```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>

#include "vector.h"
#include "matrix.h"
#include "quaternion.h"

// prototypes
int     main(int argc, char **argv);
void    test_quaternion();
void    test_vector();
void    test_matrix();

int main(int argc, char **argv)
{
//test_quaternion();
//test_vector();
  test_matrix();
}

void test_quaternion()
{

  // typical usage would be:
  //   0. store q with some graphical object
  //   1. in (interactive) loop:
  //     1.1. figure out incremental rotation, q'
  //       1.1.1. (use q' = new quaternion<Type>(x1,y1,x2,y2))
  //     1.2. "add to" object's quaternion, q = q' * q (order is important!)
  //     1.3. normalize q, q.norm()
  //   2. in display routine:
  //     2.1. get transformation matrix, M = q.rotMat() (in row-major order!)
  //     2.3. feed M to OpenGL's object matrix stack (may need to transpose)
  //

        double                  theta = 45.0*M_PI/180.0;// theta (in radians)
        std::vector<double>     vzero(3,0.0);           // rotation axis
        std::vector<double>     x(3,0.0);               // rotation axis
        std::vector<double>     y(3,0.0);               // rotation axis
        std::vector<double>     z(3,0.0);               // rotation axis

x[0] = 1.0, x[1] = 0.0, x[2] = 0.0;
y[0] = 0.0, y[1] = 1.0, y[2] = 0.0;
z[0] = 0.0, z[1] = 0.0, z[2] = 1.0;

        quaternion<double>      q_0(1.0,vzero);
        quaternion<double>      q_1(1.0,x);
        quaternion<double>      q_2(1.0,y);
        quaternion<double>      q_3(1.0,z);
        quaternion<double>      q_4((double)3.0 * q_1);

        quaternion<double>      qx(cos(theta/2.0),sin(theta/2.0)*x);
        quaternion<double>      qy(cos(theta/2.0),sin(theta/2.0)*y);
        quaternion<double>      qz(cos(theta/2.0),sin(theta/2.0)*z);

        std::vector<double>     P(3,0.0);               // position (x,y,z)
        std::vector<double>     Ph(4,0.0);              // position (x,y,z,w)

 P[0] = 10.0,  P[1] = 0.0,  P[2] = 0.0;
Ph[0] = 10.0, Ph[1] = 0.0, Ph[2] = 0.0;  Ph[3] = 1.0;
```

```
        quaternion<double>      q(cos(theta/2.0),sin(theta/2.0)*z);
        quaternion<double>      p(0.0,P);
//      matrix<double>          M( (!q).R() * q.L() );  // rotation matrix
        matrix<double>          M(q.quattomat());
        quaternion<double>      qm(M);

        quaternion<double>      sq(q_1.slerp(q_2,.5));

  std::cout << " q_0 = " <<  q_0 << std::endl;
  std::cout << " q_0.quattomat() = \n" << q_0.quattomat() << std::endl;
  std::cout << " q_1 = " <<  q_1 << std::endl;
  std::cout << " q_2 = " <<  q_2 << std::endl;
  std::cout << " q_3 = " <<  q_3 << std::endl;
  std::cout << " q_1.slerp(q_2,.5) = " <<  sq << std::endl;
  std::cout << "-q_1 = " << -q_1 << std::endl;
  std::cout << " q_1 * 3.0 = " << q_1 * 3.0 << std::endl;
  std::cout << " 3.0 * q_1 = " << (double)3.0 * q_1 << std::endl;
  std::cout << " q_1 * q_2 = " << q_1 * q_2 << std::endl;
  std::cout << " q_1 * R(q_2) = " << q_1 * q_2.R() << std::endl;
  std::cout << " q_2 * L(q_1) = " << q_2 * q_1.L() << std::endl;
  std::cout << " q_4 = " <<  q_4 << std::endl;
  std::cout << " |q4| = " << q_4.length() << std::endl;
  std::cout << " q_4/|q_4| = " << q_4.norm() << std::endl;
  std::cout << " !q_4 = " << !q_4 << std::endl;
  std::cout << " q_4 * !q_4 = " << q_4 * !q_4 << std::endl;
  std::cout << " q = " <<  q << std::endl;
  std::cout << " q * p * !q = " << q * p * !q << std::endl;
  std::cout << " (!q).R() * q.L() = \n" << (!q).R() * q.L() << std::endl;
  std::cout << " q.L() * (!q).R() = \n" << q.L() * (!q).R() << std::endl;
  std::cout << " M = \n" << M << std::endl;
  std::cout << " Ph * M = " << Ph * M << std::endl;
  std::cout << " qx.quattomat() = \n" << qx.quattomat() << std::endl;
  std::cout << " qy.quattomat() = \n" << qy.quattomat() << std::endl;
  std::cout << " qz.quattomat() = \n" << qz.quattomat() << std::endl;
  std::cout << " qm = " <<  qm << std::endl;

  P = q_1.v();
  std::cout << " P = " <<  P << std::endl;
}

void test_vector()
{
        std::vector<float>      vin(3,0.0);
        std::vector<float>      v_1(3,0.0);
        std::vector<float>      v_2(3,0.0);
        std::vector<float>      v_3(3,0.0);
        std::vector<float>      v_4(4,0.0);
        std::vector<float>      v_1n(3,0.0), v_2n(3,0.0);
        matrix<float>           m_7(3,3);

  v_1[0] = 10.0, v_1[1] =  0.0, v_1[2] =  0.0;
  v_2[0] =  0.0, v_2[1] = 10.0, v_2[2] =  0.0;
  v_3[0] =  0.0, v_3[1] =  0.0, v_3[2] =  0.0;
  v_4[0] =  1.0, v_4[1] =  2.0, v_4[2] =  3.0;  v_4[3] = 4.0;

  std::cout << "v_1 = " << v_1 << std::endl << "v_2 = " << v_2 << std::endl;
  std::cout << "v_1[0] = " << v_1[0] << std::endl;
  std::cout << "v_1[1] = " << v_1[1] << std::endl;
  std::cout << "v_1[2] = " << v_1[2] << std::endl;
  std::cout << "v_1 . v_2 = " << dot(v_1,v_2) << std::endl;
  v_1n = norm(v_1);  v_2n = norm(v_2);
```

```
  std::cout << "v_1/|v_1| = " << v_1n << std::endl;
  std::cout << "v_2/|v_2| = " << v_2n << std::endl;
  std::cout << "v_1/|v_1| = " << norm(v_1) << std::endl;
  std::cout << "v_2/|v_2| = " << norm(v_2) << std::endl;
  std::cout << "v_1 = " << v_1 << std::endl << "v_2 = " << v_2 << std::endl;
  std::cout << "v_1 x v_2 = " << cross(v_1,v_2) << std::endl;
  std::cout << "v_1n = " << v_1n << std::endl;
  std::cout << "v_2n = " << v_2n << std::endl;
  std::cout << "v_1/|v_1| x v_2/|v_2| = " << cross(v_1n,v_2n) << std::endl;
  std::cout << "v_1 . v_2 = " << dot(v_1,v_2) << std::endl;
  std::cout << "v_1 + v_2 = " << v_1 + v_2 << std::endl;
  std::cout << "-v_1 = " << -v_1 << std::endl;
  v_3[0] = -1.; v_3[1] = -1.; v_3[2] = -1.;
  std::cout << "v_3 = " << v_3 << std::endl;
  std::cout << "v_4 = " << v_4 << std::endl;
  std::cout << "v_3 + v_4 = " << v_3 + v_4 << std::endl;
  std::cout << "v_4 + v_3 = " << v_4 + v_3 << std::endl;
  m_7[0][0] = 1; m_7[0][1] = 2; m_7[0][2] = 3;
  m_7[1][0] = 2; m_7[1][1] = 3; m_7[1][2] = 1;
  m_7[2][0] = 3; m_7[2][1] = 2; m_7[2][2] = 1;
  std::cout << "m_7 =" << std::endl << m_7;
  std::cout << "v_3 * m_7 = " << v_3 * m_7 << std::endl;
  v_3 *= m_7;
  std::cout << "v_3 = " << v_3 << std::endl;

  v_1[0] =  1.0, v_1[1] =  1.0, v_1[2] =  0.0;
