Path Generation

Alexander Koldy

Forest Hills Robotics League

alexanderkoldy.ak@gmail.com

September 18, 2024

Parameterization

In FTC, we typically move in $\mathbb{R}^2$, i.e., two dimensions x and y. Therefore, our waypoints are often defined as (x, y) pairs. Let's now define two waypoints on our typical (empty) FTC field:

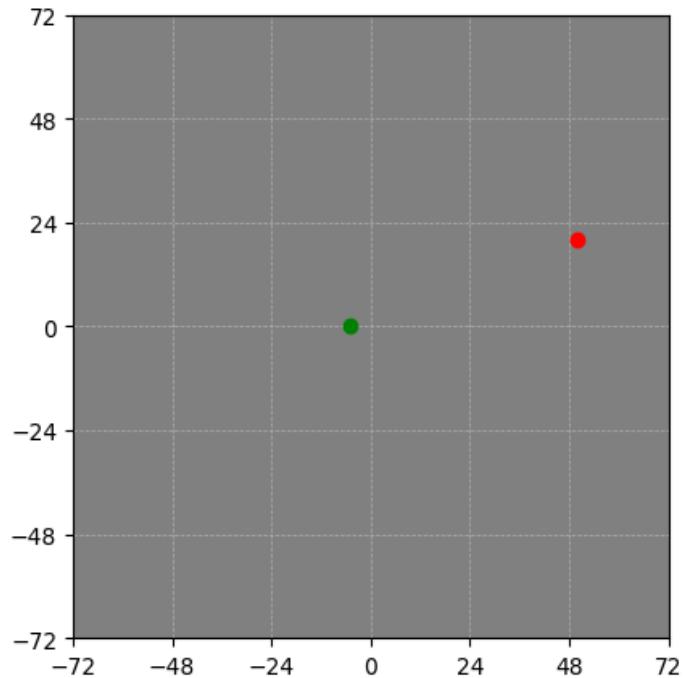
- Let p_0 represent the current position of our robot
- Let p_n represent some other point we'd like the robot to go to

Note: we use $n = 1$ in this case since we only have two points and we start our indexing at 0.

Let's see what $p_0 = (-5.0, 0.0)$ and $p_n = (50.0, 20.0)$ look like on the field (assuming our units are in inches for now).

- The green point represents p_0
- The red point represents p_n

Now, how do we generate waypoints **between** these two points?



Parameterization - lines

Since we currently only have two points (a start and end position), we can use a **line segment** to generate intermediate waypoints. But how do we generate a this line segment? Well, the naive approach would be to use the information about the two points to generate the slope and y-intercept of the line. Let's try this out.

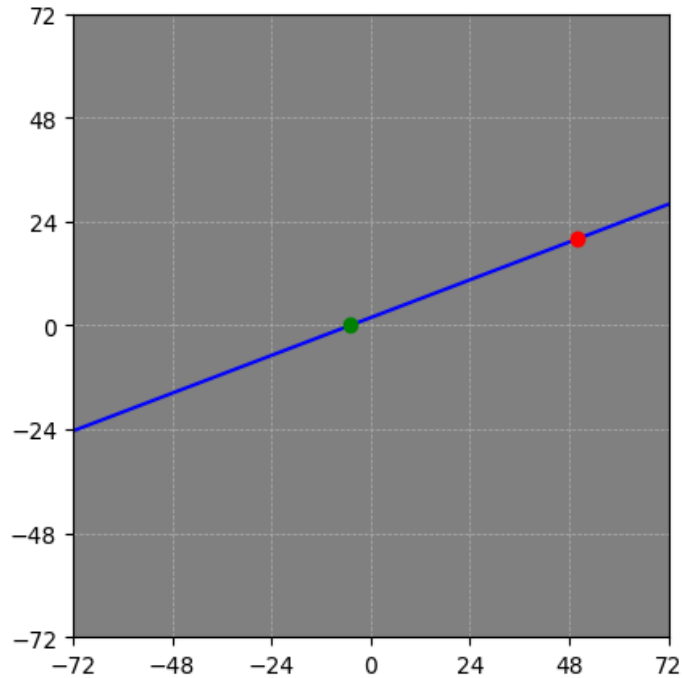
First, we find the slope m between p_n and p_0 as

$$m = \frac{y_n - y_0}{x_n - x_0} = \frac{20.0 - 0.0}{50.0 - (-5.0)} \approx 0.364$$

Next, we find the y-intercept by plugging in the (x, y) values of either point into the line formula $y = mx + b$ and solving for b

$$b = y_n - mx_n = 20.0 - 0.364 \cdot 50.0 = 1.8$$

The blue line shows that we've done our math correctly, however, we only care about the section **between** the waypoints.



