Mathematics for Robotics and Control - Assignment 005: Frames

In [ ]:
```
Worked with David Ackerson
```

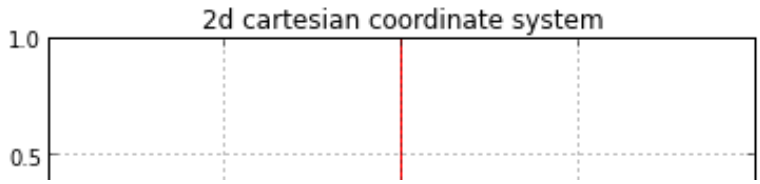
In [1]:
```
import sys
# Windows and Linux so far have no problem with the inline backend...
if not "win" in sys.platform and not "linux" in sys.platform:
    %pylab
# ...on other platforms we did run into problems in the past, so let's u
else:
    %pylab inline
```

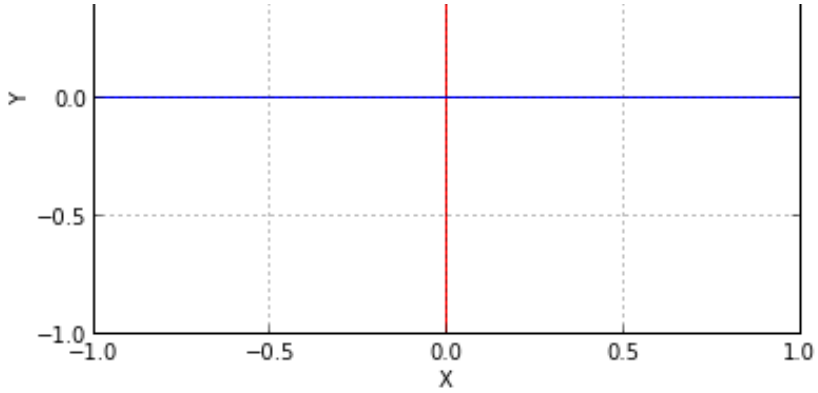
```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.kernel.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

You'll notice that this notebook is a little more verbose than previous ones - this is due to it originally being part of some form of tutorial. The "real" assignments are at the end of the notebook, so if you don't feel like reading all this, just skip to the end.

This assignment is about coordinate systems and frames. We start out with a simple, two dimensional cartesian coordinate system:

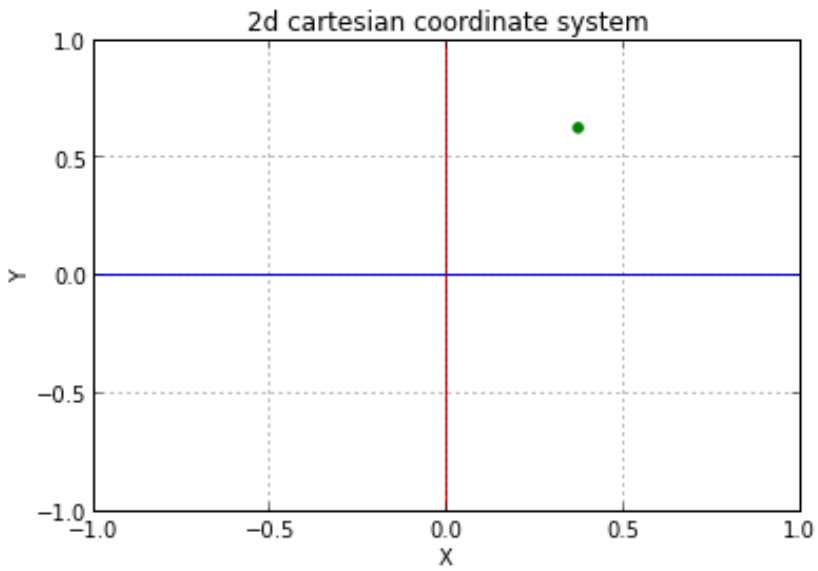
In [6]:
```
def create_figure(xlim=None, ylim=None):
    xlim = xlim if xlim is not None else (-1, 1)
    ylim = ylim if ylim is not None else (-1, 1)
    fig = figure()
    ax = fig.gca()
    ax.set_xlim(*xlim)
    ax.set_xlabel("X")
    ax.set_ylim(*ylim)
    ax.set_ylabel("Y")
    ax.set_title("2d cartesian coordinate system")
    ax.plot([0, 0], ylim, "r")
    ax.plot(xlim, [0, 0], "b")
    ax.grid()
    return fig, ax

