Time Optimal Smartmouse Controls

October 5, 2017

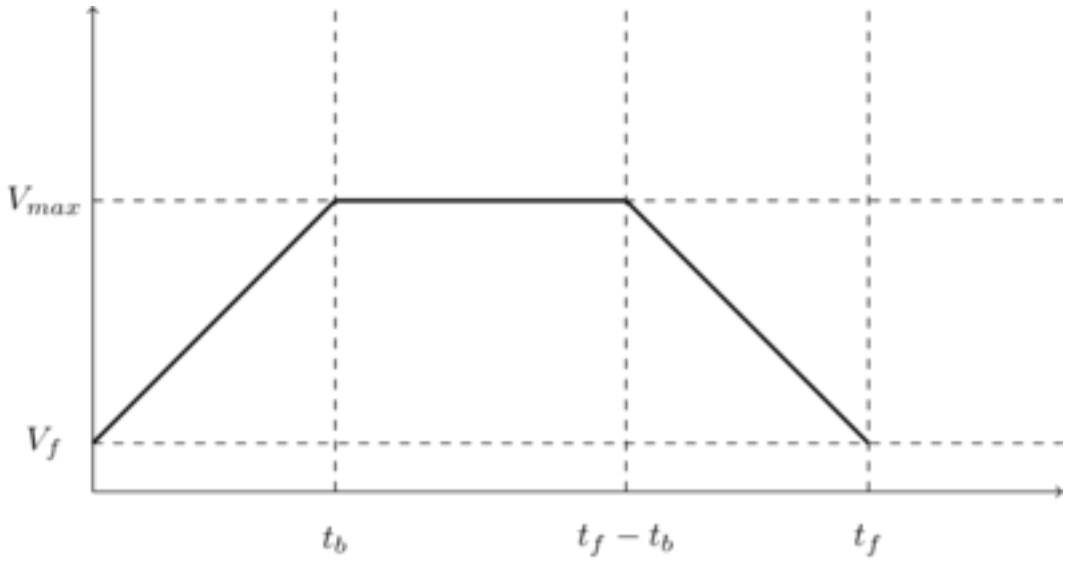
1 Time Optimal Velocity Profiles

When the maze solver commands that the robot go forward, it can say that it must go forward one or more squares depending on what it knows about the maze. When we don't know what is after the square we pass through, we must be going slow enough to handle any scenario. In other words, there is some V_f that we must reach by the end of our motion. We also begin motions at this speed, since between we arrived where we are we required that we reach V_f to get there. Therefore, we start and end at V_f, and we want to cover some distance d in the fast possible time. To do so, we accelerate at our fixed a until we reach max speed, or until we need to start slowing down (whichever comes first). This gives us a trapezoid shaped velocity profile.

1.1 Going Straight

```
In [20]: %load_ext tikzmagic

The tikzmagic extension is already loaded. To reload it, use:
  %reload_ext tikzmagic


In [21]: %%tikz -s 400,400
         \draw[->] (0,0) -- (10,0);
         \draw[->] (0,0) -- (0,5);

         \draw[line width=1] (0,0.5) -- (2.5,3);
         \draw[line width=1] (2.5,3) -- (5.5,3);
         \draw[line width=1] (5.5,3) -- (8,0.5);
         \draw[dashed] (0,0.5) -- (10,0.5);
         \draw[dashed] (0,3) -- (10,3);
         \draw[dashed] (2.5,0) -- (2.5,5);
         \draw[dashed] (5.5,0) -- (5.5,5);
         \draw[dashed] (8,0) -- (8,5);

         \draw (-0.5, 0.5) node {$V_{f}$};
         \draw (-0.5, 3) node {$V_{max}$};
         \draw (2.5, -0.5) node {$t_b$};
```

```
\draw (5.5, -0.5) node {$t_f-t_b$};
\draw (8, -0.5) node {$t_f$};
```



The time to accelerate from V_f to V_{max} is $t_b = \frac{V-V_f}{a}$. We can substitute this into newtons first equation of motion as follows.

$$d = Vt_b - \frac{1}{2}at_b{}^2 \tag{1}$$
$$= V\Big(\frac{V-V_f}{a}\Big) - \frac{1}{2}a\Big(\frac{V-V_f}{a}\Big)^2 \tag{2}$$
$$= \Big(\frac{V^2 - VV_f}{a}\Big) - \Big(\frac{a(V-V_f)^2}{2a^2}\Big) \tag{3}$$
$$= \Big(\frac{2V^2 - 2VV_f}{2a}\Big) - \Big(\frac{V^2 - 2VV_f + V_f{}^2}{2a}\Big) \tag{4}$$
$$= \frac{2V^2 - 2VV_f - V^2 + 2VV_f - V_f{}^2}{2a} \tag{5}$$
$$d = \frac{V^2 - V_f{}^2}{2a} \tag{6}$$
$$\tag{7}$$

For example, if you're at starting at $V_f = 0.2\frac{m}{s}$, and you're ramping up to $V = 0.5\frac{m}{s}$, and you're acceleration is fixed at the $a = 2\frac{m}{s^2}$, the distance you'll need to do that is $d = \frac{0.5-0.2}{2*2} = 0.075m$

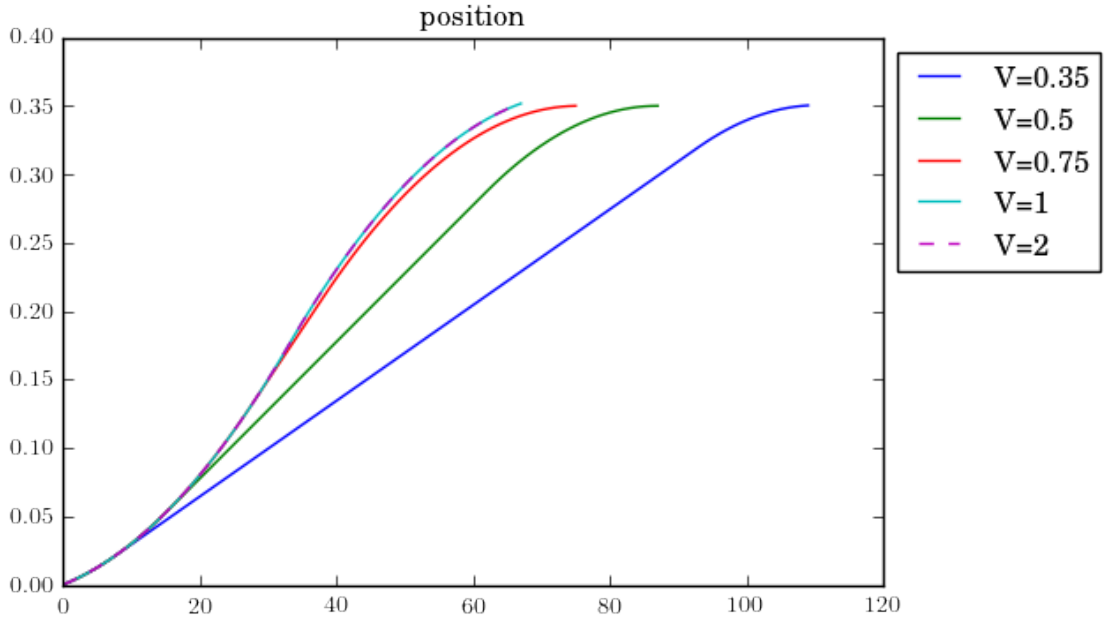
1.2 Code that proves it

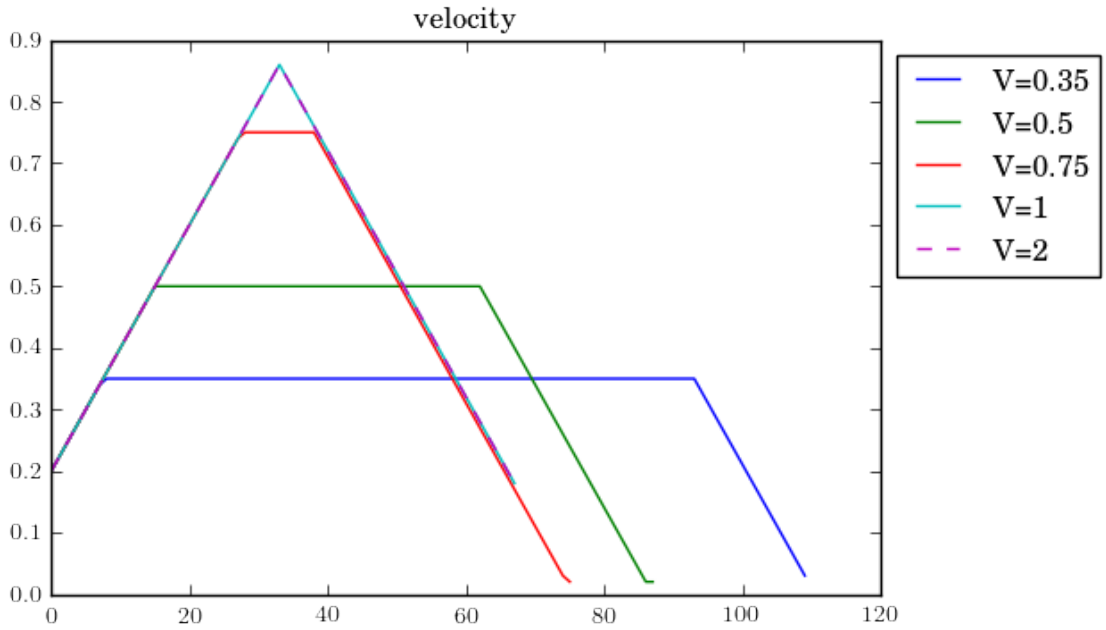
```
In [22]: # dependencies and global setup
         import numpy as np
         import matplotlib.pyplot as plt
```

