism is not particularly robust. The Visual

| Feature | VC++ 6.0 | g++ 2.95.2 | MIPSpro 7.3.1.1m |
|---|---|---|---|
| Support for `explicit` keyword | *OK* | *OK* | *OK* |
| Templated member functions | *OK* | *OK* | *OK* |
| Specialization of templated classes | *OK* | *OK* | *OK* |
| Partial specialization of templated classes | | *OK* | *OK* |
| Specialization of templated functions | *OK* | *OK* | *OK* |
| Partial specialization of templated functions | | *OK* | *OK* |

Table 1: Current support for some key C++ template language features in the Microsoft Visual C++ 6.0, gnu g++ 2.95.2 and SGI MIPSpro 7.3.1.1m compilers.

C++ compiler does handle simple matching cases correctly; perhaps the compiler developers have not encountered enough pressure yet from their user base to make the correct handling of more sophisticated templates a priority.

As for the gnu and MIPSpro compilers, both were capable of handling our vector library code. With any compiler, one should be sure to get the current version, as compliance with the C++ template standard is a recent occurrence at best. One specific note for the g++ compiler: the g++ shipped with Redhat 6.2 Linux is not especially current. In particular, the shipped compiler does not support all the features listed in Table 1. Users of Redhat 6.2 should be prepared to download and install a current g++ in order to successfully compile `FM_vector`.

# 5 Performance and Optimization

Table 2 lists example timings for the evaluation of a routine typically used in point location codes. The routine defines a predicate indicating whether point `d` is above, coplanar with, or below the plane defined by points `a`, `b` and `c` (as exactly as we can calculate using standard floating-point). The rows correspond to increasing levels of compile-time optimization. The first two `float` and `double` columns list performance obtained using the generic `FM_vector` class. The `float` and `double` headings indicate whether the vectors were composed of single-precision or double-precision components, respectively. The second pair of `float` and `double` columns repeats the timings, but with the added availability of the class partial specialization for length-3 vectors. The third pair of columns lists timings when total specializations for vectors of 3 `float`s and 3 `double`s were available.

Clearly, timings for one particular expression do not provide a thorough analysis of the performance of the vector library. Nevertheless, we include the table to make two points. First, note the range in performance obtained by varying the -O optimization. The `FM_vector` class is carefully written to use patterns that we expect to optimize well, but we do not get the benefits at default (-O0) optimization. Authors of high-performance numerical codes should already be familiar with the importance of using compiler optimizations; the table shows that optimizations are especially important when using classes such as `FM_vector` to represent low-level objects.

The second point of Table 2 is to provide a confirmation that implementing vector

| | Generic | | Partial Specialization | | Full Specialization | |
|---|---|---|---|---|---|---|
| -O$n$ | `float` | `double` | `float` | `double` | `float` | `double` |
| 0 | 2.090 | 2.000 | 0.654 | 0.610 | 0.648 | 0.600 |
| 1 | 0.362 | 0.337 | 0.238 | 0.218 | 0.175 | 0.149 |
| 2 | 0.354 | 0.324 | 0.237 | 0.216 | 0.179 | 0.145 |
| 3 | 0.360 | 0.341 | 0.172 | 0.145 | 0.172 | 0.145 |

Table 2: Example timings (usec/call) for a routine used in point location consisting of the following statement: `return sign(FM_dot(d - a, FM_cross(b - a, c - a)))`. The timings were computed on a dual 500-MHz Xeon processor workstation, using the g++ 2.95.2 compiler. All vectors had length `N` = 3.

class specializations does indeed significantly improve performance. Comparing the columns under the **Generic** heading to those under the **Partial Specialization** and **Full Specialization** headings, we see that the specializations can cut the execution time roughly in half, compared to the generic implementation. In Section 3.3, we introduced the idea of providing class specializations for our vector implementation. Here we confirm that there is a worthwhile performance pay-off due to specializations.

Table 2 also provides a comparison between partial and full specialization. Both the partial and full specializations essentially rely upon the same optimizations, based on exploiting specific vector lengths. From the table we can see that the compiler has an easier time recognizing opportunities for optimization when we provide full specializations, but at a high enough optimization we get the same performance with partial specializations alone. Of course, these timings are for one compiler and one expression, so they are clearly not the final word on the subject. Our overall philosophy is to make a concerted effort to maintain high performance over a wide range of vector instantiation types. To that end we favor optimizations that apply to ordinary (unspecialized) templates over partially specialized templates, and partial specializations over full specializations. We see two problems with full specializations. First, supplying full specializations puts the library authors in the position of having to anticipate which instantiation types are important to users. Some may be fairly obvious, but if the library is utilized by a variety of users, then this job can quickly become difficult. Second, liberally adding full specializations can lead to a code maintenance nightmare. Part of the original appeal of templates is that they enable the compiler to generate customized code at compile/link time, working from a relatively small set of templated classes. Adding many full specializations nullifies the software engineering advantages of such an approach and leads to much more nearly-the-same code, with the problems inherent in such proliferation inevitably following.

# 6 Conclusion

We have presented the design and implementation of a templated C++ class for small vectors. The design balances the need for extensibility and flexibility on one hand with the desire for high performance on the other. We hope that this report may enlighten

readers to some of the more interesting possibilities inherent to the C++ template mechanism. In the appendix that follows, we provide the actual source to our `FM_vector` class and a few supporting classes. We encourage the reader to utilize the code to try some experiments on his or her own. C++ compilers are only now starting to fully support the template standard. The classes presented in this report are only the beginning of what is possible, given these new capabilities. The future should be interesting.

# References


[Bli] Blitz++. `http://oonumerics.org/blitz/`.

[ISO98] ISO/IEC. *C++ International Standard*, September 1998. 14882:1998(E).

[Mey96] S. Meyers. *More Effective C++*. Addison-Wesley Publishing Company, Menlo Park, California, 1996.

[MH00] P. Moran and C. Henze. The FEL 2.2 reference manual. Technical report, National Aeronautics and Space Administration, 2000. NAS-00-007.

[MHE00] P. Moran, C. Henze, and D. Ellsworth. The FEL 2.2 user guide. Technical report, National Aeronautics and Space Administration, 2000. NAS-00-002.

[MS96] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley Publishing Company, Menlo Park, California, 1996.

[Mye95] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995. `http://www.cantrip.org/traits.html`.