fig, ax = create_figure()
```





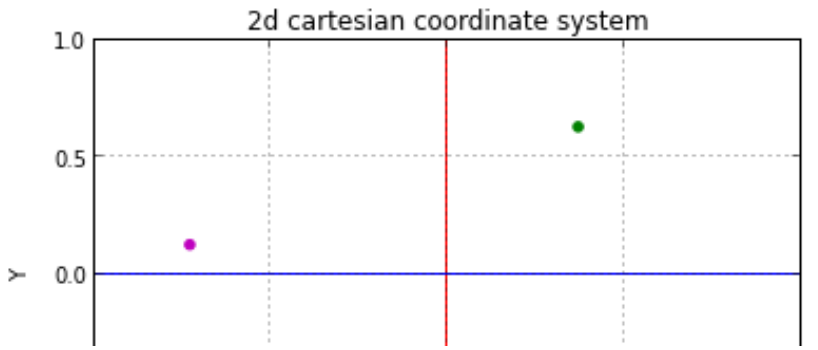
It is now possible to specify a point inside this coordinate system - we just neeed a value for each of the axes of the coordinate system. Since our coordinate system has two dimensions **X** and **Y**, we need two values. Let's call them **x** and **y** and put a point at $x = 0.37$ and $y = 0.63$:

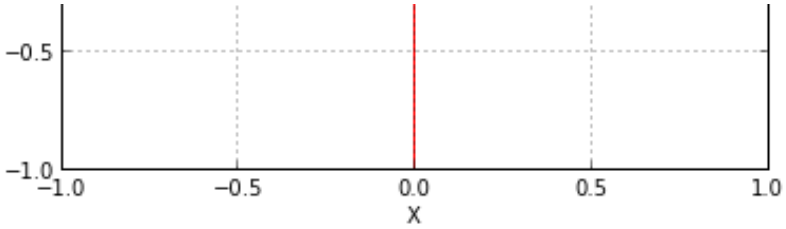
In [2]:
```
#fig, ax = create_figure()
x = [0.37]
y = [0.63]
ax.scatter(x, y, color="g")
display(fig)
```



The green dot is our point p_1. Here is another point p_2 at $x_2 = -0.73$ and $y_2 = 0.13$:

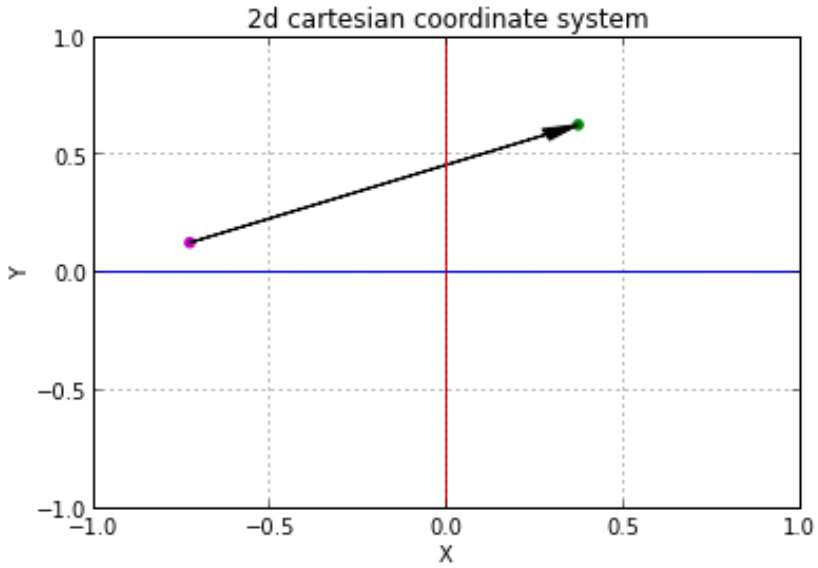
In [3]:
```
x2 = [-0.73]
y2 = [0.13]
ax.scatter(x2, y2, color="m")
display(fig)
```





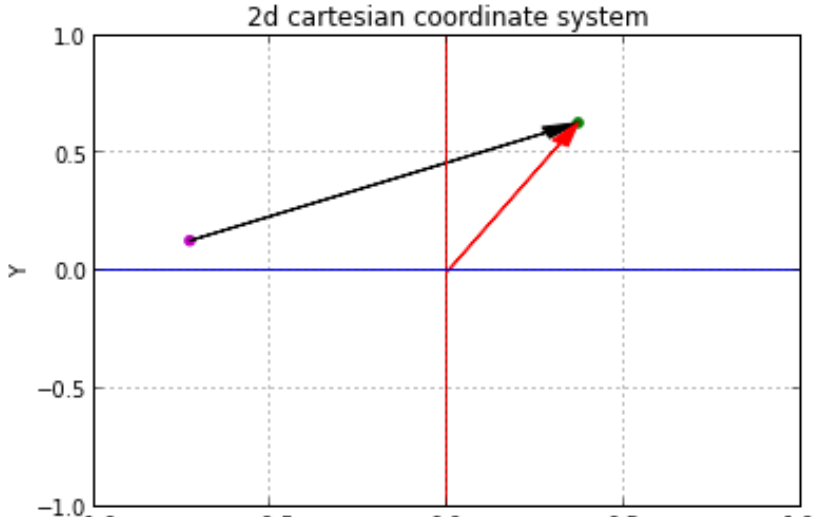
The shortest distance between those points is a line. Let's draw this line:

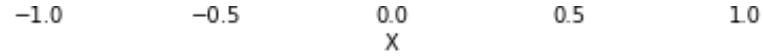
In [4]:
```
x_distance = 0.37 - -0.73
y_distance = 0.63 - 0.13
ax.arrow(-0.73, 0.13, x_distance, y_distance, fc="k", ec="k", head_width
display(fig)
```



You will notice that the line connecting p_1 and p_2 has a direction (it points from p_2 to p_1) and a magnitude (it's length). Recall that a geometric object that has both direction and magnitude can be expressed using a vector. Now think of what happens if we add a line from the coordinate system's origin at $(0, 0)$ to a given point, say p_1:

In [5]:
```
ax.arrow(0, 0, 0.37, 0.63, fc="r", ec="r", head_width=0.05, head_length=
display(fig)
```





This is in fact no different than what we did before, i.e. having the vector point from p_2 to p_1. This has two rather obvious consequences:

I. Any point in a coordinate system can be expressed as a vector pointing from the origin to the point's coordinates
