Write up Project MPC, Term 2.

I worked on main.cpp and MPC.cpp

Implementation of model:

1. Describe model in detail. This includes the state, actuators and update equations.

The goal of MPC is to get values for actuators like acceleration (a) and Steering angle (delta and not psi_dot) so that the cross track error (cte) and heading error is minimum. This optimization problem is solved through Ipopt and it comes up with values of (a, psi_dot) for each short step the car takes.
Here are some detailed steps:

```
// Solver takes  state variables and actuator invariables in a singular vector.
// Thus, we should to establish when one variable starts and another ends to make
// our lifes easier.
size_t x_start = 0;
size_t y_start = x_start + N;
size_t psi_start = y_start + N;
size_t v_start = psi_start + N;
size_t cte_start = v_start + N;
size_t epsi_start = cte_start + N;
size_t delta_start = epsi_start + N;
size_t a_start = delta_start + N - 1;
```

Next, FG eval:

important here are the weights:

```
  // individual optimization weights
    const double cte_cost_weight = 1800;
    const double epsi_cost_weight = 1200;
    const double v_cost_weight = 0.9;
    const double delta_cost_weight = 249;
    const double a_cost_weight = 33;
    const double delta_change_cost_weight = 85;
    const double a_change_cost_weight = 12;
```

Next, setting up constraints:

```
    // Setup model constraints

    // Initial constraints
    // We add 1 to each of the starting indices due to cost being located at
    // index 0 of `fg`. This bumps up the position of all the other values.
    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];

  // The rest of the constraints
  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 = 0.0;
    for (int i = 0; i < coeffs.size(); i++) {
      f0 += coeffs[i] * CppAD::pow(x0, i);
    }

    AD<double> psides0 = 0.0;
    for (int i = 1; i < coeffs.size(); i++) {
      psides0 += i*coeffs[i] * CppAD::pow(x0, i-1); // f'(x0)
```

Next, MPC class:

define Testvector

```
vector<double> MPC::Solve(Eigen::VectorXd state, Eigen::VectorXd coeffs) {
  bool ok = true;
  typedef CPPAD_TESTVECTOR(double) Dvector;

  double x = state[0];
  double y = state[1];
  double psi = state[2];
  double v = state[3];
  double cte = state[4];
  double epsi = state[5];
```

Number of model variables:

```
size_t n_vars = N * 6 + (N - 1) * 2;
```

```
// Number of model constraints
  size_t n_constraints = N * 6;
```

Next: Object that computes objective and constraints

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

Solving for all variables:

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

2. Reasoning behind chosen N (timestep length) and dt (elapsed duration between timesteps) values.

N and dt are parameters that determine discrete points which are used by the polynomial fit function to predict the path. We don't want too much gap or too little. More specifically:
The factors which matter when choosing N and dt are the amount of distance covered by the car between 2 successive update cycles. If N*dt to be too long (lets say more than 3 seconds) since the car would cover a significant distance which may not entirely match with the received reference trajectory. Too short is not good, either. Experimenting with values yields an optimal value of N=10 and dt=0.1.

3. A polynomial is fitted to waypoints.

Preprocessing the waypoints:
line 107 et. all in main.cpp

```
for (int i=0;i < ptsx .size(); i++)
          {
             double shift_x = ptsx[i] - px;
             double shift_y = ptsy[i] - py;

           ptsx[i] = (shift_x * cos (0-psi) - shift_y * sin (0 - psi));
           ptsy[i] = (shift_x * sin (0-psi) + shift_y * cos (0 - psi));
          }
```

Next we find the coefficients for fitting the polynomial

```
          / Let's use the given function polyfit to "fit" the polynomial using 3rd degree power
           We have to makesure we pass vectors as Eigen vectors topolyfit
           */

          double* ptrx = &ptsx[0];
          Eigen::Map<Eigen::VectorXd> ptsx_transform(ptrx, 6);


          double* ptry = &ptsy[0];
          Eigen::Map<Eigen::VectorXd> ptsy_transform(ptry, 6);
```

```
auto coeffs = polyfit (ptsx_transform, ptsy_transform, 3 );

// calculate cross track error(cte)

double cte = polyeval(coeffs,0);

//calculate orientation error
// double epsi = psi - atan ( coeffs[1] + 2 * px * coeffs[2] + 3 * px * coeffs[3] * pow (px,2));
double epsi = -atan(coeffs[1]);//as psi and px are 0
```

We fit the model in line 119

```
auto coeffs = polyfit(ptsx_veh, ptsy_veh, 3);
```

4. Implement Model Predictive Control that handles a 100 millisecond latency. How to deal with latency.

```
// Latency for predicting time at actuation
	const double dt = 0.1;

	// Predict state after latency
	// x, y and psi are all zero after transformation above
	double pred_px = 0.0 + v * dt; // Since psi is zero, cos(0) = 1, can leave out
	const double pred_py = 0.0; // Since sin(0) = 0, y stays as 0 (y + v * 0 * dt)
	double pred_psi = 0.0 + v * -delta / Lf * dt;
	double pred_v = v + a * dt;
	double pred_cte = cte + v * sin(epsi) * dt;
	double pred_epsi = epsi + v * -delta / Lf * dt;

	// Feed in the predicted state values
	Eigen::VectorXd state(6);
	state << pred_px, pred_py, pred_psi, pred_v, pred_cte, pred_epsi;

	// Solve for new actuations (and to show predicted x and y in the future)
	auto vars = mpc.Solve(state, coeffs);
```