# Appendix A

This appendix lists the C++ source code provided by the following files:

| | |
|---|---|
| `FM_vector.h` | The header file providing our vector definitions. |
| `FM_matrix.h` | The header file providing our matrix definitions. |
| `FM_timer.h` | A simple timer class used to produce the values in Table 2. |
| `vector_tests.C` | An example program that exercises `FM_vector` features. |

```
// Emacs mode -*-c++-*- //
#ifndef _FM_VECTOR_H_
#define _FM_VECTOR_H_
/*
 * NAME: FM_vector.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <iostream>
#include <math.h>

#ifndef FM_COORD
#define FM_COORD
typedef float FM_coord;
#endif


template <int N, typename T> class FM_vector;


template <typename T>
struct FM_traits
{
  typedef T element_type;
};

template <int N, typename T>
struct FM_traits<FM_vector<N,T> >
{
  typedef typename FM_traits<T>::element_type element_type;
};

template <int N, typename T>
T FM_dot(const FM_vector<N,T>&, const FM_vector<N,T>&);

template <typename T>
FM_vector<3,T> FM_cross(const FM_vector<3,T>&, const FM_vector<3,T>&);


template <int N, typename T = FM_coord>
class FM_vector
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    for (int i = 0; i < N; i++)
      d[i] = dat[i];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<N,S>& dat)
  {
    for (int i = 0; i < N; i++)
      d[i] = (T) dat[i];
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  operator const T*() const
  {
    return (typename FM_traits<T>::element_type*) d;
  }

  friend bool operator==(const FM_vector<N,T>& lhs,
                         const FM_vector<N,T>& rhs)
  {
```

```
  bool res = true;
  for (int i = 0; i < N; i++) {
    if (!(lhs[i] == rhs[i])) {
      res = false;
      break;
    }
  }
  return res;
}

FM_vector<N,T>& operator+=(const FM_vector<N,T>& v) {
  for (int i = 0; i < N; i++)
    d[i] += v[i];
  return *this;
}

FM_vector<N,T>& operator-=(const FM_vector<N,T>& v) {
  for (int i = 0; i < N; i++)
    d[i] -= v[i];
  return *this;
}

FM_vector<N,T>& operator*=(typename FM_traits<T>::element_type s) {
  for (int i = 0; i < N; i++)
    d[i] *= s;
  return *this;
}

FM_vector<N,T>& operator/=(typename FM_traits<T>::element_type s) {
  for (int i = 0; i < N; i++)
    d[i] /= s;
  return *this;
}

friend FM_vector<N,T> operator-(const FM_vector<N,T>& u)
{
  T tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = -u[i];
  return FM_vector<N,T>(tmp);
}

friend FM_vector<N,T> operator+(const FM_vector<N,T>& lhs,
                                const FM_vector<N,T>& rhs)
{
  T tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] + rhs[i];
  return FM_vector<N,T>(tmp);
}
friend FM_vector<N,T> operator-(const FM_vector<N,T>& lhs,
                                const FM_vector<N,T>& rhs)
{
  T tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] - rhs[i];
  return FM_vector<N,T>(tmp);
}

friend FM_vector<N,T>
operator*(typename FM_traits<T>::element_type lhs,
          const FM_vector<N,T>& rhs)
{
  T tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs * rhs[i];
  return FM_vector<N,T>(tmp);
}
```

```
  friend FM_vector<N,T>
  operator*(const FM_vector<N,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    T tmp[N];
    for (int i = 0; i < N; i++)
      tmp[i] = lhs[i] * rhs;
    return FM_vector<N,T>(tmp);
  }

private:
  T d[N];
};


template <typename T>
class FM_vector<1,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<1,S>& dat)
  {
    d[0] = (T) dat[0];
  }
  FM_vector(const T& a0)
  {
    d[0] = a0;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  operator const T*() const
  {
    return (typename FM_traits<T>::element_type*) d;
  }

  friend bool operator==(const FM_vector<1,T>& lhs,
                         const FM_vector<1,T>& rhs)
  {
    return
      lhs.d[0] == rhs.d[0];
  }

  FM_vector<1,T>& operator+=(const FM_vector<1,T>& v) {
    d[0] += v[0];
    return *this;
  }

  FM_vector<1,T>& operator-=(const FM_vector<1,T>& v) {
    d[0] -= v[0];
    return *this;
  }

  FM_vector<1,T>& operator*=(typename FM_traits<T>::element_type s) {
    d[0] *= s;
    return *this;
  }

  FM_vector<1,T>& operator/=(typename FM_traits<T>::element_type s) {
    d[0] /= s;
    return *this;
```

```
  }

  friend FM_vector<1,T> operator-(const FM_vector<1,T>& u)
  {
    return FM_vector<1,T>(-u.d[0]);
  }

  friend FM_vector<1,T> operator+(const FM_vector<1,T>& lhs,
                                  const FM_vector<1,T>& rhs)
  {
    return FM_vector<1,T>(lhs.d[0] + rhs.d[0]);
  }
  friend FM_vector<1,T> operator-(const FM_vector<1,T>& lhs,
                                  const FM_vector<1,T>& rhs)
  {
    return FM_vector<1,T>(lhs.d[0] - rhs.d[0]);
  }

  friend FM_vector<1,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<1,T>& rhs)
  {
    return FM_vector<1,T>(lhs * rhs.d[0]);
  }
  friend FM_vector<1,T>
  operator*(const FM_vector<1,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<1,T>(lhs.d[0] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<1,T>&,
                     const FM_vector<1,T>&);

private:
  T d[1];
};


template <typename T>
class FM_vector<2,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
    d[1] = dat[1];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<2,S>& dat)
  {
    d[0] = (T) dat[0];
    d[1] = (T) dat[1];
  }
  FM_vector(const T& a0, const T& a1)
  {
    d[0] = a0;
    d[1] = a1;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  operator const T*() const
  {
    return (typename FM_traits<T>::element_type*) d;
  }
```

```
friend bool operator==(const FM_vector<2,T>& lhs,
                       const FM_vector<2,T>& rhs)
{
  return
    lhs.d[0] == rhs.d[0] &&
    lhs.d[1] == rhs.d[1];
}

FM_vector<2,T>& operator+=(const FM_vector<2,T>& v) {
  d[0] += v[0];
  d[1] += v[1];
  return *this;
}

FM_vector<2,T>& operator-=(const FM_vector<2,T>& v) {
  d[0] -= v[0];
  d[1] -= v[1];
  return *this;
}

FM_vector<2,T>& operator*=(typename FM_traits<T>::element_type s) {
