In this quiz you'll use MPC to follow the trajectory along a line Steps:

1. Set *N* and *dt*.
2. Fit the polynomial to the waypoints.
3. Calculate initial cross track error and orientation error values.
4. Define the components of the cost function (state, actuators, etc). You may use the methods previously discussed or make up something, up to you!
5. Define the model constraints. These are the state update equations defined in the *Vehicle Models* module.

Before you begin let's go over the libraries you'll use for this quiz and the following project.

Ipopt

Ipopt is the tool we'll be using to optimize the control inputs $[\delta_1, a_1, ..., \delta_{N-1}, a_{N-1}]$. It's able to find locally optimal values (non-linear problem!) while keeping the constraints set directly to the actuators and the constraints defined by the vehicle model. Ipopt requires we give it the jacobians and hessians directly - it does not compute them for us. Hence, we need to either manually compute them or have a library do this for us. Luckily, there is a library called CppAD which does exactly this.

CppAD

CppAD is a library we'll use for automatic differentiation. By using CppAD we don't have to manually compute derivatives, which is tedious and prone to error.

In order to use CppAD effectively, we have to use its types instead of regular `double` or `std::vector` types.
Additionally math functions must be called from CppAD. Here's an example of calling `pow`:

```
CppAD::pow(x, 2);
// instead of
pow(x, 2);
```

Luckily most elementary math operations are overloaded. So calling `*`, `+`, `-`, `/` will work as intended as long as it's called on `CppAD<double>` instead of `double`. Most of this is done for you and there are examples to draw from in the code we provide.

Code Structure

We've filled in most of the quiz starter code for you. The goal of this quiz is really just about getting everything to work as intended.

That said, it may be tricky to decipher some elements of the starter code, so we will walk you through it.

There are two main components in **`MPC.cpp`**:

1. `vector<double> MPC::Solve(Eigen::VectorXd x0, Eigen::VectorXd coeffs)` method
2. `FG_eval` class

MPC::Solve

`x0` is the initial state $[x, y, \psi, v, cte, e\psi]$, `coeffs` are the coefficients of the fitting polynomial. The bulk of this method is setting up the vehicle model constraints (`constraints`) and variables (`vars`) for Ipopt.

Variables

```
double x = x0[0];
double y = x0[1];
double psi = x0[2];
double v = x0[3];
double cte = x0[4];
double epsi = x0[5];
...
// Set the initial variable values
vars[x_start] = x;
vars[y_start] = y;
vars[psi_start] = psi;
vars[v_start] = v;
vars[cte_start] = cte;
vars[epsi_start] = epsi;
```

Note Ipopt expects all the constraints and variables as vectors. For example, suppose `N` is 5, then the structure of `vars` a 38-element vector:

vars[0],...,vars[4] -> $[x_1, ..., x_5]$

vars[5],...,vars[9] -> $[y_1, ..., y_5]$

vars[10],...,vars[14] -> $[\psi_1, ..., \psi_5]$

vars[15],...,vars[19] -> $[v_1,....,v_5]$

vars[20],...,vars[24] -> $[cte_1,....,cte_5]$

vars[25],...,vars[29] -> $[e\psi_1,....,e\psi_5]$

vars[30],...,vars[33] -> $[\delta_1,....,\delta_4]$

vars[34],...,vars[37] -> $[a_1,....,a_4]$

We then set lower and upper bounds on the variables. Here we set the range of values δ to [-25, 25] in radians:

```
for (int i = delta_start; i < a_start; i++) {
    vars_lowerbound[i] = -0.436332;
    vars_upperbound[i] = 0.436332;
  }
```

Constraints Next the we set the lower and upper bounds on the constraints.

Consider, for example:

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

This expresses that x_{t+1} **MUST** be equal to $x_t + v_t * cos(\psi_t) * dt$. Put differently:

$$x_{t+1} - (x_t + v_t * cos(\psi_t) * dt) = 0$$

The equation above simplifies the upper and lower bounds of the constraint: both must be 0.

This can be generalized to the other equations as well:

```
for (int i = 0; i < n_constraints; i++) {
  constraints_lowerbound[i] = 0;
  constraints_upperbound[i] = 0;
}
```

FG_eval

The `FG_eval` class has the constructor:

```
FG_eval(Eigen::VectorXd coeffs) { this->coeffs = coeffs; }
```

where `coeffs` are the coefficients of the fitted polynomial. `coeffs` will be used by the cross track error and heading error equations.

The `FG_eval` class has only one method:

```
void operator()(ADvector& fg, const ADvector& vars)
```

`vars` is the vector of variables (from the previous section) and `fg` is the vector of constraints.

One complication: `fg[0]` stores the cost value, so the `fg` vector is 1 element larger than it was in `MPC::Solve`.

Here in `operator()` you'll define the cost function and constraints. *x* is already completed:

```
for (int t = 1; t <= N; t++) {
    AD<double> x1 = vars[x_start + t];

    AD<double> x0 = vars[x_start + t - 1];
    AD<double> psi0 = vars[psi_start + t - 1];
    AD<double> v0 = vars[v_start + t - 1];

    // Here's `x` to get you started.
    // The idea here is to constraint this value to be 0.
    //
    // NOTE: The use of `AD<double>` and use of `CppAD`!
    // This is also CppAD can compute derivatives and pass
    // these to the solver.

    // TODO: Setup the rest of the model constraints
    fg[1 + x_start + i] = x1 - (x0 + v0 * CppAD::cos(psi0) * dt);
  }
```

Note that we start the loop at `t=1`, because the values at `t=0` are set to our initial state - those values are not calculated by the solver.

An `FG_eval` object is created in `MPC::Solve`:

```
FG_eval fg_eval(coeffs);
```

This is then used by Ipopt to find the lowest cost trajectory:

```
// place to return solution
CppAD::ipopt::solve_result<Dvector> solution;

// solve the problem
CppAD::ipopt::solve<Dvector, FG_eval>(
    options, vars, vars_lowerbound, vars_upperbound, constraints_lowerbound,
    constraints_upperbound, fg_eval, solution);
```

The filled in `vars` vector is stored as `solution.x` and the cost as `solution.obj_value`.

Complete the *Model Predictive Control* quiz from **here**.