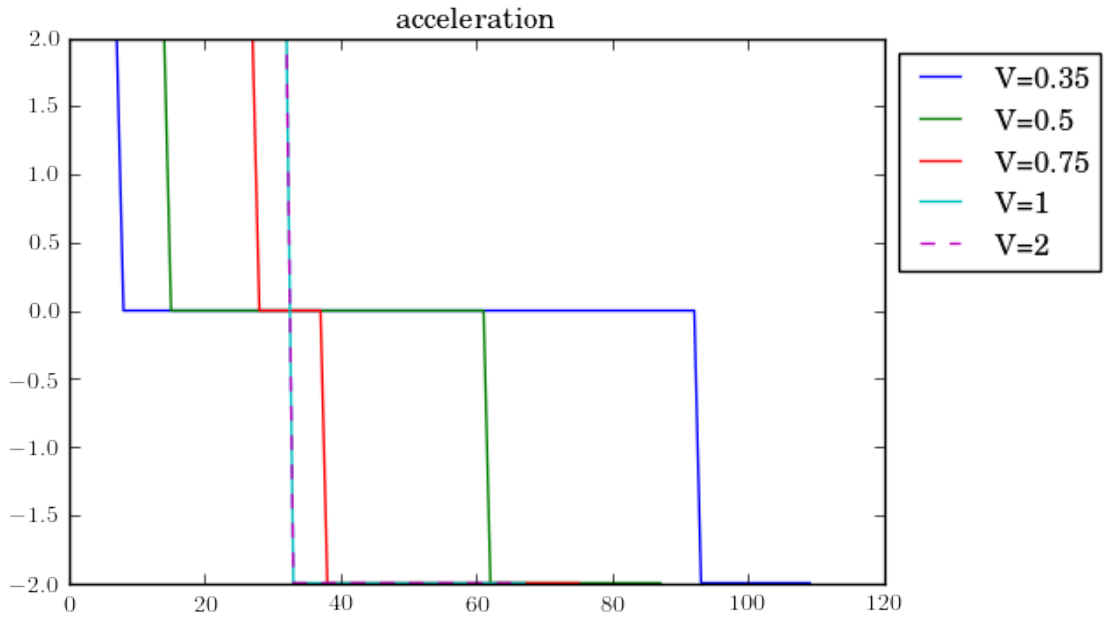
```
        np.set_printoptions(suppress=True, precision=3, linewidth=100)

        LOG_LVL = 2

        def debug(*args):
            if LOG_LVL <= 0:
                print(*args)

        def info(*args):
            if LOG_LVL <= 1:
                print(*args)

        def warning(*args):
            if LOG_LVL <= 2:
                print(*args)

        def log(*args):
            if LOG_LVL < 100:
                print(*args)
```

In [23]:
```
        def profile(V0, Vf, Vmax, d, A, buffer=3e-3):
            v = V0
            x = 0
            a = A
            vs = [v]
            xs = [x]
            a_s = [a]

            dt = 0.01
            while x < d:
                x = x + v*dt + a*dt*dt/2.0
                v = v + a*dt
                ramp_d = (v*v+ - Vf*Vf) / (2.0*A)
                if (d-x) < ramp_d + buffer:
                    a = -A
                elif v < Vmax:
                    a = A
                else:
                    a = 0

                if v > Vmax:
                    v = Vmax
                elif v < Vf:
                    v = Vf

                xs.append(x)
                vs.append(v)
                a_s.append(a)
```

```
    return xs, vs, a_s

def graph(title, idx):
    plt.figure()
    plt.title(title)
    Vs = [0.35, 0.5, 0.75, 1, 2]
    Vf = 0.02
    V0 = 0.2
    d = 0.35
    a = 2
    for V in Vs:
        results  = profile(V0, Vf, V, d, a)
        vs = results[1]
        if V == 2: # make V=2 dashed so we can see it over V=1
            plt.plot(results[idx], label='V={}'.format(V), linestyle='dashed')
        else:
            plt.plot(results[idx], label='V={}'.format(V))
        plt.legend(bbox_to_anchor=(1, 1), loc=2)

graph("position", 0)
graph("velocity", 1)
graph("acceleration", 2)
plt.show()
```







1.3 General Form Trajectory Planning

Let's start out with a generating trajectories that are not time optimal, but rely on specifying the final time v_f. For smartmouse, our state space is $[x, y, \theta]$, and a turn can be defined as starting at a

point $[x_0, y_0, \theta_0]$ and going to $[x_f, y_f, \theta_0]$. Of course, we also want to specify the velocities at these point, $[\dot{x}_0, \dot{y}_0, \dot{\theta}_0]$ and $[\dot{x}_f, \dot{y}_f, \dot{\theta}_f]$. We have four constraints, so if we want to fit a smooth polynomial to those points we need a 4th order polynomial.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$
$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2$$

If we sub in our constraints, we get the following system of equations.

$$q(0) = a_0 \tag{8}$$
$$\dot{q}(0) = a_1 \tag{9}$$
$$q(t_f) = a_0 + a_1 t_f + a_2 {t_f}^2 + a_3 {t_f}^3 \tag{10}$$
$$\dot{q}(t_f) = a_1 + 2a_2 t_f + 3a_3 {t_f}^2 \tag{11}$$
$$\tag{12}$$

In matrix form that looks like:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q(0) \\ \dot{q}(0) \\ q(t_f) \\ \dot{q}(t_f) \end{bmatrix} \tag{13}$$

It can be shown that the matrix on the left is invertable, so long as $t_f - t_0 > 0$. So we can invert and solve this equation and get all the a coefficients. We can then use this polynomial to generate the $q(t)$ and $\dot{q}(t)$ -- our trajectory.

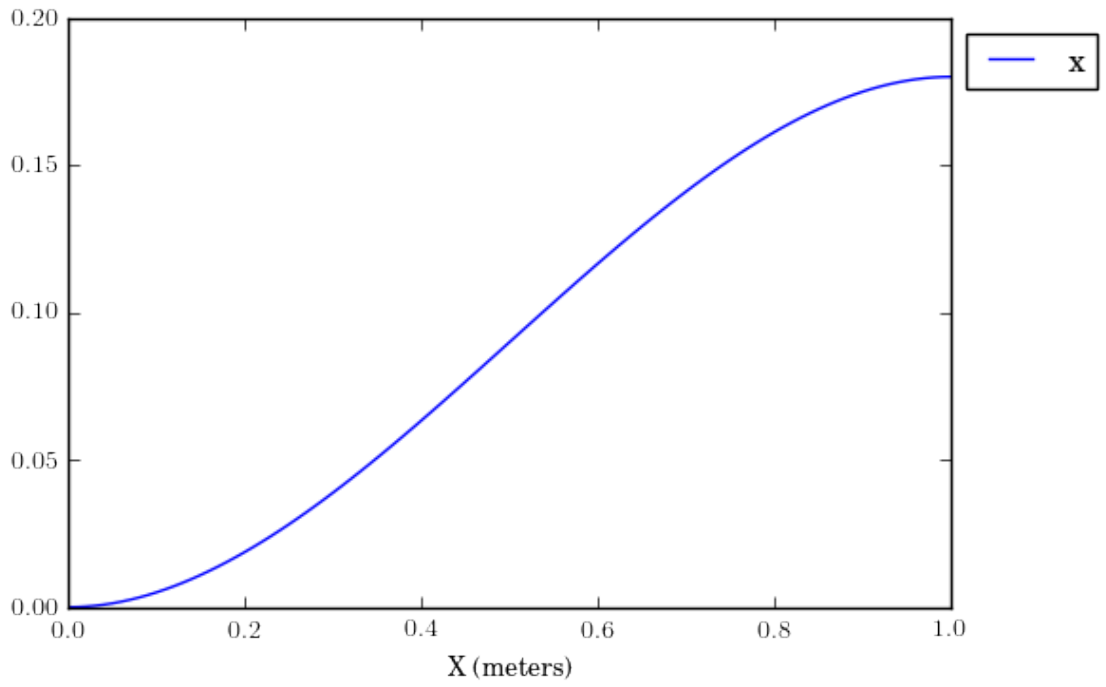
```
In [24]: def simple_traj_solve(q_0, q_f, q_dot_0, q_dot_t_f, t_f):
             # Example: you are a point in space (one dimension) go from rest at the origin to a
             q_0 = np.array([0])
             q_dot_0 = np.array([0])
             q_t_f = np.array([0.18])
             q_dot_t_f = np.array([0])

             b = np.array([q_0, q_dot_0, q_t_f, q_dot_t_f])
             a = np.array([[1,0,0,0],[0,1,0,0],[1, t_f, pow(t_f,2),pow(t_f,3)],[0,1,2*t_f,3*pow(
             log(a, b)
             coeff = np.linalg.solve(a, b)
             log(coeff)

             return coeff

         simple_traj_info = (0, 0, 0.18, 0, 1)
         simple_traj_coeff = simple_traj_solve(*simple_traj_info)
```

```
[[1 0 0 0]
 [0 1 0 0]
 [1 1 1 1]
```

```
 [0 1 2 3]] [[ 0.  ]
 [ 0.  ]
 [ 0.18]
 [ 0.  ]]
[[ 0.  ]
 [ 0.  ]
 [ 0.54]
 [-0.36]]
```