II. Any point in the existing coordinate system can be the origin of a new coordinate system that is "somehow" related to the original coordinate system

We can thus express a point as follows: $p =< x, y >$, where $< x, y >$ denotes a vector with two components (coordinates). Let us now talk about how the two coordinate systems are related. In order to do this, it is helpful to assign names to both coordinate systems. We assign the letter W to the coordinate system that has it's origin at $(0, 0)$, while we name the coordinate system that has it's origin at $p_2 =< -0.73, 0.13 >$ L. If we look at a point p in coordinate system C, we denote this as ${}^{C}p$, i.e. if we want to talk about p_1 having the coordinate values $x = 0.37$ and $y = 0.63$ in coordinate system W, we write: ${}^{W}p_1 =< 0.37, 0.63 >$.

Now, the relationship between those two coordinate frames is simple. Think about how to express the point ${}^{W}p_1 =< 0.37, 0.63 >$ in L. First, look at the coordinates of p_1 in L: ${}^{L}p_1 =< 1.1, 0.5 >$. Obviously, the relation between the two points in both coordinate systems W and L is the distance of the respective origins of W and L. A coordinate transformation can then be carried out by a simple addition of vectors, whereas the inverse transformation is available by just inverting the sign of the displacement vector.

Talking only about translation is not enough, though - imagine what happens when a robot changes it's direction while following a path, or it's sensors suddenly start looking into another direction. It is easy to see that we also need to be able to express rotational transformations. In order to visualize those, let's first have a look at the unit circle. Imagine your robot is located at ${}^{W}p_{robot} =< 0, 0 >$ and wants to inspect it's complete environment by doing a scan