[WS13/14] Mathematics for Robotics and Control: Assignment 001 - Frames

In [15]:
```
import IPython.core.display
import sys
if not "win" in sys.platform and not "linux" in sys.platform:
    %pylab
else:
    %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: ['angle', 'linalg', 'ra
ndom', 'power', 'info', 'fft']
`%pylab --no-import-all` prevents importing * from pylab and numpy
```

Assignments are usually marked with a difference level based on what your TA thinks the average student should be capable of:

- [L1]: The required solution is very similar to what we already discussed in the lecture / lab class. No surprises here.
- [L2]: The solution requires the application of theoretical and practical knowledge you already have, but might require applying said knowledge in a different manner / to a different problem. If there is new stuff, there really is nothing to it, i.e. no new material will be introduced that would offer any depth or require you to understand any of it before you'd be able to apply it.
- [L3]: Oh-Em-Gee, you actually have to think to solve this assignment. You might even have to look up new stuff. What a drag!

These ratings are included to help you plan on how you want to work on the assignment, to better schedule your time etc.

Function and Module List

Each week, the assignment sheet will contain a list of Python modules and functions you are supposed to know. These modules and functions will help you to solve the assignments and most of them are required in later assignments. You are expected to familiarize yourself with each module / function listed here and to know when and how to apply them. Remember this is especially important in the exam!

Modules:

- numpy
- matplotlib.pyplot
- sympy.physics.mechanics

Functions:

numpy:

- arange, apply_along_axis, array, asarray, diag, dstack, flat, flatten, hstack, linspace, load, ones, ravel, save, shape, split, squeeze, vstack, zeros

matplotlib.pyplot:

- cla, clf, figure, gca, legend, plot, show, subplot, title, xlabel, xlim, ylabel, ylim

sympy.physics.mechanics.ReferenceFrame:

- dcm, orient, orientnew, x, y, z

Assignment 1.1 [L1]

Have a look at the following video that depicts the KUKA youBot during the 2012 hackathon at BRSU drawing a picture of itself.

```
In [16]: IPython.display.YouTubeVideo("1OYyTs3DrPo?html5=1")
```

Out[16]: youBot hackathon 2012 - youBot drawing picture

Assume the robot has a local reference frame (images/youbot_render.png) attached to it's base, called R_{YB}. The drawing software on the youBot sends the coordinates of the tip of the pen used to create the drawing to an external softwarepackage. All point coordinates are expressed in R_{YB}. Furthermore, the youBot is drawing autonomously in an arena while you are observing it from a elevated, remote location behind a glass panel. See the picture below if you are unsure what the scenario looks like.

In [17]:
```
IPython.core.display.Image(r"images/observationlab.jpg", embed=True)
```

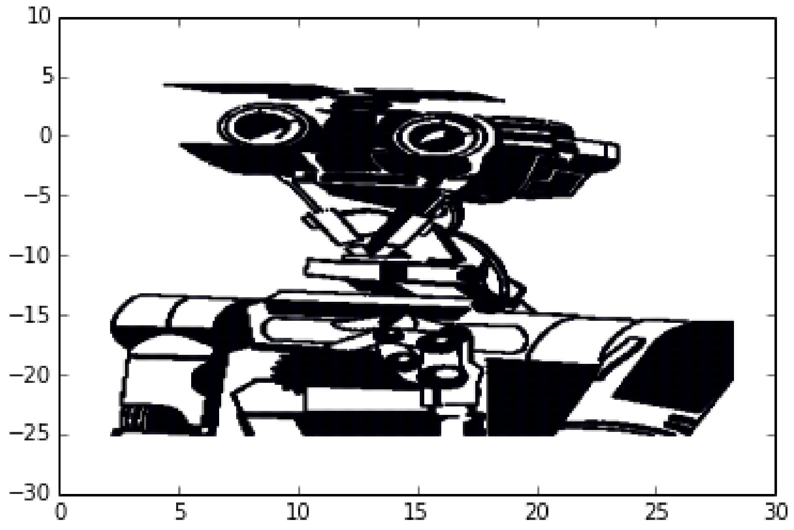
Out[17]:

