An Eigen::Affine3d is a 4x4 homogeneous transformation matrix

Eigen::Affine3d pose;

pose = Eigen::Translation3d(0, 0, 1);

Eigen::Matrix3d m;

m = AngleAxisd(angle1, Vector3f::UnitZ())

\* AngleAxisd(angle2, Vector3f::UnitY())

\* AngleAxisd(angle3, Vector3f::UnitZ());

pose.linear() = m;

#include <Eigen/Geometry>

Eigen::Affine3d create\_rotation\_matrix(double ax, double ay, double az) {

Eigen::Affine3d rx =

Eigen::Affine3d(Eigen::AngleAxisd(ax, Eigen::Vector3d(1, 0, 0)));

Eigen::Affine3d ry =

Eigen::Affine3d(Eigen::AngleAxisd(ay, Eigen::Vector3d(0, 1, 0)));

Eigen::Affine3d rz =

Eigen::Affine3d(Eigen::AngleAxisd(az, Eigen::Vector3d(0, 0, 1)));

return rz \* ry \* rx;

}

int main() {

Eigen::Affine3d r = create\_rotation\_matrix(1.0, 1.0, 1.0);

Eigen::Affine3d t(Eigen::Translation3d(Eigen::Vector3d(1,1,2)));

Eigen::Matrix4d m = (t \* r).matrix(); // Option 1

Eigen::Matrix4d m = t.matrix(); // Option 2

m \*= r.matrix();

return 0;

}

Another method is to do the following:

Eigen::Matrix3d R;

// Find your Rotation Matrix

Eigen::Vector3d T;

// Find your translation Vector

Eigen::Matrix4d Trans; // Your Transformation Matrix

Trans.setIdentity(); // Set to Identity to make bottom row of Matrix 0,0,0,1

Trans.block<3,3>(0,0) = R;

Trans.rightCols<1>() = T;

Explanation below from <http://pointclouds.org/documentation/tutorials/matrix_transform.php>

***/\* Reminder: how transformation matrices work :***

***|-------> This column is the translation***

***| 1 0 0 x | \***

***| 0 1 0 y | }-> The identity 3x3 matrix (no rotation) on the left***

***| 0 0 1 z | /***

***| 0 0 0 1 | -> We do not use this line (and it has to stay 0,0,0,1)***

***METHOD #1: Using a Matrix4f***

***This is the "manual" method, perfect to understand but error prone !***

***\*/***

**Eigen::Matrix4f transform\_1 = Eigen::Matrix4f::Identity();**

***// Define a rotation matrix (see https://en.wikipedia.org/wiki/Rotation\_matrix)***

**float theta = M\_PI/4; *// The angle of rotation in radians***

***// Define a translation of 2.5 meters on the x axis.***

***/\* METHOD #2: Using a Affine3f***

***This method is easier and less error prone***

***\*/***

**Eigen::Affine3f transform\_2 = Eigen::Affine3f::Identity();**

***// Define a translation of 2.5 meters on the x axis.***

**transform\_2.translation() << 2.5, 0.0, 0.0;**

***// The same rotation matrix as before; theta radians around Z axis***

**transform\_2.rotate (Eigen::AngleAxisf (theta, Eigen::Vector3f::UnitZ()));**

***// Print the transformation***

**printf ("\nMethod #2: using an Affine3f\n");**

**std::cout << transform\_2.matrix() << std::endl;**

**int main()**

**{**

**Eigen::ArrayXf v(6);**

**v << 1, 2, 3, 4, 5, 6;**

**cout << "v.head(3) =" << endl << v.head(3) << endl << endl;**

**cout << "v.tail<3>() = " << endl << v.tail<3>() << endl << endl;**

**v.segment(1,4) \*= 2;**

**cout << "after 'v.segment(1,4) \*= 2', v =" << endl << v << endl;**

**}**

**Spherical interpolation   
(**[Rotation2D](https://eigen.tuxfamily.org/dox/classEigen_1_1Rotation2D.html)**and**[Quaternion](https://eigen.tuxfamily.org/dox/classEigen_1_1Quaternion.html)**only)**

