Ok, let's walk through the solution. We'll go through each of the TODO tags.

**N & dt**

*// TODO: Set N and dt*

**size\_t** N = 25 ;

**double** dt = 0.05 ;

Here we had to assign values to N and dt. It's likely you set these variables to slightly different values. That's fine as long as the cross track error decreased to 0. It's a good idea to play with different values here.

For example, if we were to set N to 100, the simulation would run much slower. This is because the solver would have to optimize 4 times as many control inputs. Ipopt, the solver, permutes the control input values until it finds the lowest cost. If you were to open up Ipopt and plot the x and y values as the solver mutates them, the plot would look like a worm moving around trying to fit the shape of the reference trajectory.

**Cost function**

**void** **operator**()(ADvector& fg, **const** ADvector& vars) {

*// The cost is stored is the first element of `fg`.*

*// Any additions to the cost should be added to `fg[0]`.*

fg[0] = 0;

*// Cost function*

*// TODO: Define the cost related the reference state and*

*// any anything you think may be beneficial.*

*// The part of the cost based on the reference state.*

**for** (**int** t = 0; t < N; t++) {

fg[0] += CppAD::pow(vars[cte\_start + t], 2);

fg[0] += CppAD::pow(vars[epsi\_start + t], 2);

fg[0] += CppAD::pow(vars[v\_start + t] - ref\_v, 2);

}

*// Minimize the use of actuators.*

**for** (**int** t = 0; t < N - 1; t++) {

fg[0] += CppAD::pow(vars[delta\_start + t], 2);

fg[0] += CppAD::pow(vars[a\_start + t], 2);

}

*// Minimize the value gap between sequential actuations.*

**for** (**int** t = 0; t < N - 2; t++) {

fg[0] += CppAD::pow(vars[delta\_start + t + 1] - vars[delta\_start + t], 2);

fg[0] += CppAD::pow(vars[a\_start + t + 1] - vars[a\_start + t], 2);

}

There's a lot to unwind here.

Let's start with the function arguments: fg and vars.

The vector fg is where the cost function and vehicle model/constraints is defined. We'll go the fg vector in more detail shortly.

The other function argument is the vector vars. This vector contains all variables used by the cost function and model:

[*x*,*y*,*ψ*,*v*,*cte*,*eψ*]

[*δ*,*a*]

This is all one long vector, so if N is 25 then the indices are assigned as follows:

vars[0], ..., vars[24] -> *x*1,...,*x*25

vars[25], ..., vars[49] -> *y*1,...,*y*25

vars[50], ..., vars[74] -> *ψ*1,...,*ψ*25

vars[75], ..., vars[99] -> *v*1,...,*v*25

vars[100], ..., vars[124] -> *cte*1,...,*cte*25

vars[125], ..., vars[149] -> *eψ*1,...,*eψ*25

vars[150], ..., vars[173] -> *δ*1,...,*δ*24

vars[174], ..., vars[197] -> *a*1,...,*a*24

Now let's focus on the actual cost function. Since 0 is the index at whichIpopt expects fg to store the cost value, we sum all the components of the cost and store them at index 0.

In each iteration through the loop, we sum three components to reach the aggregate cost: our cross-track error, our heading error, and our velocity error.

*// The part of the cost based on the reference state.*

**for** (**int** t = 0; t < N; t++) {

fg[0] += CppAD::pow(vars[cte\_start + t] , 2);

fg[0] += CppAD::pow(vars[epsi\_start + t], 2);

fg[0] += CppAD::pow(vars[v\_start + t], 2);

}

We've already taken care of the main objective - to minimize our cross track, heading, and velocity errors. A further enhancement is to constrain erratic control inputs.

For example, if we're making a turn, we'd like the turn to be smooth, not sharp. Additionally, the vehicle velocity should not change too radically.

*// Minimize change-rate.*

**for** (**int** t = 0; t < N - 1; t++) {

fg[0] += CppAD::pow(vars[delta\_start + t], 2);

fg[0] += CppAD::pow(vars[a\_start + t], 2);

}

The goal of this final loop is to make control decisions more consistent, or smoother. The next control input should be similar to the current one.

*// Minimize the value gap between sequential actuations.*

**for** (**int** t = 0; t < N - 2; t++) {

fg[0] += CppAD::pow(vars[delta\_start + t + 1] - vars[delta\_start + t], 2);

fg[0] += CppAD::pow(vars[a\_start + t + 1] - vars[a\_start + t], 2);

}

**Initialization & constraints**

We initialize the model to the initial state. Recall fg[0] is reserved for the cost value, so the other indices are bumped up by 1.

fg[1 + x\_start] = vars[x\_start];

fg[1 + y\_start] = vars[y\_start];

fg[1 + psi\_start] = vars[psi\_start];

fg[1 + v\_start] = vars[v\_start];

fg[1 + cte\_start] = vars[cte\_start];

fg[1 + epsi\_start] = vars[epsi\_start];

All the other constraints based on the vehicle model:

*x*​*t*+1​​=*x*​*t*​​+*v*​*t*​​∗*cos*(*ψ*​*t*​​)∗*dt*

*y*​*t*+1​​=*y*​*t*​​+*v*​*t*​​∗*sin*(*ψ*​*t*​​)∗*dt*

*ψ*​*t*+1​​=*ψ*​*t*​​+​*L*​*f*​​​​*v*​*t*​​​​∗*δ*​*t*​​∗*dt*

*v*​*t*+1​​=*v*​*t*​​+*a*​*t*​​∗*dt*

*cte*​*t*+1​​=*f*(*x*​*t*​​)−*y*​*t*​​+(*v*​*t*​​∗*sin*(*eψ*​*t*​​)∗*dt*)

*eψ*​*t*+1​​=*ψ*​*t*​​−*ψdes*​*t*​​+(​*L*​*f*​​​​*v*​*t*​​​​∗*δ*​*t*​​∗*dt*)

Let's look how to model *ψ*. Based on the above equations, we need to constrain the value of *ψ* at time t+1:

*ψ*​*t*+1​​=*ψ*​*t*​​+​*L*​*f*​​​​*v*​*t*​​​​∗*δ*​*t*​​∗*dt*

We do that by setting a value within fg to the difference of ps1 and the above formula.