  v_2[0] =  1.0, v_2[1] =  1.0, v_2[2] =  1.0;
  std::cout << "v_1 = " << v_1 << std::endl << "v_2 = " << v_2 << std::endl;
  std::cout << "norm(v_1 - v_2) = " << norm(v_1 - v_2) << std::endl;
  std::cout << "v_1 = " << v_1 << std::endl << "v_2 = " << v_2 << std::endl;

  std::cout << "4.0 * v_1 = " << (float)4.0 * v_1 << std::endl;

  v_2 = v_2 + (float)4.0 * v_1;
  std::cout << "v_2 += 4.0 * v_1 = " << v_2 << std::endl;

  std::cout << std::endl;

//std::cout << "Enter in vector (format: x [y [z]])" << std::endl;
//cin >> vin;
//std::cout << vin << std::endl;
}

void test_matrix()
{
        matrix<double>  m_1(2,2),m_2(2,2),m_3(2,2);
        matrix<double>  m_4(1,3),m_5(3,3),m_6(1,3);
        matrix<double>  m_7(3,3);
        matrix<double>  *m_8 = new matrix<double>(3,3);
        matrix<double>  *m_9 = new matrix<double>(1,1);

  m_1.identity();  m_2.identity();

  // should be 2x2 identity matrices
  std::cout << "m_1, m_2 = " << std::endl;
  std::cout << m_1 << std::endl << m_2 << std::endl;

  // should be 2x2 0 matrix
  std::cout << "m_1 - m_2 = " << std::endl;
  std::cout << m_1 - m_2 << std::endl;

  // should be 2x2 negative identity matrix
```

```
m_3 = -m_1;
std::cout << "m_3 = -m_1" << std::endl;
std::cout << m_3 << std::endl;

m_3[0][0] = 1; m_3[0][1] = 2;
m_3[1][0] = 3; m_3[1][1] = 4;

m_4[0][0] = 2; m_4[0][1] = 2; m_4[0][2] = 2;

m_5[0][0] = 1; m_5[0][1] = 2; m_5[0][2] = 3;
m_5[1][0] = 2; m_5[1][1] = 3; m_5[1][2] = 1;
m_5[2][0] = 3; m_5[2][1] = 2; m_5[2][2] = 1;

// should be 1x3 matrix
m_6 = m_4 * m_5;
std::cout << "m_6 = m_4 * m_5" << std::endl;
std::cout << m_6 << std::endl;

// should be 1x3 and 3x3 matrices
std::cout << "m_4, m_5 = " << std::endl;
std::cout << m_4 << std::endl << m_5 << std::endl;

// should be new 1x3 matrix
m_4 *= m_5;
std::cout << "m_4 *= m_5" << std::endl;
std::cout << m_4 << std::endl;

// should be 3x3 inverse matrix
m_7 = !m_5;
std::cout << "m_7 = !m_5" << std::endl;
std::cout << m_7 << std::endl;

// should be 3x3 identity matrix
std::cout << "m_5 * m_7" << std::endl;
std::cout << m_5 * m_7 << std::endl;

// should be 3x3 matrix
std::cout << "m_5 =" << std::endl;
std::cout << m_5 << std::endl;

// should be 3x3 matrix
m_5 = m_5 - 2.0;
std::cout << "m_5 = m_5 - 2.0" << std::endl;
std::cout << m_5 << std::endl;

// should be 3x3 matrix
std::cout << "2.0 + m_5" << std::endl;
std::cout << (double)2.0 + m_5 << std::endl;

// should be 3x3 matrix
m_7 = (double)2.0 + m_5;
std::cout << "m_7 = 2.0 + m_5" << std::endl;
std::cout << m_7 << std::endl;

// should be 3x3 matrix
std::cout << "2.0 - m_5" << std::endl;
std::cout << (double)2.0 - m_5 << std::endl;

// should be 3x3 matrix
m_7 = (double)2.0 - m_5;
std::cout << "m_7 = 2.0 - m_5" << std::endl;
std::cout << m_7 << std::endl;
```

```
// should be 3x3 matrix
std::cout << "2.0 * m_5" << std::endl;
std::cout << (double)2.0 * m_5 << std::endl;

// should be 3x3 matrix
m_7 = (double)2.0 * m_5;
std::cout << "m_7 = 2.0 * m_5" << std::endl;
std::cout << m_7 << std::endl;

// should be 3x3 matrix
std::cout << "2.0 / m_5" << std::endl;
std::cout << (double)2.0 / m_5 << std::endl;

// should be 3x3 matrix
m_7 = (double)2.0 / m_5;
std::cout << "m_7 = 2.0 / m_5" << std::endl;
std::cout << m_7 << std::endl;

// should be 3x3 0 matrix
std::cout << "m_8 =" << std::endl;
std::cout << m_8 << std::endl;

delete m_8;

// should be 1x1 0 matrix
std::cout << "m_9 =" << std::endl;
std::cout << m_9 << std::endl;

delete m_9;

matrix<double>        H(1,2),P(2,2),R(1,1);
matrix<double> denom(1,1);

H[0][0] = 1.0; H[0][1] = 0.0;
matrix<double> Ht(2,1);
Ht = H.transpose();

P[0][0] = 1.0/16.0; P[0][1] = 0.0;
P[1][0] = 0.0;      P[1][1] = 1.0/16.0;

R[0][0] = 102.0;

std::cout << "H =" << std::endl;
std::cout << H << std::endl;
std::cout << "Ht =" << std::endl;
std::cout << Ht << std::endl;
std::cout << "P =" << std::endl;
std::cout << P << std::endl;

denom = H * P * Ht + R;

std::cout << "denom =" << std::endl;
std::cout << denom << std::endl;

std::cout << "!denom =" << std::endl;
std::cout << !denom << std::endl;

std::cout << "P * Ht =" << std::endl;
std::cout << P * Ht << std::endl;

std::cout << "P * Ht * !denom =" << std::endl;
```

```
  std::cout << P * Ht * !denom << std::endl;

  std::cout << std::endl;
}
```

```
#include        <iostream>
#include        <iomanip>
#include        <vector>
#include        <cstdlib>
#include        <cstring>
#include        <cmath>

#include        "matrix.h"
#include        "vector.h"

//////////////////////////// operators: copy assignment ////////////////////////
template <typename T>
matrix<T> matrix<T>::operator=(const matrix<T>& rhs)
{
  if(this != &rhs) {
    arr.resize(rhs.rows());
    for(int i=0;i<(int)arr.size();++i) arr[i] = rhs.arr[i];
  }
  return *this;
}

//////////////////////////// operators: unary ///////////////////////////////////
template <typename T>
matrix<T> matrix<T>::operator-(void)
{
        matrix<T>       result(rows(),cols());

  // create 0 matrix, dim(r,c), subtract this matrix from 0 to get -ve matrix

  return (result -= *this);
}

template <typename T>
matrix<T> matrix<T>::operator!(void)
{
        int             n=rows();
        int             *indx = new int [n];
        T               d, *col = new T [n];
        matrix<T>       copy(*this), inverse(rows(),cols());

  if(rows() == cols()) {
    copy.ludcmp(indx,&d);
    for(int j=0; j<n; ++j) {
      for(int i=0; i<n; ++i) col[i] = (T)0.0;
      col[j] = (T)1.0;
      copy.lubksb(indx,col);
      for(int i=0; i<n; ++i) inverse[i][j] = col[i];
    }
  }
  delete [] indx;  delete [] col;
  return inverse;
}

//////////////////////////// operators: (const matrix&, const matrix&) /////////
template <typename T>
int matrix<T>::operator<(const matrix<T>& rhs)
{
  if(this == &rhs)                              // compare _pointers_ !
    return(0);
  else {
    if((rows() != rhs.rows()) || (cols() != rhs.cols()))
      return(0);
```

```
    else {
      for(int i=0; i<rows(); ++i)
        for(int j=0; j<cols(); ++j)
          if((*this)[i][j] >= rhs[i][j])
            return(0);
    }
  }
  return(1);                                    // if we get here, lhs < rhs
}

template <typename T>
int matrix<T>::operator>(const matrix<T>& rhs)
{
  if(this == &rhs)                              // compare _pointers_ !
    return(0);
  else {
    if((rows() != rhs.rows()) || (cols() != rhs.cols()))
      return(0);
    else {
      for(int i=0; i<rows(); ++i)
        for(int j=0; j<cols(); ++j)
          if((*this)[i][j] <= rhs[i][j])
            return(0);
    }
  }
  return(1);                                    // if we get here, lhs > rhs
}

template <typename T>
int matrix<T>::operator==(const matrix<T>& rhs)
{
        float   precision=0.01;

  if(this == &rhs)                              // compare _pointers_ !
    return(1);
  else {
    if((rows() != rhs.rows()) || (cols() != rhs.cols()))
      return(0);
    else {
      for(int i=0; i<rows(); ++i)
        for(int j=0; j<cols(); ++j)
          if(fabs((double)((*this)[i][j] - rhs[i][j])) > precision)
            return(0);
    }
  }
  return(1);                                    // if we get here, lhs == rhs
}

template <typename T>
matrix<T> matrix<T>::operator*=(const matrix<T>& rhs)
{
  return((*this) = (*this) * rhs);
}

template <typename T>
matrix<T> matrix<T>::operator*(const matrix<T>& rhs)
{
        matrix<T>    result(rows(),rhs.cols());

  if(cols() == rhs.rows()) {
    for(int i=0; i<result.rows(); ++i)
      for(int j=0; j<result.cols(); ++j)
```

```
        for(int k=0; k<cols(); ++k)
          result[i][j] += (*this)[i][k] * rhs[k][j];
  }
  return result;
}

template <typename T>
matrix<T> matrix<T>::operator/=(matrix<T>& rhs)
{
  return((*this) = (*this) * !rhs);
}

template <typename T>
matrix<T> matrix<T>::operator/(matrix<T>& rhs)
{
        matrix<T>    result(rows(),rhs.cols());

  return(result = (*this) * !rhs);
}

template <typename T>
matrix<T> matrix<T>::operator+=(const matrix<T>& rhs)
{
        int r = rows() <= rhs.rows() ? rows() : rhs.rows();
        int c = cols() <= rhs.cols() ? cols() : rhs.cols();

  for(int i=0; i<r; ++i)
    for(int j=0; j<c; ++j)
      (*this)[i][j] += rhs[i][j];

  return *this;
}

template <typename T>
matrix<T> matrix<T>::operator+(const matrix<T>& rhs)
{
        matrix<T>    result(*this);

  return(result += rhs);
}

template <typename T>
matrix<T> matrix<T>::operator-=(const matrix<T>& rhs)
{
  for(int i=0; i<rows(); ++i)
    for(int j=0; j<cols(); ++j)
      (*this)[i][j] -= rhs[i][j];

  return *this;
}

template <typename T>
matrix<T> matrix<T>::operator-(const matrix<T>& rhs)
{
        matrix<T>    result(*this);

  return(result -= rhs);
}

template <typename T>
matrix<T> matrix<T>::ludcmp(int *indx,T *d)
{
```

```
// Given a square matrix, this routine replaces it by the LU decomposition
// of a rowwise permutation of itself.  indx is an output vector that
// records the row permutation effected by the partial pivoting; d is
// ouput as 1 or -1 depending on whether the number of row interchanges
// was even or odd, respectively.  This routine is meant to be used in
// combination with lubksb to solve linear equations or invert a matrix.
//
// From: Press, William H., Teukolsky, Saul A., Vetterling, William T., and
//       Flannery, Brian P., ''Numerical Recipes in C:  The Art of Scientific
//       Computing'', Cambridge University Press, (Cambridge:1992), 2nd ed.

      int     i,j,k;
      int     n = rows();     // r must == c
      int     imax=0;
      T       big,dum,sum,temp;
                              // v stores the implicit scaling of each row
      T       *v = new T [n];

*d = (T)1.0;                  // no row interchanges yet
for(i=0; i<n; i++) {          // loop over rows to get the impl. scaling info
  big = (T)0.0;
  for(j=0; j<n; j++)
    if((temp = (T)fabs((double)((*this)[i][j]))) > big) big = temp;
  if(big == 0.0) {
    std::cerr << "ludcmp: Warning! singular matrix!" << std::endl;
    return(*this);
  }
  // no nonzero largest element
  v[i] = (T)(1.0/big);                // save the scaling
}
for(j=0; j<n; j++) {          // this is the loop over col's of Crout's method
  for(i=0; i<j; i++) {        // this is equation (2.3.12) except for i = j
    sum = (*this)[i][j];
    for(k=0; k<i; k++) sum -= (*this)[i][k] * (*this)[k][j];
    (*this)[i][j] = sum;
  }
  big = (T)0.0;               // init for search for largest pivot element
  for(i=j; i<n; i++) {        // this is i=j of equation (2.3.12) and
    sum = (*this)[i][j];      // i=j+1..n of equation (2.3.13)
    for(k=0; k<j; k++) sum -= (*this)[i][k] * (*this)[k][j];
    (*this)[i][j] = sum;
    if((dum = (T)(v[i]*fabs((double)sum))) >= big) {
      // is the figure of merit for the pivot better than the best so far?
      big = dum;
      imax = i;
    }
  }
  if(j != imax) {             // do we need to interchange rows?
    for(k=0; k<n; k++) {      // yes, do so...
      dum = (*this)[imax][k];
      (*this)[imax][k] = (*this)[j][k];
      (*this)[j][k] = dum;
    }
    *d = -(*d);               // ...and change the parity of d
    v[imax] = v[j];           // also interchange the scale factor
  }
  indx[j] = imax;
  if((*this)[j][j] == 0.0) {
    std::cerr << "ludcmp: Warning! possible singular matrix!" << std::endl;
    (*this)[j][j] = (T)TINY;
  }
  // if the pivot element is zero the matrix is singular (at least to the
```

```
    // precision of the algorithm).  for some applications on singular
    // matrices, it is desirable to substitute TINY for zero
    if(j != n-1) {              // now, finally, divide by the pivot element
      dum = (T)(1.0/((*this)[j][j]));
      for(i=j+1; i<n; i++) (*this)[i][j] *= dum;
    }
  }                             // go back for the next column in the reduction
  delete [] v;

  return *this;
}

template <typename T>
matrix<T> matrix<T>::lubksb(int *indx,T *b)
{
  // Solves the set of n linear equations A . X = B.  Here (*this) square
  // matrix is assumed to be its LU decomposition, determined by the member
  // ludcmp.  indx (of dimension n, which is equal to either of r or c since
  // r == c), is input as the permutation vector returned by ludcmp. b (also
  // of dimension n) is input as the right-hand side vector B, and returns
  // with the solution vector X.  The (*this) matrix, n, and indx are not
  // modified by this member and can be left in place for successive calls
  // with different right-hand sides b.  This routine takes into account
  // the possibility that b will begin with many zero elements, so it is
  // efficient for use in matrix inversion.
  //
  // From: Press, William H., Teukolsky, Saul A., Vetterling, William T., and
  //       Flannery, Brian P., ‘‘Numerical Recipes in C:  The Art of Scientific
  //       Computing’’, Cambridge University Press, (Cambridge:1992), 2nd ed.

        int     n = rows();           // rows() must == cols()
        int     i,j,ii=-1,ip;
        T       sum;

  for(i=0; i<n; i++) {          // when ii is set to a positive value, it will
    ip = indx[i];               //   become the index of the first non-
    sum = b[ip];                //   vanishing element of b.  we now do the
    b[ip] = b[i];               //   forward substitution, equation (2.3.6).
                                //   the only new wrinkle is to unsramble the
                                //   permutation as we go
    if(ii >= 0)
      for(j=ii; j<=i-1; j++) sum -= (*this)[i][j] * b[j];
    else if(sum)                // a nonzero element was encountered, so from
      ii = i;                   //   now on we will have to do the sums in
    b[i] = sum;                 //   the loop above
  }
  for(i=n-1; i>=0; i--) {       // now do the backsubstitution, eq. (2.3.7)
    sum = b[i];
    for(j=i+1; j<n; j++) sum -= (*this)[i][j] * b[j];
    b[i] =                      // store a component of the solution vector X
      sum/(*this)[i][i];
  }                             // all done!

  return *this;
}

////////////////////////// operators: (const matrix&, scalar) ////////////////
template <typename T>
matrix<T> matrix<T>::operator*=(const T& rhs)
{
  for(int i=0; i<rows(); ++i)
    for(int j=0; j<cols(); ++j)
```

```
      (*this)[i][j] *= rhs;

  return *this;
}

template <typename T>
matrix<T> matrix<T>::operator*(const T& rhs)
{
        matrix<T>    result(*this);

  return(result *= rhs);
}

template <typename T>
matrix<T> matrix<T>::operator/=(const T& rhs)
{
  for(int i=0; i<rows(); i++)
    for(int j=0; j<cols(); j++)
      (*this)[i][j] /= rhs;

  return *this;
}

template <typename T>
matrix<T> matrix<T>::operator/(const T& rhs)
{
        matrix<T>    result(*this);

  return(result /= rhs);
}

template <typename T>
matrix<T> matrix<T>::operator+=(const T& rhs)
{
  for(int i=0; i<rows(); i++)
    for(int j=0; j<cols(); j++)
      (*this)[i][j] += rhs;

  return *this;
}

template <typename T>
matrix<T> matrix<T>::operator+(const T& rhs)
{
        matrix<T>    result(*this);

  return(result += rhs);
}

template <typename T>
matrix<T> matrix<T>::operator-=(const T& rhs)
{
  for(int i=0; i<rows(); i++)
    for(int j=0; j<cols(); j++)
      (*this)[i][j] -= rhs;

  return *this;
}

template <typename T>
matrix<T> matrix<T>::operator-(const T& rhs)
{
```

```
        matrix<T>    result(*this);

  return(result -= rhs);
}

////////////////////////// friend operators: (const scalar&, const matrix&) ///
template <typename T>
matrix<T> operator*(const T& lhs,const matrix<T>& rhs)
{
        matrix<T>       result(rhs);

  return(result *= lhs);
}

template <typename T>
matrix<T> operator/(const T& lhs,const matrix<T>& rhs)
{
        double          prec = 0.001;
        matrix<T>       result(rhs);

  // tensor product?
  // shouldn't this be lhs * !rhs?
  // this routine calculates (scalar / matrix), giving
  // [ scalar/element scalar/element ... ]
  // [ scalar/element scalar/element ... ]
  // [ ...                           ... ]

  for(int i=0; i<result.rows(); ++i)
    for(int j=0; j<result.cols(); ++j)
      result[i][j] =
        // check for div by zero
        (T)(fabs(result[i][j] - prec) <= prec ? HUGE : lhs/result[i][j]);

  return(result);
}

template <typename T>
matrix<T> operator+(const T& lhs,const matrix<T>& rhs)
{
        matrix<T>    result(rhs);

  return(result += lhs);
}

template <typename T>
matrix<T> operator-(const T& lhs,const matrix<T>& rhs)
{
        matrix<T>    result(rhs);

  result = -result;
  return(result += lhs);
}

////////////////////////// friend operators: (const vector<T>&, const matrix&)
template <typename T>
std::vector<T> operator*=(std::vector<T>& lhs,const matrix<T>& rhs)
{
  return(lhs = lhs * rhs);
}

template <typename T>
std::vector<T> operator*(const std::vector<T>& lhs,const matrix<T>& rhs)
```

```
{
       std::vector<T>  result((int)rhs.cols(),(T)0.0);

  if((int)lhs.size() == rhs.rows()) {
    for(int j=0; j<(int)result.size(); j++)
      for(int k=0; k<rhs.cols(); k++)
        result[j] += lhs[k] * rhs[k][j];
  }
  return result;
}

////////////////////////// friends ////////////////////////////////////////
template <typename T>
std::ostream& operator<<(std::ostream& s,const matrix<T>& rhs)
{
  s.setf(std::ios::fixed,std::ios::floatfield); s.precision(2);
  for(int i=0; i<rhs.rows(); ++i) {
    s << "[";
    for(int j=0; j<rhs.cols(); ++j) {
      s.width(6);
      j == rhs.cols()-1 ? s << rhs[i][j] : s << rhs[i][j] << ", ";
    }
    s << "]" << std::endl;
  }
  return s;
}

////////////////////////// public member functions ////////////////////////////
template <typename T>
void matrix<T>::zero(void)
{
  for(int i=0; i<rows(); ++i)
    for(int j=0; j<cols(); ++j)
      (*this)[i][j] = static_cast<T>(0.0);
}

template <typename T>
void matrix<T>::one(void)
{
  for(int i=0; i<rows(); ++i)
    for(int j=0; j<cols(); ++j)
      (*this)[i][j] = static_cast<T>(1.0);
}

template <typename T>
void matrix<T>::identity(void)
{
  for(int i=0; i<rows(); ++i) {
    for(int j=0; j<cols(); ++j) (*this)[i][j]=(T)0.0;
    (*this)[i][i]=(T)1.0;
  }
}

template <typename T>
matrix<T> matrix<T>::clamp(T lo,T hi)
{
  for(int i=0; i<rows(); ++i) {
    for(int j=0; j<cols(); ++j) {
      if( (*this)[i][j] < lo ) (*this)[i][j] = lo;
      if( (*this)[i][j] > hi ) (*this)[i][j] = hi;
    }
  }
```

```
  return *this;
}

template <typename T>
matrix<T> matrix<T>::transpose(void)
{
        matrix<T>    result(cols(),rows());

  for(int i=0;i<cols();++i)
    for(int j=0;j<rows();++j)
      result[i][j] = (*this)[j][i];

  return(result);
}

template <typename T>
T matrix<T>::trace(void) const
{
        T   sum=(T)0.0;

  for(int i=0; i<rows(); ++i)
    for(int j=0; j<cols(); ++j)
      if(i==j) sum += (*this)[i][j];

  return sum;
}

//////////////////////////// specializations ////////////////////////////////////
template class matrix<int>;
template std::ostream& operator<<(std::ostream&, const matrix<int>&);
template matrix<int> operator*(const int&, const matrix<int>&);
template matrix<int> operator/(const int&, const matrix<int>&);
template matrix<int> operator+(const int&, const matrix<int>&);
template matrix<int> operator-(const int&, const matrix<int>&);
template std::vector<int> operator*(const std::vector<int>&, const matrix<int>&);
template std::vector<int> operator*=(std::vector<int>&, const matrix<int>&);

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

template class matrix<double>;
template std::ostream& operator<<(std::ostream&, const matrix<double>&);
template matrix<double> operator*(const double&, const matrix<double>&);
template matrix<double> operator/(const double&, const matrix<double>&);
template matrix<double> operator+(const double&, const matrix<double>&);
template matrix<double> operator-(const double&, const matrix<double>&);
template std::vector<double> operator*(const std::vector<double>&, const matrix<double>
&);
template std::vector<double> operator*=(std::vector<double>&, const matrix<double>&);
```

```
#include        <iostream>
#include        <iomanip>
#include        <vector>
#include        <cmath>

#include        "vector.h"
#include        "matrix.h"
#include        "quaternion.h"

//////////////////////////// operators: unary ////////////////////////////////
template <typename T>
quaternion<T> quaternion<T>::operator-(void)
{
        quaternion<T>    result((T)0.0);

  // create 0 quaternion, subtract this quaternion from 0 to get -ve quaternion

  return(result -= this);
}

template <typename T>
quaternion<T> quaternion<T>::operator!(void)
{
        T               one_over_mag = static_cast<T>(1.0)/dot(*this);
        quaternion<T>   result((T)0.0);

  // let q = (*this) = (s,v), then quaternion inverse is
  //   q^{-1} = \frac{1}{|q|^2} * (s,-v)
  //   where |q|^2 = s^2 + (v . v), e.g., the quaternion's length squared
  //                                (or its dot product, e.g., dot(this))
  //
  result.q[3] = one_over_mag *  q[3];
  for(int i=0; i<3; ++i) result.q[i] = one_over_mag * -q[i];

  return(result);
}

//////////////////////////// operators: (const quaternion&, const quaternion&) //
template <typename T>
int quaternion<T>::operator==(const quaternion<T>& rhs)
{
        float   precision=0.01;

  if(this == &rhs) return 1;
  else {
    if(fabs(q[3] - rhs.q[3]) > precision)
      return 0;
  }
  return(q == rhs.q); // if we get here, lhs.q[3] == rhs.q[3]
}

template <typename T>
int quaternion<T>::operator<(const quaternion<T>& rhs)
{
  if(this == &rhs) return 0;
  else {
    if(q[3] >= rhs.q[3])
      return 0;
  }

  return(q < rhs.q); // if we get here, lhs.q[3] < rhs.q[3]
}
```

```
template <typename T>
int quaternion<T>::operator>(const quaternion<T>& rhs)
{
  if(this == &rhs) return 0;
  else {
    if(q[3] <= rhs.q[3])
      return 0;
  }

  return(q > rhs.q); // if we get here, lhs.q[3] > rhs.q[3]
}

template <typename T>
quaternion<T> quaternion<T>::operator=(const quaternion<T>& rhs)
{
  if(this != &rhs) q = rhs.q;

  return *this;
}

template <typename T>
quaternion<T> quaternion<T>::operator*=(const quaternion<T>& rhs)
{
  return((*this) = (*this) * rhs);
}

template <typename T>
quaternion<T> quaternion<T>::operator*(const quaternion<T>& rhs)
{
        quaternion<T>   result((T)0.0);
        std::vector<T>  v1(3,(T)0.0), v2(3,(T)0.0), vc(3,(T)0.0);

  // let q_1 = (*this) = (s_1,v_1), q_2 = rhs = (s_2,v_2), then
  //   q_1 q_2 = (s_1 s_2 - v_1.dot(v_2), s_1 v_2 + s_2 v_1 + v_1.cross(v_2))
  //             (    <scalar>          ,          <vector>                  )
  //
  // Note: should also be able to obtain this using the L() and R() routines,
  //       e.g., (*result) = rhs * L(); or (*result) = (*this) * rhs.R();
  //       ONLY IFF class quaternion were stored as a 1x4 vector, [x y z s]
  //

  for(int i=0; i<3; ++i) v1[i] = q[i];
  for(int i=0; i<3; ++i) v2[i] = rhs.q[i];

  result.q[3] = q[3] * rhs.q[3] - ::dot(v1,v2);

  vc = cross(v1,v2);

  for(int i=0; i<3; ++i) v1[i] *= rhs.q[3];
  for(int i=0; i<3; ++i) v2[i] *=     q[3];

  for(int i=0; i<3; ++i) result.q[i] = v2[i] + v1[i] + vc[i];

  return result;
}

template <typename T>
quaternion<T> quaternion<T>::operator*(const matrix<T>& rhs)
{
        quaternion<T>   quat(q * rhs);
```

```
  return quat;
}

template <typename T>
quaternion<T> quaternion<T>::operator/=(quaternion<T>& rhs)
{
  return((*this) = (*this) * !rhs);
}

template <typename T>
quaternion<T> quaternion<T>::operator/(quaternion<T>& rhs)
{
        quaternion<T>   result((T)0.0);

  return(result = (*this) * !rhs);
}

template <typename T>
quaternion<T> quaternion<T>::operator+=(const quaternion<T>& rhs)
{
  for(int i=0; i<4; ++i) q[i] += rhs.q[i];

  return *this;
}

template <typename T>
quaternion<T> quaternion<T>::operator+(const quaternion<T>& rhs)
{
        quaternion<T>   result(this);

  return(result += rhs);
}

template <typename T>
quaternion<T> quaternion<T>::operator-=(const quaternion<T>& rhs)
{
  for(int i=0; i<4; ++i) q[i] -= rhs.q[i];

  return *this;
}

template <typename T>
quaternion<T> quaternion<T>::operator-(const quaternion<T>& rhs)
{
        quaternion<T>   result(this);

  return(result -= rhs);
}

/////////////////////////// operators: (const quaternion&, scalar) /////////////
template <typename T>
quaternion<T> quaternion<T>::operator*=(const T& rhs)
{
  for(int i=0; i<4; ++i) q[i] *= rhs;

  return *this;
}

template <typename T>
quaternion<T> quaternion<T>::operator*(const T& rhs)
{
        quaternion<T>   result(this);
```

```
  return(result *= rhs);
}

template <typename T>
quaternion<T> quaternion<T>::operator/=(const T& rhs)
{