While you are looking down at the youBot drawing stuff, the little fellow transmits its local coordinates to a computer near you. Now, you want to print a copy of whatever the youBot is drawing using an old 70s plotter sitting on your desk inside the observation room (if you've never seen a plotter, here is an example image (images/plotter.jpg)). Now, the plotter only accepts drawing coordinates in it's local coordinate frame, so you have to convert whatever coordinates the youBot is sending. You can assume that the youBot is drawing on the floor of the observation lab. The floor of your chamber is located $5m$ above the lab floor, and the plotter is sitting on a desk that is $72cm$ high. Note that the plotter is a vertical model, and it's drawing area is facing away from the observation window. The distance from the origin of the drawing area of the plotter and the origin of R_{YB} is exactly $7.5m$. You can assume that the size of the drawing area of the youBot and the plotter's drawing area size are identical, and that the X axes of both frames are aligned.

Write a program that loads the data file youbot_drawing_points.npy and simulate the plotter using matplotlib to obtain a copy of whatever the youBot is drawing. Display the image.

In [18]:
```
# Solution 1.1
from sympy.physics.mechanics import ReferenceFrame
import numpy as NP
coordinates = load("youbot_drawing_points.npy")
world = ReferenceFrame('W')
newFrame = world.orientnew("newFrame", "Axis", (radians(-90), world.x))
rotationMatrix = world.dcm(newFrame)
rotationMatrix = NP.array(NP.asarray(rotationMatrix), dtype=numpy.float64)
for i,element in enumerate(coordinates):
    coordinates[i] = rotationMatrix.dot(element)
scatter(coordinates[:,0], coordinates[:,1], s = .5)
```

Out[18]:
```
<matplotlib.collections.PathCollection at 0x49a7c50>
```



Assignment 1.1 took me minutes.

Assignment 1.2 L2

Look at slide number 47 for this week's lecture, titled "*Computational Considerations*". The slide discusses two different methods of going from a frame D to a destination frame A, using two intermediate frames B and C. The first method suggests combining the matrices ${}^A_BR$, ${}^B_CR$ and ${}^C_DR$ into a new matrix ${}^A_DR$ and apply this matrix repeatedly, while the second approach would go through each single matrix separately when transforming a new point to the destination frame.

So, in general: ${}^AP = {}^A_B R \cdot {}^B_C R \cdot {}^C_D R \cdot {}^D P$

1. Method 1: ${}^AP = {}^A_D R \cdot {}^D P$
2. Method 2:
   A. ${}^AP = {}^A_B R \cdot {}^B_C R \cdot {}^C_D R \cdot {}^D P$
   B. ${}^AP = {}^A_B R \cdot {}^B_C R \cdot {}^C P$
   C. ${}^AP = {}^A_B R \cdot {}^B P$

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 changing as well, but updated with a frequency of 30Hz.

1. **Describe a situation in which this scenario could occur. Why would the update rates be different? Why would the rotation matrices change?**
2. **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 a metric (see lecture slide 47).**
3. **Write a little Python script that implements both methods and simulate both use cases using the update rates mentioned above. Use IPython's %timeit macro to measure the time for both scenarios. Examine if your simulation support your previous claims.**