  d[0] *= s;
  d[1] *= s;
  return *this;
}

FM_vector<2,T>& operator/=(typename FM_traits<T>::element_type s) {
  d[0] /= s;
  d[1] /= s;
  return *this;
}

friend FM_vector<2,T> operator-(const FM_vector<2,T>& u)
{
  return FM_vector<2,T>(-u.d[0], -u.d[1]);
}

friend FM_vector<2,T> operator+(const FM_vector<2,T>& lhs,
                                const FM_vector<2,T>& rhs)
{
  return FM_vector<2,T>(lhs.d[0] + rhs.d[0],
                        lhs.d[1] + rhs.d[1]);
}
friend FM_vector<2,T> operator-(const FM_vector<2,T>& lhs,
                                const FM_vector<2,T>& rhs)
{
  return FM_vector<2,T>(lhs.d[0] - rhs.d[0],
                        lhs.d[1] - rhs.d[1]);
}

friend FM_vector<2,T>
operator*(typename FM_traits<T>::element_type lhs,
          const FM_vector<2,T>& rhs)
{
  return FM_vector<2,T>(lhs * rhs.d[0],
                        lhs * rhs.d[1]);
}
friend FM_vector<2,T>
operator*(const FM_vector<2,T>& lhs,
          typename FM_traits<T>::element_type rhs)
{
  return FM_vector<2,T>(lhs.d[0] * rhs,
                        lhs.d[1] * rhs);
}

friend T FM_dot<T>(const FM_vector<2,T>&,
                   const FM_vector<2,T>&);
```

```
private:
  T d[2];
};


template <typename T>
class FM_vector<3,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
    d[1] = dat[1];
    d[2] = dat[2];
  }
  template <typename S>
  explicit FM_vector(const FM_vector<3,S>& dat)
  {
    d[0] = (T) dat[0];
    d[1] = (T) dat[1];
    d[2] = (T) dat[2];
  }
  FM_vector(const T& a0, const T& a1, const T& a2)
  {
    d[0] = a0;
    d[1] = a1;
    d[2] = a2;
  }

  T& operator[](int i) { return d[i]; }
  const T& operator[](int i) const { return d[i]; }

  operator const T*() const
  {
    return (typename FM_traits<T>::element_type*) d;
  }

  friend bool operator==(const FM_vector<3,T>& lhs,
                         const FM_vector<3,T>& rhs)
  {
    return
      lhs.d[0] == rhs.d[0] &&
      lhs.d[1] == rhs.d[1] &&
      lhs.d[2] == rhs.d[2];
  }

  FM_vector<3,T>& operator+=(const FM_vector<3,T>& v) {
    d[0] += v[0];
    d[1] += v[1];
    d[2] += v[2];
    return *this;
  }

  FM_vector<3,T>& operator-=(const FM_vector<3,T>& v) {
    d[0] -= v[0];
    d[1] -= v[1];
    d[2] -= v[2];
    return *this;
  }

  FM_vector<3,T>& operator*=(typename FM_traits<T>::element_type s) {
    d[0] *= s;
    d[1] *= s;
    d[2] *= s;
    return *this;
  }
```

```
  FM_vector<3,T>& operator/=(typename FM_traits<T>::element_type s) {
    d[0] /= s;
    d[1] /= s;
    d[2] /= s;
    return *this;
  }

  friend FM_vector<3,T> operator-(const FM_vector<3,T>& u)
  {
    return FM_vector<3,T>(-u.d[0], -u.d[1], -u.d[2]);
  }

  friend FM_vector<3,T> operator+(const FM_vector<3,T>& lhs,
                                  const FM_vector<3,T>& rhs)
  {
    return FM_vector<3,T>(lhs.d[0] + rhs.d[0],
                          lhs.d[1] + rhs.d[1],
                          lhs.d[2] + rhs.d[2]);
  }
  friend FM_vector<3,T> operator-(const FM_vector<3,T>& lhs,
                                  const FM_vector<3,T>& rhs)
  {
    return FM_vector<3,T>(lhs.d[0] - rhs.d[0],
                          lhs.d[1] - rhs.d[1],
                          lhs.d[2] - rhs.d[2]);
  }

  friend FM_vector<3,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<3,T>& rhs)
  {
    return FM_vector<3,T>(lhs * rhs.d[0],
                          lhs * rhs.d[1],
                          lhs * rhs.d[2]);
  }
  friend FM_vector<3,T>
  operator*(const FM_vector<3,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<3,T>(lhs.d[0] * rhs,
                          lhs.d[1] * rhs,
                          lhs.d[2] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<3,T>&,
                     const FM_vector<3,T>&);
  friend FM_vector<3,T> FM_cross<T>(const FM_vector<3,T>&,
                                    const FM_vector<3,T>&);

private:
  T d[3];
};


template <typename T>
class FM_vector<4,T>
{
public:
  FM_vector() {}
  FM_vector(const T dat[])
  {
    d[0] = dat[0];
    d[1] = dat[1];
    d[2] = dat[2];
    d[3] = dat[3];
  }
  template <typename S>
```

```
explicit FM_vector(const FM_vector<3,S>& dat)
{
  d[0] = (T) dat[0];
  d[1] = (T) dat[1];
  d[2] = (T) dat[2];
  d[3] = (T) dat[3];
}
FM_vector(const T& a0, const T& a1, const T& a2, const T& a3)
{
  d[0] = a0;
  d[1] = a1;
  d[2] = a2;
  d[3] = a3;
}

T& operator[](int i) { return d[i]; }
const T& operator[](int i) const { return d[i]; }

operator const T*() const
{
  return (typename FM_traits<T>::element_type*) d;
}

friend bool operator==(const FM_vector<4,T>& lhs,
                       const FM_vector<4,T>& rhs)
{
  return
    lhs.d[0] == rhs.d[0] &&
    lhs.d[1] == rhs.d[1] &&
    lhs.d[2] == rhs.d[2] &&
    lhs.d[3] == rhs.d[3];
}

FM_vector<4,T>& operator+=(const FM_vector<4,T>& v) {
  d[0] += v[0];
  d[1] += v[1];
  d[2] += v[2];
  d[3] += v[3];
  return *this;
}

FM_vector<4,T>& operator-=(const FM_vector<4,T>& v) {
  d[0] -= v[0];
  d[1] -= v[1];
  d[2] -= v[2];
  d[3] -= v[3];
  return *this;
}

FM_vector<4,T>& operator*=(typename FM_traits<T>::element_type s) {
  d[0] *= s;
  d[1] *= s;
  d[2] *= s;
  d[3] *= s;
  return *this;
}

FM_vector<4,T>& operator/=(typename FM_traits<T>::element_type s) {
  d[0] /= s;
  d[1] /= s;
  d[2] /= s;
  d[3] /= s;
  return *this;
}

friend FM_vector<4,T> operator-(const FM_vector<4,T>& u)
{
  return FM_vector<4,T>(-u.d[0], -u.d[1], -u.d[2], -u.d[3]);
```

```
  }

  friend FM_vector<4,T> operator+(const FM_vector<4,T>& lhs,
                                  const FM_vector<4,T>& rhs)
  {
    return FM_vector<4,T>(lhs.d[0] + rhs.d[0],
                          lhs.d[1] + rhs.d[1],
                          lhs.d[2] + rhs.d[2],
                          lhs.d[3] + rhs.d[3]);
  }
  friend FM_vector<4,T> operator-(const FM_vector<4,T>& lhs,
                                  const FM_vector<4,T>& rhs)
  {
    return FM_vector<4,T>(lhs.d[0] - rhs.d[0],
                          lhs.d[1] - rhs.d[1],
