Introduction to the Eigen Library

Wyatt Newman

October, 2105

ROS messages require serialization for network communication. As a result, these messages can be inconvenient to work with when one desires to perform operations on data. One common need is to perform linear-algebra operations. A useful C++ library for linear algebra is the "Eigen" library (see http://eigen.tuxfamily.org). The Eigen open-source project is independent of ROS. However, one can still use Eigen with ROS.

To use Eigen with ROS, one must make changes in the usual three places: package.xml, CmakeLists.txt, and the header files in your .cpp source code.

In the package.xml, one must add the following line:

```
<build_depend>cmake_modules</build_depend>
```

(see: https://github.com/ros/cmake_modules/blob/0.3-devel/README.md#usage).

In the CmakeLists.txt file, if using catkin_simple, add the following:

```
find_package(cmake_modules REQUIRED)
find_package(Eigen3 REQUIRED)

include_directories(${EIGEN3_INCLUDE_DIR})
add_definitions(${EIGEN_DEFINITIONS})
```

In your C++ source code, add appropriate headers. In our first example, we will use classes that require the following header files:

```
#include <Eigen/Eigen>
#include <Eigen/Dense>
#include <Eigen/Geometry>
#include <Eigen/Eigenvalues>
```

An illustrative program: fitting points to a plane: In the package "example_eigen_plane_fit", a source file "example_eigen_plane_fit.cpp" illustrates a variety of Eigen capabilities. Eigen objects have member functions and member data. Operators are defined for these functions that perform the equivalent algebraic computations that you would expect.

An example vector may be defined e.g. as follows:

```
Eigen::Vector3d normal_vec(1,2,3);
```

This instantiates an Eigen object that is a column vector comprised of 3 double-precision values. The object is named "normal_vec" and it is initialized to the values [1;2;3].

One of the member functions is "norm()", which computes the Euclidean length of the vector (square root of the sum of squares of the components). The vector can be coerced to unit length (if it is a non-zero vector!) as follows:

```
    normal_vec/=normal_vec.norm(); // make this vector unit length
```

Thus, vector*scalar operations behave as expected (where normal_vec.norm() returns a scalar value).

Here is an example of instantiating a 3x3 matrix object comprised of double-precision values:

```
Eigen::Matrix3d Rot_z;
```

With the following notation, one can fill the matrix with data, one row at a time:

```
Rot_z.row(0)<<0,1,0;  // populate the first row--shorthand method
Rot_z.row(1)<<1,0,0;  //second row
Rot_z.row(2)<<0,0,1;  // yada, yada
```

There are a variety of other methods for initializing or populating matrices and vectors. For example, one can fill a vector or matrix with zeros using:

```
Eigen::Vector3d centroid;
 // here's a handy way to initialize data to all zeros; more variants exist
centroid = Eigen::MatrixXd::Zero(3,1); // http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt
```

The arguments (3,1) specify 3 rows, 1 column. (A vector is simply a special case of a matrix, for which there is either a single row or a single column).

Alternatively, to initialize to values of 1:

```
Eigen::VectorXd  ones_vec= Eigen::MatrixXd::Ones(npts,1);
```

In the illustrative example, we generate a set of points that lie on (or near) a pre-determined plane. We then invoke Eigen methods to try to discover what was the original plane, using only the data points. This operation is valuable in point-cloud processing, where we may wish to find flat surfaces of interest, e.g. tables, walls, floors, doors, etc.

In the example code, the plane of interest is defined to have a surface normal called “normal_vec”, and the plane is offset from the origin by a distance “dist.” A plane has a unique definition of distance from the origin. If one cares about positive vs negative surfaces of a plane, then the distance from the origin can be a signed number, where the offset is measured from origin to plane in the direction of the plane's normal (and thus can result in a negative offset).

To generate some sample data, we construct a pair of vectors perpendicular to the plane's normal vector. We can do so starting with some vector v1 that is *not* colinear with the plane normal. In the example code, this vector is generated by rotating normal_vec 90 degrees about the z axis, which is accomplished with the following matrix*vector multiply:

```
v1 = Rot_z*normal_vec; //here is how to multiply a matrix times a vector
```

(note: if normal_vec is parallel to the z axis, v1 will be equal to normal_vec, and the subsequent operations will not work).