Parameterization - lines

Let's now take a different approach to generate the line segment between the lines instead of the line that intersects both points. This way we ensure all waypoints are bounded by the start and end points. We introduce a new variable t and make x and y functions of this variable. If we bound t to $[0, 1]$, i.e., $0 \leq t \leq 1$, we can generate functions for the line segment between p_0 and p_n:

$$x(t) = (1 - t)x_0 + (t)x_n, \ \forall t \in [0, 1]$$
$$y(t) = (1 - t)y_0 + (t)y_n, \ \forall t \in [0, 1]$$

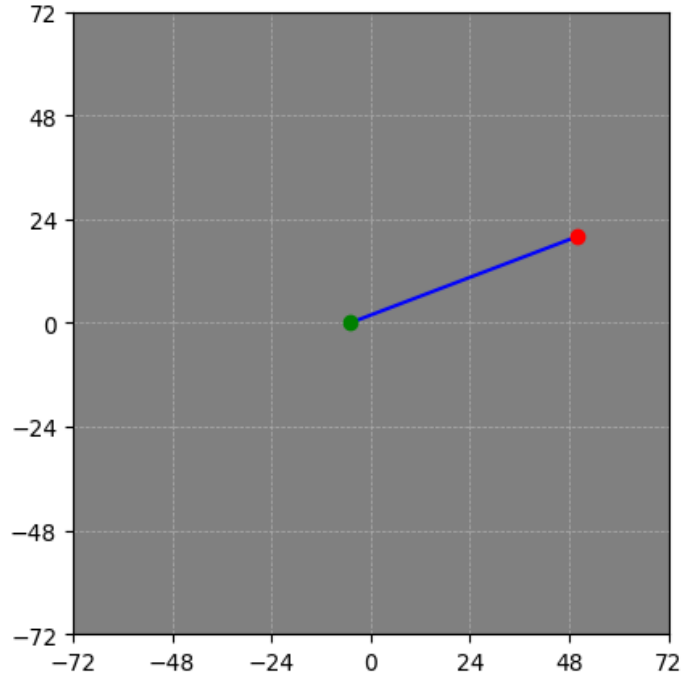
Note: $\forall t \in [0, 1]$ *translates to "for all* t *in* $[0, 1]$*"*

Notice when $t = 0$, $x(0) = x_0$ and $y(0) = y_0$ and when $t = 1$, $x(1) = x_n$ and $y(0) = y_n$. When we parameterize, we will often bound t between 0 and 1 to get clean results like this.

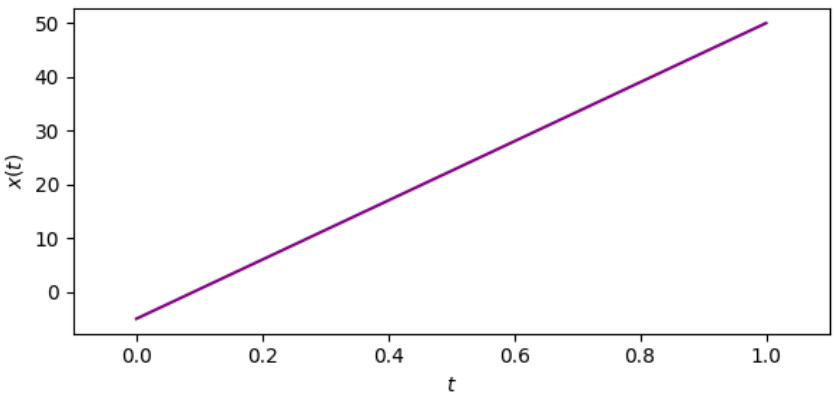
Note: t is often the name given to a parameterization variable, but it does not necessarily have units of time.

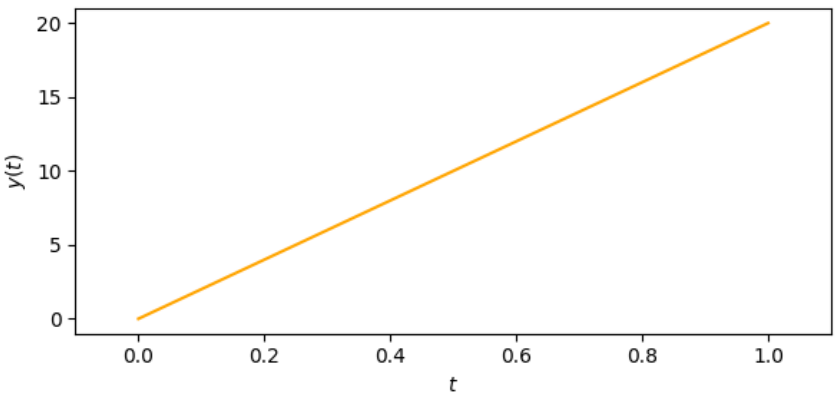
Parameterization - lines

Let's take a look at the result of the previous slide. We now have a line segment running between the start and endpoint!



Moreover, we can see graph $x(t)$ and $y(t)$ to see the individual contribution of each function.





Parameterization - lines

Let's also verify if that this line segment follows the equation of a line. First, solve for t in either $x(t)$ or $y(t)$ (let's choose x):

$$x(t) = (1-t)x_0 + (t)x_n = x_0 + (x_n - x_0)t$$
$$\implies t = \frac{x(t) - x_0}{x_n - x_0}$$

Now, we plug this result into $y(t)$ as follows

$$\begin{aligned} y(t) &= (1-t)y_0 + (t)y_n = y_0 + (y_n - y_0)t \\ &= y_0 + (y_n - y_0)\left(\frac{x(t) - x_0}{x_n - x_0}\right) \\ &= \left(\frac{y_n - y_0}{x_n - x_0}\right)x(t) + \left(-\left(\frac{y_n - y_0}{x_n - x_0}\right)x_0 + y_0\right) \\ &= mx(t) + b \end{aligned}$$

We see that we get back to the equation of a line! If we plug in our values for p_0 and p_n, we get the exact same line we got earlier.

Parameterization - lines

Now that we understand simple parameterizations for lines, let's start to get a little bit more general so we can build up to more advanced movements around the field.

Note: these next few parts will seem redundant and silly, but they will be necessary to build up to spline generation later.