Here you can see that the resulting coeffictions are $a_0 = 0$, $a_1 = 0$, $a_2 = 0.54$, $a_0 = -0.36$. Intuitively, this says that we're going to have positive acceleration, but our acceleration is going to slow down over time. Let's graph it!

```
In [25]: def simple_traj_plot(coeff, t_f):
             dt = 0.01
             ts = np.array([[1, t, pow(t,2), pow(t,3)] for t in np.arange(0, t_f+dt,  dt)])
             qs = ts@coeff
             plt.plot(ts[:,1], qs, label="x")
             plt.xlabel("time (seconds)")
             plt.xlabel("X (meters)")
             plt.legend(bbox_to_anchor=(1,1), loc=2)
             plt.show()

         simple_traj_plot(simple_traj_coeff, simple_traj_info[-1])
```



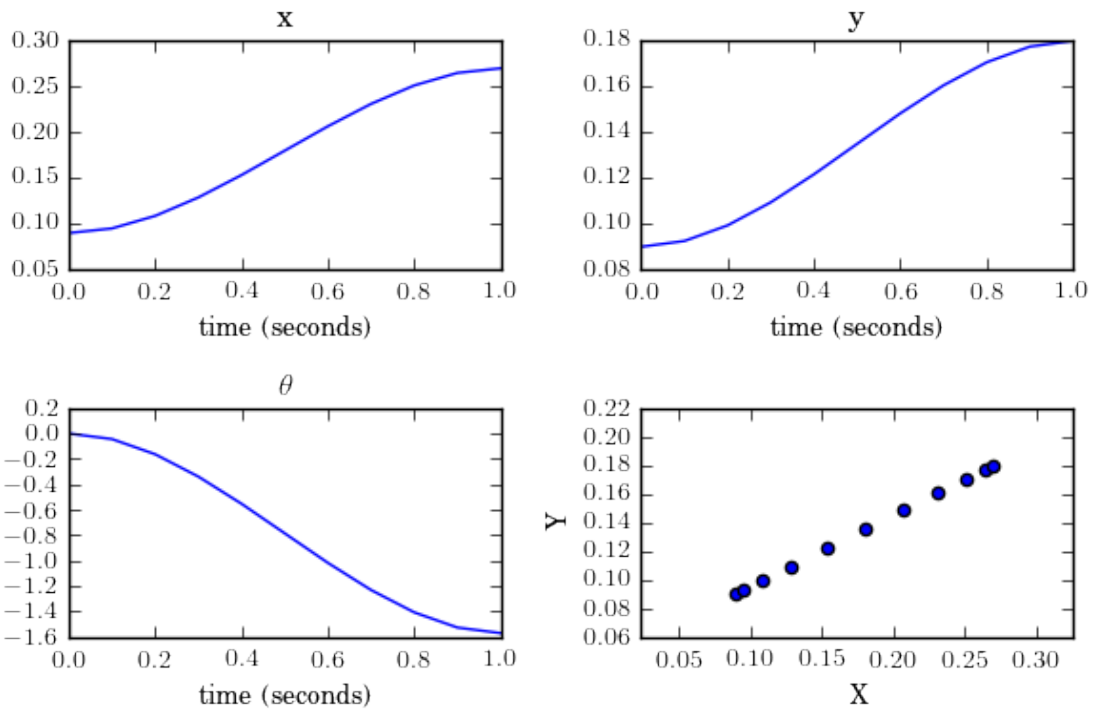
oooooooooh so pretty

Let's try another example, now with our full state space of $[x, y, \theta]$.

```
In [26]: def no_dynamics():
             # In this example, we go from (0.18, 0.09, 0) to (0.27,0.18, -1.5707). Our starting
             q_0 = np.array([0.09,0.09,0])
             q_dot_0 = np.array([0,0,0])
             q_f = np.array([0.27,0.18,-1.5707])
             q_dot_f = np.array([0,0,0])
             t_f = 1

             b = np.array([q_0, q_dot_0, q_f, q_dot_f])
             a = np.array([[1,0,0,0],[0,1,0,0],[1, t_f, pow(t_f,2),pow(t_f,3)],[0,1,2*t_f,3*pow(
             coeff = np.linalg.solve(a, b)
             log(coeff)

             dt = 0.1
             ts = np.array([[1, t, pow(t,2), pow(t,3)] for t in np.arange(0, t_f+dt,  dt)])
             qs = ts@coeff

             plt.rc('text', usetex=True)
             plt.rc('font', family='serif')
             plt.gca().set_adjustable("box")
             plt.subplot(221)
             plt.plot(ts[:,1], qs[:,0])
             plt.xlabel("time (seconds)")
             plt.title("x")
             plt.subplot(222)
             plt.plot(ts[:,1], qs[:,1])
             plt.xlabel("time (seconds)")
             plt.title("y")
             plt.subplot(223)
             plt.plot(ts[:,1], qs[:,2])
             plt.xlabel("time (seconds)")
             plt.title(r"$\theta$")
             plt.subplot(224)
             plt.scatter(qs[:,0], qs[:,1])
             plt.axis('equal')
             plt.xlabel("X")
             plt.ylabel("Y")
             plt.tight_layout()
             plt.show()

         no_dynamics()
```

```
[[ 0.09   0.09   0.   ]
 [ 0.     0.     0.   ]
 [ 0.54   0.27  -4.712]
```

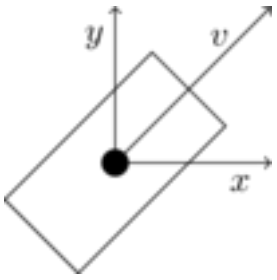
```
[-0.36  -0.18   3.141]]
```



Well, they are smooth, but these are not possible to execute! The robot cannot simply translate sideways.

2 Trajectory Planning With a Simple Dynamics Model

```
In [27]: %%tikz -s 100,100

         \draw [rotate around={-45:(0,0)}] (-.5,-1) rectangle (0.5,1);
         \filldraw (0,0) circle (0.125);

         \draw [->] (0,0) -- (0,1.5);
         \draw [->] (0,0) -- (1.5,0);
         \draw [->] (0,0) -- (1.5,1.5);
         \draw (1.2, -0.2) node {$x$};
         \draw (-0.2, 1.2) node {$y$};
         \draw (1, 1.2) node {$v$};
```



We need to change our constraints to the system of equations. Specifically, we need our dynamics model. For now, let's assume a simplified car model.

$$\dot{x} = v\cos(\theta)$$
$$\dot{y} = v\sin(\theta)$$

This basically claims that for any instant in time the robot is moving a constant velocity along θ. This isn't very accurate, but let's just start with that since the real dynamics of our robot are more complex.

First we will bring in the constraints from before. We must satisfy specific initial and final positions in $[x, y, \theta]$. I've used new letters for cofficients to avoid confusion.

$$x_0 = c_0 + c_1(0) + c_2(0)^2 + c_3(0)^3 + c_3(0)^4 + c_3(0)^5 \tag{14}$$
$$y_0 = d_0 + d_1(0) + d_2(0)^2 + d_3(0)^3 + d_3(0)^4 + d_3(0)^5 \tag{15}$$
$$x_{t_f} = c_0 + c_1(t_f) + c_2(t_f)^2 + c_3(t_f)^3 + c_3(t_f)^4 + c_3(t_f)^5 \tag{16}$$
$$y_{t_f} = d_0 + d_1(t_f) + d_2(t_f)^2 + d_3(t_f)^3 + d_3(t_f)^4 + d_3(t_f)^5 \tag{17}$$
$$\tag{18}$$

Notice here we have 12 unknowns, $c_0 \ldots c_5$ and $d_0 \ldots d_5$. So we're gonna need more equations for there to be a unique solution. Also notice we haven't defined any constraints related to our dynamics model. That would be a good place to get our other equations!

First, we want to be able to specify initial velocity v_0 and final velocity v_{t_f}. It is easlier to just constrain $\dot{x}_0$, $\dot{y}_0$, $\dot{x}_{t_f}$, $\dot{y}_{t_f}$. So if we want to specify that we start facing $\frac{\pi}{2}$ going 1m/s, we'd just specify $cos(\frac{\pi}{2})$ for $\dot{x}_0$ and $sin(\frac{\pi}{2})$ for $\dot{y}_0$.

$$\dot{x}_0 = c_1 \tag{19}$$
$$\dot{y}_0 = d_1 \tag{20}$$
$$\dot{x}_{t_f} = (0)c_0 + (1)c_1 + 2t_f c_2 + 3{t_f}^2 c_3 + 4{t_f}^3 c_4 + 5{t_f}^4 c_5 \tag{21}$$
$$\dot{y}_{t_f} = (0)d_0 + (1)d_1 + 2t_f d_2 + 3{t_f}^2 d_3 + 4{t_f}^3 d_4 + 5{t_f}^4 d_5 \tag{22}$$

Let's also make sure x and y components obey trigonometry.

$$v\cos(\theta)\sin(\theta) + v\cos(\theta)\sin(\theta) = v\sin(2\theta) \tag{23}$$
$$\dot{x}\sin(\theta) + \dot{y}\sin(\theta) = v\sin(2\theta) \tag{24}$$

We can get two equations out of this by specifying initial and final velocities

$$v_0 \sin(2\theta_0) = \dot{x}_0 \sin(\theta_0) + \dot{y}_0 \cos(\theta_0) \tag{25}$$
$$v_{t_f} \sin(2\theta_{t_f}) = \dot{x}_{t_f} \sin(\theta_{t_f}) + \dot{y}_{t_f} \cos(\theta_{t_f}) \tag{26}$$

We should write out the full form though, to make things in terms of our coefficients.

