

// In this quiz you'll implement the global kinematic model.

#include <math.h>

#include <iostream>

#include "Dense"

//

// Helper functions

//

double pi() { return M\_PI; }

double deg2rad(double x) { return x \* pi() / 180; }

double rad2deg(double x) { return x \* 180 / pi(); }

const double Lf = 2;

Eigen::VectorXd globalKinematic(Eigen::VectorXd state,

Eigen::VectorXd actuators, double dt) {

Eigen::VectorXd next\_state(state.size());

//TODO: Implement the Global Kinematic Model, to return

// the next state from inputs

// NOTE: state is [x, y, psi, v]

// NOTE: actuators is [delta, a]

next\_state[0]=state[0]+state[3]\*cos(state[2])\*dt;

next\_state[1]=state[1]+state[3]\*sin(state[2])\*dt;

next\_state[2]=state[2]+state[3]/Lf\*actuators[0]\*dt;

next\_state[3]=state[3]+actuators[1]\*dt;

//Add your code below

return next\_state;

}

int main() {

// [x, y, psi, v]

Eigen::VectorXd state(4);

// [delta, v]

Eigen::VectorXd actuators(2);

state << 0, 0, deg2rad(45), 1;

actuators << deg2rad(5), 1;

Eigen::VectorXd next\_state = globalKinematic(state, actuators, 0.3);

std::cout << next\_state << std::endl;

}

// In this quiz you'll fit a polynomial to waypoints.

#include <iostream>

#include "Dense"

using namespace Eigen;

// Evaluate a polynomial.

double polyeval(Eigen::VectorXd coeffs, double x) {

double result = 0.0;

for (int i = 0; i < coeffs.size(); i++) {

result += coeffs[i] \* pow(x, i);

}

return result;

}

// Fit a polynomial.

// Adapted from

// https://github.com/JuliaMath/Polynomials.jl/blob/master/src/Polynomials.jl#L676-L716

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;

}

int main() {

Eigen::VectorXd xvals(6);

Eigen::VectorXd yvals(6);

// x waypoint coordinates

xvals << 9.261977, -2.06803, -19.6663, -36.868, -51.6263, -66.3482;

// y waypoint coordinates

yvals << 5.17, -2.25, -15.306, -29.46, -42.85, -57.6116;

// TODO: use `polyfit` to fit a third order polynomial to the (x, y)

// coordinates.

// Hint: call Eigen::VectorXd polyfit() and pass xvals, yvals, and the

// polynomial degree/order

// YOUR CODE HERE

for (double x = 0; x <= 20; x += 1.0) {

// TODO: use `polyeval` to evaluate the x values.

std::cout << polyeval(polyfit(xvals,yvals,3),x) << std::endl;

}

// Expected output

// -0.905562

// -0.226606

// 0.447594

// 1.11706

// 1.7818

// 2.44185

// 3.09723

// 3.74794

// 4.39402

// 5.03548

// 5.67235

// 6.30463

// 6.93236

// 7.55555

// 8.17423

// 8.7884

// 9.3981

// 10.0033

// 10.6041

// 11.2005

// 11.7925

}

We can express the error between the center of the road and the vehicle's position as the cross track error (CTE).

*ctet*+1​=*yt*​−*f*(*xt*​)+(*vt*​∗*sin*(*eψt*​)∗*dt*)

This can be broken up into two parts:

1. y\_t - f(x\_t)*yt*​−*f*(*xt*​) being current cross track error.
2. v\_t \* sin(e\psi\_t) \* dt*vt*​∗*sin*(*eψt*​)∗*dt* being the change in error caused by the vehicle's movement.

e\psi\_{t+1} = \psi\_t - \psi{des}\_t + (\frac{v\_t} { L\_f} \* \delta\_t \* dt)*eψt*+1​=*ψt*​−*ψdest*​+(*Lf*​*vt*​​∗*δt*​∗*dt*)

Similarly to the cross track error this can be interpreted as two parts:

1. \psi\_t - \psi{des}\_t*ψt*​−*ψdest*​ being current orientation error.
2. \frac{v\_t} { L\_f} \* \delta\_t \* dt*Lf*​*vt*​​∗*δt*​∗*dt* being the change in error caused by the vehicle's movement.