                          lhs.d[2] - rhs.d[2],
                          lhs.d[3] - rhs.d[3]);
  }

  friend FM_vector<4,T>
  operator*(typename FM_traits<T>::element_type lhs,
            const FM_vector<4,T>& rhs)
  {
    return FM_vector<4,T>(lhs * rhs.d[0],
                          lhs * rhs.d[1],
                          lhs * rhs.d[2],
                          lhs * rhs.d[3]);
  }
  friend FM_vector<4,T>
  operator*(const FM_vector<4,T>& lhs,
            typename FM_traits<T>::element_type rhs)
  {
    return FM_vector<4,T>(lhs.d[0] * rhs,
                          lhs.d[1] * rhs,
                          lhs.d[2] * rhs,
                          lhs.d[3] * rhs);
  }

  friend T FM_dot<T>(const FM_vector<4,T>&,
                     const FM_vector<4,T>&);

private:
  T d[4];
};


template <int N, typename T>
bool operator!=(const FM_vector<N,T>& lhs,
                const FM_vector<N,T>& rhs)
{
  return !(lhs == rhs);
}


template <int N, typename T>
std::ostream& operator<<(std::ostream& lhs,
                         const FM_vector<N,T>& rhs)
{
  lhs << "(";
  lhs << rhs[0];
  for (int i = 1; i < N; i++)
    lhs << ", " << rhs[i];
  return lhs << ")";
}


template <int N, typename T>
T FM_dot(const FM_vector<N,T>& lhs, const FM_vector<N,T>& rhs)
```

```
{
  T res = lhs[0] * rhs[0];
  for (int i = 1; i < N; i++)
    res += lhs[i] * rhs[i];
  return res;
}


template <typename T>
T FM_dot(const FM_vector<1,T>& lhs,
         const FM_vector<1,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0];
}

template <typename T>
T FM_dot(const FM_vector<2,T>& lhs,
         const FM_vector<2,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0] +
    lhs.d[1] * rhs.d[1];
}

template <typename T>
T FM_dot(const FM_vector<3,T>& lhs,
         const FM_vector<3,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0] +
    lhs.d[1] * rhs.d[1] +
    lhs.d[2] * rhs.d[2];
}

template <typename T>
T FM_dot(const FM_vector<4,T>& lhs,
         const FM_vector<4,T>& rhs)
{
  return
    lhs.d[0] * rhs.d[0] +
    lhs.d[1] * rhs.d[1] +
    lhs.d[2] * rhs.d[2] +
    lhs.d[3] * rhs.d[3];
}


template <typename T>
FM_vector<3,T> FM_cross(const FM_vector<3,T>& lhs,
                        const FM_vector<3,T>& rhs)
{
  return FM_vector<3,T>(lhs.d[1] * rhs.d[2] - rhs.d[1] * lhs.d[2],
                        rhs.d[0] * lhs.d[2] - lhs.d[0] * rhs.d[2],
                        lhs.d[0] * rhs.d[1] - rhs.d[0] * lhs.d[1]);
}


template <int N, typename T>
T FM_mag(const FM_vector<N,T>& v)
{
  return (T) sqrt(FM_dot(v, v));
}


template <int N, typename T>
T FM_distance2(const FM_vector<N,T>& lhs, const FM_vector<N,T>& rhs)
{
  FM_vector<N,T> d = rhs - lhs;
```

```
  return FM_dot(d, d);
}


template <int N>
FM_vector<N,bool> operator!(const FM_vector<N,bool>& u)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = !u[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
FM_vector<N,bool> operator&&(const FM_vector<N,bool>& lhs,
                             const FM_vector<N,bool>& rhs)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] && rhs[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
FM_vector<N,bool> operator||(const FM_vector<N,bool>& lhs,
                             const FM_vector<N,bool>& rhs)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] || rhs[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
FM_vector<N,bool> operator^(const FM_vector<N,bool>& lhs,
                            const FM_vector<N,bool>& rhs)
{
  bool tmp[N];
  for (int i = 0; i < N; i++)
    tmp[i] = lhs[i] ^ rhs[i];
  return FM_vector<N,bool>(tmp);
}

template <int N>
bool operator<=(const FM_vector<N,bool>& lhs,
                const FM_vector<N,bool>& rhs)
{
  bool res = true;
  for (int i = 0; i < N; i++) {
    if (lhs[i] && !rhs[i]) {
      res = false;
      break;
    }
  }
  return res;
}

template <int N>
bool operator>=(const FM_vector<N,bool>& lhs,
                const FM_vector<N,bool>& rhs)
{
  bool res = true;
  for (int i = 0; i < N; i++) {
    if (!lhs[i] && rhs[i]) {
      res = false;
      break;
    }
  }
```

```
  return res;
}


typedef FM_vector<2,int> FM_vector2i;
typedef FM_vector<2,float> FM_vector2f;
typedef FM_vector<2,double> FM_vector2d;
typedef FM_vector<3,int> FM_vector3i;
typedef FM_vector<3,float> FM_vector3f;
typedef FM_vector<3,double> FM_vector3d;
typedef FM_vector<4,int> FM_vector4i;
typedef FM_vector<4,float> FM_vector4f;
typedef FM_vector<4,double> FM_vector4d;


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 * LOG:
 *      $Log$
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_MATRIX_H_
#define _FM_MATRIX_H_
/*
 * NAME: FM_matrix.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include "FM_vector.h"


template <int M, int N, typename T>
FM_vector<N,T>
operator*(const FM_vector<M,T>& lhs,
          const FM_vector<M,FM_vector<N,T> >& rhs)
{
  T tmp[N], sum;
  for (int n = 0; n < N; n++) {
    sum = (T) 0;
    for (int m = 0; m < M; m++) {
      sum += lhs[m] * rhs[m][n];
    }
    tmp[n] = sum;
  }
  return FM_vector<N,T>(tmp);
}

template <typename T>
FM_vector<3,T>
operator*(const FM_vector<3,T>& lhs,
          const FM_vector<3,FM_vector<3,T> >& rhs)
{
  return FM_vector<3,T>(lhs[0] * rhs[0][0] +
                        lhs[1] * rhs[1][0] +
                        lhs[2] * rhs[2][0],
                        lhs[0] * rhs[0][1] +
                        lhs[1] * rhs[1][1] +
                        lhs[2] * rhs[2][1],
                        lhs[0] * rhs[0][2] +
                        lhs[1] * rhs[1][2] +
                        lhs[2] * rhs[2][2]);
}

template <typename T>
FM_vector<4,T>
operator*(const FM_vector<4,T>& lhs,
          const FM_vector<4,FM_vector<4,T> >& rhs)
{
  return FM_vector<4,T>(lhs[0] * rhs[0][0] +
                        lhs[1] * rhs[1][0] +
                        lhs[2] * rhs[2][0] +
                        lhs[3] * rhs[3][0],
                        lhs[0] * rhs[0][1] +
                        lhs[1] * rhs[1][1] +