Let's take another look at the line segment parameterization between our two points. When we rearranged $x(t)$ and $y(t)$ in the previous slide, we got

$$\begin{aligned} x(t) &= (1-t)x_0 + (t)x_n \\ &= (x_n - x_0)t + x_0 \\ y(t) &= (1-t)y_0 + (t)y_n \\ &= (y_n - y_0)t + y_0 \end{aligned}$$

Note: we will be dropping the $\forall t \in [0,1]$ notation, but remember that t is still bound!

Parameterization - lines

What we have are actually two **first-order polynomials**. In other words, since t is our variable and each equation *technically* has 1 as the maximum exponent of t, i.e., t^1, we are working with polynomials of the first order. A second-order polynomial, i.e., a quadratic may look something like

$$f(t) = at^2 + bt + c$$

Now, let's pretend we do not know the constants of our parameterization. Therefore, we parameterize the line between our two points with some general first-order polynomials (lines):

$$\ell_x(t) = a_x t + b_x$$
$$\ell_y(t) = a_y t + b_y$$

We will solve for a_x, b_x, a_y and b_y by considering the constraints of our path.

Note: we are essentially redefining $x(t)$ to be $\ell_x(t)$ and $y(t)$ to be $\ell_y(t)$ here. This is to reduce the confusion in notation later down the line.

Parameterization - lines

The set of constraints for a first-order polynomial consist of our **endpoint constraints**. We know that at $\ell_x(0)$ our line must be at x_0, at $\ell_x(1)$ our line must be at x_n, at $\ell_y(0)$ or line must be at y_0 and at $\ell_y(1)$ our line must be at y_n:

$$\ell_x(0) = b_x = x_0$$
$$\ell_x(1) = a_x + b_x = x_n$$
$$\ell_y(0) = b_y = y_0$$
$$\ell_y(1) = a_y + b_y = y_n$$

Remember, our a and b values are **unknown** right now. We can arrange these constraints into matrix format as

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} a_x \\ b_x \\ a_y \\ b_y \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} x_0 \\ x_n \\ y_0 \\ y_n \end{bmatrix}}_{\mathbf{b}}$$

Note: **x** *is referring to the vector of unknowns and* **b** *is referring to the vector of constants in our constraints. Do not get these confused with x and b.*

Parameterization - lines

We now solve for $\mathbf{x}$ by taking the inverse of matrix $\mathbf{A}$ as follows:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Note: recall that $\mathbf{A}$ *must be invertible!*

We see that the solution is

$$\mathbf{x} = \begin{bmatrix} a_x \\ b_x \\ a_y \\ b_y \end{bmatrix} = \begin{bmatrix} 55.0 \\ -5.0 \\ 20.0 \\ 0.0 \end{bmatrix}$$

Plugging these values into our parameterization yields

$$\ell_x(t) = 55.0t - 5.0$$
$$\ell_y(t) = 20.0t$$

This gives us the same line segment we graphed earlier!

Note: with EJML and Java you can write `SimpleMatrix x = A.solve(b);`

Parameterization - lines

Before we move on, notice how **A** is a square matrix. This means our system of equations is **fully constrained**. In other words, the number of unknowns equals the number of constraints.

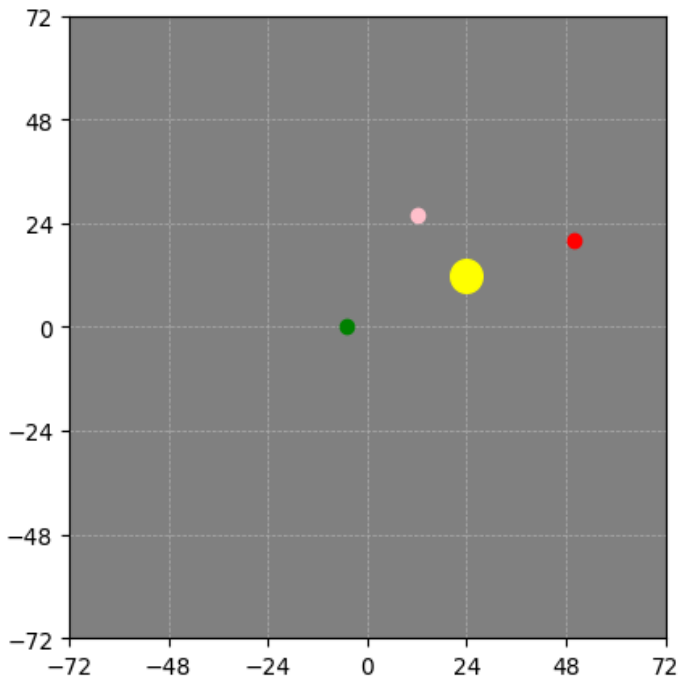
In general, you can under-constrain and over-constrain your system of equations (though the matrix will no longer be invertible, so other methods of solving may come into play), however, this may yield issues with the solution outside the scope of this presentation.

Moving forward, when we build the constraint matrix **A**, we will keep it fully constrained.

Note: the number of rows of **A** *represents the number of constraints, while the number of columns represents the number of unknowns.*

Path generation - piecewise linear

Suppose we now have an obstacle (in FTC, this will likely be a part of the game field) centered at $p_{obs} = (24.0, 12.0)$ (shown in yellow). Now if our robot tries to follow the line segment between our two waypoints, it'll hit the obstacle! Let's add an intermediate point $p_1 = (12.0, 26.0)$ (shown in pink) to generate a path which will avoid the obstacle.



