```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>
#include <cmath>

#define TINY    1.0e-20;

// forward declarations
template <typename T> class matrix;
template <typename T> std::ostream& operator<<(std::ostream&, const matrix<T>&);
template <typename T> matrix<T> operator*(const T&, const matrix<T>&);
template <typename T> matrix<T> operator/(const T&, const matrix<T>&);
template <typename T> matrix<T> operator+(const T&, const matrix<T>&);
template <typename T> matrix<T> operator-(const T&, const matrix<T>&);
template <typename T> std::vector<T> operator*(const std::vector<T>&, const matrix<T>&)
;
template <typename T> std::vector<T> operator*=(std::vector<T>&, const matrix<T>&);

template <typename T>
class matrix {
        public:
        // constructors (overloaded)
        matrix(int r=4, int c=4) : arr(r)
          { for(int i=0;i<r;++i) arr[i].resize(c); }
        matrix(const matrix& rhs) : arr(rhs.rows())
          { for(int i=0;i<rows();++i) arr[i] = rhs.arr[i]; }

        // destructors -- default should be ok

        // operators (dereferencing)
        const std::vector<T> & operator[](int r) const  { return arr[r]; }
        std::vector<T> & operator[](int r)              { return arr[r]; }

        // operators (copy assignment)
        matrix  operator=(const matrix& rhs);

        // operators (unary)
        matrix  operator-(void);
        matrix  operator!(void);

        // operators (const matrix&, const matrix&)
        int     operator==(const matrix& rhs);
        int     operator<(const matrix& rhs);
        int     operator>(const matrix& rhs);
        matrix  operator*=(const matrix& rhs);
        matrix  operator*(const matrix& rhs);
        matrix  operator/=(matrix& rhs);
        matrix  operator/(matrix& rhs);
        matrix  operator+=(const matrix& rhs);
        matrix  operator+(const matrix& rhs);
        matrix  operator-=(const matrix& rhs);
        matrix  operator-(const matrix& rhs);

        // operators (const matrix&, scalar)
        matrix  operator*=(const T& rhs);
        matrix  operator*(const T& rhs);
        matrix  operator/=(const T& rhs);
```

```
        matrix  operator/(const T& rhs);
        matrix  operator+=(const T& rhs);
        matrix  operator+(const T& rhs);
        matrix  operator-=(const T& rhs);
        matrix  operator-(const T& rhs);

        // operators (const scalar&, const matrix&)
        // note: g++ outputs error 'declaration of 'operator*' as non-function'
        // the problem is that the declaration of member operator* hides all
        // overloaded declarations from global scope---to access the global
        // function, you need to explicitly qualify it:
        //   friend matrix ::operator* <> (const T&, const matrix&);
        // but this will not compile since :: gets associated with matrix<T>;
        // instead, put parenthesis around the function name
        //   friend matrix (::operator* <>) (const T&, const matrix&);
#ifdef GCC3_3
        // since we need to use g++3.3 on the Macs...