                        lhs[2] * rhs[2][1] +
                        lhs[3] * rhs[3][1],
                        lhs[0] * rhs[0][2] +
                        lhs[1] * rhs[1][2] +
                        lhs[2] * rhs[2][2] +
                        lhs[3] * rhs[3][2],
                        lhs[0] * rhs[0][3] +
                        lhs[1] * rhs[1][3] +
                        lhs[2] * rhs[2][3] +
                        lhs[3] * rhs[3][3]);
}
```

```
template <int M, int N, typename T>
FM_vector<M,T>
operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
          const FM_vector<N,T>& rhs)
{
  T tmp[M];
  for (int m = 0; m < M; m++) {
    tmp[m] = FM_dot(lhs[m], rhs);
  }
  return FM_vector<M,T>(tmp);
}

template <typename T>
FM_vector<3,T>
operator*(const FM_vector<3,FM_vector<3,T> >& lhs,
          const FM_vector<3,T>& rhs)
{
  return FM_vector<3,T>(FM_dot(lhs[0], rhs),
                        FM_dot(lhs[1], rhs),
                        FM_dot(lhs[2], rhs));
}

template <typename T>
FM_vector<4,T>
operator*(const FM_vector<4,FM_vector<4,T> >& lhs,
          const FM_vector<4,T>& rhs)
{
  return FM_vector<4,T>(FM_dot(lhs[0], rhs),
                        FM_dot(lhs[1], rhs),
                        FM_dot(lhs[2], rhs),
                        FM_dot(lhs[3], rhs));
}


template <int M, int N, int P, typename T>
FM_vector<M,FM_vector<P,T> >
operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
          const FM_vector<N,FM_vector<P,T> >& rhs)
{
  FM_vector<P,T> tmp[M];
  T sum;
  for (int m = 0; m < M; m++) {
    for (int p = 0; p < P; p++) {
      sum = (T) 0;
      for (int n = 0; n < N; n++) {
        sum += lhs[m][n] * rhs[n][p];
      }
      tmp[m][p] = sum;
    }
  }
  return FM_vector<M,FM_vector<P,T> >(tmp);
}


template <int M, int N, typename T>
FM_vector<N,FM_vector<M,T> >
FM_transpose(const FM_vector<M,FM_vector<N,T> >& in)
{
  FM_vector<M,T> tmp[N];
  for (int m = 0; m < M; m++) {
    for (int n = 0; n < N; n++) {
      tmp[n][m] = in[m][n];
    }
  }
  return FM_vector<N,FM_vector<M,T> >(tmp);
}

template <typename T>
```

```
FM_vector<3,FM_vector<3,T> >
FM_transpose(const FM_vector<3,FM_vector<3,T> >& in)
{
  return FM_vector<3,FM_vector<3,T> >
    (FM_vector<3,T>(in[0][0], in[1][0], in[2][0]),
     FM_vector<3,T>(in[0][1], in[1][1], in[2][1]),
     FM_vector<3,T>(in[0][2], in[1][2], in[2][2]));
}


template <int N, typename T>
T FM_det(const FM_vector<N,FM_vector<N,T> >&);

template <int N, typename T>
T FM_minor(const FM_vector<N,FM_vector<N,T> >& in, int row, int col)
{
  FM_vector<N-1,FM_vector<N-1,T> > tmp;
  int dst_row, dst_col;
  dst_row = 0;
  for (int src_row = 0; src_row < N; src_row++) {
    if (src_row == row) continue;
    dst_col = 0;
    for (int src_col = 0; src_col < N; src_col++) {
      if (src_col == col) continue;
      tmp[dst_row][dst_col] = in[src_row][src_col];
      dst_col++;
    }
    dst_row++;
  }
  return FM_det(tmp);
}

template <int N, typename T>
T FM_det(const FM_vector<N,FM_vector<N,T> >& in)
{
  T sum = (T) 0;
  for (int n = 0; n < N; n++) {
    T minor = FM_minor(in, n, 0);
    T cofactor = (n & 1) ? -minor : minor;
    sum += in[n][0] * cofactor;
  }
  return sum;
}

template <typename T>
T FM_det(const FM_vector<1,FM_vector<1,T> >& in)
{
  return in[0][0];
}

template <typename T>
T FM_det(const FM_vector<2,FM_vector<2,T> >& in)
{
  return in[0][0] * in[1][1] - in[1][0] * in[0][1];
}

template <typename T>
T FM_det(const FM_vector<3,FM_vector<3,T> >& in)
{
  return
    in[0][0] * (in[1][1] * in[2][2] - in[2][1] * in[1][2]) -
    in[1][0] * (in[0][1] * in[2][2] - in[2][1] * in[0][2]) +
    in[2][0] * (in[0][1] * in[1][2] - in[1][1] * in[0][2]);
}

template <typename T>
T FM_det(const FM_vector<4,FM_vector<4,T> >& in)
{
```

```
  // columns 2,3
  T r0r1 = in[0][2] * in[1][3] - in[1][2] * in[0][3];
  T r0r2 = in[0][2] * in[2][3] - in[2][2] * in[0][3];
  T r0r3 = in[0][2] * in[3][3] - in[3][2] * in[0][3];
  T r1r2 = in[1][2] * in[2][3] - in[2][2] * in[1][3];
  T r1r3 = in[1][2] * in[3][3] - in[3][2] * in[1][3];
  T r2r3 = in[2][2] * in[3][3] - in[3][2] * in[2][3];

  // column 0
  T minor0 = in[1][1] * r2r3 - in[2][1] * r1r3 + in[3][1] * r1r2;
  T minor1 = in[0][1] * r2r3 - in[2][1] * r0r3 + in[3][1] * r0r2;
  T minor2 = in[0][1] * r1r3 - in[1][1] * r0r3 + in[3][1] * r0r1;
  T minor3 = in[0][1] * r1r2 - in[1][1] * r0r2 + in[2][1] * r0r1;

  return
    in[0][0] * minor0 -
    in[1][0] * minor1 +
    in[2][0] * minor2 -
    in[3][0] * minor3;
}


template <int N, typename T>
FM_vector<N,FM_vector<N,T> >
FM_adj(const FM_vector<N,FM_vector<N,T> >& in)
{
  FM_vector<N,FM_vector<N,T> > res;
  for (int row = 0; row < N; row++) {
    for (int col = 0; col < N; col++) {
      T minor = FM_minor(in, row, col);
      T cofactor = ((row + col) & 1) ? -minor : minor;
      res[col][row] = cofactor;  // transpose
    }
  }
  return res;
}


template <int N, typename T>
int FM_inv(const FM_vector<N,FM_vector<N,T> >& in,
           FM_vector<N,FM_vector<N,T> >* out)
{
  T det = FM_det(in);
  if (det == (T) 0)
    return 1;
  *out = (T) 1 / det * FM_adj(in);
  return 0;
}

template <typename T>
int FM_inv(const FM_vector<2,FM_vector<2,T> >& in,
           FM_vector<2,FM_vector<2,T> >* out)
{
  T det = FM_det(in);
  if (det == (T) 0)
    return 1;
  T inv_det = (T) 1 / det;
  (*out)[0][0] = inv_det * in[1][1];
  (*out)[0][1] = inv_det * -in[0][1];
  (*out)[1][0] = inv_det * -in[1][0];
  (*out)[1][1] = inv_det * in[0][0];
  return 0;
}

template <typename T>
int FM_inv(const FM_vector<3,FM_vector<3,T> >& in,
           FM_vector<3,FM_vector<3,T> >* out)
{
```

```
  // column 0
  T minor0 = in[1][1] * in[2][2] - in[2][1] * in[1][2];
  T minor1 = in[0][1] * in[2][2] - in[2][1] * in[0][2];
  T minor2 = in[0][1] * in[1][2] - in[1][1] * in[0][2];

  T det =
    in[0][0] * minor0 -
    in[1][0] * minor1 +
    in[2][0] * minor2;

  if (det == (T) 0)
    return 1;
  T inv_det = (T) 1 / det;

  (*out)[0][0] = inv_det * minor0;