**rot3 = rot1.slerp**(alpha,rot2);

**Component** **accessors**

|  |  |
| --- | --- |
| full read-write access to the internal matrix | t.matrix() = matN1xN1; // N1 means N+1  matN1xN1 = t.matrix(); |
| coefficient accessors | t(i,j) = scalar; <=> t.matrix()(i,j) = scalar;  scalar = t(i,j); <=> scalar = t.matrix()(i,j); |
| translation part | t.translation() = vecN;  vecN = t.translation(); |
| linear part | t.linear() = matNxN;  matNxN = t.linear(); |
| extract the rotation matrix | matNxN = t.rotation(); |

|  |  |
| --- | --- |
|  | **procedural API** |
| [**Translation**](https://eigen.tuxfamily.org/dox/classEigen_1_1Translation.html) | t.translate(Vector\_(tx,ty,..));  t.pretranslate(Vector\_(tx,ty,..)); |

TIME: from <http://library.isr.ist.utl.pt/docs/roswiki/roscpp%282f%29Overview%282f%29Time.html>

## Time and Duration

ROS has builtin time and duration primitive types, which [roslib](http://library.isr.ist.utl.pt/docs/roswiki/roslib.html) provides as the ros::Time and ros::Duration classes, respectively. A Time is a specific moment (e.g. "today at 5pm") whereas a Duration is a period of time (e.g. "5 hours"). Durations can be negative.

Times and durations have identical representations:

int32 sec

int32 nsec

ROS has the ability to setup a simulated [Clock](http://library.isr.ist.utl.pt/docs/roswiki/Clock.html) for nodes. Instead of using platform time routines, you should use roscpp's time routines for accessing the current time, which will work seamlessly with simulated [Clock](http://library.isr.ist.utl.pt/docs/roswiki/Clock.html) time as well as wall-clock time.

### Getting the Current Time

ros::Time::now()

* Get the current time as a ros::Time instance:

ros::Time begin = ros::Time::now();

#### Time zero

When using simulated [Clock](http://library.isr.ist.utl.pt/docs/roswiki/Clock.html) time, now() returns time 0 until first message has been received on /clock, so 0 means essentially that the client does not know clock time yet. A value of 0 should therefore be treated differently, such as looping over now() until non-zero is returned.

### Creating Time and Duration Instances

You can create a Time or Duration to a specific value as well, either floating-point seconds:

1 ros::Time a\_little\_after\_the\_beginning(0.001);

2 ros::Duration five\_seconds(5.0);

or through the two-integer constructor:

1 ros::Time a\_little\_after\_the\_beginning(0, 10000);

2 ros::Duration five\_seconds(5, 0);

### Converting Time and Duration Instances

Time and Duration objects can also be turned into floating point seconds:

1 double secs =ros::Time::now().toSec();

2

3 ros::Duration d(0.5);

4 secs = d.toSec();

### Time and Duration Arithmetic

Like other primitive types, you can perform arithmetic operations on Times and Durations. People are often initially confused on what arithmetic with these instances is like, so it's good to run through some examples:

* 1 hour + 1 hour = 2 hours (duration + duration = duration)

2 hours - 1 hour = 1 hour (duration - duration = duration)

Today + 1 day = tomorrow (time + duration = time)

Today - tomorrow = -1 day (time - time = duration)

Today + tomorrow = error (time + time is undefined)

Arithmetic with Time and Duration instances is similar to the above examples:

1 ros::Duration two\_hours = ros::Duration(60\*60) + ros::Duration(60\*60);

2 ros::Duration one\_hour = ros::Duration(2\*60\*60) - ros::Duration(60\*60);

3 ros::Time tomorrow = ros::Time::now() + ros::Duration(24\*60\*60);

4 ros::Duration negative\_one\_day = ros::Time::now() - tomorrow;

## Sleeping and Rates

bool ros::Duration::sleep()

* Sleep for the amount of time specified by the duration:

1 ros::Duration(0.5).sleep(); // sleep for half a second

ros::Rate

* roslib provides a ros::Rate convenience class which makes a best effort at maintaining a particular rate for a loop. For example:

ros::Rate r(10); // 10 hz

while (ros::ok())

{

... do some work ...

r.sleep();

}

In the above example, the Rate instance will attempt to keep the loop at 10hz by accounting for the time used by the work done during the loop.

**Note:** It is generally recommended to use Timers instead of Rate. See the [Timers Tutorial](http://library.isr.ist.utl.pt/docs/roswiki/roscpp_tutorials%282f%29Tutorials%282f%29Timers.html) for details.

## Wall Time

For cases where you want access to the actual wall-clock time even if running inside simulation, roslib provides Wall versions of all its time constructs, i.e. ros::WallTime, ros::WallDuration, and ros::WallRate which have identical interfaces to ros::Time, ros::Duration, and ros::Rate respectively.