        friend matrix operator* <>(const T& lhs, const matrix& rhs);
        friend matrix operator/ <>(const T& lhs, const matrix& rhs);
        friend matrix operator+ <>(const T& lhs, const matrix& rhs);
        friend matrix operator- <>(const T& lhs, const matrix& rhs);

        friend std::vector<T> operator* <>(const std::vector<T>&, const matrix&);
#else
        friend matrix (::operator* <>)(const T& lhs, const matrix& rhs);
        friend matrix (::operator/ <>)(const T& lhs, const matrix& rhs);
        friend matrix (::operator+ <>)(const T& lhs, const matrix& rhs);
        friend matrix (::operator- <>)(const T& lhs, const matrix& rhs);

        friend std::vector<T> (::operator* <>)(const std::vector<T>&, const matrix&);
#endif

        // friends -- note the extra <> telling the compiler to instantiate
        // a templated version of operator<< -- <T> is also legal, i.e.,
        // friend std::ostream& operator<< <T>(std::ostream&, const matrix&);
//      friend std::istream&    operator>>(std::istream&, matrix&);
//      friend std::istream&    operator>>(std::istream&, matrix*)
//        { return(s >> (*rhs)); }
        friend std::ostream&    operator<< <>(std::ostream&, const matrix&);
        friend std::ostream&    operator<<(std::ostream& s, matrix* rhs)
          { return(s << (*rhs)); }

        // member functions
        int     rows(void) const        { return arr.size(); }
        int     cols(void) const        { return rows() ? arr[0].size() : 0; }
        void    zero(void);
        void    one(void);
        void    identity(void);
        matrix  clamp(T lo,T hi);
        matrix  transpose(void);
        T       trace(void) const;

        private:
        std::vector<std::vector<T> >    arr;
        matrix<T>                       ludcmp(int *indx,T *d);
        matrix<T>                       lubksb(int *indx,T *b);
};

#endif
```

```
#ifndef QUAT_H
#define QUAT_H

#include        "matrix.h"

#define TRACKBALLSIZE   (0.8)
#define ROOT2           (1.41421356237309504880)
#define ROOT2OVER2      (.70710678118654752440)
#define EPSILON         (0.00001)

// forward declarations
template <typename T> class quaternion;
template <typename T> std::ostream& operator<<(std::ostream&, const quaternion<T>&);
template <typename T> quaternion<T> operator*(T lhs, quaternion<T>& rhs);
template <typename T> quaternion<T> operator/(T lhs, quaternion<T>& rhs);

template <typename T>
class quaternion {
        public:
        // constructors (overloaded)
        quaternion(T sin=(T)0.0) : \
                q(4,(T)0.0)                     { q[3] = (T)sin; }
        quaternion(const quaternion& rhs) : \
                q(rhs.q)                        { };
        quaternion(quaternion *rhs) : \
                q(rhs->q)                       { };
        quaternion(std::vector<T> qin) : \
                q(4,(T)0.0)                     { q[0] = qin[0];
                                                  q[1] = qin[1];
                                                  q[2] = qin[2];
                                                  q[3] = (T)1.0;
                                                }
        quaternion(matrix<T>& m) : \
                q(4,(T)0.0)                     { q[3] = (T)1.0; mattoquat(m); }
        quaternion(T sin, std::vector<T> qin) : \
                q(4,(T)0.0)                     { q[0] = qin[0];
                                                  q[1] = qin[1];
                                                  q[2] = qin[2];
                                                  q[3] = (T)sin;
                                                }
        quaternion(T x1,T y1,T x2,T y2) : \
                q(4,(T)0.0)                     { q[3] = (T)1.0;
                                                  trackball(x1,y1,x2,y2);
                                                }
        // destructors
        //~quaternion()                         { };

        const T& operator[](int r) const        { return q[r]; }
        T& operator[](int r)                    { return q[r]; }

        // operators: unary
        quaternion operator-(void);
        quaternion operator!(void);                     // quaternion inverse

        // operators: (quaternion, const matrix&)
        quaternion operator*(const matrix<T>& rhs);

        // operators: (quaternion, const quaternion&)
        int operator==(const quaternion& rhs);
        int operator<(const quaternion& rhs);
        int operator>(const quaternion& rhs);