$$v(0) \sin(2\theta_0) = \left[c_1 + 2(0)c_2 + 3(0)^2 c_3 + 4(0)^3 c_4 + 5(0)^4 c_5\right] \sin(\theta_0) + \left[d_1 + 2(0)d_2 + 3(0)^2 d_3 + 4(0)^3 d_4 + 5(0)^4 d_5\right] \tag{27}$$
$$v(0) \sin(2\theta_0) = \sin(\theta_0)c_1 + \cos(\theta_0)d_1 \tag{28}$$

$$v(t_f) \sin(2\theta_{t_f}) = \left[c_1 + 2(t_f)c_2 + 3(t_f)^2 c_3 + 4(t_f)^3 c_4 \ + 5(t_f)^4 c_5\right] \sin(\theta_{t_f}) + \left[d_1 + 2(t_f)d_2 + 3(t_f)^2 d_3 + 4(t_f)^3 d_4 + 5\right. \tag{29}$$
$$v(t_f) \sin(2\theta_{t_f}) = \sin(\theta_{t_f})c_1 + 2\sin(\theta_{t_f})t_f c_2 + 3\sin(\theta_{t_f}){t_f}^2 c_3 + 4\sin(\theta_{t_f}){t_f}^3 c_4 + 5\sin(\theta_{t_f}){t_f}^4 c_5 + \cos(\theta_{t_f})d_1 + 2\cos(\tag{30}$$
$$\tag{31}$$

The last two equations constrains the robot from moving in any direction other than its heading. Of course it must relate $\dot{x}$ to $\dot{y}$. Still not totally sure how we got this equation so I'm just copying it from some slides.... However you can plug in some example values and check. For instance translating sideways violates this equation: set $\dot{x} = 1, \dot{y} = 0, v = 1, \theta = \frac{\pi}{2}$.

$$v\cos(\theta)\sin(\theta) - v\cos(\theta)\sin(\theta) = 0 \tag{32}$$
$$v\cos(\theta)\sin(\theta) - v\sin(\theta)\cos(\theta) = 0 \tag{33}$$
$$\dot{x}\sin(\theta) - \dot{y}\cos(\theta) = 0 \tag{34}$$

and again written out fully in terms of our coefficients

$$\left[c_1 + 2(0)c_2 + 3(0)^2 c_3 + 4(0)^3 c_4 + 5(0)^4 c_5\right] \sin(\theta_0) - \left[d_1 + 2(0)d_2 + 3(0)^2 d_3 + 4(0)^3 d_4 + 5(0)^4 d_5\right] \cos(\theta_0) = 0 \tag{35}$$
$$\sin(\theta_0)c_1 - \cos(\theta_0)d_1 = 0 \tag{36}$$

$$\left[c_1 + 2(t_f)c_2 + 3(t_f)^2 c_3 + 4(t_f)^3 c_4 + 5(t_f)^4 c_5\right] \sin(\theta_{t_f}) - \left[d_1 + 2(t_f)d_2\right. \tag{37}$$
$$\sin(\theta_{t_f})c_1 + 2\sin(\theta_{t_f})t_f c_2 + 3\sin(\theta_{t_f}){t_f}^2 c_3 + 4\sin(\theta_{t_f}){t_f}^3 c_4 + 5\sin(\theta_{t_f}){t_f}^4 c_5 - \cos(\theta_{t_f})d_1 - 2\cos(\theta_{t_f})t_f d_2 - 3\cos(\theta \tag{38}$$

Ok, that should work. Now let's write it out in matrix form. We use c and s to shorten sin and cos.

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & s(\theta_0) & 0 & 0 & 0 & 0 & 0 & c(\theta_0) & 0 & 0 & 0 \\
0 & s(\theta_0) & 0 & 0 & 0 & 0 & 0 & -c(\theta_0) & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & t & {t_f}^2 & {t_f}^3 & {t_f}^4 & {t_f}^5 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & t_f & {t_f}^2 & {t_f}^3 & {t_f}^4 \\
0 & s(\theta_{t_f}) & 2s(\theta_{t_f})t_f & 3s(\theta_{t_f}){t_f}^2 & 4s(\theta_{t_f}){t_f}^3 & 5s(\theta_{t_f}){t_f}^4 & 0 & c(\theta_{t_f}) & 2c(\theta_{t_f})t_f & 3c(\theta_{t_f}){t_f}^2 & 4c(\theta_{t_f}){t_f}^3 & 5c \\
0 & s(\theta_{t_f}) & 2s(\theta_{t_f})t_f & 3s(\theta_{t_f}){t_f}^2 & 4s(\theta_{t_f}){t_f}^3 & 5s(\theta_{t_f}){t_f}^4 & 0 & -c(\theta_{t_f}) & -2c(\theta_{t_f})t_f & -3c(\theta_{t_f}){t_f}^2 & -4c(\theta_{t_f}){t_f}^3 & -5 \\
0 & 1 & 2t_f & 3{t_f}^2 & 4{t_f}^3 & 5{t_f}^4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2t_f & 3{t_f}^2 & 4{t_f}^3
\end{bmatrix} \tag{39}$$

```
In [28]: # Let's solve this in code like we did before
         def plot_vars(traj_plan):
             dt = 0.001
             T = np.arange(0, traj_plan.get_t_f()+dt, dt)
             xts = np.array([[1, t, pow(t,2), pow(t,3), pow(t,4), pow(t,5), 0, 0, 0, 0, 0, 0] fo
             xdts = np.array([[0, 1, 2*t, 3*pow(t,2), 4*pow(t,3), 5*pow(t,4), 0, 0, 0, 0, 0, 0]
             yts = np.array([[0, 0, 0, 0, 0, 0, 1, t, pow(t,2), pow(t,3), pow(t,4), pow(t,5)] fo
             ydts = np.array([[0, 0, 0, 0, 0, 0, 0, 1, 2*t, 3*pow(t,2), 4*pow(t,3), 5*pow(t,4)]
             xs = xts@traj_plan.get_coeff()
             ys = yts@traj_plan.get_coeff()
             xds = xdts@traj_plan.get_coeff()
             yds = ydts@traj_plan.get_coeff()

             plt.rc('text', usetex=True)
             plt.rc('font', family='serif')
             plt.rc('axes.formatter', useoffset=False)
             plt.figure(figsize=(10, 2.5))

             plt.subplot(141)
             plt.plot(T, xs, linewidth=3)
             plt.xlabel("time (seconds)")
             plt.title("X")

             plt.subplot(142)
             plt.plot(T, ys, linewidth=3, color='r')
             plt.xlabel("time (seconds)")
             plt.title("Y")

             plt.subplot(143)
             plt.plot(T, xds, linewidth=3, color='g')
             plt.xlabel("time (seconds)")
             plt.title("$\dot{x}$")
```

```
        plt.tight_layout()

        plt.subplot(144)
        plt.plot(T,yds, linewidth=3, color='y')
        plt.xlabel("time (seconds)")
        plt.title("$\dot{y}$")
        plt.tight_layout()
        plt.show()

    def plot_traj(traj_plan):
        dt = 0.01
        T = np.arange(0, traj_plan.get_t_f()+dt, dt)
        xts = np.array([[1, t, pow(t,2), pow(t,3), pow(t,4), pow(t,5), 0, 0, 0, 0, 0, 0] fo
        yts = np.array([[0, 0, 0, 0, 0, 0, 1, t, pow(t,2), pow(t,3), pow(t,4), pow(t,5)] fo
        xs = xts@traj_plan.get_coeff()
        ys = yts@traj_plan.get_coeff()
        plot_traj_pts(xs, ys, T)

    def plot_traj_pts(xs, ys, T):
        plt.figure(figsize=(5, 5))
        W = 3
        plt.scatter(xs, ys, marker='.', linewidth=0, c=T)
        plt.xlim(0, W * 0.18)
        plt.ylim(0, W * 0.18)
        plt.xticks(np.arange(W+1)*0.18)
        plt.yticks(np.arange(W+1)*0.18)
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.title("Trajectory")
        plt.suptitle("time goes from blue to red")
        plt.grid(True)

        plt.show()
```

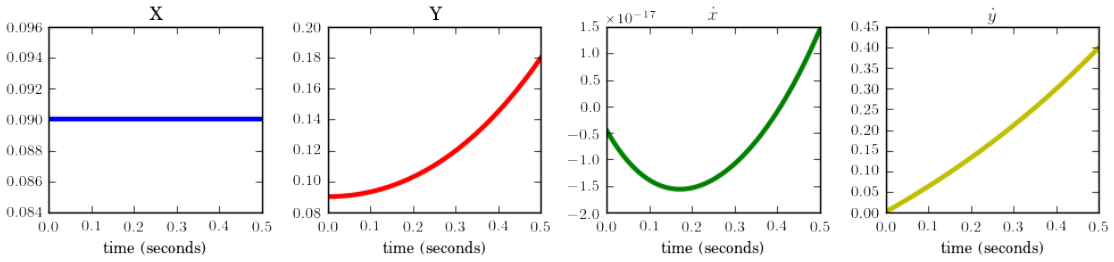
In [29]:
```
    from math import sin, cos, pi
    from collections import namedtuple

    WayPoint = namedtuple('WayPoint', ['x', 'y', 'theta', 'v'])

    class TrajPlan:
        def x_constraint(t):
            return [1, t, pow(t, 2), pow(t, 3), pow(t, 4), pow(t, 5), 0, 0, 0, 0, 0, 0]

        def y_constraint(t):
            return [0, 0, 0, 0, 0, 0, 1, t, pow(t, 2), pow(t, 3), pow(t, 4), pow(t, 5)]

        def non_holonomic_constraint(theta_t, t):
            s_t = sin(theta_t)
```