of it's surroundings, i.e. it wants to do a full 360° turn. The point the robot is looking at could thus be described as moving along a unit circle. As you know from trigonometry, the points along a unit circle (i.e. $r = 1$) can be described by the following equation:

$$x^2 + y^2 = 1$$

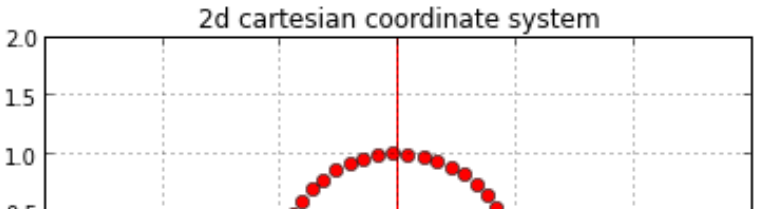
Using the parametric form, we can write this as

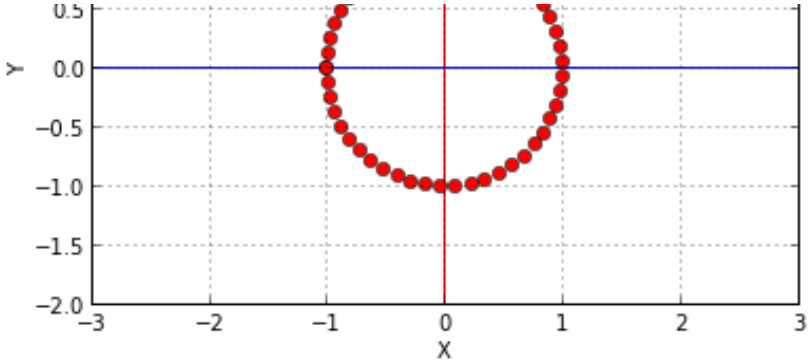
$$x = cos(t)$$

$$y = sin(t)$$

Let's see if those equations make any sense by plotting them:

In [7]:
```
import numpy
fig, ax = create_figure((-3, 3), (-2, 2))
for t in numpy.linspace(-numpy.pi, numpy.pi, 50):
    x = cos(t)
    y = sin(t)
    ax.plot(x, y, "ro")
```

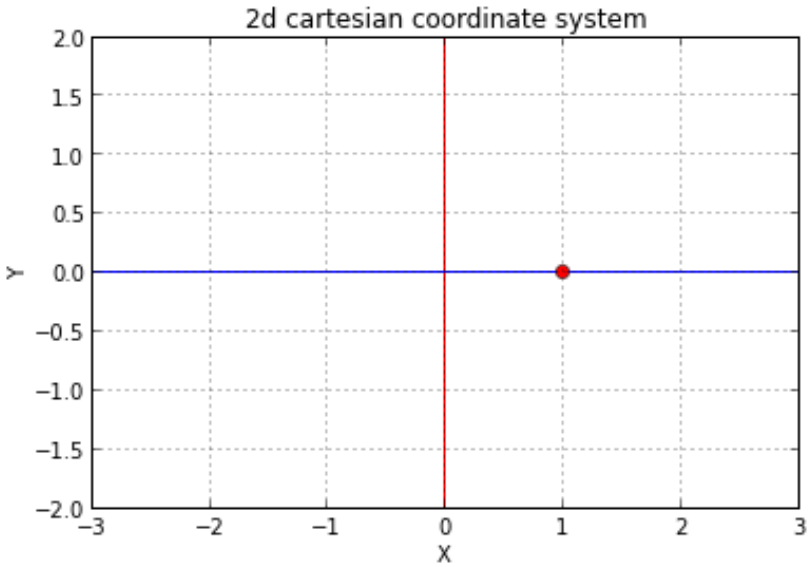




Ok, that looks about right - we can now describe motions along a circle. What if we want to express these motions as a more closed-form term that can be apply to a single point to move it along that circle? Let's start with ${}^{W}p_2 =< 1, 0 >$:

In [8]:
```
import numpy
fig, ax = create_figure((-3, 3), (-2, 2))
W_p2 = [1, 0]
ax.plot([W_p2[0]], [W_p2[1]],"ro")
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0xafe14ac>]
```



How can we move ${}^{W}p_2$ along the same circle as before, but using a transform we apply to ${}^{W}p_2$? The answer is: by using a rotation matrix. Let's express the circle equations from above using a rotation matrix R:

$$R = \begin{bmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{bmatrix}$$

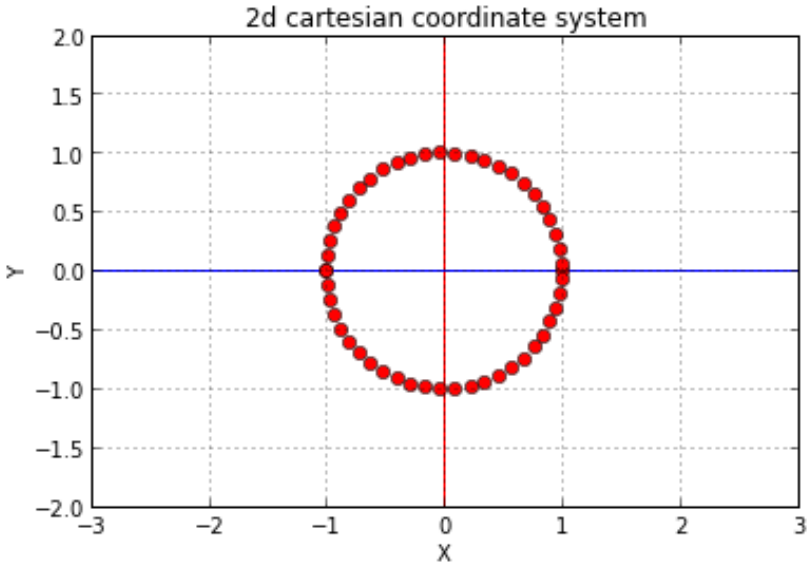
Here is a Python function to create such a matrix, given α:

In [9]:
```
create_rotation_matrix = lambda alpha: numpy.array([ [numpy.cos(alpha),
```

Here is what happens when you apply this matrix to ${}^{W}p_2$:

In [15]:
```
fig, ax = create_figure((-3, 3), (-2, 2))
W_p2 = [1, 0]
```