        quaternion operator=(const quaternion& rhs);
```

```
        quaternion operator*=(const quaternion& rhs);
        quaternion operator*(const quaternion& rhs);
        quaternion operator/=(quaternion& rhs);
        quaternion operator/(quaternion& rhs);
        quaternion operator+=(const quaternion& rhs);
        quaternion operator+(const quaternion& rhs);
        quaternion operator-=(const quaternion& rhs);
        quaternion operator-(const quaternion& rhs);

        // operators: (quaternion, quaternion*)
        int operator==(quaternion* rhs)         { return((*this) == (*rhs)); }
        int operator<(quaternion* rhs)          { return((*this) == (*rhs)); }
        int operator>(quaternion* rhs)          { return((*this) == (*rhs)); }
        quaternion operator=(quaternion* rhs)   { return((*this)  = (*rhs)); }
        quaternion operator*=(quaternion* rhs)  { return((*this) *= (*rhs)); }
        quaternion operator*(quaternion* rhs)   { return((*this) *  (*rhs)); }
        quaternion operator/=(quaternion* rhs)  { return((*this) /= (*rhs)); }
        quaternion operator/(quaternion* rhs)   { return((*this) /  (*rhs)); }
        quaternion operator+=(quaternion* rhs)  { return((*this) += (*rhs)); }
        quaternion operator+(quaternion* rhs)   { return((*this) +  (*rhs)); }
        quaternion operator-=(quaternion* rhs)  { return((*this) -= (*rhs)); }
        quaternion operator-(quaternion* rhs)   { return((*this) -  (*rhs)); }

        // operators (const quaternion&, scalar)
        quaternion operator*=(const T& rhs);
        quaternion operator*(const T& rhs);
        quaternion operator/=(const T& rhs);
        quaternion operator/(const T& rhs);

        // friend operators: (const scalar*, const quaternion&)
#ifdef GCC3_3
        friend quaternion operator* <>(T lhs,quaternion& rhs);
        friend quaternion operator/ <>(T lhs,quaternion& rhs);
#else
        friend quaternion (::operator* <>)(T lhs,quaternion& rhs);
        friend quaternion (::operator/ <>)(T lhs,quaternion& rhs);
#endif

        // friends
        friend std::ostream& operator<< <>(std::ostream&, const quaternion&);
//      friend std::ostream& operator<<(std::ostream& s, quaternion* rhs)
//        { return(s << (*rhs)); }

        // member functions
        const T&        s()                     { return((const T&)q[3]); }
        std::vector<T>  v()                     { std::vector<T> r(3,(T)0.0);
                                                  for(int i=0;i<3;++i)
                                                    r[i] = q[i];
                                                  return(r);
                                                }
        T               dot(quaternion& rhs);
        T               dot(quaternion* rhs)    { return((T)dot((*rhs))); }
        T               sqlength(void)          { return((T)dot(this)); }
        T               length(void)            { return((T)sqrt(dot(this))); }
        quaternion      norm(void)              { return(*this / length()); }
        matrix<T>       L(void);
        matrix<T>       R(void);
        matrix<T>       quattomat(void);        // return (4x4) rot matrix
        void            mattoquat(matrix<T>& m);
        quaternion      slerp2(quaternion rhs,T t);// spherical linear interp.
        quaternion      slerp(quaternion rhs,T t); // spherical linear interp.
```

```
        // protected: available to this (base) class and subs. derived classes
        protected:
        void            trackball(T x1,T y1,T x2,T y2);
        T               project_to_sphere(T r,T x,T y);

        // private: only available to this class
        private:
        std::vector<T>  q;                      // [x,y,z,s]
};

#endif
```

```
#ifndef VECTOR_H
#define VECTOR_H

#include <iostream>
#include <vector>

template <typename T>
std::ostream&   operator<<(std::ostream& s, const std::vector<T>&);

template <typename T>
std::vector<T>  operator-(const std::vector<T>&);

template <typename T>
std::vector<T>  operator*(const T&, const std::vector<T>&);

template <typename T>
std::vector<T>  operator+(const std::vector<T>&, const std::vector<T>&);

template <typename T>
std::vector<T>  operator-(const std::vector<T>&, const std::vector<T>&);

template <typename T>
T               dot(const std::vector<T>&, const std::vector<T>&);

template <typename T>
std::vector<T>  cross(const std::vector<T>&, const std::vector<T>&);

template <typename T>
std::vector<T>  norm(const std::vector<T>&);

#endif
```