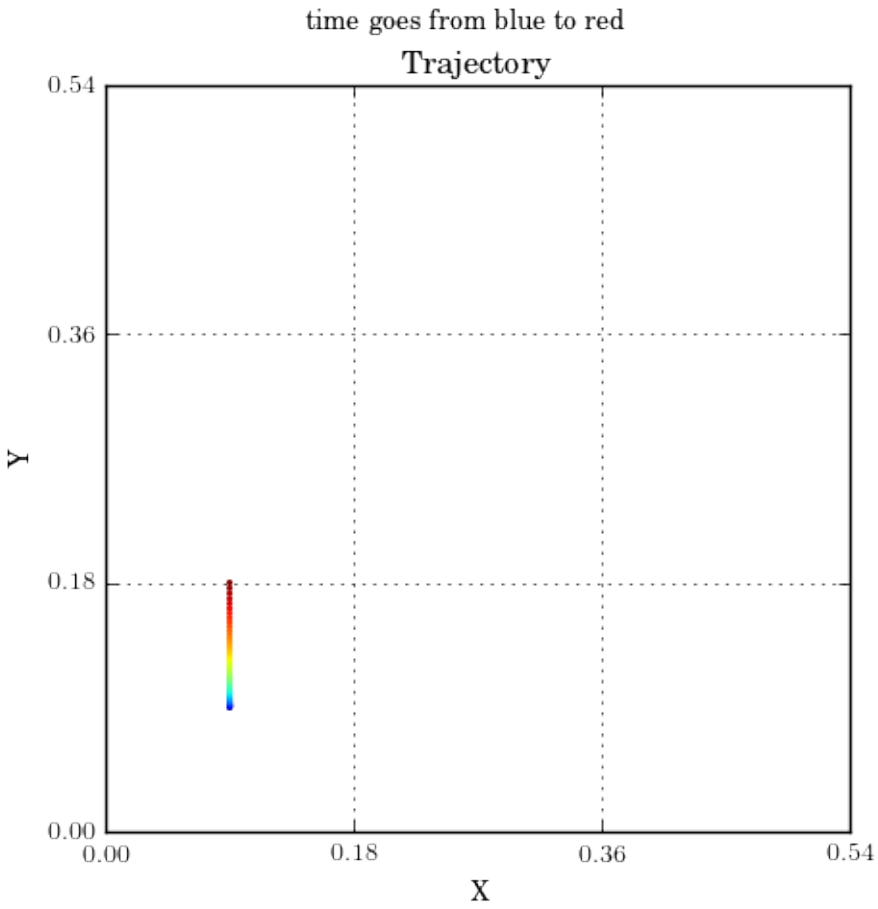
```
        c_t = cos(theta_t)
        t_2 = pow(t, 2)
        t_3 = pow(t, 3)
        t_4 = pow(t, 4)
        return [0, s_t, 2 * s_t * t, 3 * s_t * t_2, 4 * s_t * t_3, 5 * s_t * t_4, 0, c_

    def trig_constraint(theta_t, t):
        s_t = sin(theta_t)
        c_t = cos(theta_t)
        t_2 = pow(t, 2)
        t_3 = pow(t, 3)
        t_4 = pow(t, 4)
        return [0, s_t, 2 * s_t * t, 3 * s_t * t_2, 4 * s_t * t_3, 5 * s_t * t_4, 0, -c

    def x_dot_constraint(t):
        return [0, 1, 2 * t, 3 * pow(t, 2), 4 * pow(t, 3), 5 * pow(t, 4), 0, 0, 0, 0, 0

    def y_dot_constraint(t):
        return [0, 0, 0, 0, 0, 0, 0, 1, 2 * t, 3 * pow(t, 2), 4 * pow(t, 3), 5 * pow(t,

    def solve(self, waypoints):
        # Setup the matrices to match the equation above
        A = []
        b = []

        for t, pt in waypoints:
            A += [TrajPlan.x_constraint(t),
                  TrajPlan.y_constraint(t),
                  TrajPlan.non_holonomic_constraint(pt.theta, t),
                  TrajPlan.trig_constraint(pt.theta, t),
                  TrajPlan.x_dot_constraint(t),
                  TrajPlan.y_dot_constraint(t)]
            b += [pt.x,
                  pt.y,
                  0,
                  pt.v*sin(2*pt.theta),
                  cos(pt.theta)*pt.v,
                  sin(pt.theta)*pt.v]

        A = np.array(A)
        b = np.array(b)
        rank = np.linalg.matrix_rank(A)

        if rank == A.shape[1]:
            if A.shape[0] == A.shape[1]:
                coeff = np.linalg.solve(A, b)
            else:
                warning("not square, using least squares.".format(A.shape))
```

```
                coeff, resid, rank, s = np.linalg.lstsq(A, b)
            else:
                warning("Ranks don't match! {} equations {} variables, using least squares"
                coeff, resid, rank, s = np.linalg.lstsq(A, b)

            debug("rank {}".format(rank))
            debug("A: \n{}".format(A))
            debug("coeff: \n{}".format(coeff))
            error = np.sum(np.power(A@coeff - b, 2))
            if error > 1e-10:
                info("These two vectors should be equal! But there is error.")
                info("b is: \n{}".format(b))
                info("A@coeff is: \n{}".format(A@coeff))
            info("RMS Error of solution to equations")
            info(error)

            self.coeff = coeff
            self.waypoints = waypoints

        def get_coeff(self):
            return self.coeff

        def get_t_f(self):
            return self.waypoints[-1][0]
```

2.1 Example Plots

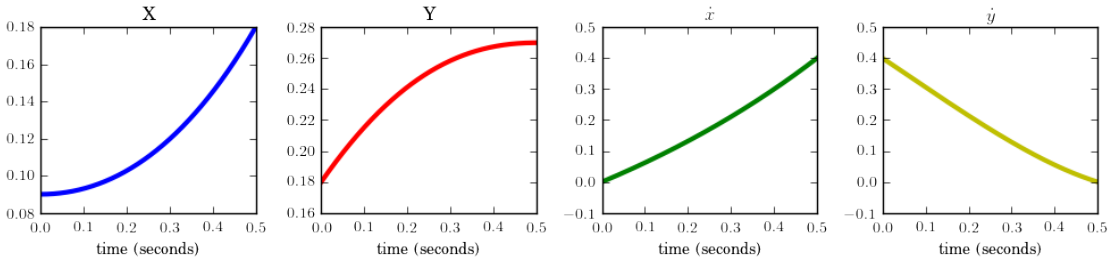
```
In [30]: # forward 1 cell, start from rest, end at 40cm/s, do it in .5 seconds
         LOG_LVL = 5
         fwd_1 = TrajPlan()
         fwd_1.solve([(0, WayPoint(0.09, 0.09, pi/2, 0)), (0.5, WayPoint(0.09, 0.18, pi/2, 0.4))
         plot_vars(fwd_1)
         plot_traj(fwd_1)
```

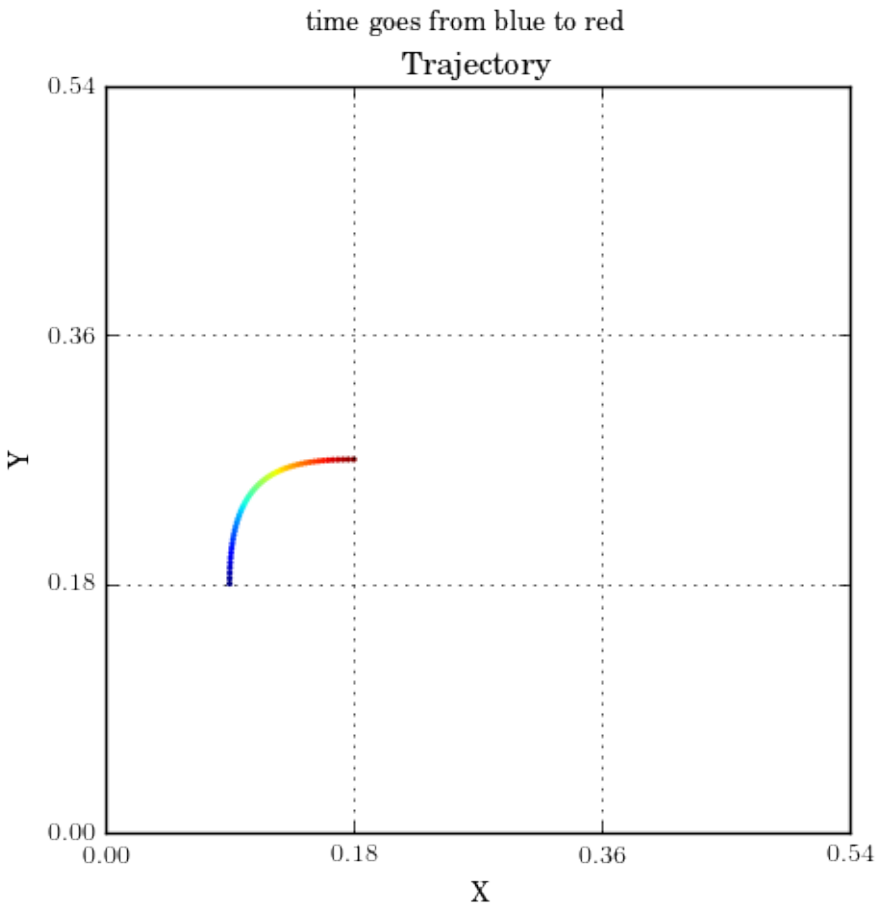




In [31]:
```
# continue by turning right 90 degrees
LOG_LVL = 1
turn_right = TrajPlan()
turn_right.solve([(0, WayPoint(0.09, 0.18, pi/2, 0.4)), (0.5, WayPoint(0.18, 0.27, 0, 0
plot_vars(turn_right)
plot_traj(turn_right)
```

