1. <u>Model description</u>

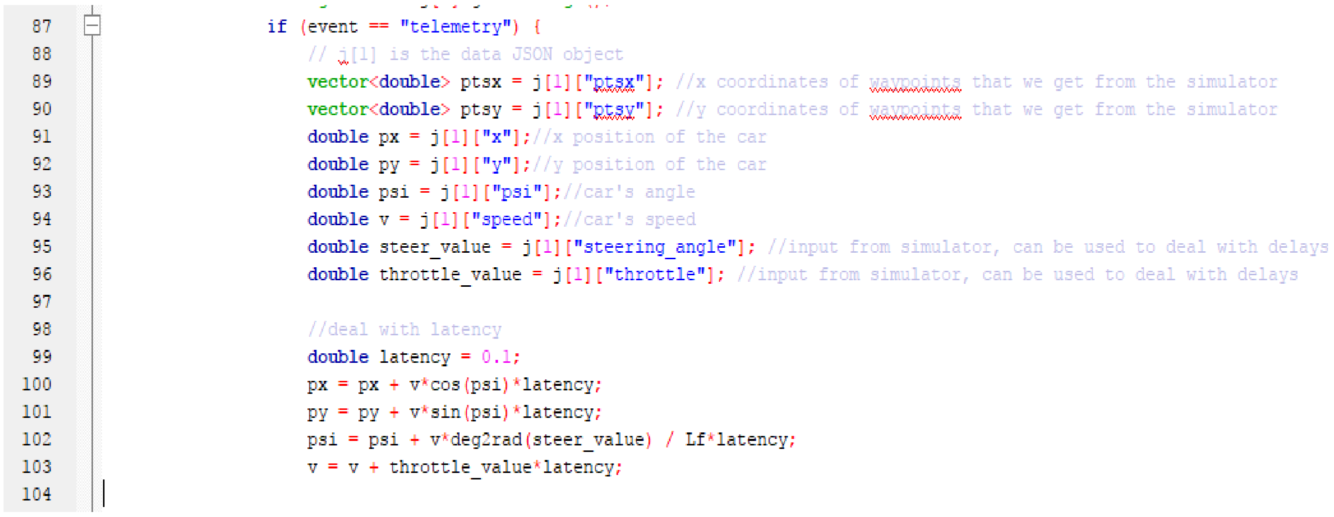
Student describes their model in detail. This includes the state, actuators and update equations.

The model is built up of 2 main code files: main.cpp and MPC.cpp.

- <u>Main.cpp:</u>
  - Communicates with our simulator:

In code lines 89-96, the simulator provides the model with inputs. ptsx and ptsy are the x-y coordinates of the waypoints and px, py the position of the car. Psi contains the angle of the direction of the car and v contains the speed.

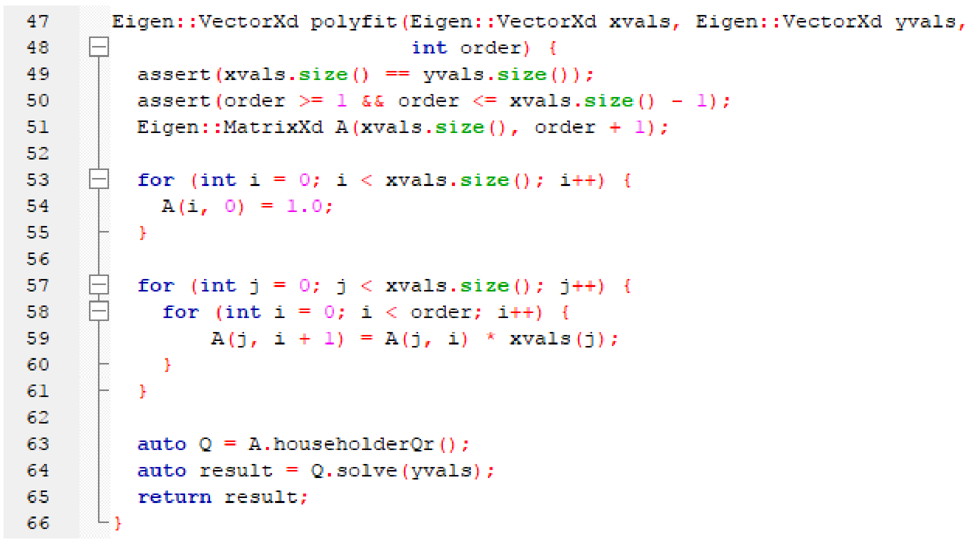
Steer_value contains the current steering angle and throttle_value the current throttle of the car. As explained in section 4, we use these 2 variables in our code to compensate for latency.



```
 87              if (event == "telemetry") {
 88                // j[1] is the data JSON object
 89                vector<double> ptsx = j[1]["ptsx"]; //x coordinates of waypoints that we get from the simulator
 90                vector<double> ptsy = j[1]["ptsy"]; //y coordinates of waypoints that we get from the simulator
 91                double px = j[1]["x"];//x position of the car
 92                double py = j[1]["y"];//y position of the car
 93                double psi = j[1]["psi"];//car's angle
 94                double v = j[1]["speed"];//car's speed
 95                double steer_value = j[1]["steering_angle"]; //input from simulator, can be used to deal with delays
 96                double throttle_value = j[1]["throttle"]; //input from simulator, can be used to deal with delays
 97
 98                //deal with latency
 99                double latency = 0.1;
100                px = px + v*cos(psi)*latency;
101                py = py + v*sin(psi)*latency;
102                psi = psi + v*deg2rad(steer_value) / Lf*latency;
103                v = v + throttle_value*latency;
104
```

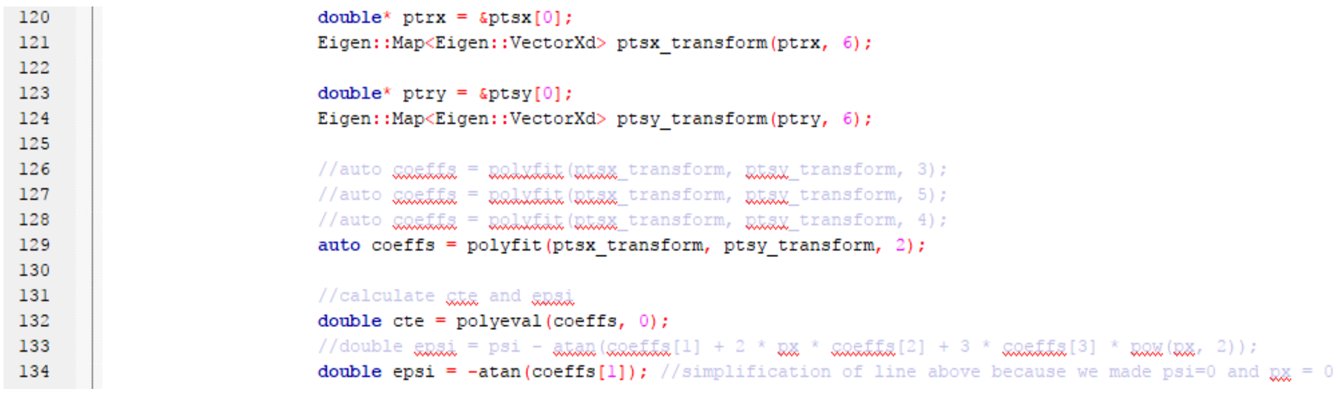
Code lines 99-103 deal with the 100 milliseconds latency between the actual steering and throttle inputs and the response of the car on these inputs. The details of these code lines are provided below in section 4.

  - polyfit() function



```
Eigen::VectorXd polyfit(Eigen::VectorXd xvals, Eigen::VectorXd yvals,
                        int order) {
  assert(xvals.size() == yvals.size());
  assert(order >= 1 && order <= xvals.size() - 1);
  Eigen::MatrixXd A(xvals.size(), order + 1);

  for (int i = 0; i < xvals.size(); i++) {
    A(i, 0) = 1.0;
  }

  for (int j = 0; j < xvals.size(); j++) {
    for (int i = 0; i < order; i++) {
      A(j, i + 1) = A(j, i) * xvals(j);
    }
  }

  auto Q = A.householderQr();
  auto result = Q.solve(yvals);
  return result;
}
```

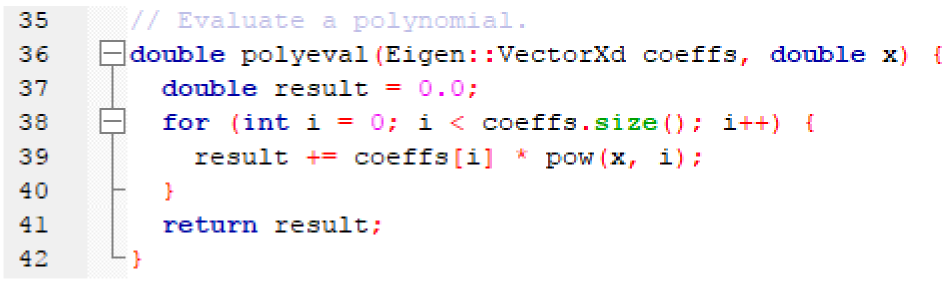
The polyfit() function calculates the coefficients of the polynomial fitted through the waypoints.



```
120                         double* ptrx = &ptsx[0];
121                         Eigen::Map<Eigen::VectorXd> ptsx_transform(ptrx, 6);
122
123                         double* ptry = &ptsy[0];
124                         Eigen::Map<Eigen::VectorXd> ptsy_transform(ptry, 6);
125
126                         //auto coeffs = polyfit(ptsx_transform, ptsy_transform, 3);
127                         //auto coeffs = polyfit(ptsy_transform, ptsx_transform, 5);
128                         //auto coeffs = polyfit(ptsx_transform, ptsy_transform, 4);
129                         auto coeffs = polyfit(ptsx_transform, ptsy_transform, 2);
130
131                         //calculate cte and epsi
132                         double cte = polyeval(coeffs, 0);
133                         //double epsi = psi - atan(coeffs[1] + 2 * px * coeffs[2] + 3 * coeffs[3] * pow(px, 2));
134                         double epsi = -atan(coeffs[1]); //simplification of line above because we made psi=0 and px = 0
```

In code line 129, I use this function to calculate the coefficients. I tried polynomials of different degrees (2nd, 3rd, 4th and 5th) and 2nd degree seemed to give the best performance.

- polyeval() function



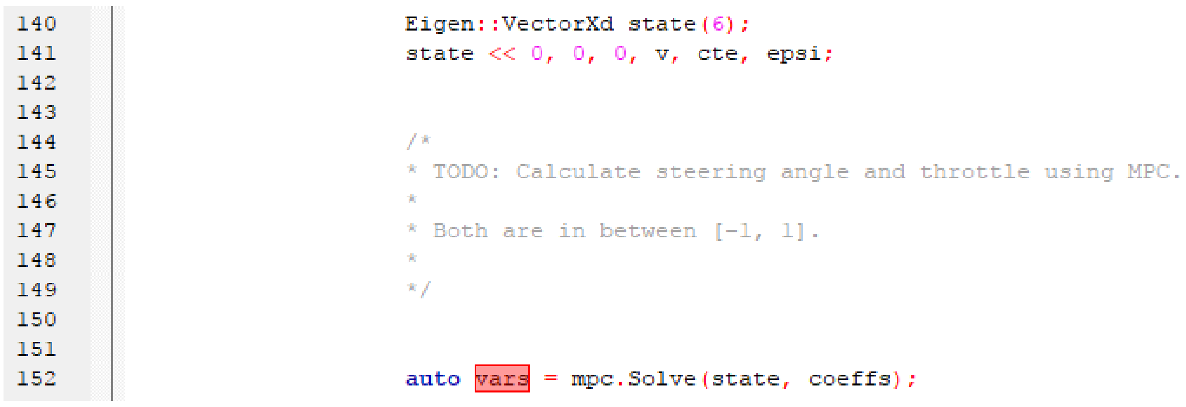
```
35  // Evaluate a polynomial.
36  double polyeval(Eigen::VectorXd coeffs, double x) {
37    double result = 0.0;
38    for (int i = 0; i < coeffs.size(); i++) {
39      result += coeffs[i] * pow(x, i);
40    }
41    return result;
42  }
```

The polyeval() function calculates the y value for a given x value of the fitted polynomial. It is used in line 132 to calculate the cte. Because we shifted the waypoints in 106-117 (see section 3), we can now use this function to calculate the distance of our car from the polynomial through the waypoints.

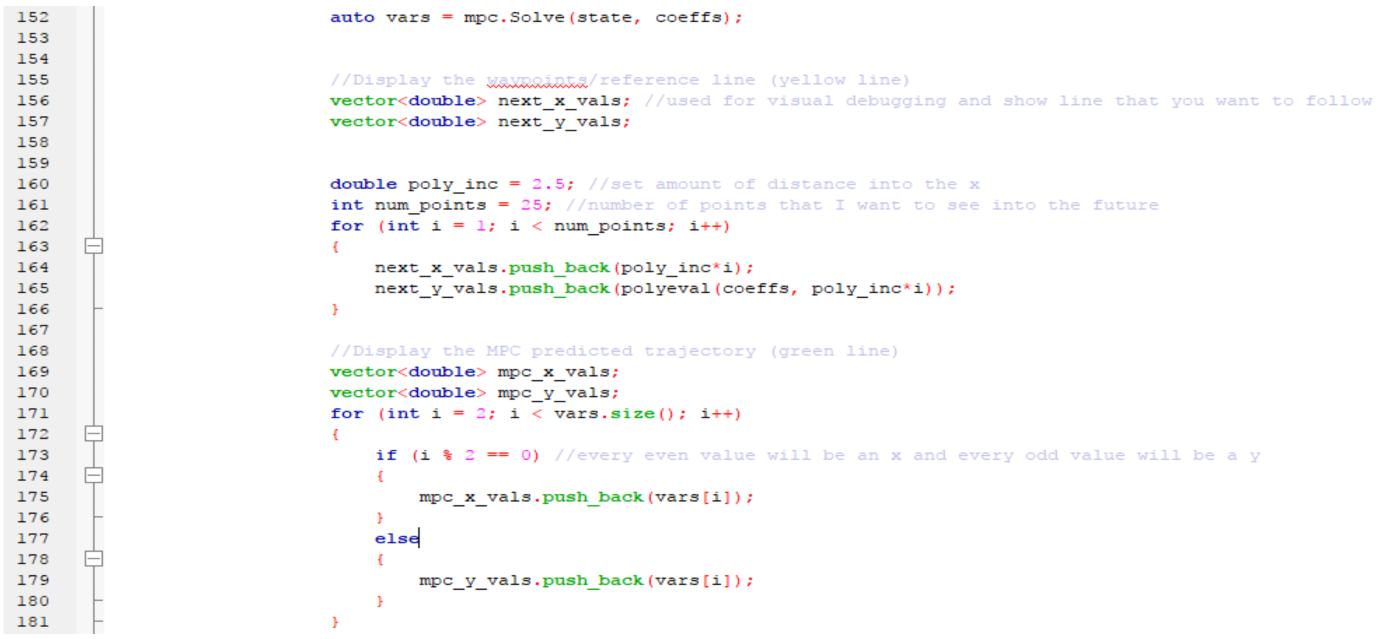
In line 134 we calculate the psi error. Line 134 is a simplification of the code commented in line 133. Because of the pre-processing in lines 106-117, psi and px are zero, which simplifies line 133 to line 134.

Finally we can fill our state vector in line 141. The x, y, and psi have been set to zero in lines 106-117. The speed v comes straight from the simulator and cte and epsi have been calculated in lines 132 and 134.

Now we have the current state vector and the fitted coefficients of the polynomial that the car should follow. With these inputs we can calculate in line 152 the optimal values for the actuators (steering angle and throttle). The steering angle is saved in vars[0] and the throttle in vars[1].



```
140                      Eigen::VectorXd state(6);
141                      state << 0, 0, 0, v, cte, epsi;
142
143
144                      /*
145                      * TODO: Calculate steering angle and throttle using MPC.
146                      *
147                      * Both are in between [-1, 1].
148                      *
149                      */
150
151
152                      auto vars = mpc.Solve(state, coeffs);
```



```
152                 auto vars = mpc.Solve(state, coeffs);
153
154
155                 //Display the waypoints/reference line (yellow line)
156                 vector<double> next_x_vals; //used for visual debugging and show line that you want to follow
157                 vector<double> next_y_vals;
158
159
160                 double poly_inc = 2.5; //set amount of distance into the x
161                 int num_points = 25; //number of points that I want to see into the future
162                 for (int i = 1; i < num_points; i++)
163                 {
164                     next_x_vals.push_back(poly_inc*i);
165                     next_y_vals.push_back(polyeval(coeffs, poly_inc*i));
166                 }
167
168                 //Display the MPC predicted trajectory (green line)
169                 vector<double> mpc_x_vals;
170                 vector<double> mpc_y_vals;
171                 for (int i = 2; i < vars.size(); i++)
172                 {
173                     if (i % 2 == 0) //every even value will be an x and every odd value will be a y
174                     {
175                         mpc_x_vals.push_back(vars[i]);
176                     }
177                     else
178                     {
179                         mpc_y_vals.push_back(vars[i]);
180                     }
181                 }
```

Finally in line 160-166 we calculate the points of the yellow line and in lines 169-181 the points from the green line in the simulator.