Previously, we have set the corresponding constraints\_lowerbound and the constraints\_upperbound values to 0. That means the solver will force this value of fg to always be 0.

**for** (**int** t = 1; t < N ; t++) {

*// psi, v, delta at time t*

AD<**double**> psi0 = vars[psi\_start + t - 1];

AD<**double**> v0 = vars[v\_start + t - 1];

AD<**double**> delta0 = vars[delta\_start + t - 1];

*// psi at time t+1*

AD<**double**> psi1 = vars[psi\_start + t];

*// how psi changes*

fg[1 + psi\_start + t] = psi1 - (psi0 + v0 \* delta0 / Lf \* dt);

}

The oddest line above is probably fg[1 + psi\_start + t].

fg[0] stores the cost value, so there's always an offset of 1. So fg[1 + psi\_start] is where we store the initial value of *ψ*. Finally, fg[1 + psi\_start + t] is reserved for the *t*th of *N* values of *ψ* that the solver computes.

Coding up the other parts of the model is similar.

**for** (**int** t = 1; t < N; t++) {

*// The state at time t+1 .*

AD<**double**> x1 = vars[x\_start + t];

AD<**double**> y1 = vars[y\_start + t];

AD<**double**> psi1 = vars[psi\_start + t];

AD<**double**> v1 = vars[v\_start + t];

AD<**double**> cte1 = vars[cte\_start + t];

AD<**double**> epsi1 = vars[epsi\_start + t];

*// The state at time t.*

AD<**double**> x0 = vars[x\_start + t - 1];

AD<**double**> y0 = vars[y\_start + t - 1];

AD<**double**> psi0 = vars[psi\_start + t - 1];

AD<**double**> v0 = vars[v\_start + t - 1];

AD<**double**> cte0 = vars[cte\_start + t - 1];

AD<**double**> epsi0 = vars[epsi\_start + t - 1];

*// Only consider the actuation at time t.*

AD<**double**> delta0 = vars[delta\_start + t - 1];

AD<**double**> a0 = vars[a\_start + t - 1];

AD<**double**> f0 = coeffs[0] + coeffs[1] \* x0;

AD<**double**> psides0 = CppAD::atan(coeffs[1]);

*// Here's `x` to get you started.*

*// The idea here is to constraint this value to be 0.*

*//*

*// Recall the equations for the model:*

*// x\_[t] = x[t-1] + v[t-1] \* cos(psi[t-1]) \* dt*

*// y\_[t] = y[t-1] + v[t-1] \* sin(psi[t-1]) \* dt*

*// psi\_[t] = psi[t-1] + v[t-1] / Lf \* delta[t-1] \* dt*

*// v\_[t] = v[t-1] + a[t-1] \* dt*

*// cte[t] = f(x[t-1]) - y[t-1] + v[t-1] \* sin(epsi[t-1]) \* dt*

*// epsi[t] = psi[t] - psides[t-1] + v[t-1] \* delta[t-1] / Lf \* dt*

fg[1 + x\_start + t] = x1 - (x0 + v0 \* CppAD::cos(psi0) \* dt);

fg[1 + y\_start + t] = y1 - (y0 + v0 \* CppAD::sin(psi0) \* dt);

fg[1 + psi\_start + t] = psi1 - (psi0 + v0 \* delta0 / Lf \* dt);

fg[1 + v\_start + t] = v1 - (v0 + a0 \* dt);

fg[1 + cte\_start + t] =

cte1 - ((f0 - y0) + (v0 \* CppAD::sin(epsi0) \* dt));

fg[1 + epsi\_start + t] =

epsi1 - ((psi0 - psides0) + v0 \* delta0 / Lf \* dt);

}

**Fitting a polynomial to the waypoints**

*// TODO: fit a polynomial to the above x and y coordinates*

**auto** coeffs = polyfit(ptsx, ptsy, 1);

The x and y coordinates are contained in the ptsx and ptsy vectors. Since these are 2-element vectors a 1-degree polynomial (straight line) is sufficient.

**Calculating the cross track and orientation error**

**double** x = -1;

**double** y = 10;

**double** psi = 0;

**double** v = 10;

*// TODO: calculate the cross track error*

**double** cte = polyeval(coeffs, x) - y;

*// TODO: calculate the orientation error*

**double** epsi = psi - atan(coeffs[1]);

The cross track error is calculated by evaluating at polynomial at x (-1) and subtracting y.

Recall orientation error is calculated as follows *eψ*=*ψ*−*ψdes*, where *ψdes* is can be calculated as *arctan*(*f*​′​​(*x*)).

*f*(*x*)=*a*​0​​+*a*​1​​∗*x*

*f*​′​​(*x*)=*a*​1​​

hence the solution double epsi = psi - atan(coeffs[1]);