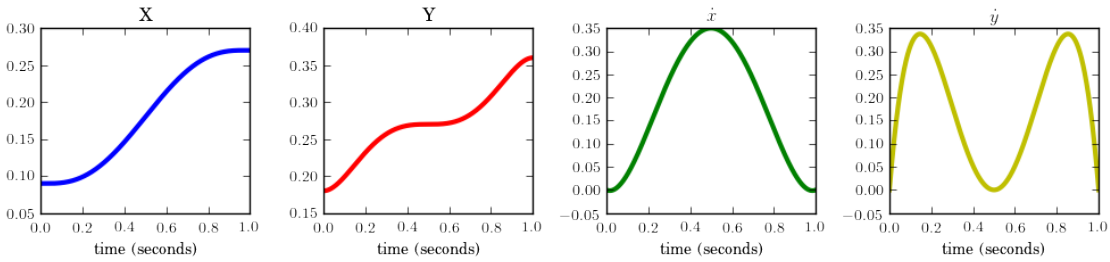
```
Ranks don't match! 8 equations 12 variables, using least squares
RMS Error of solution to equations
3.90120475645e-31
```

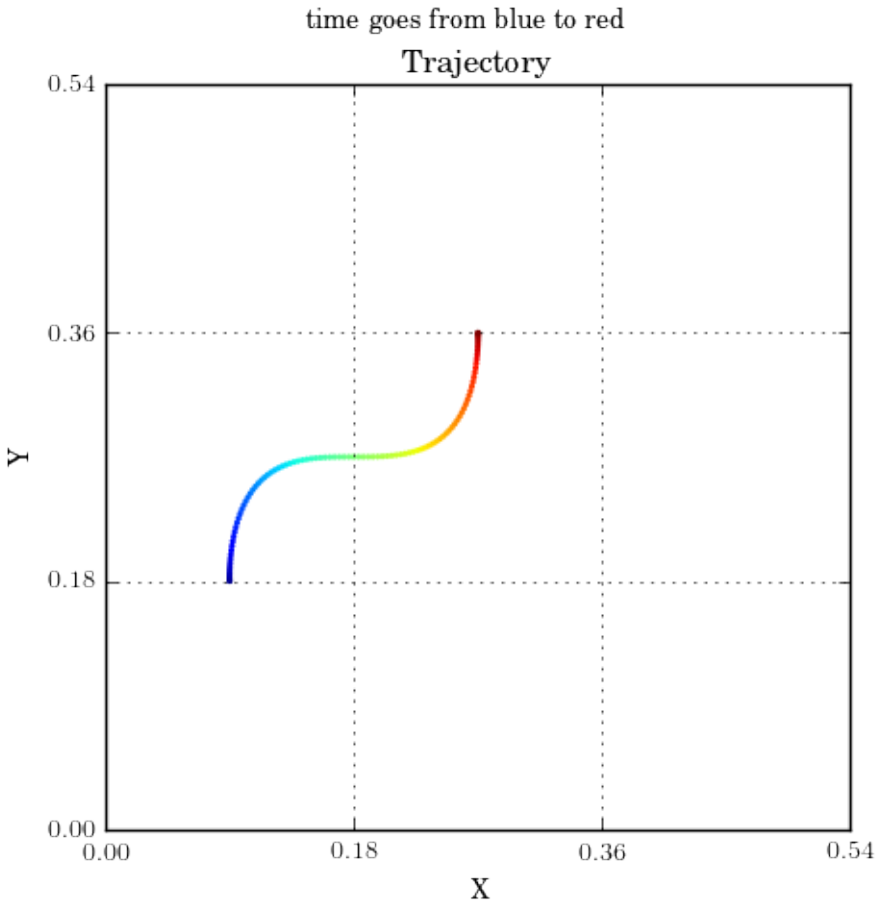




In [32]:
```
# 3 waypoints!
LOG_LVL = 1
turn_right = TrajPlan()
turn_right.solve([(0, WayPoint(0.09, 0.18, pi/2, 0.0)), (0.5, WayPoint(0.18, 0.27, 0, 0
plot_vars(turn_right)
plot_traj(turn_right)
```

```
not square, using least squares.
RMS Error of solution to equations
2.2370801519e-29
```





Note for this system of equations with 3 waypoints, there is no solution. However, the error of the solution found is very small.

3 Trajectory Following

Now that we have a trajectory, we want to design a controller that will follow it as closely as possible. To do this, I'm just going to do a proportional controller. Later we will design an optimal controller. We want to make sure the robot is on the path, facing along the path, and going the right speed. When all of these are true the change in speed should be zero. Let's come up with an equation to relate current pose and velocity to the desired pose and velocity. Let our outputs be the linear velocity v and the rotational velocity w.

$$w = w_d + d * P_1 + (\theta_d - \theta)P_2$$

$$v = v_d + l * P_3$$

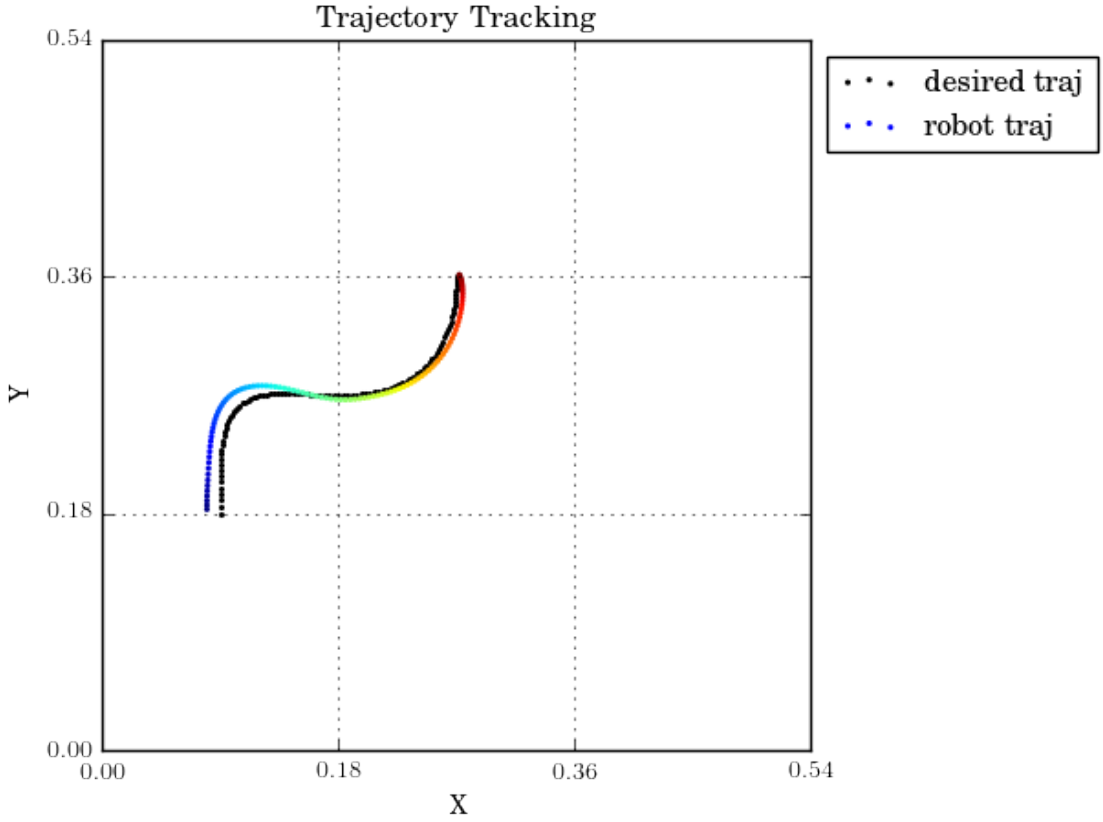
where v_d is desired velocity, θ_d is the desired angle, d is signed distance to the planned trajectory (to the right of the plan is positive), v_d and w_d are the desired velocities of the robot, and P_1, P_2, and P_3 are constants. Essentially what we're saying with the first equation is that when you're far off the trajectory you need to turn harder to get back on to it, but you also need to be aligned with it. The second equation says if you're lagging behind your plan speed up, or slow down if you're overshooting.

In [33]:
```
from math import atan2

LOG_LVL = 5
def simulate(robot_q_0, waypoints, P_1, P_2, P_3, P_4):
    traj = TrajPlan()
    traj.solve(waypoints)

    dt = 0.01
    robot_x = robot_q_0[0]
    robot_y = robot_q_0[1]
    robot_theta = robot_q_0[2]
    robot_v = robot_q_0[3]
    robot_w = robot_q_0[4]
    actual_robot_v = robot_q_0[3]
    actual_robot_w = robot_q_0[4]
    v_acc = 2 * dt
    TRACK_WIDTH = 0.0633
    w_acc = v_acc / (TRACK_WIDTH/2)
    T = np.arange(0, traj.get_t_f()+dt, dt)
    x_des_list = []
    y_des_list = []
    robot_x_list = []
    robot_y_list = []
    for t in T:
        x_des = [1, t, pow(t,2), pow(t,3), pow(t,4), pow(t,5), 0, 0, 0, 0, 0, 0] @ traj
        dx_des = [0, 1, 2*t, 3*pow(t,2), 4*pow(t,3), 5*pow(t,4), 0, 0, 0, 0, 0, 0] @ tr
        ddx_des = [0, 0, 0, 0, 0, 0, 0, 0, 2, 6*t, 12*pow(t,2), 20*pow(t,3)] @ traj.get
        y_des = [0, 0, 0, 0, 0, 0, 1, t, pow(t,2), pow(t,3), pow(t,4), pow(t,5)] @ traj
        dy_des = [0, 0, 0, 0, 0, 0, 0, 1, 2*t, 3*pow(t,2), 4*pow(t,3), 5*pow(t,4)] @ tr
        ddy_des = [0, 0, 0, 0, 0, 0, 0, 0, 2, 6*t, 12*pow(t,2), 20*pow(t,3)] @ traj.get
        theta_des = atan2(dy_des, dx_des)
```