  for(int i=0; i<4; ++i) q[i] /= rhs;

  return *this;
}

template <typename T>
quaternion<T> quaternion<T>::operator/(const T& rhs)
{
        quaternion<T>   result(this);

  return(result /= rhs);
}

////////////////////////// friend operators: (const scalar&, cosnt quaternion&)
template <typename T>
quaternion<T> operator*(T lhs,quaternion<T>& rhs)
{
        quaternion<T>    result(rhs);

  return(result *= lhs);
}

template <typename T>
quaternion<T> operator/(T lhs,quaternion<T>& rhs)
{
        quaternion<T>    result(!rhs);

  return(result *= lhs);
}

////////////////////////// friends /////////////////////////////////////////
template <typename T>
std::ostream& operator<<(std::ostream& s,const quaternion<T>& rhs)
{
  s.setf(std::ios::fixed,std::ios::floatfield);
  s.precision(2);
  s.width(6);
  s << "(" << rhs.q[3] << ", " <<
       "<" << rhs.q[0] << ", " <<
              rhs.q[1] << ", " <<
              rhs.q[2] << ">)";
  return s;
}

////////////////////////// public member functions //////////////////////////
template <typename T>
T quaternion<T>::dot(quaternion& rhs)
{
        T       dot=(T)0.0;

  for(int i=0; i<4; ++i) dot += q[i] * rhs.q[i];

  return(dot);
}
```

```
template <typename T>
matrix<T> quaternion<T>::L(void)
{
        matrix<T>       L_q(4,4);
        register T      x = q[0], y = q[1], z = q[2], s = q[3];

  // return L_row(q) in row-major order  (transpose result if col-major order)
  //
  L_q[0][0] =  s;  L_q[0][1] =  z;  L_q[0][2] = -y;  L_q[0][3] = -x;
  L_q[1][0] = -z;  L_q[1][1] =  s;  L_q[1][2] =  x;  L_q[1][3] = -y;
  L_q[2][0] =  y;  L_q[2][1] = -x;  L_q[2][2] =  s;  L_q[2][3] = -z;
  L_q[3][0] =  x;  L_q[3][1] =  y;  L_q[3][2] =  z;  L_q[3][3] =  s;

  return L_q;
}

template <typename T>
matrix<T> quaternion<T>::R(void)
{
        matrix<T>       R_q(4,4);
        register T      x = q[0], y = q[1], z = q[2], s = q[3];

  // return R_row(q) in row-major order  (transpose result if col-major order)
  //
  R_q[0][0] =  s;  R_q[0][1] = -z;  R_q[0][2] =  y;  R_q[0][3] = -x;
  R_q[1][0] =  z;  R_q[1][1] =  s;  R_q[1][2] = -x;  R_q[1][3] = -y;
  R_q[2][0] = -y;  R_q[2][1] =  x;  R_q[2][2] =  s;  R_q[2][3] = -z;
  R_q[3][0] =  x;  R_q[3][1] =  y;  R_q[3][2] =  z;  R_q[3][3] =  s;

  return R_q;
}

template <typename T>
matrix<T> quaternion<T>::quattomat(void)
{
        matrix<T>  Q(4,4);
        register T x = q[0], y = q[1], z = q[2], s = q[3];
        register T ss = s*s, sx = (T)2.0*s*x, sy = (T)2.0*s*y, sz = (T)2.0*s*z;
        register T           xx =     x*x,    xy = (T)2.0*x*y, xz = (T)2.0*x*z;
        register T                            yy =     y*y,    yz = (T)2.0*y*z;
        register T                                             zz =     z*z;

  // return (4x4) rot matrix, in row-major order we'd have:
  //   R(q^-1) . L(q) = Q
  // +-             -+ +-             -+
  // | s   z  -y   x | | s   z  -y  -x |
  // |-z   s   x   y | |-z   s   x  -y | =
  // | y  -x   s   z | | y  -x   s  -z |
  // |-x  -y  -z   s | | x   y   z   s |
  // +-             -+ +-             -+
  //
  // +-                                                                  -+
  // | (s^2+x^2-y^2-z^2)     (2xy+2sz)         (2xz-2sy)             0    |
  // |     (2xy-2sz)     (s^2-x^2+y^2-z^2)     (2yz+2sx)             0    |
  // |     (2xz+2sy)         (2yz-2sx)     (s^2-x^2-y^2+z^2)         0    |
  // |         0                 0                 0         (s^2+x^2+y^2+z^2) |
  // +-                                                                  -+
  //

  // both of these work, but the inverse and matrix mult involved are too
  // expensive
  //
```

```
  // return( (!q).R() * q.L() );
  // Q = (!q).R() * q.L();

  Q[0][0]=(T)(ss+xx-yy-zz); Q[0][1]=(T)(xy+sz); Q[0][2]=(T)(xz-sy); Q[0][3]=(T)0.0;
  Q[1][0]=(T)(xy-sz); Q[1][1]=(T)(ss-xx+yy-zz); Q[1][2]=(T)(yz+sx); Q[1][3]=(T)0.0;
  Q[2][0]=(T)(xz+sy); Q[2][1]=(T)(yz-sx); Q[2][2]=(T)(ss-xx-yy+zz); Q[2][3]=(T)0.0;
  Q[3][0]=(T)0.0;     Q[3][1]=(T)0.0;     Q[3][2]=(T)0.0; Q[3][3]=(T)(ss+xx+yy+zz);

  return(Q);
}

template <typename T>
void quaternion<T>::mattoquat(matrix<T>& mat)
{
  // Routine to convert rotation matrix to a quaternion assuming matrix is 4x4
  // and quaternion is of unit magnitue.  The corresponding quaternion rotation
  // matrix (R(q^-1) . L(q)) is given below:
  //
  // +-                                                  -+
  // | (1-2y^2-2z^2)       (2xy+2sz)        (2xz-2sy)     0 |
  // |    (2xy-2sz)     (1-2x^2-2z^2)       (2yz+2sz)     0 |
  // |    (2xz+2sy)        (2yz-2sx)     (1-2x^2-2y^2)    0 |
  // |        0               0                0         1 |
  // +-                                                  -+
  //
  // Note that if the quaternion is of unit magnitude, the diagonal entries
  // can be rewritten from the original (R(q^-1) . L(q)) form given above
  // in routine quattomat (i.e., the (0,0) entry (s^2+x^2-y^2-z^2) =
  // (1 - 2y^2 - 2z^2) iff s^2 + x^2 + y^2 + z^2 = 1).
  //
  // Code modified from:
  //   Watt, Alan and Watt, Mark, ‘‘Advanced Animation and Rendering
  //   Techniques: Theory and Practice’’, Addison-Wesley, Reading, MA, 1992.
  // (p.363)
  //

        T       trace = mat.trace();
        T       s=(T)0.0;
        int     i,j,k,nxt[3] = {1,2,0};

  if( (mat.rows() != 4) || (mat.cols() != 4) ) return;

  if(trace > 0.0) {
    s = (T)sqrt(trace);
    q[3] = s * (T)0.5;

    s *= (T)2.0;
    q[0] = (mat[1][2] - mat[2][1])/s;
    q[1] = (mat[2][0] - mat[0][2])/s;
    q[2] = (mat[0][1] - mat[1][0])/s;
  }
  else {
    i = 0;
    if(mat[1][1] > mat[0][0]) i = 1;
    if(mat[2][2] > mat[i][i]) i = 2;
    j = nxt[i];  k = nxt[j];

    s = (T)sqrt( (mat[i][i] - (mat[j][j] + mat[k][k])) + 1.0 );

    q[i] = s * (T)0.5;          // s is one of x,y,z depending on i

    s *= (T)2.0;
```

```
    q[3] = (mat[j][k] - mat[k][j])/s;
    q[j] = (mat[i][j] + mat[j][i])/s;
    q[k] = (mat[i][k] + mat[k][i])/s;
  }
}

template <typename T>
quaternion<T> quaternion<T>::slerp2(quaternion<T> rhs, T t)
{
  // Routine to calculate spherical linear interpolation (slerp) between
  // two quaternions (this and rhs) given paramter t.
  //