In [37]:

```
# Solution 1.2
# ...
#
#2.1: Suppose you throw ball to someone, the ball not only rotates but also
changes it position.
############################################################################
#########################################
#
#2.2:
#   Blockwise results in 54 multiplication and 36 additions.
#   Stepwise results in 27 multiplications and 18 additions.
#   Blockwise should be slower, however in our task, rotations are changing
30 times per second, and
#   Poistion vector dP is changing 100 times per second.
#
#   So for blockwise computation:
#   (54 x 30) + (100 x 9) = 2520 multiplications
#   (36 x 30) + (100 x 6) = 1080 additions.
#
#   For stepwise computation:
#   (27 x 100)            = 2700 multiplications
#   (18 x 100)            = 1800 additions.
#
#   So for our case blockwise should be faster.
#
############################################################################
#########################################
import sympy.physics.mechanics

W = ReferenceFrame('W')
rotationalMatrix = GF.orientnew("W", "Axis", (radians(-90), GF.x))
rA = GF.dcm(rotationalMatrix)
rotationalMatrix = GF.orientnew("W", "Axis", (radians(-90), GF.y))
rB = GF.dcm(rotationalMatrix)
```

```
rotationalMatrix = GF.orientnew("W", "Axis", (radians(-90), GF.z))
rC = GF.dcm(rotationalMatrix)
pD = [1,1,1]

def blockWise():
    for y in range(100):
        if y < 30:
            rAB = rA.dot(rB)
            rABC = rAB.dot(rC)
            pA = rABC.dot(pD.transpose())

def stepWise():
    for y in range(100):
        pC = rC.dot(pD.transpose())
        pB = rB.dot(pD)
        pA = rA.dot(pB)

print "Time taken by blockwise: "
%timeit blockWise
print "Time taken by stepwise:"
%timeit stepWise
```

```
Time taken by blockwise:
10000000 loops, best of 3: 28.4 ns per loop
Time taken by stepwise:
10000000 loops, best of 3: 25.3 ns per loop
```

Assignment 1.2 took me minutes.

Assignment 1.3 [L2]

You are the processing unit of the Mars rover Spirit (http://xkcd.com/695/), currently stuck in the soil of Mars (images/mars.jpg), unable to move. Power is failing, your creators seem to have abandoned you, and yet your mission continues. With only a single LIDAR (http://en.wikipedia.org/wiki/Lidar) sensor remaining, you notice a distant movement (images/marsdustdevil.gif). Recalling your position, you remember mission control commanded you to established a reference frame W a while ago, and that the center of your base is currently residing at ${}^{W}P_S = (3.7m, 3.5m, 0m)$. Accessing your construction schematics tells you your LIDAR is sitting on top of a $50cm$ long pole whose center is located at polar coordinates $r = 43cm, \phi = 37^\circ$, where the center of your base is the origin of the polar coordinate system. The LIDAR is a long-range laser scanner with an opening angle of 180°, directly transmitting the distance of the moving phenomenon relative to the LIDAR's own position into your memory banks in steps of 1°. Degree zero is exactly on your right, degree 180 exactly on your left. Now, given your new status as "Stationary Research Platform" (pff!), you decide to watch the phenomenon for a while and record it's movement.

In [ ]:
```
IPython.core.display.Image(r"images/lidar_scanner.png", embed=True)
```

Over time, you obtain 1000 LIDAR scans and detect the object at different positions. You save your observations in the file lidar_scan.npy. **Reconstruct, extrapolate and visualize the object's movement according to your sensor scans. Is there any logic to it's movement?**

In [ ]:
```
# Solution 1.3
# ...
import numpy as NP

from pylab import *
scans = load("lidar_scan.npy")
polarCoordinates = []
for scan in scans:
    for angle, distance in enumerate(scan):
            if distance != 0:
                polarCoordinates.append([180 - radians(angle), distance])
polarCoordinates = NP.array(polarCoordinates, dtype=NP.float64)
cartesianCoordinates = []
for polarCoordinate in polarCoordinates:
    angle = polarCoordinate[0]
    distance = polarCoordinate[1]
    xAxis = distance * cos(angle)
    yAxis = distance * sin(angle)
    cartesianCoordinates.append([xAxis, yAxis])
cartesianCoordinates = NP.array(cartesianCoordinates, dtype=NP.float64)
plot(cartesianCoordinates[:, 0], cartesianCoordinates[:, 1])
scatter(cartesianCoordinates[:, 0], cartesianCoordinates[:, 1])
```

Assignment 1.3 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

In [ ]: