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:
$[x,y,\psi,v,cte,e\psi]$
$[\delta,a]$
This is all one long vector, so if `N` is 25 then the indices are assigned as follows:

vars[0], ..., vars[24] -> $x1,...,x25$
vars[25], ..., vars[49] -> $y1,...,y25$
vars[50], ..., vars[74] -> $\psi 1,...,\psi 25$
vars[75], ..., vars[99] -> $v1,...,v25$
vars[100], ..., vars[124] -> $cte1,...,cte25$
vars[125], ..., vars[149] -> $e\psi 1,...,e\psi 25$
vars[150], ..., vars[173] -> $\delta 1,...,\delta 24$
vars[174], ..., vars[197] -> $a1,...,a24$
Now let's focus on the actual cost function. Since 0 is the index at which `Ipopt` 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:

$x_{t+1}=x_t+v_t*cos(\psi_t)*dt$

$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$
$\psi_{t+1} = \psi_t + L_f v_t * \delta_t * dt$
$v_{t+1} = v_t + a_t * dt$
$cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt)$
$e\psi_{t+1} = \psi_t - \psi des_t + (L_f v_t * \delta_t * dt)$
Let's look how to model ψ. Based on the above equations, we need to constrain the value of ψ at time t+1:
$\psi_{t+1} = \psi_t + L_f v_t * \delta_t * dt$
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 tth 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 $e\psi = \psi - \psi des$, 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]);`