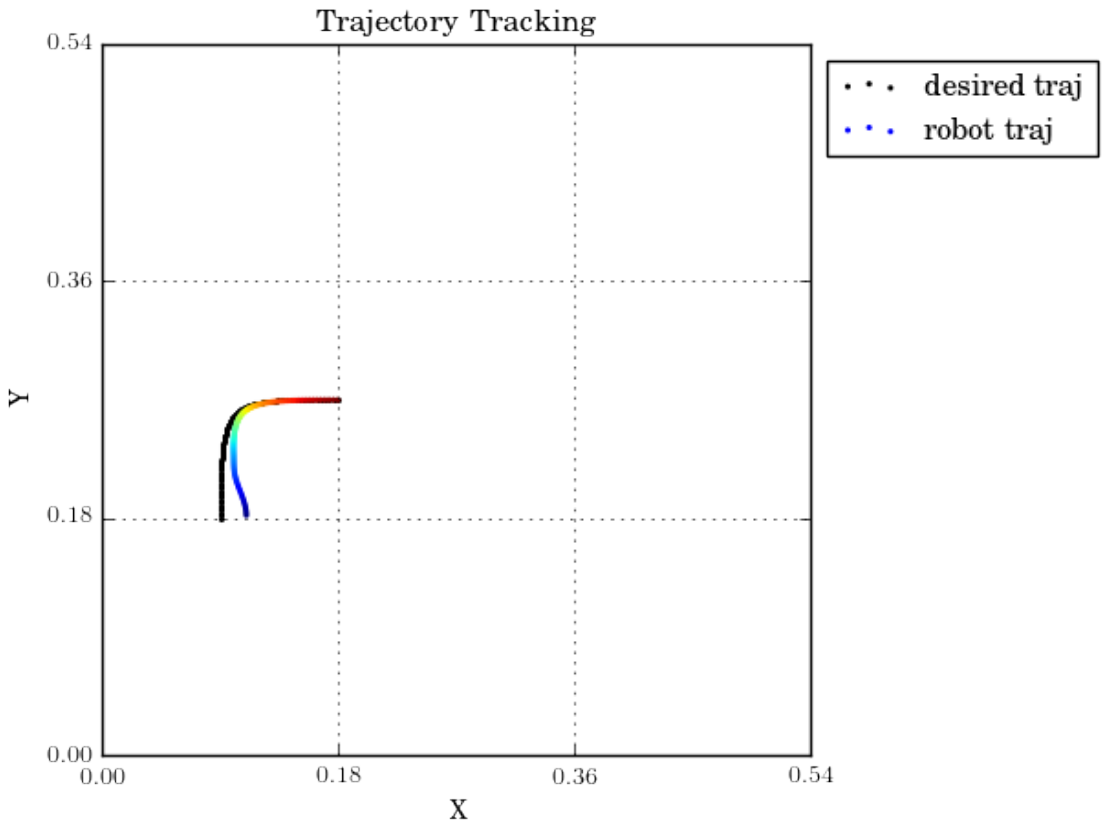
```
            v_des = cos(theta_des)*dx_des + sin(theta_des)*dy_des
            w_des = 1/v_des * (ddy_des*cos(theta_des) - ddx_des*sin(theta_des));

            # simple Dubin's Car forward kinematics
            robot_x += cos(robot_theta) * actual_robot_v * dt
            robot_y += sin(robot_theta) * actual_robot_v * dt
            robot_theta += actual_robot_w * dt

            # control
            euclidian_error = np.sqrt(pow(x_des - robot_x, 2) + pow(y_des - robot_y, 2))
            transformed_x = (robot_x - x_des) * cos(-theta_des) + (robot_y - y_des) * -sin(
            transformed_y = (robot_x - x_des) * sin(-theta_des) + (robot_y - y_des) * cos(-
            right_of_traj = transformed_y < 0
            signed_euclidian_error = euclidian_error if right_of_traj else -euclidian_error
            lag_error = -transformed_x
            robot_w = w_des + signed_euclidian_error * P_1 + (theta_des - robot_theta) * P_
            robot_v = v_des + lag_error * P_3

            # simple acceleration model
            if robot_v < actual_robot_v:
                actual_robot_v = max(robot_v, actual_robot_v - v_acc)
            elif robot_v > actual_robot_v:
                actual_robot_v = min(robot_v, actual_robot_v + v_acc)
            if robot_w < actual_robot_w:
                actual_robot_w = max(robot_w, actual_robot_w - w_acc)
            elif robot_w > actual_robot_w:
                actual_robot_w = min(robot_w, actual_robot_w + w_acc)

            x_des_list.append(x_des)
            y_des_list.append(y_des)
            robot_x_list.append(robot_x)
            robot_y_list.append(robot_y)

        plt.figure(figsize=(5, 5))
        W = 3
        plt.scatter(x_des_list, y_des_list, marker='.', linewidth=0, c='black', label='desi
        plt.scatter(robot_x_list, robot_y_list, marker='.', linewidth=0, c=T, label='robot
        plt.xlim(0, W * 0.18)
        plt.ylim(0, W * 0.18)
        plt.xticks(np.arange(W+1)*0.18)
        plt.yticks(np.arange(W+1)*0.18)
        plt.grid(True)
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.title("Trajectory Tracking")
        plt.legend(bbox_to_anchor=(1,1), loc=2)
```

In [34]:
```
test_P_1=500
```

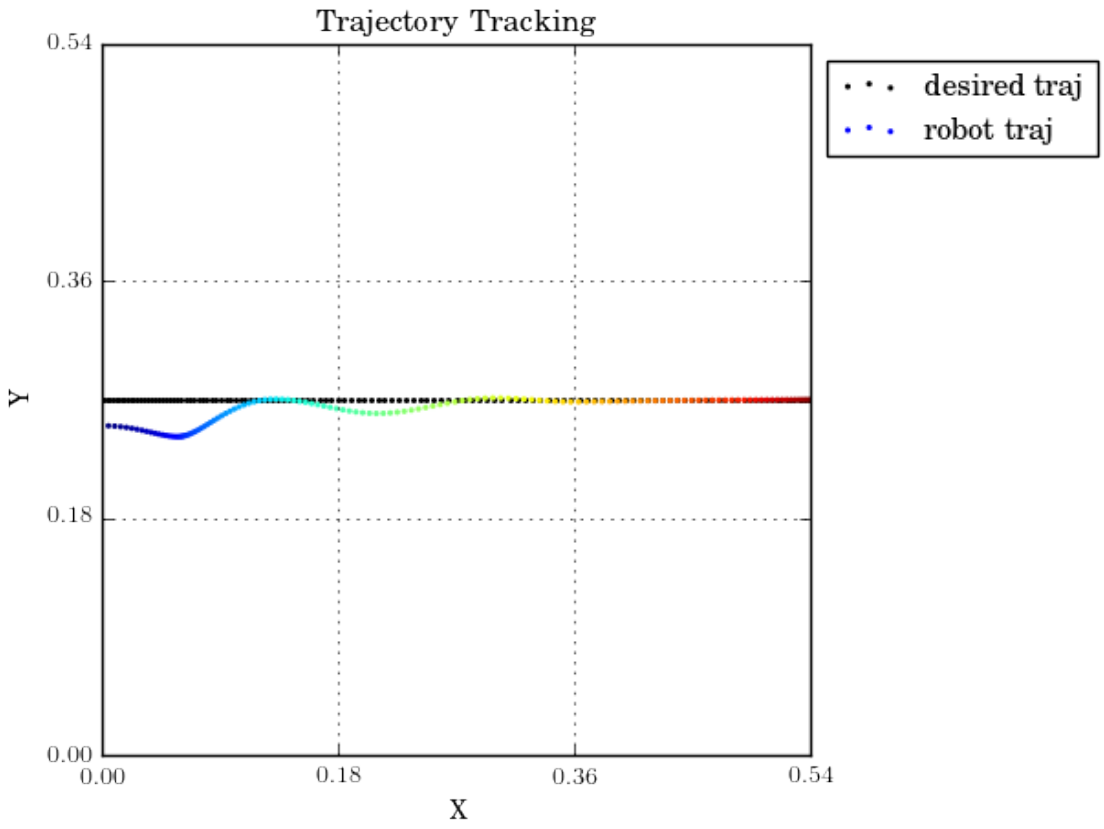
```
test_P_2=50
test_P_3=10
test_P_4=0.00
robot_q_0 = (0.08, 0.18, pi/2, 0.3, 0)
traj = [(0, WayPoint(0.09, 0.18, pi/2, 0.5)), (0.5, WayPoint(0.18, 0.27, 0, 0.35)), (1,
simulate(robot_q_0, traj, test_P_1, test_P_2, test_P_3, test_P_4)
plt.show()
```