  // Code modified from:
  //   Watt, Alan and Watt, Mark, ''Advanced Animation and Rendering
  //   Techniques: Theory and Practice'', Addison-Wesley, Reading, MA, 1992.
  // (p.364)
  //
        T               omega,sinom,cosom = dot(rhs);
        T               sclp,sclq;
        quaternion<T>   qp(this), qq(rhs);
        quaternion<T>   qt(this);

  if((1.0+cosom) > EPSILON) {
    if((1.0-cosom) > EPSILON) {
      omega = (T)acos((double)cosom);
      sinom = (T)sin((double)omega);
      sclp = (T)sin((1.0-t)*omega)/sinom;
      sclq = (T)sin((    t)*omega)/sinom;
    }
    else {
      // if cos(omega) is very small, don't bother calculating sin(omega)
      // since sin(t) -> 0 as t -> 0.
      sclp = (T)1.0-t;
      sclq =     t;
    }
    qt = (sclp * qp) + (sclq * qq);
  }
  else {
    // Not quite sure what this piece of code does---see Watt & Watt.
    // I suspect this is the part that figures out whether we need to
    // calculate slerp(p,q,t) or slerp(p,-q,t), but it looks like it's
    // calculating slerp(p,-p,t) instead.  Possible bug in the book?
    // Since I'm a little leary about this, I added a type of API to slerp
    // which checks to see which arc we're traversing (see slerp()) based
    // on the distance of (p-q).(p-q) vs. (p+q).(p+q).  Due to that API routine
    // we may never actually get in here...
    //
    qq.q[0] = -qp.q[1];
    qq.q[1] =  qp.q[0];
    qq.q[2] = -qp.q[3];

    sclp = (T)sin((1.0-t)*M_PI_2);
    sclq = (T)sin((    t)*M_PI_2);
    qt = (sclp * qp) + (sclq * qq);

    qt.q[3]   =  qp.q[2];
  }

  return(qt);
}

template <typename T>
```

```
quaternion<T> quaternion<T>::slerp(quaternion<T> rhs, T t)
{
  // Routine to calculate spherical linear interpolation (slerp) between
  // two quaternions (this and rhs) given paramter t.  This is sort of an
  // API to slerp2 which does most of the work.  Here, I put in the test
  // to see whether we need to negate q (rhs) or not as per the text:
  //   Watt, Alan and Watt, Mark, ''Advanced Animation and Rendering
  //   Techniques: Theory and Practice'', Addison-Wesley, Reading, MA, 1992.
  // (p.365)
  //
        quaternion<T>   qp(this),qq(rhs);
        quaternion<T>   p_pos_q_diff(qp - qq),p_neg_q_diff(qp + qq);

  if(p_pos_q_diff.sqlength() < p_neg_q_diff.sqlength())
    return(slerp2( rhs,t));
  else
    return(slerp2(-rhs,t));
}

////////////////////////// protected helper functions (for constructors) //////
template <typename T>
void quaternion<T>::trackball(T x1,T y1,T x2,T y2)
{
  // Routine to simulate a trackball, modified from code originally written by
  // Gavin Bell for Silicon Graphics, November 1988.  His (presumably) comments
  // follow.  (My comments appear in <>'s.)
  //
  // Ok, simulate a track-ball.  Project the points onto the virtual trackball,
  // then figure out the axis of rotation, which is the cross product of P1 P2
  // and O P1 (O is the center of the ball, 0,0,0).
  //
  // Note: This is a deformed trackball--is a trackball in the center, but is
  // deformed into a hyperbolic sheet of rotation away from the center.  This
  // particular function was chosen after trying out several variations.
  //
  // It is assumed that the arguments to this routine are in the range
  // (-1.0 ... 1.0)
  //

        T               rangle;
        std::vector<T>  p1(3,(T)0.0),p2(3,(T)0.0);
        std::vector<T>  p1n(3,(T)0.0),p2n(3,(T)0.0);
        std::vector<T>  axis(3,(T)0.0);

  if( (x1 == x2) && (y1 == y2) ){
    // zero rotation
    //
    // <note: this routine is intended as part of a constructor, so we really
    // don't need this stuff below, since vector and scalar have already been
    // initialized to these values---I left the lines in though just in case
    // I ever decide to make this a publicly accessible routine.>
    //
    q[0] = (T)0.0;
    q[1] = (T)0.0;
    q[2] = (T)0.0;
    q[3] = (T)1.0;
    return;
  }

  // First, figure out z-coordinates for projection of P1 and P2 to deformed
  // sphere.
  p1[0] = x1;
```

```
  p1[1] = y1;
  p1[2] = project_to_sphere((T)TRACKBALLSIZE,x1,y1);
  p2[0] = x2;
  p2[1] = y2;
  p2[2] = project_to_sphere((T)TRACKBALLSIZE,x2,y2);

  // Now, we want the cross product of P1 and P2.
  axis = cross(p2,p1);
  axis = ::norm(axis);

  // Figure out how much to rotate around that axis.
  p1n = ::norm(p1);
  p2n = ::norm(p2);
  rangle = (T)acos(::dot(p1n,p2n));

  // <Set this quaternion's s and v values.>
  q[3] = (T)cos(rangle/2.0);
  q = ::norm(axis); // normalize axis of rotation (e.g., use unit vector)
  for(int i=0; i<3; ++i) q[i] *= (T)sin(rangle/2.0);
}

template <typename T>
T quaternion<T>::project_to_sphere(T r,T x,T y)
{
  // Helper routine for trackball simulation, modified from code originally
  // written by // Gavin Bell for Silicon Graphics, November 1988.  His
  // (presumably) comments follow.  (My comments, if any, appear in <>'s.)
  //
  // Project an x,y pair onto a sphere of radius r OR a hyperbolic sheet if
  // we are away from the center of the sphere.
  //
        T       d,t,z;

  // as per book
  d = (T)sqrt(1.0 - x*x + y*y);
  return d;

  // as per Gavin
  d = (T)sqrt(x*x + y*y);
  if(d < r * ROOT2OVER2)        // inside sphere
    z = (T)sqrt(r*r - d*d);
  else {                        // on hyperbola
    t = (T)(r / ROOT2);
    z = (T)(t*t / d);
  }
  return z;
}

////////////////////////// specializations ///////////////////////////////////
template class quaternion<int>;
template std::ostream& operator<<(std::ostream& s, const quaternion<int>& rhs);
template quaternion<int> operator*(int, quaternion<int>&);
template quaternion<int> operator/(int, quaternion<int>&);

template class quaternion<float>;
template std::ostream& operator<<(std::ostream& s, const quaternion<float>& rhs);
template quaternion<float> operator*(float, quaternion<float>&);
template quaternion<float> operator/(float, quaternion<float>&);

template class quaternion<double>;
template std::ostream& operator<<(std::ostream&, const quaternion<double>&);
template quaternion<double> operator*(double, quaternion<double>&);
```

```
template quaternion<double> operator/(double, quaternion<double>&);
```