For display purposes, Eigen types are nicely formatted for “cout”. Matrices are nicely formatted with new lines for each row, e.g. using:

```
cout<<Rot_z<<endl;
```

Rather than use “cout”, it is preferable to use “ROS_INFO_STREAM()”, which is more versatile than “ROS_INFO()”. This function outputs the data via network communication, and it is thus visible via rqt_console and is loggable. To display Rot_z as a formatted matrix with ROS_INFO_STREAM(), one can use:

```
ROS_INFO_STREAM(endl<<Rot_z); // start w/ endl, so get a clean first line of data display
```

For short column vectors, it is more convenient to display the values on a single line. To do so, one can output the transpose of the vector, e.g. as follows:

```
ROS_INFO_STREAM("v1: "<<v1.transpose()<<endl);
```

Two common vector operations are the dot product and the cross product. From the example code, here are some excerpts:

```
double dotprod = v1.dot(normal_vec); //using the "dot()" member function
double dotprod2 = v1.transpose()*normal_vec;// alt: turn v1 into a row vector,
                                                    // then multiply times normal_vec
```

and for the cross product, v1 crossed into normal_vec:

```
v2 = v1.cross(normal_vec);
```

Note that the result of v1 x normal_vec must be mutually orthogonal. Since the result, v2, is perpendicular to normal_vec, it qualifies as one of our desired vectors parallel to our defined plane. A second vector in the plane can be computed as:

```
v1 = v2.cross(normal_vec);  // re-use v1; make it the cross product of v2 into normal_vec
```

Using our vectors v1, v2 and normal_vec, we can define any point in our desired plane as:

```
p = a*v1 + b*v2 + c*normal_vec
```

where we must constrain c=dist, but a and b can be any values.

We generate random points in our plane and store them as column vectors in a 3xN matrix. The matrix is instantiated with the line:

```
Eigen::MatrixXd points_mat(3,npts); //create a matrix, double-precision values, 3 rows and npts cols
```

To generate values for “a” and “b” in our point-generation equation, we can use:

```
Eigen::Vector2d rand_vec; //a 2x1 vector of doubles
```

A second illustrative program; coordinate transforms in Eigen: The second illustrative program, “example_eigen_xforms.cpp”, shows use of the Eigen class “Affine”. This program assumes that the minimal robot is running, and thus the “tf” topic includes the individual transforms among links, sensor and world. The program creates a tfListener, which looks up the transform from sensor frame to world frame, as well as each of the involved sequential transforms.

A utility function “transformTFToEigen()” converts a tf::Transform into an equivalent Eigen::Afffine3d. The affine type can hold the rotational and translational parts of the transformation, though it does not contain the named parent and child frames, nor the time stamp at which the transform was known to be valid.

Four incremental transforms are requested from the transform listener, and these are converted to corresponding Eigen::Affine types. These four transforms are multiplied times each other to compute the overall transform from sensor frame to world, as follows:

```
affine_kinect_pc_wrt_world=
affine_link1_wrt_world*affine_link2_wrt_link1*affine_kinect_link_wrt_link2
*affine_kinect_pc_wrt_kinect_link;
```

This transform is compared to the result obtained by converting the result from the tfListener directly from sensor frame to world frame. The example program displays the results, showing that the two approaches yield identical results.

It can also be convenient to save affines in a vector. The line:

```
vector <Eigen::Affine3d> vec_of_affines;
```

creates a suitable vector. The push_back() member function is used to append transforms to this vector, e.g. as:

```
vec_of_affines.push_back(affine_kinect_pc_wrt_kinect_link);
```

After filling the vector with the sequential transforms, the product of transforms can be computed as:

```
affine_kinect_pc_wrt_world = vec_of_affines[0];
for (int i=1;i<vec_of_affines.size();i++) {
    affine_kinect_pc_wrt_world = vec_of_affines[i]*affine_kinect_pc_wrt_world;
}
```

Note that the order of multiplication is important, since A*B typically does not equal B*A for matrices. One must take care to multiply transforms in the correct, logical order. The result of multiplying the affine's stored in a vector is identical to the result from explicitly multiplying the transforms together manually. The vector approach, however, quickly becomes more practical as the robot becomes more complex.

Finally, use of the transform is illustrated by converting a fictitious sensor point from the sensor frame to the world frame, which is performed by multiplying as follows:

```
p_wrt_world = affine_kinect_pc_wrt_world*p_wrt_sensor;
```

This point can be converted back into the sensor frame using the inverse transform, which can be invoked as follows:

```
p_back_in_sensor_frame = affine_kinect_pc_wrt_world.inverse()*p_wrt_world;
```

The Eigen::Affine objects are particularly convenient for implementing forward and inverse kinematics based for kinematic chains.