```
ax.plot([W_p2[0]], [W_p2[1]],"ro")
for alpha in numpy.linspace(-numpy.pi, numpy.pi, 50):
    R = create_rotation_matrix(alpha)
    W_p2 = numpy.dot(R, [1, 0])
    ax.plot([W_p2[0]], [W_p2[1]],"ro")
```



That looks precisely like the circle above. Good - you now know how to translate points and rotate them to move them backwards and forwards between different coordinate systems. Can we combine both the translation and the rotatio in a single expression to form a more general transformation? The answer is of course: yes, we can. Here is how:

$$T = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & offset_x \\ sin(\alpha) & cos(\alpha) & offset_y \\ 0 & 0 & 1 \end{bmatrix}$$

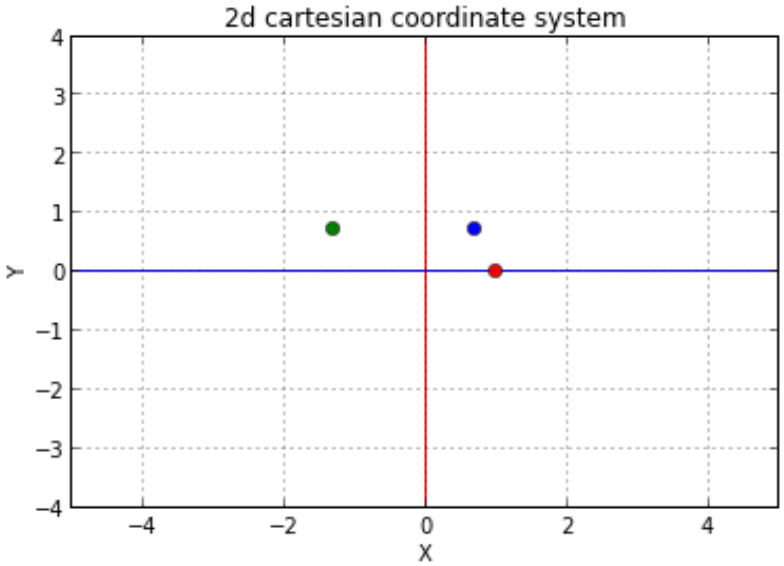
α is our rotation angle again, while $offset_x$ and $offset_y$ are the distances of the coordinate systems with respect to x and y, i.e. the vector $o =< offset_x, offset_y >$ is the vector pointing from one coordinate system origin to the other. Now, given that our point is in 2d and our new transformation matrix is 3×3, we need to add another coordinate to our point to enable matrix multiplication. We pick 1 and simply write $p =< x, y, 1 >$, resulting in a homogeneous representation. Again, moving and rotationg a point can now be done using a matrix-vector product:

In [30]:
```
create_transformation_matrix = lambda alpha, offset: numpy.array([ [nump