Path generation - piecewise linear

Now, since we have three waypoints instead of two, we must parameterize two lines instead of one:

$$\ell_{x,i}(t) = a_{x,i}t + b_{x,i}$$
$$\ell_{y,i}(t) = a_{y,i}t + b_{y,i}$$

where $i \in \{1, 2\}$, meaning we have Line 1 and Line 2.

Path generation - piecewise linear

Let's recall our **endpoint constraints**. This time, we will set the start of Line 1 to be equal to p_0 and the end of Line 2 to be equal to p_n. Note that we will parameterize *each* line with $0 \leq t \leq 1$.

$$\ell_{x,1}(0) = b_{x,1} = x_0$$
$$\ell_{y,1}(0) = b_{y,1} = y_0$$
$$\ell_{x,2}(1) = a_{x,2} + b_{x,2} = x_n$$
$$\ell_{y,2}(1) = a_{y,2} + b_{y,2} = y_n$$

Now, we must introduce **continuity constraints**, so that the end of Line 1 matches up with the start of Line 2:

$$\ell_{x,1}(1) = a_{x,1} + b_{x,1} = x_1$$
$$\ell_{y,1}(1) = a_{y,1} + b_{y,1} = y_1$$
$$\ell_{x,2}(0) = b_{x,2} = x_1$$
$$\ell_{y,2}(0) = b_{y,2} = y_1$$

Path generation - piecewise linear

Using both sets of constraints we can define a new system of equations to solve for our eight unknowns, $a_{x,1}$, $b_{x,1}$, $a_{y,1}$, $b_{y,1}$, $a_{x,2}$, $b_{x,2}$, $a_{y,2}$, $b_{y,2}$:

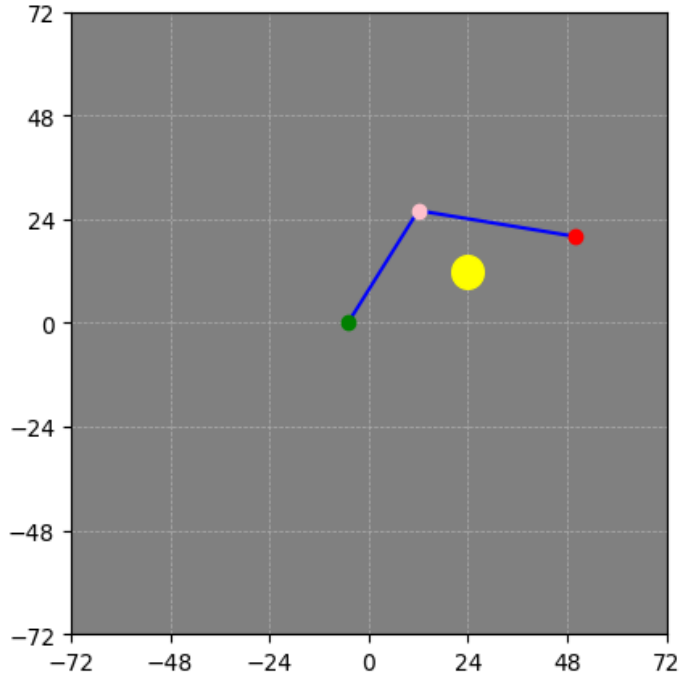
$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} a_{x,1} \\ b_{x,1} \\ a_{y,1} \\ b_{y,1} \\ a_{x,2} \\ b_{x,2} \\ a_{y,2} \\ b_{y,2} \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} x_0 \\ y_0 \\ x_n \\ y_n \\ x_1 \\ y_1 \\ x_1 \\ y_1 \end{bmatrix}}_{\mathbf{b}}$$

Path generation - piecewise linear

Solving $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ gives us

$$\mathbf{x} = \begin{bmatrix} a_{x,1} \\ b_{x,1} \\ a_{y,1} \\ b_{y,1} \\ a_{x,2} \\ b_{x,2} \\ a_{y,2} \\ b_{y,2} \end{bmatrix} = \begin{bmatrix} 17.0 \\ -5.0 \\ 26.0 \\ 0.0 \\ 38.0 \\ 12.0 \\ -6.0 \\ 26.0 \end{bmatrix}$$

Plugging these values into our four line equations and graphing show us two lines connecting the three waypoints.



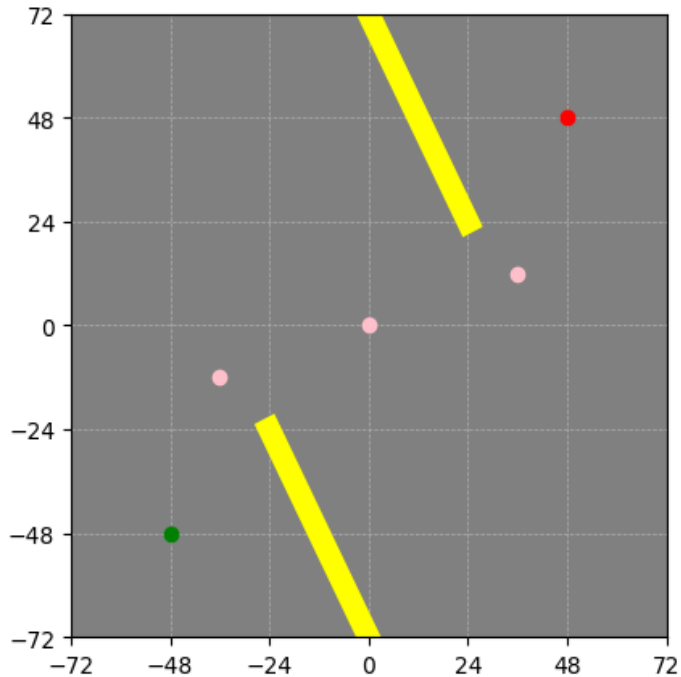
Note: This concludes the introductory portion of the material for path generation. In the next slides, we will get go through the concepts which should be implemented in code.