- MPC.cpp:

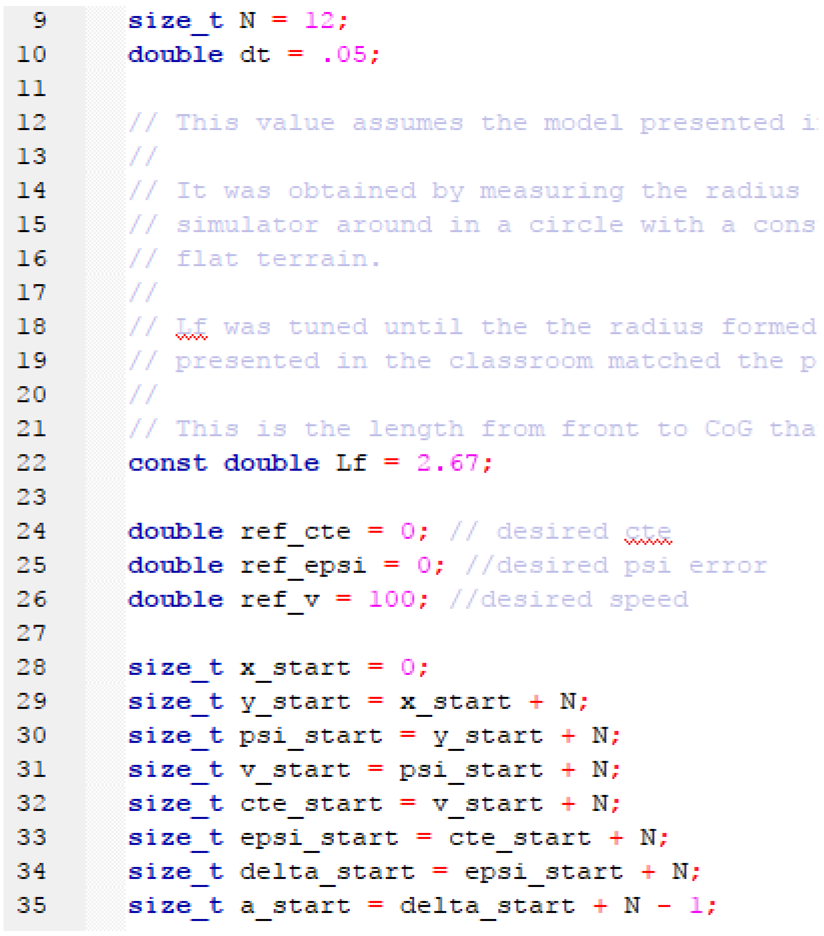
This file contains the code to optimise the cost function and produce the optimal steer and throttle inputs.

The choice of N and dt is explained in section 2.

The value of Lf is where the center of gravity is located in the car. I tried values of 2 and 3, but it didn't seem to affect the behaviour of the car too much.

In lines 24-26, we tell the model what the desired values are for cte, epsi and speed. Cte and epsi should be as close as possible because we want the car to follow the waypoints curve as close as possible. Desired speed is set to 100 because we want the car to go as fast as possible. In this case 100 mph is the max speed the car can go.

In lines 28-35, we generate variables that help us to find the values of the state and actuator variables. This is needed because all these variables are stored in 1 dimensional vectors.



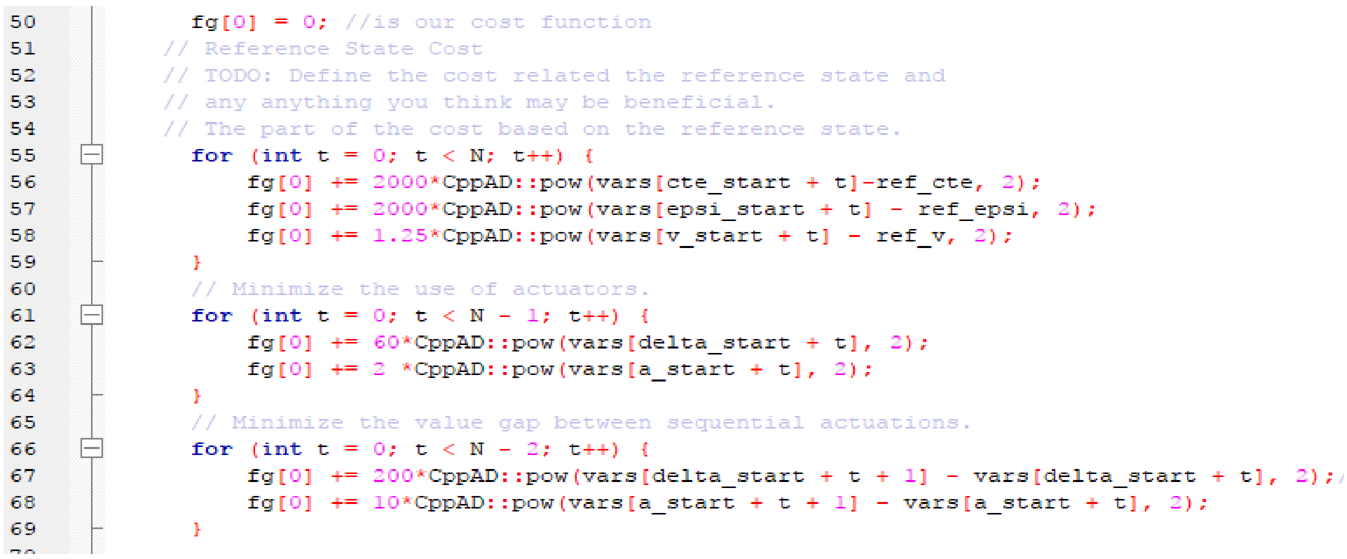
```
 9    size_t N = 12;
10    double dt = .05;
11
12    // This value assumes the model presented i
13    //
14    // It was obtained by measuring the radius
15    // simulator around in a circle with a cons
16    // flat terrain.
17    //
18    // Lf was tuned until the the radius formed
19    // presented in the classroom matched the p
20    //
21    // This is the length from front to CoG tha
22    const double Lf = 2.67;
23
24    double ref_cte = 0; // desired cte
25    double ref_epsi = 0; //desired psi error
26    double ref_v = 100; //desired speed
27
28    size_t x_start = 0;
29    size_t y_start = x_start + N;
30    size_t psi_start = y_start + N;
31    size_t v_start = psi_start + N;
32    size_t cte_start = v_start + N;
33    size_t epsi_start = cte_start + N;
34    size_t delta_start = epsi_start + N;
35    size_t a_start = delta_start + N - 1;
```