T = create_transformation_matrix(numpy.radians(45), [0, 0])
print "T = \n", T

W_p3 = [1, 0, 1]
L_p3 = numpy.dot(T, W_p3)
print "\nL_p3 =\n", L_p3

fig, ax = create_figure((-5, 5), (-4, 4))
ax.plot([W_p3[0]], [W_p3[1]],"ro")
ax.plot([L_p3[0]], [L_p3[1]],"bo")

T2 = create_transformation_matrix(numpy.radians(45), [-2, 0])
print "T2 = \n", T2
L2_p3 = numpy.dot(T2, W_p3)
print "\nL2_p3 =\n", L2_p3
ax.plot([L2_p3[0]], [L2_p3[1]],"go")
```

```
T =
```

```
[[ 0.70710678 -0.70710678  0.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]]

L_p3 =
[ 0.70710678  0.70710678  1.        ]
T2 =
[[ 0.70710678 -0.70710678 -2.        ]
 [ 0.70710678  0.70710678  0.        ]
 [ 0.          0.          1.        ]]

L2_p3 =
[-1.29289322  0.70710678  1.        ]
```

Out[30]: `[<matplotlib.lines.Line2D at 0xc00bb4c>]`



So far, all our points were 2-dimensional, and we only rotated about one axis. Let's change that. While doing so, you will learn how to create coordinate frames and the transformation matrices to convert between them. We move to three dimensions and create two reference frames that are related to each other, using Python and SymPy. In our case, the reference frame **I** will be a fixed, global reference frame (i.e. the 'world' frame), while **K** will be a frame that is related to **I** by one or more rotations around **I**'s axes.

In [15]:
```
from sympy.physics.mechanics import ReferenceFrame
from sympy import Symbol

import sympy
sympy.init_printing(use_latex=True)

# create a fixed (global) reference frame
I = ReferenceFrame("I")

# create a new reference frame that is rotated around I's X axis by an a
alpha = Symbol("alpha")
K_x = I.orientnew("K_x", "Axis", (alpha, I.x))

# create a Direction Cosine Matrix (rotation matrix) that describes this
R_x = I.dcm(K_x)

display(R_x)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

Let's create additional rotation matrices for the remaining **Y** and **Z** axes:

In [1]:
```
# create rotation matrices for rotation around I's Y and Z axes
#beta = Symbol("beta")
K_y = I.orientnew("K_y", "Axis", (beta, I.y))
R_y = I.dcm(K_y)


gamma = Symbol("gamma")
K_z = I.orientnew("K_z", "Axis", (gamma, I.z))
R_z = I.dcm(K_z)

R_x, R_y, R_z
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-e5a19dd20cb8> in <module>()
      1 # create rotation matrices for rotation around I's Y and Z axes
      2 beta = Symbol("beta")
----> 3 K_y = I.orientnew("K_y", "Axis", (beta, I.y))
      4 R_y = I.dcm(K_y)
      5

AttributeError: 'ImaginaryUnit' object has no attribute 'orientnew'
```

The three rotation matrices can be combined via matrix multiplication to create a final rotation matrix that takes any point in **I** to **K**:

In [17]:
```
R_z * R_y * R_xq
```

Out[17]:
$$\begin{bmatrix} \cos(\beta)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma)-\sin(\gamma)\cos(\alpha) & \sin(\alpha)\sin(\gamma)+\sin(\beta)\cos(\alpha)\cos(\gamma) \\ \sin(\gamma)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma)+\cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\gamma)+\sin(\beta)\sin(\gamma)\cos(\alpha) \\ -\sin(\beta) & \sin(\alpha)\cos(\beta) & \cos(\alpha)\cos(\beta) \end{bmatrix}$$

Assignment 5.1:

Now that you have the tools to describes frames and transformations, let us look at a concrete example. Imagine the following system:

Three dice **A**, **B** and **C** are stacked on top of each other. Each die has a side length of 50mm and it's own coordinate frame with three axes that originate in the center of gravity (CoG) of the die. For each die, the vectors describing it's axes can be determined by connecting the center of the die with the center of one of it's faces, depending on which axis one is talking about. We enumerate the faces

based on the number of eyes they show, prefixed by the name of the die, i.e. $A1, A2, \ldots, A6$, $B1, B2, \ldots, B6$ and $C1, C2, \ldots, C6$. See the picture below for an illustration.

The complete stack is located in a larger universe with a global coordinate frame called **W**. The center of gravity of die A is given as ${}^{W}P_{orgA} =< 0, 0, 0 >$, i.e. the die's center is located at the origin of the global coordinate frame W. A is oriented in such a way that the axis intersecting with $A1$ points in the direction of the **X** axis in W, whereas the axis intersecting $A3$ points to W's **Y** and the vector from A's center to the center of $A2$ points in the **Z** direction in W. Again, have a look at the picture above if you are getting confused.

While the stack is located in a larger world, each die is home to a different population of sentient, microscopic organisms that can communicate with each other, and with both of the two remaining poplations on the other dice. Since the individuals that make up the populations are pretty small, each die feels like a whole world to them, so each population has established it's own coordinate system for their respective world.