Path generation - piecewise cubic

Let's set up a new start point $p_0 = (-48.0, -48.0)$ and a new end point $p_n = (48.0, 48.0)$. Assume we have new large obstacles in the robot's way, so we will be unable to use a straight line. Instead, we select points p_1, p_2 and p_3 such that we generate a path which avoids the obstacle. We can use our piecewise linear path generation strategy, however we can generate, better *smoother* paths!

In total we have $n + 1 = 5$ waypoints:

- $p_0 = (-48.0, -48.0)$
- $p_1 = (-36.0, -12.0)$
- $p_2 = (0.0, 0.0)$
- $p_3 = (36.0, 12.0)$
- $p_n = (48.0, 48.0)$



Note: the piecewise linear paths are actually perfectly fine for a robot using Pure Pursuit, however we may be able to squeeze out better performance by generating smooth paths in the first place!

Path generation - piecewise cubic

Instead of parameterizing with first-order polynomials (lines), we will parameterize with third-order polynomials or **cubic splines**. The next few slides will go through a general technique for generating splines between any $n+1$ number of waypoints. Since we have $n+1$ waypoints, we will have n splines between them. The cubic parameterization for the ith spline (in both x and y) looks like

$$s_{x,i}(t) = a_{x,i}t^3 + b_{x,i}t^2 + c_{x,i}t + d_{x,i},\ \forall t \in [0,1];\ i \in \{1,2,...,n\}$$
$$s_{y,i}(t) = a_{y,i}t^3 + b_{y,i}t^2 + c_{y,i}t + d_{y,i},\ \forall t \in [0,1];\ i \in \{1,2,...,n\}$$

In this case, we have n splines in 2 dimensions with 4 unknowns each (namely a, b, c, d), meaning we have $n \cdot 2 \cdot 4 = 8n$ unknowns. We will therefore need $8n$ constraints to fully define the system of equations. The unknowns vector for the ith spline is represented by

$$\mathbf{x}_i = \begin{bmatrix} a_{x,i} & b_{x,i} & c_{x,i} & d_{x,i} & a_{y,i} & b_{y,i} & c_{y,i} & d_{y,i} \end{bmatrix}^\top$$

meaning our full unknowns vector looks like

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1^\top & \mathbf{x}_2^\top & \cdots & \mathbf{x}_{n-1}^\top & \mathbf{x}_n^\top \end{bmatrix}^\top$$

Note: we'll refer to the combination of a single x and y spline as a single spline

Path generation - piecewise cubic

Let's start by combining the total **endpoint constraints** and **continuity constraints**, such that in general, the start of the ith spline is equal to waypoint p_{i-1} and the end of the ith spline is equal to waypoint p_i

$$
\begin{aligned}
s_{x,i}(0) &= d_{x,i} = x_{i-1} \\
s_{y,i}(0) &= d_{y,i} = y_{i-1} \\
s_{x,i}(1) &= a_{x,i} + b_{x,i} + c_{x,i} + d_{x,i} = x_i \\
s_{y,i}(1) &= a_{y,i} + b_{y,i} + c_{y,i} + d_{y,i} = y_i
\end{aligned}
$$

Using these constraints, we can generate constraint matrices $\mathbf{A}_{i,cont}$, and $\mathbf{b}_{i,cont}$.

$$
\mathbf{A}_{i,cont} = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}, \quad \mathbf{b}_{i,cont} = \begin{bmatrix}
x_{i-1} \\ y_{i-1} \\ x_i \\ y_i
\end{bmatrix}
$$

Notice how we already have $4n$, or half, of our total needed constraints!

Path generation - piecewise cubic

We now need to generate the full set of continuinty constraints via the matrices $\mathbf{A}_{cont}$ and $\mathbf{b}_{cont}$. These matrices look like

$$\mathbf{A}_{cont} = \begin{bmatrix} \mathbf{A}_{1,cont} & \mathbf{0} & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{2,cont} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \mathbf{A}_{n-1,cont} & \mathbf{0} \\ \mathbf{0} & \cdots & \cdots & \mathbf{0} & \mathbf{A}_{n,cont} \end{bmatrix}, \quad \mathbf{b}_{cont} = \begin{bmatrix} \mathbf{b}_{1,cont} \\ \mathbf{b}_{2,cont} \\ \vdots \\ \mathbf{b}_{n-1,cont} \\ \mathbf{b}_{n,cont} \end{bmatrix}$$

Note: the $\mathbf{0}$*s represent matrices of zeros with the same size as* $\mathbf{A}_{i,cont}$*.*

One thing to notice is that $\mathbf{A}_{0,cont} = \mathbf{A}_{i,cont} = \mathbf{A}_{n,cont}$, meaning $\mathbf{A}_{i,cont}$ is the same and constant for every i! This makes building the large matrix $\mathbf{A}_{cont}$ much simpler.