  (*out)[0][1] = inv_det * -minor1;
  (*out)[0][2] = inv_det * minor2;
  (*out)[1][0] = inv_det * (in[2][0] * in[1][2] - in[1][0] * in[2][2]);
  (*out)[1][1] = inv_det * (in[0][0] * in[2][2] - in[2][0] * in[0][2]);
  (*out)[1][2] = inv_det * (in[1][0] * in[0][2] - in[0][0] * in[1][2]);
  (*out)[2][0] = inv_det * (in[1][0] * in[2][1] - in[2][0] * in[1][1]);
  (*out)[2][1] = inv_det * (in[2][0] * in[0][1] - in[0][0] * in[2][1]);
  (*out)[2][2] = inv_det * (in[0][0] * in[1][1] - in[1][0] * in[0][1]);

  return 0;
}


template <typename T>
int FM_inv(const FM_vector<4,FM_vector<4,T> >& in,
           FM_vector<4,FM_vector<4,T> >* out)
{
  // compute minors column by column, but fill in (*out) row
  // by row to effectively transpose

  // columns 2,3
  T r0r1 = in[0][2] * in[1][3] - in[1][2] * in[0][3];
  T r0r2 = in[0][2] * in[2][3] - in[2][2] * in[0][3];
  T r0r3 = in[0][2] * in[3][3] - in[3][2] * in[0][3];
  T r1r2 = in[1][2] * in[2][3] - in[2][2] * in[1][3];
  T r1r3 = in[1][2] * in[3][3] - in[3][2] * in[1][3];
  T r2r3 = in[2][2] * in[3][3] - in[3][2] * in[2][3];

  // column 0
  T minor0 = in[1][1] * r2r3 - in[2][1] * r1r3 + in[3][1] * r1r2;
  T minor1 = in[0][1] * r2r3 - in[2][1] * r0r3 + in[3][1] * r0r2;
  T minor2 = in[0][1] * r1r3 - in[1][1] * r0r3 + in[3][1] * r0r1;
  T minor3 = in[0][1] * r1r2 - in[1][1] * r0r2 + in[2][1] * r0r1;

  T det =
    in[0][0] * minor0 -
    in[1][0] * minor1 +
    in[2][0] * minor2 -
    in[3][0] * minor3;

  if (det == (T) 0)
    return 1;
  T inv_det = (T) 1 / det;

  (*out)[0][0] = inv_det * minor0;
  (*out)[0][1] = inv_det * -minor1;
  (*out)[0][2] = inv_det * minor2;
  (*out)[0][3] = inv_det * -minor3;

  // column 1
  minor0 = in[1][0] * r2r3 - in[2][0] * r1r3 + in[3][0] * r1r2;
  minor1 = in[0][0] * r2r3 - in[2][0] * r0r3 + in[3][0] * r0r2;
  minor2 = in[0][0] * r1r3 - in[1][0] * r0r3 + in[3][0] * r0r1;
```

```
  minor3 = in[0][0] * r1r2 - in[1][0] * r0r2 + in[2][0] * r0r1;

  (*out)[1][0] = inv_det * -minor0;
  (*out)[1][1] = inv_det * minor1;
  (*out)[1][2] = inv_det * -minor2;
  (*out)[1][3] = inv_det * minor3;

  // columns 0,1
  r0r1 = in[0][0] * in[1][1] - in[1][0] * in[0][1];
  r0r2 = in[0][0] * in[2][1] - in[2][0] * in[0][1];
  r0r3 = in[0][0] * in[3][1] - in[3][0] * in[0][1];
  r1r2 = in[1][0] * in[2][1] - in[2][0] * in[1][1];
  r1r3 = in[1][0] * in[3][1] - in[3][0] * in[1][1];
  r2r3 = in[2][0] * in[3][1] - in[3][0] * in[2][1];

  // column 2
  minor0 = in[1][3] * r2r3 - in[2][3] * r1r3 + in[3][3] * r1r2;
  minor1 = in[0][3] * r2r3 - in[2][3] * r0r3 + in[3][3] * r0r2;
  minor2 = in[0][3] * r1r3 - in[1][3] * r0r3 + in[3][3] * r0r1;
  minor3 = in[0][3] * r1r2 - in[1][3] * r0r2 + in[2][3] * r0r1;

  (*out)[2][0] = inv_det * minor0;
  (*out)[2][1] = inv_det * -minor1;
  (*out)[2][2] = inv_det * minor2;
  (*out)[2][3] = inv_det * -minor3;

  // column 3
  minor0 = in[1][2] * r2r3 - in[2][2] * r1r3 + in[3][2] * r1r2;
  minor1 = in[0][2] * r2r3 - in[2][2] * r0r3 + in[3][2] * r0r2;
  minor2 = in[0][2] * r1r3 - in[1][2] * r0r3 + in[3][2] * r0r1;
  minor3 = in[0][2] * r1r2 - in[1][2] * r0r2 + in[2][2] * r0r1;

  (*out)[3][0] = inv_det * -minor0;
  (*out)[3][1] = inv_det * minor1;
  (*out)[3][2] = inv_det * -minor2;
  (*out)[3][3] = inv_det * minor3;

  return 0;
}


template <int N, typename T>
void FM_identity(FM_vector<N,FM_vector<N,T> >* out)
{
  T zero = (T) 0;
  T one = (T) 1;
  for (int row = 0; row < N; row++)
    for (int col = 0; col < N; col++)
      (*out)[row][col] = row == col ? one : zero;
}


typedef FM_vector<2,FM_vector<2,float> > FM_matrix22f;
typedef FM_vector<2,FM_vector<3,float> > FM_matrix23f;
typedef FM_vector<3,FM_vector<2,float> > FM_matrix32f;
typedef FM_vector<3,FM_vector<3,float> > FM_matrix33f;
typedef FM_vector<3,FM_vector<3,double> > FM_matrix33d;
typedef FM_vector<4,FM_vector<4,float> > FM_matrix44f;
typedef FM_vector<4,FM_vector<4,double> > FM_matrix44d;


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
```

```
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 * LOG:
 *      $Log$
 */
#endif
```

```
// Emacs mode -*-c++-*- //
#ifndef _FM_TIMER_H_
#define _FM_TIMER_H_
/*
 * NAME: FM_timer.h
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <sys/time.h>

class FM_timer
{
public:
  FM_timer() { reset(); }
  void reset() { total = 0.0; }
  void start() { gettimeofday(&start_tv, (struct timezone *) 0); }
  void stop()
  {
    struct timeval stop_tv;
    gettimeofday(&stop_tv, (struct timezone *) 0);
    long dts = stop_tv.tv_sec - start_tv.tv_sec;
    long dtus = stop_tv.tv_usec - start_tv.tv_usec;
    double dt = (double) dts + (double) dtus * 1.0e-6;

    // round to milliseconds
    long millisec = (long) (dt * 1000.0 + 0.5);
    total += millisec * 1000.0;
  }
  double elapsed() { return total; }

private:
  struct timeval start_tv;
  double total;
};

/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
 * to any person obtaining a copy of this software
 * and associated documentation files (the "Software"),
 * to deal in the Software without restriction,
 * including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit
 * persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission
 * notice shall be included in all copies or substantial
 * portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
 * OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
 * FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
 * THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 *
 * LOG:
 *      $Log$
 */
#endif
```

```
/*
 * NAME: vector_tests
 *
 * WRITTEN BY:
 *      Patrick Moran           pmoran@nas.nasa.gov
 */
#include <assert.h>
#include <stdlib.h>
#include "FM_vector.h"
#include "FM_matrix.h"
#include "FM_timer.h"

void random_init(int* i)
{
  *i = rand() % 10 - 5;
}

void random_init(float* f)
{
  *f = (float) (rand() % 20 - 10);
}

void random_init(double* d)
{
  *d = (double) (rand() % 20 - 10);
}

template <int N, typename T>
void random_init(FM_vector<N,T>* v)
{