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

template <typename T>
std::ostream& operator<<(std::ostream& s,const std::vector<T>& rhs)
{
  s.setf(std::ios::fixed,std::ios::floatfield);
  s.precision(1);
  s.width(3);
  s << "<";
  for(int i=0;i < (int)rhs.size(); i++)
    i == (int)rhs.size()-1 ? s << rhs[i] : s << rhs[i] << ", ";
  s << ">";

  return s;
}

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

  for(int i=0; i<(int)result.size(); ++i) result[i] = -rhs[i];

  return(result);
}

template <typename T>
std::vector<T> operator+(const std::vector<T>& lhs,const std::vector<T>& rhs)
{
        int             dim = lhs.size() < rhs.size() ? lhs.size() : rhs.size();
        std::vector<T>  result(dim,(T)0.0);

  for(int i=0; i<(int)result.size(); ++i) result[i] = lhs[i] + rhs[i];

  return(result);
}

template <typename T>
std::vector<T> operator-(const std::vector<T>& lhs,const std::vector<T>& rhs)
{
        int             dim = lhs.size() < rhs.size() ? lhs.size() : rhs.size();
        std::vector<T>  result(dim,(T)0.0);

  for(int i=0; i<(int)result.size(); ++i) result[i] = lhs[i] - rhs[i];

  return(result);
}

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

  for(int i=0; i<(int)result.size(); ++i) result[i] *= lhs;

  return(result);
```

```
}

template <typename T>
T dot(const std::vector<T>& lhs, const std::vector<T>& rhs)
{
        int     dim = lhs.size() < rhs.size() ? lhs.size() : rhs.size();
        T       dot=(T)0.0;

  for(int i=0; i<dim; ++i) dot += lhs[i] * rhs[i];

  return dot;
}

template <typename T>
std::vector<T> cross(const std::vector<T>& lhs, const std::vector<T>& rhs)
{
        int             c;
        std::vector<T>  result;

  if(lhs.size() <= rhs.size())          result = rhs;
  else                                  result = lhs;

  c = (int)result.size();

  // result = this x rhs
  for(int j=0;j<c;j++)
    result[j] = \
      rhs[((c+j-1)%c)] * lhs[((c+j+1)%c)] - \
      lhs[((c+j-1)%c)] * rhs[((c+j+1)%c)];

  /* result = rhs x this
  for(int j=0;j<c;j++)
     result[j] = \
       lhs[((c+j-1)%c)] * rhs[((c+j+1)%c)] - \
       rhs[((c+j-1)%c)] * lhs[((c+j+1)%c)];
  */

  return result;
}

template <typename T>
std::vector<T> norm(const std::vector<T>& v)
{
        std::vector<T>  result(v);
        T               length = (T)sqrt(dot(v,v));

  for(int i=0; i<(int)result.size(); ++i) result[i] /= length;

  return result;
}

////////////////////   <int> specializations //////////////////////////////
template
std::ostream& operator<<(std::ostream&, const std::vector<int>& rhs);
template
std::vector<int> operator-(const std::vector<int>&);
template
std::vector<int> operator*(const int&, const std::vector<int>&);
template
std::vector<int> operator+(const std::vector<int>&, const std::vector<int>&);
template
std::vector<int> operator-(const std::vector<int>&, const std::vector<int>&);
```

```
template int dot(const std::vector<int>&, const std::vector<int>&);
template
std::vector<int> cross(const std::vector<int>&, const std::vector<int>&);
template
std::vector<int> norm(const std::vector<int>&);

//////////////////// <float> specializations ////////////////////////////////
template
std::ostream& operator<<(std::ostream&, const std::vector<float>& rhs);
template
std::vector<float> operator-(const std::vector<float>&);
template
std::vector<float> operator*(const float&, const std::vector<float>&);
template
std::vector<float> operator+(const std::vector<float>&, const std::vector<float>&);
template
std::vector<float> operator-(const std::vector<float>&, const std::vector<float>&);
template float dot(const std::vector<float>&, const std::vector<float>&);
template
std::vector<float> cross(const std::vector<float>&, const std::vector<float>&);
template
std::vector<float> norm(const std::vector<float>&);

//////////////////// <double> specializations ////////////////////////////////
template
std::ostream& operator<<(std::ostream&, const std::vector<double>& rhs);
template
std::vector<double> operator-(const std::vector<double>&);
template
std::vector<double> operator*(const double&, const std::vector<double>&);
template
std::vector<double> operator+(const std::vector<double>&, const std::vector<double>&);
template
std::vector<double> operator-(const std::vector<double>&, const std::vector<double>&);
template
double dot(const std::vector<double>&, const std::vector<double>&);
template
std::vector<double> cross(const std::vector<double>&, const std::vector<double>&);
template
std::vector<double> norm(const std::vector<double>&);
```