Note: the reason we need these large matrices is so that we properly multiply each constant in $\mathbf{A}_{cont}$ *with it's corresponding unknown in* $\mathbf{x}$*.*

Path generation - piecewise cubic

To build $\mathbf{A}_{cont}$ we use

$$\mathbf{A}_{cont} = \mathbf{I}_{n \times n} \otimes \mathbf{A}_{i,cont}$$

where $\mathbf{I}_{n \times n}$ is the n by n identity matrix and $\otimes$ is the Kronecker product. The Kroncker product between an identity matrix and some other matrix essentially replaces each 1 with that matrix. In Java (using EJML), assuming we've set up $\mathbf{A}_{i,cont}$ in code already, we can use something like:

```
SimpleMatrix I = SimpleMatrix.identity(n);
SimpleMatrix A_cont = I.kron(A_i_cont);
```

to generate the large matrix $\mathbf{A}_{cont}$. To build $\mathbf{b}_{cont}$ in code, we can first instantiate a vector of size $4n$ using EJML, then simply insert each $\mathbf{b}_{i,cont}$ at each $4(i-1)$ height using a loop:

```
b_cont.insertIntoThis(4*(i-1), 0, b_i_cont);
```

A good way to validate your work is to check that the size of $\mathbf{A}_{cont}$ is $4n \times 8n$.

Note: we use $4(i-1)$ as the index because $\mathbf{b}_{i,cont}$ has size 4. This way we perfectly stack each $\mathbf{b}_{0,cont}$ through $\mathbf{b}_{n,cont}$ without overlap!

Path generation - piecewise cubic

We will now introduce **smoothness** constraints between each spline. These constraints ensure there are no rugged edges at intersection points. Let's start by taking the first and second derivative of the ith spline (assume the same conditions on t and i).

$$\dot{s}_{x,i}(t) = 3a_{x,i}t^2 + 2b_{x,i}t + c_{x,i}$$
$$\dot{s}_{y,i}(t) = 3a_{y,i}t^2 + 2b_{y,i}t + c_{y,i}$$
$$\ddot{s}_{x,i}(t) = 6a_{x,i}t + 2b_{x,i}$$
$$\ddot{s}_{y,i}(t) = 6a_{y,i}t + 2b_{y,i}$$

Note: $\dot{f}$ is the same as f' or $\frac{df}{dt}$.

Smoothness constraints only happen at intermediate (pink) waypoints, as the starting and ending splines are connected to the start and end points, respectively. To deal with this, we introduce $j \in \{2, ..., n\}$.

Path generation - piecewise cubic

To ensure smoothness, the first and second derivatives at the end of the $(j-1)$th spline should be equal to the first and second derivatives at the start of the jth spline. Mathematically this looks like:

$$
\begin{aligned}
&\dot{s}_{x,j-1}(1) = \dot{s}_{x,j}(0) \\
&\quad \implies 3a_{x,j-1} + 2b_{x,j-1} + c_{x,j-1} = c_{x,j} \\
&\quad \implies 3a_{x,j-1} + 2b_{x,j-1} + c_{x,j-1} - c_{x,j} = 0 \\
&\dot{s}_{y,j-1}(1) = \dot{s}_{y,j}(0) \\
&\quad \implies 3a_{y,j-1} + 2b_{y,j-1} + c_{y,j-1} = c_{y,j} \\
&\quad \implies 3a_{y,j-1} + 2b_{y,j-1} + c_{y,j-1} - c_{y,j} = 0 \\
&\ddot{s}_{x,j-1}(1) = \ddot{s}_{x,j}(0) \\
&\quad \implies 6a_{x,j-1} + 2b_{x,j-1} = 2b_{x,j} \\
&\quad \implies 3a_{x,j-1} + b_{x,j-1} - b_{x,j} = 0 \\
&\ddot{s}_{y,j-1}(1) = \ddot{s}_{y,j}(0) \\
&\quad \implies 6a_{y,j-1} + 2b_{y,j-1} = 2b_{y,j} \\
&\quad \implies 3a_{y,j-1} + b_{y,j-1} - b_{y,j} = 0
\end{aligned}
$$

Path generation - piecewise cubic

Using these constraints, we can generate constraint matrices $\mathbf{A}_{j,smoo}$, and $\mathbf{b}_{j,smoo}$.

$$\mathbf{A}_{j,smoo} = \left[\begin{array}{cccccccc|cccccccc} 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{array}\right]$$

However, if we take a closer look at the constraints, we notice that the right hand side of each equation is 0. Therefore $\mathbf{b}_{j,smoo}$ is simply a column vector of zeros of size 4.

The line through the middle of the matrix is simply an indicator for us to remember that the left side (first eight columns) corresponds to the $(j-1)$th unknowns, while the right side corresponds to the jth unknowns. This will be important when we start stacking the matrices, as simply using the Kronecker product with an identity matrix is not enough.

