Project 5 - CarND-MPC-Project-master

For this project, major part is to identify what is the error we wish to minimize, and also the weight on each error also play an important role from my experience.

Also another tedious thing has to highlight here is the environment set up for this project. To be specific, install ipopt really took me more than 1 day figuring out the correct way of doing it, below link is my way of doing that, that is reference working for me:

https://medium.com/@notus.li/install-ipopt-on-ubuntu-16-04-e2644fa93545

First part is to define cost function, and here is the hard code:

I would say the error is straight forward, also all has been mentioned inside lecture, but the weight is something has to play around with.

For instance, if “delta_error_weight” is low like 500/ 600, then in my case, car angel for sure will be unstable especially for area turning right/ left.

Even I change into 700, it also not stable, till 950. now it is stable.

```
//why it is putting previous vector
FG_eval(Eigen::VectorXd coeffs, vector<double> previous_actuations) {
  this->coeffs = coeffs;
  this->previous_actuations = previous_actuations;
}

typedef CPPAD_TESTVECTOR(AD<double>) ADvector;
void operator()(ADvector& fg, const ADvector& vars) {
  // TODO: implement MPC
  // `fg` a vector of the cost constraints, `vars` is a vector of variable values (state & actuators)
  // NOTE: You'll probably go back and forth between this function and
  // the Solver function below.
  fg[0] = 0;
  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);
  }

  for (int t = 0; t < N - 1; t++) {
    fg[0] += CppAD::pow(vars[delta_start + t], 2);
    fg[0] += a_error_weight*CppAD::pow(vars[a_start + t], 2);
  }

  for (int t = 0; t < N - 2; t++) {
    fg[0] += delta_error_weight*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);
  }
```

Below is area where we define the route function, you can see the coeffs parameter stands for route function parameter.

```
    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 = 0; t < N - 1; t++) {
      // The state at time t+1 .
      AD<double> x1 = vars[x_start + t + 1];
      AD<double> y1 = vars[y_start + t + 1];
      AD<double> psi1 = vars[psi_start + t + 1];
      AD<double> v1 = vars[v_start + t + 1];
      AD<double> cte1 = vars[cte_start + t + 1];
      AD<double> epsi1 = vars[epsi_start + t + 1];

      // The state at time t.
      AD<double> x0 = vars[x_start + t];
      AD<double> y0 = vars[y_start + t];
      AD<double> psi0 = vars[psi_start + t];
      AD<double> v0 = vars[v_start + t];
      AD<double> cte0 = vars[cte_start + t];
      AD<double> epsi0 = vars[epsi_start + t];

      // Only consider the actuation at time t.
      AD<double> delta0 = vars[delta_start + t];
      AD<double> a0 = vars[a_start + t];

      AD<double> f0 = coeffs[0] + coeffs[1] * x0 + coeffs[2]*x0*x0 + coeffs[3]*x0*x0*x0;
      AD<double> psides0 = CppAD::atan(coeffs[1]+2*coeffs[2]*x0 + 3 * coeffs[3]*x0*x0);


      fg[2 + x_start + t] = x1 - (x0 + v0 * CppAD::cos(psi0) * dt);
      fg[2 + y_start + t] = y1 - (y0 + v0 * CppAD::sin(psi0) * dt);
      fg[2 + psi_start + t] = psi1 - (psi0 + v0 * delta0 / Lf * dt);
      fg[2 + v_start + t] = v1 - (v0 + a0 * dt);
      fg[2 + cte_start + t] =
          cte1 - ((f0 - y0) + (v0 * CppAD::sin(epsi0) * dt));
      fg[2 + epsi_start + t] =
          epsi1 - ((psi0 - psides0) + v0 * delta0 / Lf * dt);
    }
  }
;
```

And below is area we give lower & upper bound for each parameter like delta has to be less than 25 degree for both left & right side.

The way I am choosing N & dt is quick straight forward, I didn't worry too much about dt, just always give the recommended value 0.05, but N really need to modify even based on different speed of the car. The fast max speed is, the low N should be, this is my finding. I tried from 15, down into 14, 13 etc,.in the end 12 meets my expectation. Model can run properly. I feel like if what to modify dt, need to keep in mind N*dt*max_car_speed should not be changing too much. This should be the distance we are using for forecasting future route map, so too long or too short all incorrect. Since we have max x0*x0*x0.

Also I realize one thing, if N & dt can be dynamic based on current speed, then would be the best since at the beginning, once speed is low, a fix N & dt will make N*dt*car_speed short, which will impact the stability of the route forecasting. That is the reason always at the beginning that is not that stable as other time.

To add here:

First **statement**, for instance, below are starting point of each data group, we have x, y, psi, v is the speed, also CTE delta etc.

```
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;
```

Below are **update equations functions** I am using for updating, this is to refer to lesson mentioned in the Udacity lecture:

```
    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 = 0; t < N - 1; t++) {
      // The state at time t+1 .
      AD<double> x1 = vars[x_start + t + 1];
      AD<double> y1 = vars[y_start + t + 1];
      AD<double> psi1 = vars[psi_start + t + 1];
      AD<double> v1 = vars[v_start + t + 1];
      AD<double> cte1 = vars[cte_start + t + 1];
      AD<double> epsi1 = vars[epsi_start + t + 1];

      // The state at time t.
      AD<double> x0 = vars[x_start + t];
      AD<double> y0 = vars[y_start + t];
      AD<double> psi0 = vars[psi_start + t];
      AD<double> v0 = vars[v_start + t];
      AD<double> cte0 = vars[cte_start + t];
      AD<double> epsi0 = vars[epsi_start + t];

      // Only consider the actuation at time t.
      AD<double> delta0 = vars[delta_start + t];
      AD<double> a0 = vars[a_start + t];

      AD<double> f0 = coeffs[0] + coeffs[1] * x0 + coeffs[2]*x0*x0 + coeffs[3]*x0*x0*x0;
      AD<double> psides0 = CppAD::atan(coeffs[1]+2*coeffs[2]*x0 + 3 * coeffs[3]*x0*x0);


      fg[2 + x_start + t] = x1 - (x0 + v0 * CppAD::cos(psi0) * dt);
      fg[2 + y_start + t] = y1 - (y0 + v0 * CppAD::sin(psi0) * dt);
      fg[2 + psi_start + t] = psi1 - (psi0 + v0 * delta0 / Lf * dt);
      fg[2 + v_start + t] = v1 - (v0 + a0 * dt);
      fg[2 + cte_start + t] =
          cte1 - ((f0 - y0) + (v0 * CppAD::sin(epsi0) * dt));
      fg[2 + epsi_start + t] =
          epsi1 - ((psi0 - psides0) + v0 * delta0 / Lf * dt);
```

Polynomial Fitting and MPC Preprocessing:

Below is the function achive the Polynomial Fitting:

```
AD<double> f0 = coeffs[0] + coeffs[1] * x0 + coeffs[2]*x0*x0 + coeffs[3]*x0*x0*x0;
AD<double> psides0 = CppAD::atan(coeffs[1]+2*coeffs[2]*x0 + 3 * coeffs[3]*x0*x0);
```

In the end, car can run nicely like video attached.

For running the project, you can refer to below steps after your environment setup done:

- If this is not first time running, remove previous folder first using:

rm -R CarND-MPC-Project-master

- You may need to select 'y' for certain file while removing folder. Post that, you can run below code to start project running:

git clone https://github.com/MagicSHX/CarND-MPC-Project-master.git
cd CarND-MPC-Project-master
mkdir build && cd build
cmake .. && make

- Last step: making sure simulation started before running below code in ubuntu:

./mpc