  for (int i = 0; i < N; i++)
    random_init(&((*v)[i]));
}

template <typename T>
T max_mag(T t)
{
  return t >= (T) 0 ? t : -t;
}

template <int N, typename T>
typename FM_traits<T>::element_type
max_mag(const FM_vector<N,T>& v)
{
  typename FM_traits<T>::element_type res =
    (typename FM_traits<T>::element_type) 0;

  for (int i = 0; i < N; i++) {
    typename FM_traits<T>::element_type e = max_mag(v[i]);
    if (e > res)
      res = e;
  }
  return res;
}


inline static double fmax(double a, double b)
{
  return a > b ? a : b;
}


template <int N, typename T>
void general_tests()
{
  int i;
  T a[N];
  for (i = 0; i < N; i++)
```

```
    random_init(&a[i]);

  //FM_vector(const T[])
  //T& operator[](int)
  FM_vector<N,T> u(a);
  for (i = 0; i < N; i++)
    assert(u[i] == a[i]);
  for (i = 0; i < N; i++)
    u[i] = u[i];
  for (i = 0; i < N; i++)
    assert(u[i] == a[i]);

  //const T& operator[](int) const
  const FM_vector<N,T> cu = u;
  for (i = 0; i < N; i++)
    assert(u[i] == cu[i]);

  //FM_vector(T, T, ...)

  //friend bool operator==(const FM_vector<N,T>&,
  //                       const FM_vector<N,T>&)
  //friend bool operator!=(const FM_vector<N,T>&,
  //                       const FM_vector<N,T>&)
  assert(u == u);
  assert(!(u != u));
  FM_vector<N,T> v = u;
  while (u == v)
    random_init(&v);
  assert(u != v);

  //friend FM_vector<N,T> operator+(const FM_vector<N,T>&,
  //                                const FM_vector<N,T>&)
  FM_vector<N,T> w = u + v;
  for (i = 0; i < N; i++)
    assert(w[i] == u[i] + v[i]);

  //friend ostream& operator<<(ostream&,
  //                           const FM_vector<N,T>&)
  //std::cout << v << endl;

  //FM_vector<N,T> operator*(typename FM_traits<T>::element_type,
  //                         const FM_vector<N,T>&)
  typename FM_traits<T>::element_type s = 0;
  random_init(&s);
  v = s * u;
  for (i = 0; i < N; i++)
    assert(v[i] == s * u[i]);

  //FM_vector<N,T> operator*(const FM_vector<N,T>&,
  //                         typename FM_traits<T>::element_type)
  v = u * s;
  for (i = 0; i < N; i++)
    assert(v[i] == u[i] * s);

  random_init(&u);
  random_init(&v);

  //friend FM_vector<N,T> operator-(const FM_vector<N,T>&)
  w = -u;
  for (i = 0; i < N; i++)
    assert(w[i] == -u[i]);

  //friend FM_vector<N,T> operator-(const FM_vector<N,T>&,
  //                                const FM_vector<N,T>&)
  w = u - v;
  for (i = 0; i < N; i++)
    assert(w[i] == u[i] - v[i]);
}
```

```
template <int N, typename T>
void vector_of_scalars_tests()
{
  int i;
  int sum;
  FM_vector<N,T> u, v;
  random_init(&u);
  random_init(&v);

  //friend T FM_dot(const FM_vector<N,T>&,
  //                const FM_vector<N,T>&)
  sum = 0;
  for (i = 0; i < N; i++)
    sum += u[i] * v[i];
  assert(sum == FM_dot(u, v));

  //friend FM_vector<3,T> FM_cross(const FM_vector<3,T>&,
  //                               const FM_vector<3,T>&)
  T t;
  FM_vector<3,T> t3, u3, v3, w3, x3;
  random_init(&u3);
  random_init(&v3);
  random_init(&w3);
  random_init(&x3);

  // u3 x (v3 x w3) == v3(u3 . w3) - w3(u3 . v3)
  t3 = FM_cross(u3, FM_cross(v3, w3)) -
    (v3 * FM_dot(u3, w3) - w3 * FM_dot(u3, v3));
  assert(max_mag(t3) == 0);

  // (u3 x v3) x w3 == v3(u3 . w3) - u3(v3 . w3)
  t3 = FM_cross(FM_cross(u3, v3), w3) -
    (v3 * FM_dot(u3, w3) - u3 * FM_dot(v3, w3));
  assert(max_mag(t3) == 0);

  // (u3 x v3) . (w3 x x3) == (u3 . w3)(v3 . x3) - (u3 . x3)(v3 . w3)
  t = FM_dot(FM_cross(u3, v3), FM_cross(w3, x3)) -
    (FM_dot(u3, w3) * FM_dot(v3, x3) - FM_dot(u3, x3) * FM_dot(v3, w3));
  assert(t == 0);

  // u3 x (v3 x w3) + v3 x (w3 x u3) + w3 x (u3 x v3) == 0
  t3 = FM_cross(u3, FM_cross(v3, w3)) +
    FM_cross(v3, FM_cross(w3, u3)) + FM_cross(w3, FM_cross(u3, v3));
  assert(max_mag(t3) == 0);
}


template <int N, typename T>
void real_vector_of_scalars_tests()
{
  int i;
  double epsilon = 1e-5;
  FM_vector<N,T> u;

  //friend T FM_mag(const FM_vector<N,T>&)
  random_init(&u);
  double dsum = 0.0;
  for (i = 0; i < N; i++)
    dsum += (double) (u[i] * u[i]);
  assert(fabs(sqrt(dsum) - (double) FM_mag(u)) < epsilon);
}


template <int M, int N, typename T>
void vector_matrix_tests()
{
```

```
FM_vector<M,T> uM, vM;
FM_vector<N,T> uN, vN;
FM_vector<M,FM_vector<N,T> > mMN;
FM_vector<N,FM_vector<M,T> > mNM;
FM_vector<M,FM_vector<M,T> > mMM, miMM, iMM;
FM_vector<N,FM_vector<N,T> > mNN;

//Let P = M + N + 3
FM_vector<M,FM_vector<M+N+3,T> > mMP;
FM_vector<N,FM_vector<M+N+3,T> > mNP;
FM_vector<M+N+3,FM_vector<M,T> > mPM;

int row, col;
T det;

//FM_vector<N,FM_vector<M,T> >
//FM_transpose(const FM_vector<M,FM_vector<N,T> >& in)
random_init(&mMN);
mNM = FM_transpose(mMN);
for (int row = 0; row < M; row++)
  for (int col = 0; col < N; col++)
    assert(mMN[row][col] == mNM[col][row]);

//FM_vector<N,T>
//operator*(const FM_vector<M,T>& lhs,
//          const FM_vector<M,FM_vector<N,T> >& rhs)
//FM_vector<M,T>
//operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
//          const FM_vector<N,T>& rhs)
random_init(&vM);
random_init(&mMN);
uN = vM * mMN - FM_transpose(mMN) * vM;
assert(max_mag(uN) == (T) 0);

//FM_vector<M,FM_vector<P,T> >
//operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
//          const FM_vector<N,FM_vector<P,T> >& rhs)
random_init(&mMN);
random_init(&mNP);
mMP = mMN * mNP;
mPM = FM_transpose(mNP) * FM_transpose(mMN);
assert(max_mag(mMP - FM_transpose(mPM)) == (T) 0);

//FM_vector<M,FM_vector<M,T> > FM_identity()
FM_identity(&mMM);
for (row = 0; row < M; row++)
  for (col = 0; col < M; col++)
    assert(mMM[row][col] == (row == col ? (T) 1 : (T) 0));

//T FM_det(const FM_vector<N,FM_vector<N,T> >&);
FM_identity(&mMM);
det = FM_det(mMM);
assert(det == (T) 1);

const T K = (T) 7;
mMM[0][0] = K;
det = FM_det(mMM);
assert(det == K);

random_init(&mMM);
assert(FM_det(mMM) - FM_det(FM_transpose(mMM)) == (T) 0);

//FM_vector<N,FM_vector<N,T> >
//FM_adj(const FM_vector<N,FM_vector<N,T> >& in)
random_init(&mMM);