In lines 50-69, we update our cost function fg[0]. In lines 55-59, we compare our calculated value with the desired reference values for our state variables (cte, epsi and v). We can increase the importance/weights of a specific parameter by multiplying it with a constant. For example in line 56 we gave cte a weight of 2000 which means that a lot of weight/priority should be given to minimizing the cte. If we increase the weight of speed in line 58, then the model will also try and go as fast as possible. Tuning the weights of each of the update functions in lines 55-69 will have an impact on how the car drives.

In lines 61-64, we try to minimize the use of our actuators (steering and throttle) because this will result in a more comfortable and smooth ride. Also here we could give weights to these actuator update functions.

Finally in lines 66-69, we try to smoothen the steering and throttle inputs by minimizing the gap between 2 steering or throttle inputs.

The weights of each of the update functions have been manually tuned by trying different combinations of these weigths and measuring their impact on the lap time.



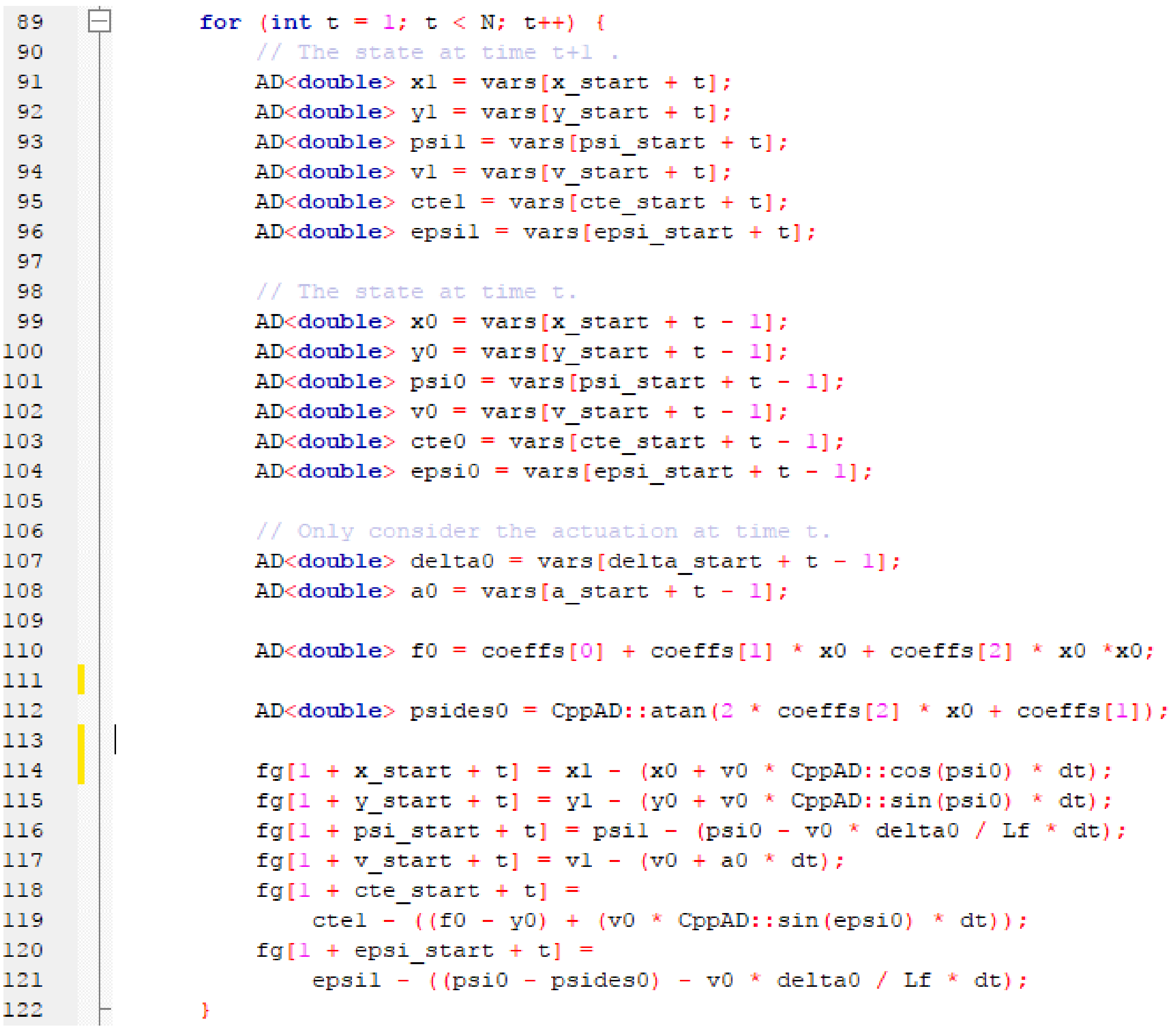
```
  fg[0] = 0; //is our cost function
// Reference State Cost
// TODO: Define the cost related the reference state and
// any anything you think may be beneficial.
// The part of the cost based on the reference state.
  for (int t = 0; t < N; t++) {
      fg[0] += 2000*CppAD::pow(vars[cte_start + t]-ref_cte, 2);
      fg[0] += 2000*CppAD::pow(vars[epsi_start + t] - ref_epsi, 2);
      fg[0] += 1.25*CppAD::pow(vars[v_start + t] - ref_v, 2);
  }
  // Minimize the use of actuators.
  for (int t = 0; t < N - 1; t++) {
      fg[0] += 60*CppAD::pow(vars[delta_start + t], 2);
      fg[0] += 2 *CppAD::pow(vars[a_start + t], 2);
  }
  // Minimize the value gap between sequential actuations.
  for (int t = 0; t < N - 2; t++) {
      fg[0] += 200*CppAD::pow(vars[delta_start + t + 1] - vars[delta_start + t], 2);
      fg[0] += 10*CppAD::pow(vars[a_start + t + 1] - vars[a_start + t], 2);
  }
```

In lines 89-122 the model constraints are defined. In lines 91-96 we define the state variables at time t+1 and in lines 99-104 the state variables at time t are defined.

In line 110 we calculate the desired car's position (f0) and in line 112 we calculate the desired psi value (psides0), which is the first derivative of f0. f0 and psides0 are used to calculate cte and epsi cost constraints in lines 118-121.

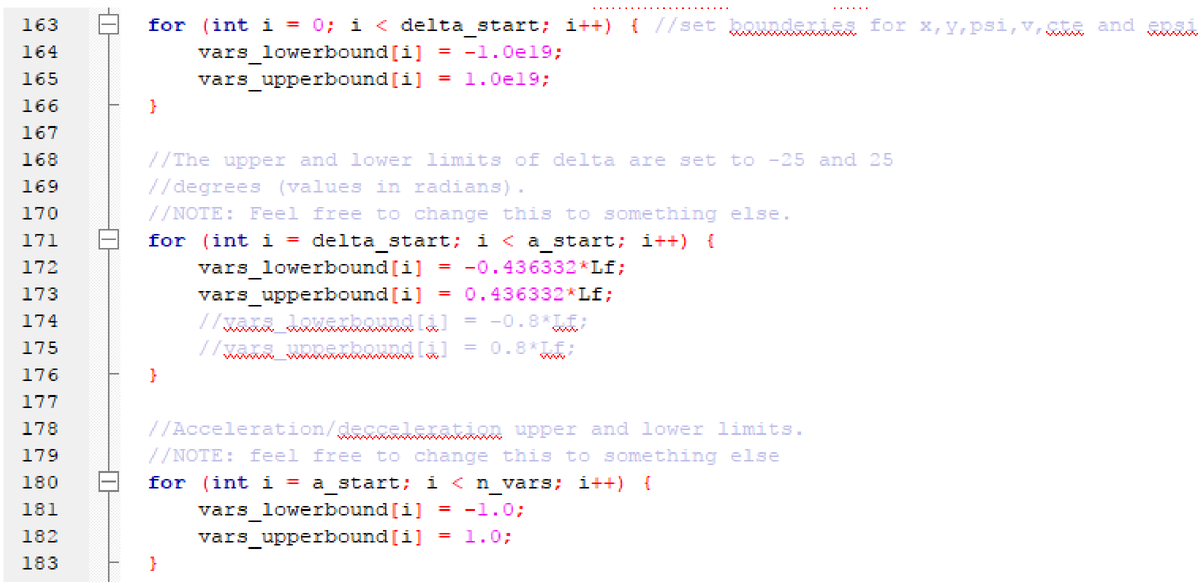
As discussed in section 3, our car seemed to perform better if we use a 2nd degree polynomial instead of 3rd degree.

Lines 114-122 are the model constraints, where we basically make sure that the state value at time t is equal to the state value at time t-1 + the rate of change of this state variable over 1 time period. The CppAD library is calculating the rate of change (derivative).



```cpp
    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 + coeffs[2] * x0 *x0;

        AD<double> psides0 = CppAD::atan(2 * coeffs[2] * x0 + coeffs[1]);

        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);
    }
```

Lines 129-263 contains the code for the Solve() function which is used in line 152 of main.cpp. It takes as arguments the state vector and the coefficients of our polynomials. It returns the optimal values for the actuators and the positions for the next time intervals.

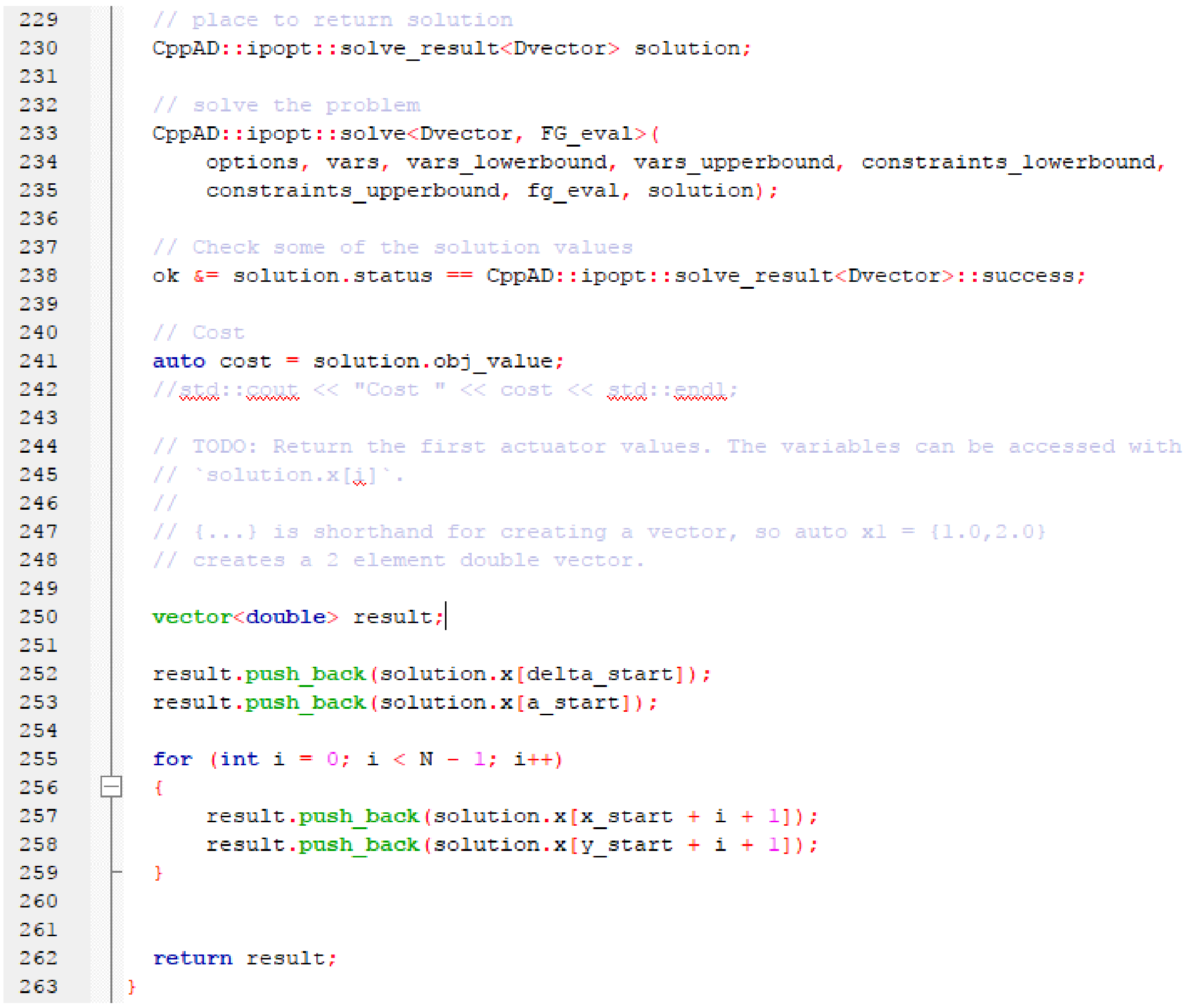


```cpp
for (int i = 0; i < delta_start; i++) { //set bounderies for x,y,psi,v,cte and epsi
    vars_lowerbound[i] = -1.0e19;
    vars_upperbound[i] = 1.0e19;
}

//The upper and lower limits of delta are set to -25 and 25
//degrees (values in radians).
//NOTE: Feel free to change this to something else.
for (int i = delta_start; i < a_start; i++) {
    vars_lowerbound[i] = -0.436332*Lf;
    vars_upperbound[i] = 0.436332*Lf;
    //vars_lowerbound[i] = -0.8*Lf;
    //vars_upperbound[i] = 0.8*Lf;
}

//Acceleration/decceleration upper and lower limits.
//NOTE: feel free to change this to something else
for (int i = a_start; i < n_vars; i++) {
    vars_lowerbound[i] = -1.0;
    vars_upperbound[i] = 1.0;
}
```

In lines 163-183 we set the upper and lower bounds for the constraints. Lines 163-166 sets the lower bounds of the state variables [x,y,psi,v,cte,epsi] to very big ans small values. This actually means that we are not constraining the state variables.

In lines 171-176 we constrain the steering angle of the car to +/- 25 degrees (0.43 radians is 25 degrees). We multiply it by Lf, because we later will devide it by Lf in the equation where this variable is used.

In lines 180-183 we set the acceleration limits to +/-1.



```
229       // place to return solution
230       CppAD::ipopt::solve_result<Dvector> solution;
231
232       // solve the problem
233       CppAD::ipopt::solve<Dvector, FG_eval>(
234           options, vars, vars_lowerbound, vars_upperbound, constraints_lowerbound,
235           constraints_upperbound, fg_eval, solution);
236
237       // Check some of the solution values
238       ok &= solution.status == CppAD::ipopt::solve_result<Dvector>::success;
239
240       // Cost
241       auto cost = solution.obj_value;
242       //std::cout << "Cost " << cost << std::endl;
243
244       // TODO: Return the first actuator values. The variables can be accessed with
245       // `solution.x[i]`.
246       //
247       // {...} is shorthand for creating a vector, so auto x1 = {1.0,2.0}
248       // creates a 2 element double vector.
249
250       vector<double> result;
251
252       result.push_back(solution.x[delta_start]);
253       result.push_back(solution.x[a_start]);
254
255       for (int i = 0; i < N - 1; i++)
256       {
257           result.push_back(solution.x[x_start + i + 1]);
258           result.push_back(solution.x[y_start + i + 1]);
259       }
260
261
262       return result;
263   }
```

In lines 230-263 the solution is written to the result vector and returned to the program. The first 2 elements of the result vector are the values for the actuators which give us the steering and acceleration that the car needs to do in the next time step.

Lines 255-259 enter the coordinates of the points where the car is going in the future (points of the green line).

2. Chosen N (timestep length) and dt (elapsed duration between timesteps)

Student discusses the reasoning behind the chosen N (timestep length) and dt (elapsed duration between timesteps) values. Additionally the student details the previous values tried.

The number of timesteps N and the timestep duration dt were the first parameters that I tuned. The optimal values that I found through trial and error were N=12 and dt=0.05. This is equal at looking 0.6 secs ahead. At a speed of 90mph, this means looking 40 meters (8 car lengths) forward.

The criterium for the optimal value is the time needed for the car to complete 1 lap. I increased N and decreased dt up to the point where the car crashed.

I initially started with a value N=10 and dt=0.1. Then I gradually increased N until the car crashed. Next I started decreasing dt until the car crashed and at that point I decreased N until the car could complete a lap with the new dt value. I repeated this process until I reached the lowest possible value for dt.