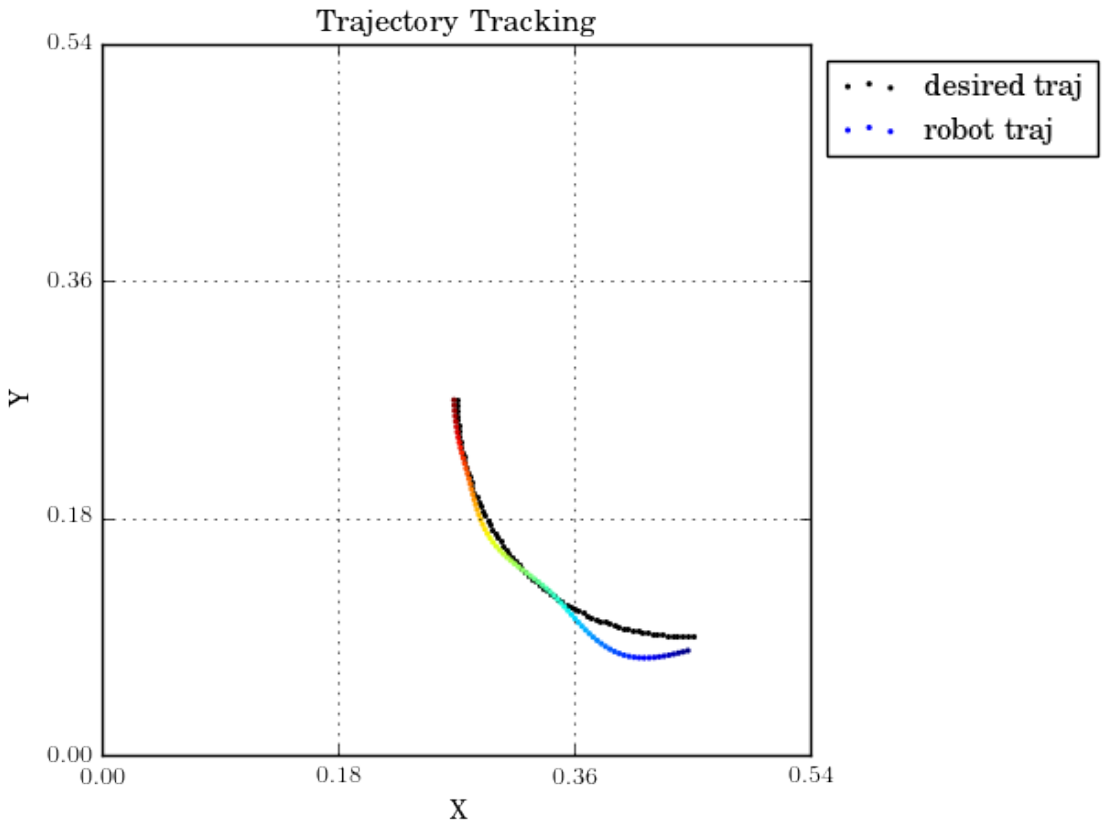
In [35]:
```
robot_q_0 = (0.11, 0.18, pi/2, 0.2, 5)
traj = [(0, WayPoint(0.09, 0.18, pi/2, 0.2)), (1, WayPoint(0.18, 0.27, 0, 0.35))]
simulate(robot_q_0, traj, test_P_1, test_P_2, test_P_3, test_P_4)
plt.show()
```



In [36]:
```
robot_q_0 = (0.0, 0.25, 0, 0.5, -5)
traj = [(0, WayPoint(0.0, 0.27, 0, 0.2)), (1.25, WayPoint(0.54, 0.27, 0, 0.2))]
simulate(robot_q_0, traj, test_P_1, test_P_2, test_P_3, test_P_4)
plt.show()
```



In [37]:
```
robot_q_0 = (0.45, 0.08, pi+0.25, 0.4, 0)
traj = [(0, WayPoint(0.45, 0.09, pi, 0.4)), (0.75, WayPoint(0.27, 0.27, pi/2, 0.4))]
simulate(robot_q_0, traj, test_P_1, test_P_2, test_P_3, test_P_4)
plt.show()
```



Note: The code above has a bug if I use `-pi` instead of `pi` in `robot_q_0`

4 LQR - The Optimal Controller

4.1 Overview of the Steps:

4.1.1 1. Write out the non-linear dynamics $\dot{x} = f(x, u)$

Here we are interested in the full blown system dynamics of the actual smartmouse robot. The forward kinematics, which depend on the current state x, y, and θ and the velocity inputs of the wheels v_l, and v_r are as follows. In the general case where the two wheels have different velocities, we have this:

$$R = \frac{W(v_l + v_r)}{2(v_r - v_l)} \qquad \text{radius of turn} \tag{40}$$

$$\theta \leftarrow \theta + \frac{v_l}{R - \frac{W}{2}}\Delta t \tag{41}$$

$$x \leftarrow x - R\left(\sin\left(\frac{v_r - v_l}{W}\Delta t - \theta\right) + \sin\theta \right) \tag{42}$$

$$y \leftarrow y - R\left(\cos\left(\frac{v_r - v_l}{W}\Delta t - \theta\right) - \cos\theta \right) \tag{43}$$

And in the special case where we're going perfectly straight:

$$\theta \leftarrow \theta \tag{44}$$

$$x \leftarrow x + v\Delta t\cos(\theta) \tag{45}$$

$$y \leftarrow y + v\Delta t\sin(\theta) \tag{46}$$

$$\tag{47}$$

We can take these equations and write them in the form of $\dot{x} = f(x, u)$. Confusingly, x here is the full state vector $[x, y, \theta]$. Most controls texts simply use x, so I'm sticking with that.

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{48}$$

$$= \begin{bmatrix} -R\left(\sin\left(\frac{v_r - v_l}{W}\Delta t - \theta\right) + \sin\theta \right) \\ -R\left(\cos\left(\frac{v_r - v_l}{W}\Delta t - \theta\right) - \cos\theta \right) \\ \frac{v_l}{R - \frac{W}{2}}\Delta t \end{bmatrix} \tag{49}$$

4.1.2 2. Identify the equilibrium points $\bar{x}$ where $\dot{x} = f(\bar{x}, \bar{u}) = 0$

Every point $[x, y, \theta]$ is an equilibrium if we input no control, $\bar{u} = 0$. This is good, because it means we can linearize around the current position of the robot no matter where it is.

4.1.3 3. Write the linearized dynamics around $\bar{x}$ as $\dot{x} \approx A\delta_x + B\delta_u$, where $\delta_x = (\bar{x} - x)$ and $\delta_u = (\bar{u} - u)$

To do this, we need the partial derivitive matrixes A and B. But since our state dynamics doesn't depend on our state, only our control, we can ignore the Ax term and just consider B.

$$B = \begin{bmatrix} \frac{\partial f_1}{\partial v_l}\Big|_{\bar{x}\bar{u}} & \frac{\partial f_1}{\partial v_r}\Big|_{\bar{x}\bar{u}} \\ \frac{\partial f_2}{\partial v_l}\Big|_{\bar{x}\bar{u}} & \frac{\partial f_2}{\partial v_r}\Big|_{\bar{x}\bar{u}} \\ \frac{\partial f_3}{\partial v_l}\Big|_{\bar{x}\bar{u}} & \frac{\partial f_3}{\partial v_r}\Big|_{\bar{x}\bar{u}} \end{bmatrix} = \begin{bmatrix} R\cos\left(\frac{(\bar{v}_r - \bar{v}_l)\Delta t}{W} - \theta\right)\frac{\Delta t}{W} & -R\cos\left(\frac{(\bar{v}_r - \bar{v}_l)\Delta t}{W} - \theta\right)\frac{\Delta t}{W} \\ -R\sin\left(\frac{(\bar{v}_r - \bar{v}_l)\Delta t}{W} - \theta\right)\frac{\Delta t}{W} & R\sin\left(\frac{(\bar{v}_r - \bar{v}_l)\Delta t}{W} - \theta\right)\frac{\Delta t}{W} \\ \frac{\Delta t}{R - \frac{W}{2}} & 0 \end{bmatrix}$$

4.1.4 4. Check if our system is controllable by looking at the rank of the controllability matrix $C = [B, AB, A^2B, \ldots, A^{n-1}B]$

We have three state variables so $n = 3$, which means $C = [B, AB, A^2B] = [B, 0, 0].'$

The rank of C is obviously 1 which is a problem since it should be equal to our number of state variables, which is 3. This indicates our system is not controllable.

4.1.5 5. Pick cost parameters Q and R

4.1.6 6. Solve for K given $LQR(A, B, Q, R)$

maybe we compute K once instead of at every time step could be consistent within one motion primitive ### 7. Apply our new controller of the form $u = -Kx$

```
In [38]: from math import atan2
         import scipy.linalg

         # source: http://www.kostasalexis.com/lqr-control.html
         def dlqr(A,B,Q,R):
             """Solve the discrete time lqr controller.


             x[k+1] = A x[k] + B u[k]

             cost = sum x[k].T*Q*x[k] + u[k].T*R*u[k]
             """
             #ref Bertsekas, p.151

             #first, try to solve the ricatti equation
             X = np.matrix(scipy.linalg.solve_discrete_are(A, B, Q, R))

             #compute the LQR gain
             K = np.matrix(scipy.linalg.inv(B.T*X*B+R)*(B.T*X*A))

             eigVals, eigVecs = scipy.linalg.eig(A-B*K)

             return K, X, eigVals

         def follow_plan(plan):
             dxdes = 1
             dydes = 1
             ddxdes = 1
             ddydes = 1
             thetades = atan2(dydes, dxdes)

             vf = dxdes*cos(thetades) + dydes*sin(thetades)
             dthetades = 1/vf*(ddydes*cos(thetades) - ddxdes*sin(thetades))
             wf = dthetades
```

```
A = np.array([[0, 0, -vf*sin(thetades)],
              [0, 0, vf*cos(thetades)],
              [0, 0, 0]])

B = np.array([[cos(thetades), 0],
              [sin(thetades), 0],
              [0, 1]]);
Q= np.eye(3);
R = np.eye(2);

K = dlqr(A, B, Q, R)
u = -K * (xvec - xdes_vec) + np.array([[vf],[wf]]);
```