Let's let the left side of $\mathbf{A}_{j,smoo}$ be $\mathbf{A}^{(L)}_{j,smoo}$ and the right side of $\mathbf{A}_{j,smoo}$ be $\mathbf{A}^{(R)}_{j,smoo}$ such that

$$\mathbf{A}_{j,smoo} = \left[\begin{array}{c|c} \mathbf{A}^{(L)}_{j,smoo} & \mathbf{A}^{(R)}_{j,smoo} \end{array}\right]$$

Path generation - piecewise cubic

We now need to generate the full set of smoothness constraints via the matrices $\mathbf{A}_{smoo}$ and $\mathbf{b}_{cont}$. The former looks like

$$\mathbf{A}_{smoo} = \begin{bmatrix} \mathbf{A}_{2,smoo}^{(L)} & \mathbf{A}_{2,smoo}^{(R)} & \mathbf{0} & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{3,smoo}^{(L)} & \mathbf{A}_{3,smoo}^{(R)} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \mathbf{A}_{n-1,smoo}^{(L)} & \mathbf{A}_{n-1,smoo}^{(R)} & \mathbf{0} \\ \mathbf{0} & \cdots & \cdots & \mathbf{0} & \mathbf{A}_{n,smoo}^{(L)} & \mathbf{A}_{n,smoo}^{(R)} \end{bmatrix}$$

Similarly to our continuity constraints, we can see that all representations of $\mathbf{A}_{j,smoo}$ are equal for any j. To build $\mathbf{A}_{smoo}$ in code, we can first establish a matrix of zeros of size $4(n-1) \times 8n$. Then, we can simply use an iterator (Let's call it k) between 0 and $n-1$ and place an $\mathbf{A}_{j,smoo}$ matrix at each each $(4k, 8k)$ row-column index of $\mathbf{A}_{smoo}$.

Since $\mathbf{b}_{j,smoo}$ is just zeros, building $\mathbf{b}_{smoo}$ reduces to simply establishing a vector of zeros of size $4(n-1)$.

Path generation - piecewise cubic

Let's calculate how many constraints we now have. From continuity we have $4n$ and from smoothness we have $4(n-1)$. This gives us a total of $8n-4$ constraints. We need $8n$ constraints to define a fully-constrained system of equations, so we only need to come up with 4 **additional constraints**!

One good way to do this is to set the derivatives of the splines to be 0 at the initial and goal points. Doing this for x and y will yield $2 \cdot 2 = 4$ constraints. Since we've been parameterizing with $t \in [0, 1]$, we can simply do:

$$\dot{s}_{x,0}(0) = c_{x,0} = 0$$
$$\dot{s}_{y,0}(0) = c_{y,0} = 0$$
$$\dot{s}_{x,n}(1) = 3a_{x,n} + 2b_{x,n} + c_{x,n} = 0$$
$$\dot{s}_{y,n}(1) = 3a_{y,n} + 2b_{y,n} + c_{y,n} = 0$$

Path generation - piecewise cubic

In matrix form, these constraints look like

$$\mathbf{A}_{add} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 3 & 2 & 1 & 0 \end{bmatrix}$$

and a vector of zeros of size 4 for $\mathbf{b}_{add}$. Remember that A_{add} has a total of $8n$ columns for each unknown. Only the first and last nine columns of $\mathbf{A}_{add}$ are shown because the rest of the matrix is zeros.

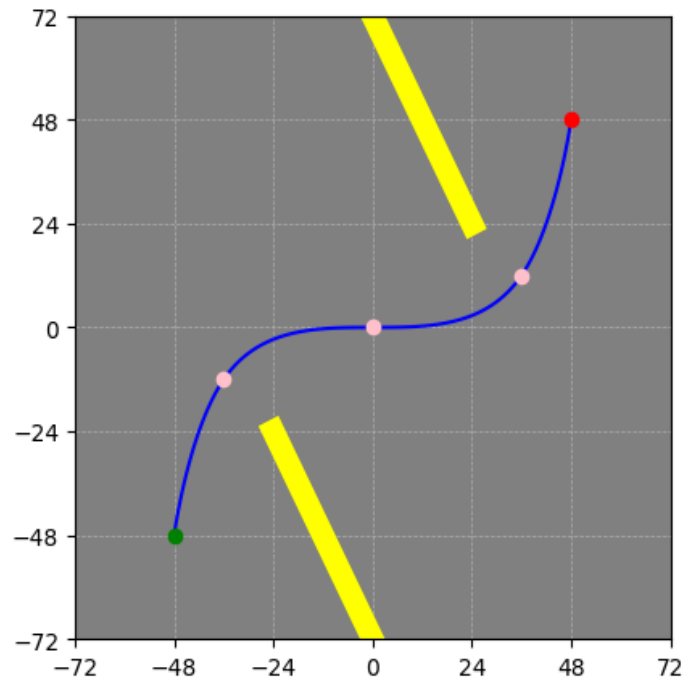
Building each matrix in code is not difficult as no stacking of constant smaller matrices is necessary. Simply establish $\mathbf{A}_{add}$ as a matrix of zeros of size $4 \times 8n$. Then fill in the first eight and final eight columns (since these correspond to the 0th and nth spline, respectively) with the constants above.

Path generation - piecewise cubic

We now have $\mathbf{A}_{cont}$ with $4n$ rows, $\mathbf{A}_{smoo}$ with $4(n-1)$ rows and $\mathbf{A}_{add}$ with 4 rows for a total of $8n$ constraints. Now, all that is left to do is vertically stack our constraint matrices and solve the fully-constrained system of equations for our spline coefficients.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{cont} \\ \mathbf{A}_{smoo} \\ \mathbf{A}_{add} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_{cont} \\ \mathbf{b}_{smoo} \\ \mathbf{b}_{add} \end{bmatrix}$$

After solving $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, we can verify our work by graphing the splines. We see that our splines hit each waypoint and smoothly transition with no rugged edges. We have successfully interpolated a smooth path for our five waypoints.