I. Not only are the organisms really small, but they are also really clever, and every single one of them is interested in astrophysics, examining the known universe surrounding the three worlds.. Given the incredible age of each population, the governments of each die-world have long put aside any conflicts and decided to cooperate and share their scientific findings. Now, the organisms living on world C discover a new celestial body **S** at ${}^{C}S =< 0, 10, 0 >$ and send their findings to the population of A. **Imagine you are part of the universe in which the stack is located, i.e. your own coordinate system is equal to W. Where would you think the body S would be located? (Transform the location of S to the frame W)**

the Units of the 10 are not defined.

${}^{W}S =< 10, 10cm, 0 >$

I. **Given that each world knows it's relative location to both other worlds, how can the scientists living on *A* determine where the body *S* is located, without first converting to an external coordinate system like W?**

They are able to subtract the location of the body *S* wrt *C* from the location of *A* wrt *C* The result is a vector originating at the origin of *A* going to *S* all in *C* coordinate space, so then only the rotations from *C* to *A* need to be accounted for.

I. **In order to return the favor, A decides to share the location of several recently discovered objects with C. The scientists in A know where the objects are located w.r.t to A, now C wants a general way to convert the received locations to their own world coordinate frame. Provide such a mechanism for the people of C.**

From A to C, rotating in X-Y-Z Fixed angles, first rotate about X by -90, then about Y by -90. then done. Use formula in slides or method above in tutorial to obtain rotation matrix.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Find homogenous transform matrix by adding on the translation in z by two dice lengths.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 10cm \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Assignment 5.1 took me minutes.

Assignment 5.2:

Assume you have a vector ${}^{A}P$ that you obtain by applying three rotations to another vector, thus mapping it to a new coordinate frame:

$${}^{A}P = {}^{A}_{B}R \cdot {}^{B}_{C}R \cdot {}^{C}_{D}R \cdot {}^{D}P$$

There are basically two ways to perform this operation:

I. Obtain a general rotation matrix that performs all steps in one pass: ${}^{A}P = {}^{A}_{D}R \cdot {}^{D}P$
II. Transform the vector one rotation at a time:
   A. ${}^{A}P = {}^{A}_{B}R \cdot {}^{B}_{C}R \cdot {}^{C}_{D}R \cdot {}^{D}P$
   B. ${}^{A}P = {}^{A}_{B}R \cdot {}^{B}_{C}R \cdot {}^{C}P$
   C. ${}^{A}P = {}^{A}_{B}R \cdot {}^{B}P$
   D. ${}^{A}P = {}^{A}P$

Imagine ${}^{D}P$ is changing at 100Hz, resulting in 100 re-computations of ${}^{A}P$ per second. Additionally, each of the three rotation matrices is also chaning, but updated by a different sensor (e.g. localization based on computer vision) and updated with a frequence of 30Hz. **What is the best way to organize the computation of ${}^{A}P$ to minimize the calculation effort? Use the number of multiplications and additions as metric.**

Using basic matrix multiplication, multiplying two N-by-N Matrices requires N multiplications and (N-1) additions for each of the NxN resulting elements. Computing:

$${}^{A}_{D}R = {}^{A}_{B}R \cdot {}^{B}_{C}R \cdot {}^{C}_{D}R$$

Therefor requires two 3x3 matrix multiplications. One 3-by-3 matrix multiplication requires:

$$9elements * (3multiplications + 2additions) = 27multiplications + 18additions$$

Twice requires 54 multiplications and 36 additions. This rotation matrix, ${}^{A}_{D}R$ would need to be updated at 30Hz.

In contrast, multiplying a Matrix by a vector reduces the dimensionality. a N-by-N matrix, multiplied by a N-by-1 vector produces another N-by-1 vector, each element of which required as before N multiplications and N-1 additions, but this time there are less elements.

in step A: ${}^{A}P = {}^{A}_{B}R \cdot {}^{B}_{C}R \cdot {}^{C}_{D}R \cdot {}^{D}P$
The first step ${}^{C}P = {}^{C}_{D}R \cdot {}^{D}P$ required 9 multiplications and 6 additions.

${}^{C}P$,^{B}P, ${}^{A}P$ each required 9 multiplcations and 6 additions. For a total of 27 multiplications and 18 additions. This needs to be done at a frequency of 100 Hz. Transforming the vector one rotation at a time, at 100Hz requires 2700 multiplications and 1800 additions per second.

Keeping the Rotation matrix up to date at 30Hz requires 1620 multiplications and 1080 additions. And that is only the Rotation matrix. there is an additional 9 multiplications and 6 additions that must be done to achieve the final answer every 100Hz. for a grand total of 2520 multiplications and 1680 additions per second. Therefor the best way to organize the computation of ${}^{A}P$ is the first method presented.