FM_identity(&iMM);
det = FM_det(mMM);
assert(max_mag(FM_adj(mMM) * mMM - det * iMM) == 0);
```

```
}


template <int M, typename T>
void real_vector_matrix_tests()
{
  int row, res;
  T epsilon = (T) 1e-4;
  T det;
  const T K = (T) 7;

  FM_vector<M,T> uM, vM;
  FM_vector<M,FM_vector<M,T> > mMM, miMM, iMM;

  //int FM_inv(const FM_vector<N,FM_vector<N,T> >& in,
  //           FM_vector<N,FM_vector<N,T> >* out)
  det = FM_det(mMM);
  while (det == (T) 0) {
    random_init(&mMM);
    det = FM_det(mMM);
  }
  res = FM_inv(mMM, &miMM);
  assert(res == 0);
  //mMM[0] = K * mMM[M - 1];
  //res = FM_inv(mMM, &miMM);
  //assert(res != 0);

  random_init(&mMM);
  det = FM_det(mMM);
  while (det == (T) 0) {
    random_init(&mMM);
    det = FM_det(mMM);
  }
  for (row = 0; row < M; row++)
    mMM[row][0] = K * mMM[row][M - 1];
  res = FM_inv(mMM, &miMM);
  assert(res != 0);

  //FM_vector<M,FM_vector<P,T> >
  //operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
  //          const FM_vector<N,FM_vector<P,T> >& rhs)
  FM_identity(&iMM);
  random_init(&mMM);
  res = FM_inv(mMM, &miMM);
  while (res != 0) {
    random_init(&mMM);
    res = FM_inv(mMM, &miMM);
  }
  epsilon = 1e-3;
  assert(max_mag(mMM * miMM - iMM) < epsilon);
  assert(max_mag(miMM * mMM - iMM) < epsilon);

  //int FM_inv(const FM_vector<N,FM_vector<N,T> >& in,
  //           FM_vector<N,FM_vector<N,T> >* out)
  det = FM_det(mMM);
  while (det == (T) 0) {
    random_init(&mMM);
    det = FM_det(mMM);
  }
  res = FM_inv(mMM, &miMM);
  assert(res == 0);
  mMM[0] = K * mMM[M - 1];
  res = FM_inv(mMM, &miMM);
  assert(res != 0);

  random_init(&mMM);
  det = FM_det(mMM);
  while (det == (T) 0) {
```

```
    random_init(&mMM);
    det = FM_det(mMM);
  }
  for (row = 0; row < M; row++)
    mMM[row][0] = K * mMM[row][M - 1];
  res = FM_inv(mMM, &miMM);
  assert(res != 0);

  //FM_vector<M,FM_vector<P,T> >
  //operator*(const FM_vector<M,FM_vector<N,T> >& lhs,
  //          const FM_vector<N,FM_vector<P,T> >& rhs)
  FM_identity(&iMM);
  random_init(&mMM);
  res = FM_inv(mMM, &miMM);
  while (res != 0) {
    random_init(&mMM);
    res = FM_inv(mMM, &miMM);
  }
  epsilon = 1e-3;
  assert(max_mag(mMM * miMM - iMM) < epsilon);
  assert(max_mag(miMM * mMM - iMM) < epsilon);
}


template <typename T>
inline static int sign(T t)
{
  return t > (T) 0 ? 1 : (t < (T) 0 ? -1 : 0);
}


template <typename T>
//inline
int orient(const FM_vector<3,T>& a,
           const FM_vector<3,T>& b,
           const FM_vector<3,T>& c,
           const FM_vector<3,T>& d)
{
  return sign(FM_dot(d - a, FM_cross(b - a, c - a)));
}


template <typename T>
void orient_tests(const FM_vector<3,T>&)
{
  int i, res = 0;
  const int N_STMTS = 1;
  const int N_TRIALS = 10000000;
  FM_timer timer;
  double total;
  FM_vector<3,T> a, b, c, d;
  random_init(&a);
  random_init(&b);
  random_init(&c);
  random_init(&d);

  res = 0;
  timer.start();
  for (i = 0; i < N_TRIALS; i++) {
    res += orient(a, b, c, d);
  }
  timer.stop();
  res = res;
  total = timer.elapsed();
  std::cout << total / (double) (N_TRIALS * N_STMTS)
            << " usec / orient" << std::endl;
}
```

```
void timing_tests()
{
  orient_tests(FM_vector<3,float>());
  orient_tests(FM_vector<3,double>());
}

int main()
{
  for (int i = 0; i < 1000; i++) {
    general_tests<1,int>();
    general_tests<2,int>();
    general_tests<3,int>();
    general_tests<4,int>();
    general_tests<17,int>();
    general_tests<4,FM_vector<3,int> >();
    general_tests<5,FM_vector<7,int> >();
    general_tests<2,FM_vector<4,FM_vector<3,int> > >();
    general_tests<8,FM_vector<5,FM_vector<7,int> > >();

    vector_of_scalars_tests<2,int>();
    vector_of_scalars_tests<3,int>();
    vector_of_scalars_tests<4,int>();
    vector_of_scalars_tests<5,int>();

    real_vector_of_scalars_tests<2,double>();
    real_vector_of_scalars_tests<3,double>();
    real_vector_of_scalars_tests<4,double>();
    real_vector_of_scalars_tests<5,double>();

    vector_matrix_tests<2,2,int>();
    vector_matrix_tests<2,3,int>();
    vector_matrix_tests<2,4,int>();
    vector_matrix_tests<2,5,int>();

    vector_matrix_tests<3,2,int>();
    vector_matrix_tests<3,3,int>();
    vector_matrix_tests<3,4,int>();
    vector_matrix_tests<3,5,int>();

    vector_matrix_tests<4,2,int>();
    vector_matrix_tests<4,3,int>();
    vector_matrix_tests<4,4,int>();
    vector_matrix_tests<4,5,int>();

    vector_matrix_tests<5,2,int>();
    vector_matrix_tests<5,3,int>();
    vector_matrix_tests<5,4,int>();
    vector_matrix_tests<5,5,int>();

    real_vector_matrix_tests<2,double>();
    real_vector_matrix_tests<3,double>();
    real_vector_matrix_tests<4,double>();
    real_vector_matrix_tests<5,double>();
  }

  timing_tests();

  std::cout << "OK" << std::endl;
  return 0;
}


/*
 * Copyright (c) 2000
 * Advanced Management Technology, Incorporated
 *
 * Permission is hereby granted, free of charge,
```

```
* to any person obtaining a copy of this software
* and associated documentation files (the "Software"),
* to deal in the Software without restriction,
* including without limitation the rights to use,
* copy, modify, merge, publish, distribute, sublicense,
* and/or sell copies of the Software, and to permit
* persons to whom the Software is furnished to do so,
* subject to the following conditions:
*
* The above copyright notice and this permission
* notice shall be included in all copies or substantial
* portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
* OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
* LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
* FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
* EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
* FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
* IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
* FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR
* THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*
* LOG:
*      $Log$
*/
```