Path generation - piecewise cubic

Now that we have formulated our method for generating splines, how can we use them in practice? We can reparameterize the spline and fit a time-based motion profile to it (Road Runner's approach), or we can discretize the spline and generate a dense set of waypoints with a distance-based motion profile.

We will opt for the latter in this lecture, as the control lectures will focus on Pure Pursuit, which does not need a time specified at each waypoint. Let's choose a discretization parameter $N > 2$ (not to be confused with n). Due to inclusivity of our sparse "original" waypoints, we will always have $N - 1$ waypoints between them.

Path generation - piecewise cubic

Let's define $\mathcal{T} = \{0, \frac{1}{N}, \frac{2}{N}, ..., \frac{N-1}{N}\}$. In code, we can define a dense waypoints matrix P, where the first row consists of x values and the second row consists of y values. The size of this matrix is $2 \times (N \cdot n) + 1$. Using loops we can fill the matrix as follows:

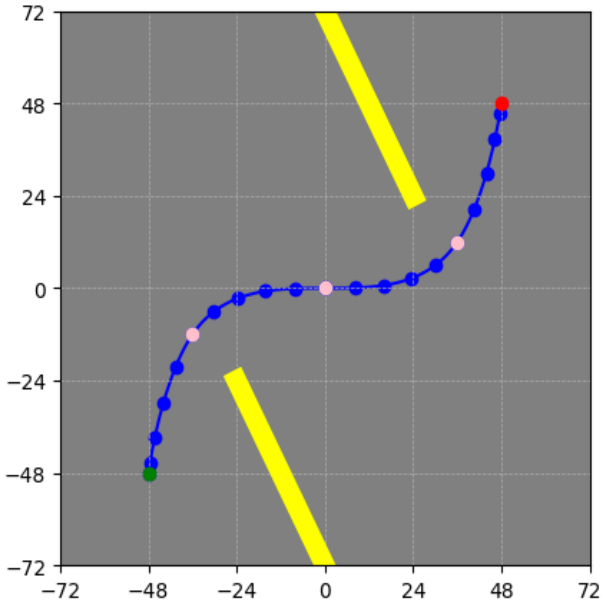
$$P_{(0,N\cdot i+j)} = s_{x,i}(\mathcal{T}_j)$$
$$P_{(1,N\cdot i+j)} = s_{y,i}(\mathcal{T}_j)$$

where $\mathcal{T}_j$ corresponds to the jth element in the set $\mathcal{T}$. The 0 and 1 refer to the row index of P, while $N \cdot i + j$ refers to the column index of P. This method does not consider the final endpoint of the spline, so we should be sure to also fill in the final column of P:

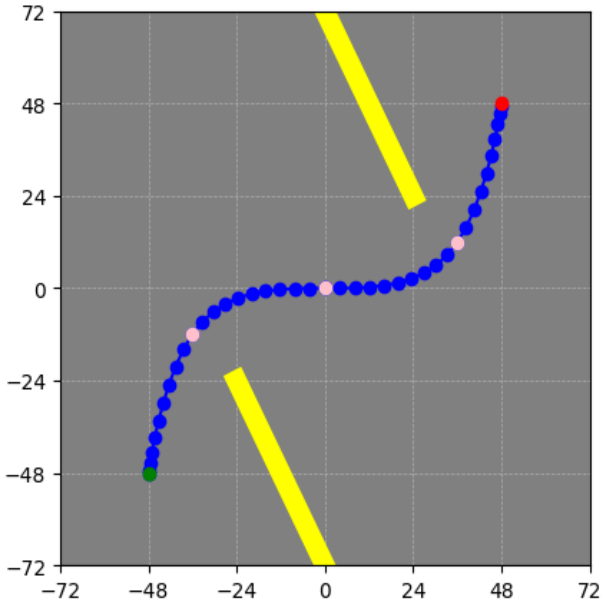
$$P_{(0,N\cdot n)} = s_{x,n}(1)$$
$$P_{(1,N\cdot n)} = s_{y,n}(1)$$

Path generation - piecewise cubic

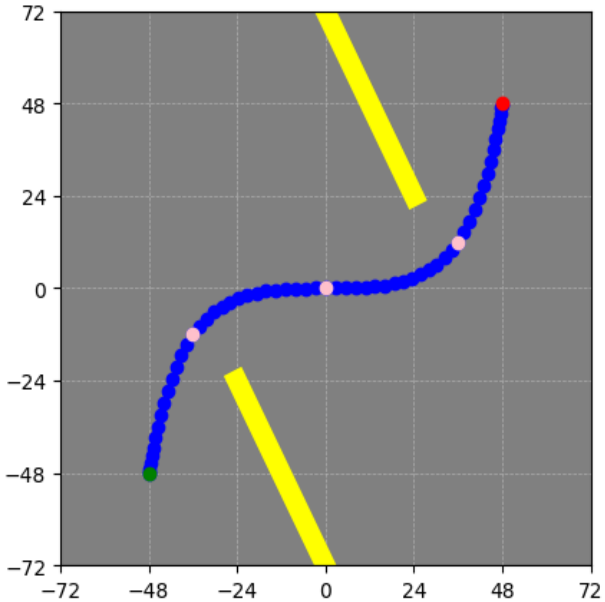
As we can see below, lower values of N correspond to a lower amount of waypoints. This is beneficial if we don't want to store very large matrices, or if we want to use Pure Pursuit without motion profiling. Higher values of N yield dense waypoints, which results in a larger matrix P, but will allow us to generate smooth distance-based motion profiles.



$N = 5$



$N = 10$



$N = 15$