Assignment 5.2 took me minutes.

Assignment 5.3:

Imagine you are constructing a mobile robot and are currently working on the base. You want to equip the robot with a sense of position. Knowing that GPS does not work indoors and the environment your robot will operate on does not feature prominent landmarks, you opt for an approach based on measuring the robot's acceleration.

I. Assume the robot only moves on a plane. How many acceleration sensors would you need, and what would be a good way to mount them? (You might remember this from your AMR lecture / assignments)
   If this robot is moving on a perfect flat horizontal plane, then the minimum requirement would be two acceleration sensors, each with their measuring axis orthogonal to the other. It may however be convienent to have a third, to measure acceleration in the Z or vertical axis. The assumption is that the robot moves only on a plane, so in a 2D plane, the best places for the accelerometers are parallel to the plane but perpendicular to each other. Or, alternitively you get a magic black box Inertial measurement unit which combines multiple accelerometers and giros. and consider it one sensor returning arrays of magic numbers
II. How do you obtain the robot's position based on the sensor readings? Specify equations.

   For each accelerometer, measuring one axis, you are being given the acceleration at time now. to obtain a position relative to were the robot started measuring you must take the double integral of accelerometer from some starting time until time now, this will give you the movement in the direction which that particular accelerometer measured in the specified time frame.

because $a(t) = \frac{dv}{dt}$, $v(t) = \frac{x}{t}$, $a(t) = \frac{d^2x}{dt^2}$

$x = \int\int a(t)$. Where x is the axis measured by that particular accelerometer.

more formally

$x = \int\int a_x(t)$
$y = \int\int a_y(t)$. in our robot with two accelerometers measuring x and y motion in a 2D plane.

3. Imagine you have constructed two nearly identical robots using this approach: robot A features a fixed arm mounted 1.7m above the base, extending 1.5m beyond the center of the base, while Robot B has a regular manipulator that has the same length and mounting point, except it is able to move and grasp objects. Robot A's manipulator always points to the direction in which A is driving. Assume both robots are located in an indoor environment using the global coordinate frame W. Robot A starts at ${}^{W}R_A =< 0, 0, 0 >$ and accelerates at $0.46\ \frac{m}{s^2}$, heading straight for ${}^{W}R_B =< 23, 15, 0 >$ while holding an object in it's hand. Robot B is itself also heading towards robot A, it's final destination being ${}^{W}D_B =< 0, 0, 0 >$. B is accelerating at $0.7\ \frac{m}{s^2}$. Both robots have a maximum travel velocity of $1.8\ \frac{m}{s}$.

4. Specify the motion of robot A over time in the coordinate system of robot B. Assume each robot's coordinate system that the coordinate system is centered at it's base.

$${}^{B}R_{A}(t) =< \sqrt{23^2 + 15^2} - min(\tfrac{1}{2} * 0.46 * t^2, 1.8 * t) - min(\tfrac{1}{2} * 0.7 * t^2, 1.8 * t), 0, 0 >.$$

5. Give the position of the object in A's hand in the coordinate system of B over time.

$${}^{B}R_{A}(t) =< \sqrt{23^2 + 15^2} - 1.5 - min(\tfrac{1}{2} * 0.46 * t^2, 1.8 * t) - min(\tfrac{1}{2} * 0.7 * t^2, 1.8 * t), 0, 1.7 >.$$

6. Specify the ideal time for B to pick up the object from A.

First: start with a similar equation as above. And from that remove the min functions by calculating the time it takes for both robot reach there max velocity and then calculating how far the robots have removed in that time, 3.5417m and 4.7293m, robot A and robot B. Then account for the arms being 1.5m infront of each robot, and from that point on both robots are moving $1.8m/s$ this gives the equation below: ${}^{B}R_{A}(t) =< \sqrt{23^2 + 15^2} - 3.5217 - 4.7293 - 1.5 - 1.5 - 2 * 1.8 * t, 0, 0 >$

for which we solve solve for the distance = 0.

$$16.2081 - 18/5 * t = 0$$

$$t = 4.5023$$

$$t = 4.5023 + 3.9130$$

t = 8.4153

adding on the 3.9130 seconds where the robots were not travelling at constant velocities.

Use transformation matrices for all your calculations.

Assignment 5.3 took me minutes.

Use this button to create a .txt file containing the time in minutes you spent working on the assignments. Make sure to include your name in the textbox below. The file will be created in the current directory.

Student's name:

Save timings