I found that if I used a value for N that was too large, then the car would be more likely to crash. N is actually how many steps you look into the future. If you have a large N, you risk to try and predict points in the future that are too uncertain to predict (too many uncertainties for the furthest points). In this way the optimiser might decide to throw the car of the track because this is benefiting the overall cost. Increasing N also means increasing the number of variables in your cost function and the computation time:

$$[\delta_1, a_1, \delta_2, a2, ..., \delta N - 1, a_{N-1}]$$

Dt is the amount of time between 2 time steps. If this is too large, there is too much time between 2 steering and throttle inputs, making the car under react and becoming unstable.

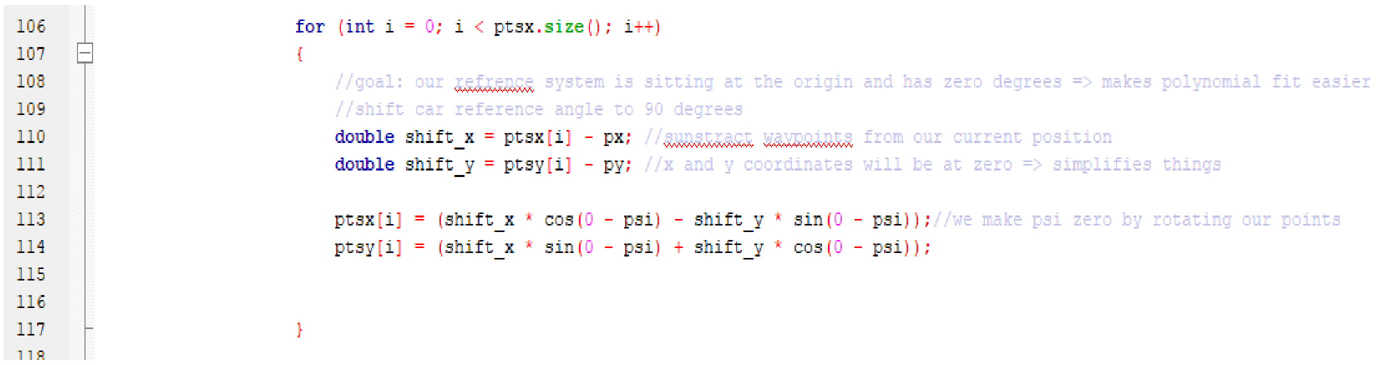
3. Fitted polynomial and waypoints preprocessing

A polynomial is fitted to waypoints.

I tried polynomials of 2nd, 3rd, 4th and 5th degree and found that 2nd degree gave me the best lap times.

If the student preprocesses waypoints, the vehicle state, and/or actuators prior to the MPC procedure it is described.

In code line 106-117 of main.cpp we shift the x and y coordinates (lines 110-11) so that they are zero.
In lines 113-114 we make psi also zero by rotating all the waypoints.
These 2 adjustments will simplify the polynomial fit and calculation of cte and epsi as explained above in section 1.

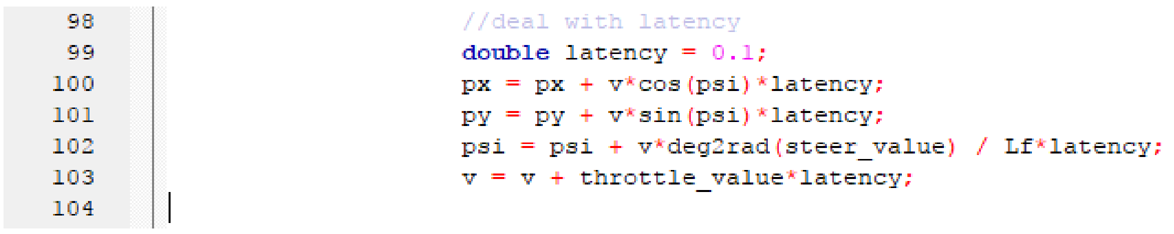


```
                    for (int i = 0; i < ptsx.size(); i++)
                    {
                        //goal: our refrence system is sitting at the origin and has zero degrees => makes polynomial fit easier
                        //shift car reference angle to 90 degrees
                        double shift_x = ptsx[i] - px; //sunstract waypoints from our current position
                        double shift_y = ptsy[i] - py; //x and y coordinates will be at zero => simplifies things

                        ptsx[i] = (shift_x * cos(0 - psi) - shift_y * sin(0 - psi));//we make psi zero by rotating our points
                        ptsy[i] = (shift_x * sin(0 - psi) + shift_y * cos(0 - psi));


                    }
```

4. <u>Implementation of 100 millisecond latency</u>

The student implements Model Predictive Control that handles a 100 millisecond latency. Student provides details on how they deal with latency.

Code lines 99-103 in the file main.cpp, deal with the 100 milliseconds latency between the actual steering and throttle inputs and the response of the car on these inputs.



```
 98                 //deal with latency
 99                 double latency = 0.1;
100                 px = px + v*cos(psi)*latency;
101                 py = py + v*sin(psi)*latency;
102                 psi = psi + v*deg2rad(steer_value) / Lf*latency;
103                 v = v + throttle_value*latency;
104
```

In line 99 we declare a variable latency which contains the 100 milliseconds delay between input and response of the car.

In lines 100-103, we adjust the state variables [x,y,psi,v] by forecasting their values after the 100 milliseconds have elapsed. We basically apply the 4 equations below so that we have adjusted state variables before we feed them to the model. In line 103, we use throttle value as a proxy for acceleration.

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$
$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$
$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$
$$v_{t+1} = v_t + a_t * dt$$

If we run our tuned model without the latency correction, the car crashes. Because of the high speed and the latency, the car responds too late on the steering and throttle inputs and starts to swing and overshoots. With the code lines above, the model looks 100 milliseconds ahead and anticipates on the effects of the current steer_value and throttle_value.