Regression Methods in the Control of Dynamic Systems

Jackson Banbury
1000331553

CSC2515
Fall, 2017

Abstract

This project seeks to explore the application of regression methods in machine learning in the space of dynamic control systems. The objective is to use regression to fit and predict trends in physical measurements used in a system that are subject to stochastic noise in order to offer better convergence to a trajectory or equilibrium. The performance of the regression methods will be compared to more traditional filtering methods commonly used in linear and nonlinear control.

Two experiments were used to offer a practical comparison between regression and filtering methods. The first is a mouse-tracking program, which adds stochastic noise to the path of a users mouse input, offering a visual, qualitative comparison, as well as a quantitative comparison in the error between predicted and true mouse paths. The second experiment is the balancing of a two-wheeled robot, which uses regression and filtering separately to mitigate the stochastic noise present in angle measurements in order to improve balance. In the latter experiment, there is no precise measurement to be used as target, so the regression methods will be used in conjunction with a filter and compared to themselves and the performance of the filter instead of a true target.

The two experiments had conflicting results in the performance of the regression methods. The mouse-tracking experiment showed that standard linear regression and lasso-regularized linear regression are viable alternatives to filters depending on the level of noise and the degree of nonlinearity in the trajectory. The balancing robot experiment showed that the performance of these methods suffer significantly without a true training target, and were significantly outperformed by the traditional filtering methods. In each case, however, the regression methods showed promising results, which with more investigation could be a useful tool in the control of oscillatory dynamic systems.

Introduction

In the control of dynamic systems, success is often measured by the proximity of a system to a desired trajectory or equilibrium state while maintaining robustness to inherent noise. Measurement devices such as accelerometers, gyroscopes, altimeters, GPS, etc. often come with varying degrees of stochastic noise. In systems that require continuous measurement to maintain equilibrium, a filter is often used to help mitigate this noise. It does so by measuring a moving average of the noisy measurements at one or higher dimensions to obtain an

underlying trend. There are a number of different types of noise filters, the majority of which observe the noisy measurements on one or higher orders to make estimates on the trend of the underlying mean. These filters have a few shortcomings, including a linear assumption in most, and the inability to recognize patterns in measurements to improve performance. A better understanding of these patterns could provide a much more effective dynamic control system. While typically only used retrospectively, evaluating patterns underneath stochastic noise is the primary use-case of regression methods in machine learning, so in theory its potential to be applied in this space is promising.

The Filters

While machine learning methods have been used in conjunction with filters such as the Wiener filter in signal processing, this project will focus purely on dynamic control. As such, the filters that will be observed are the Kalman filter, the extended Kalman filter, and to a lesser extent the complementary filter.

A Kalman filter is a Bayesian optimal estimation algorithm, often used in cases where system states cannot be measured directly in the presence of noise. In short, the simple Kalman filter uses the collected noisy data to provide a joint probability distribution of the underlying true measurement based on the noisy input. It is essentially continuously estimating the orientation by observing the angular velocity. The first step in evaluating the filtered angle is known as the prediction step. The predicted state is, assuming the noise has zero mean and u is known, is a function of the previous state prediction, the input control vector, and their corresponding transition matrices:

$$\hat{x}_{i+1|i} = E\big[x_{i+1}|z^i\big] = F_i\hat{x}_{i|i} + G_k u_k$$

The estimate variance, or the mean squared error in the estimate:

$$P_{i+1|i} = F_i P_{i|i} F_i^T + Q_k$$

Now, with this prediction $\hat{x}_{i+1|i}$, another observation is taken: z_{i+1} which is used to update the prediction in what is called the update or correction step:

$$\hat{x}_{i+1|i+1} = K'_{i+1}\hat{x}_{i+1|i} + K_{i+1}z_{i+1}$$

where K' and K are weight (gain) matrices. K' can be set to 1, and K, the Kalman gain, can be found with the following relationship:

$$K_{i+1} = P_{i+1|i}H_{i+1}^T\big[H_{i+1}P_{i+1|i}H_{i+1}^T + R_{i+1}\big]^{-1}$$

where R_{i+1} is the observation noise and H_i is the output transition matrix.

Where the simple Kalman filter makes a linear assumption about the observed measurements, the extended Kalman filter expands on this by linearizing around the current estimate by re-calculating a new Jacobian with each update, for both the prediction and update steps. Effectively this means that it is able to more adequately account for nonlinear trends. The functions are the same as above, except that the state and output transition matrices are partially differentiated, becoming Jacobians:

$$F_{i+1} = \frac{\partial f_{i+1}}{\partial x}(\hat{x}_i) \qquad \text{and} \qquad H_{i+1} = \frac{\partial h_{i+1}}{\partial x}(\hat{x}_{i+1|i})$$

The complementary filter is more specific to the application of the balancing robot, as it takes advantage of the two methods to measure angular acceleration from the onboard IMU, or Inertial Measurement Unit. The first is the simple gyroscope readings, the second is the time-derivative of the accelerometer. The complementary filter forms an estimate based on both of these measurements as well as a priori, in this case its own calculated angle from a previous calculation. The C++ application of this is a single line (appdx 0.2).
Because the measurements used in dynamic control are nearly always continuous, regression methods are used exclusively instead of classification methods. Based on the course material in CSC2515, linear regression and support vector regression (SVR) were selected as candidates for the experiments. Both methods were implemented using the free scikit-learn library in Python. More details on their implementation are explained later on a per-experiment basis

Mouse-tracking Experiment

The mouse-tracking experiment was intended to be a quick preliminary experiment in order to offer a qualitative comparison of the filters and regression methods. The mouse input from the user is captured in (x,y) coordinates, onto which a normally distributed noise is applied. The filters accept only this noisy trajectory as an input, and attempt to smooth the noise using the aforementioned methods to track a path that should be as close as possible to the actual underlying mouse trajectory. The regression methods use the noisy trajectory as input data and the true trajectory as training targets. It then uses the following noisy inputs to provide a prediction of what the unknown corresponding true trajectory is. This is conveyed below, with true and noisey trajectory in light and dark blue, respectively:

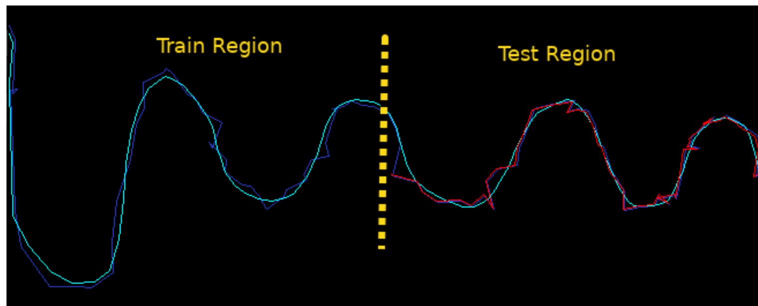


Figure 1: Simple Linear Regression on the Mouse Trajectory

Because of the multivariate nature of the experiment, SVR was not implemented, instead using lasso-regularized linear regression in its place as an alternative method to standard linear regression.
The mouse trajectories used to evaluate the filters and regression methods were intentionally oscillatory to mimic the nature of the two-wheel balancing robot system. This provides a repeating pattern of the true trajectory with which the regression methods should theoretically be able to map to offer better results. A visual of the results of the experiments is shown below:

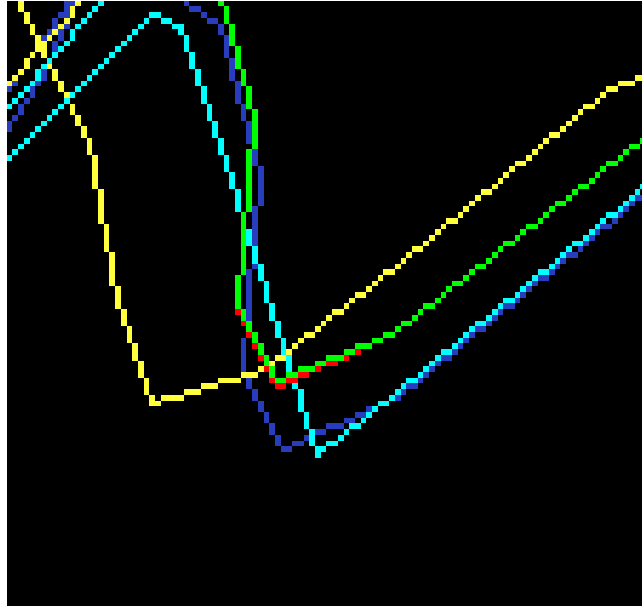


Figure 2: Close-up Comparison of Mouse-tracking Performance

From the figure above, the bright yellow KF follows the trajectory, but is still significantly diverged due to repeated nonlinearity. The standard linear regression model, red, and the lasso-regularized linear regression model, green, perform better than the filters, but they have the advantage of knowing the true underlying mouse position from the previous points for training. The EKF is a poor estimator in this case; it filters out the oscillation almost entirely, and was thus omitted from this figure. The complementary filter was almost indistinguishable from the KF and was omitted from the figure for clarity.

Quantitatively, the performance of the filters and regression methods based on the mean of the euclidean distance in pixels from the true trajectory are shown below. Note that the distance is calculated only over the prediction region of the regression methods, arbitrarily chosen as 50 mouse measurements.

For a smooth, sinusoidal oscillation:

Kalman filter performance:	11.86
Complementary filter performance:	30.77
Standard linear regression performance:	4.08
Lasso-regularized linear regression performance:	4.07

For a sharp, highly nonlinear oscillation:

Kalman filter performance:	29.47
Complementary filter performance:	50.43
Standard linear regression performance:	3.19
Lasso-regularized linear regression performance:	3.19

Unsurprisingly the Kalman and Complementary filters, which make a linear assumption of the data, perform poorly in the highly nonlinear oscillations. The regression methods, however, perform better with the nonlinear input. This is due to their ability to model the pattern of oscillation without diverging due to 'momentum' like the filters.

Robot Experiment

The purpose of the robot experiment was, like the mouse-tracking experiment, to evaluate the performance of the filtering and regression methods in estimating true trajectories in the presence of measurement noise. Unlike the two-dimensional (x,y) coordinates of the mouse-tracker, this experiment uses angle as the only dimension. In addition, unlike the mouse-tracking experiment there is no true trajectory in this case to use as a training or performance reference. As explained previously, the angle and its time-derivative are measured continuously by the inertial measurement unit mounted along the wheel axis between the motors. The unit chosen for this experiment is the popular MPU6050. The mean of its measurements is notoriously accurate despite its low cost, but it is subject to a considerable amount of stochastic noise especially at high-frequency oscillations like in the balancing robot system. The full parts list for the constructed robot can be found in appendix 0.5.

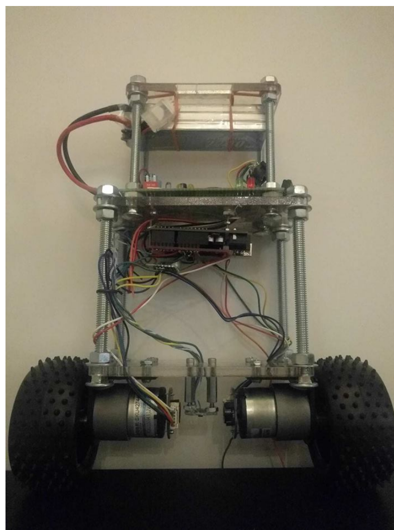


Figure 3: Constructed Balancing Robot

The two-wheeled balancing robot is based on the inverted pendulum model (appdx 0.6). The controller used to balance the robot is a simple PD controller (appdx 0.4) - while more complex controllers such as sliding mode could offer better performance, this controller was chosen so that the balancing success would intentionally depend more on the filtering. The derivations of the inverted pendulum model and the controller are omitted in favor of machine learning-related content.
The raw angle data from a single balancing run along with filtered and regression-predicted angles are shown below:

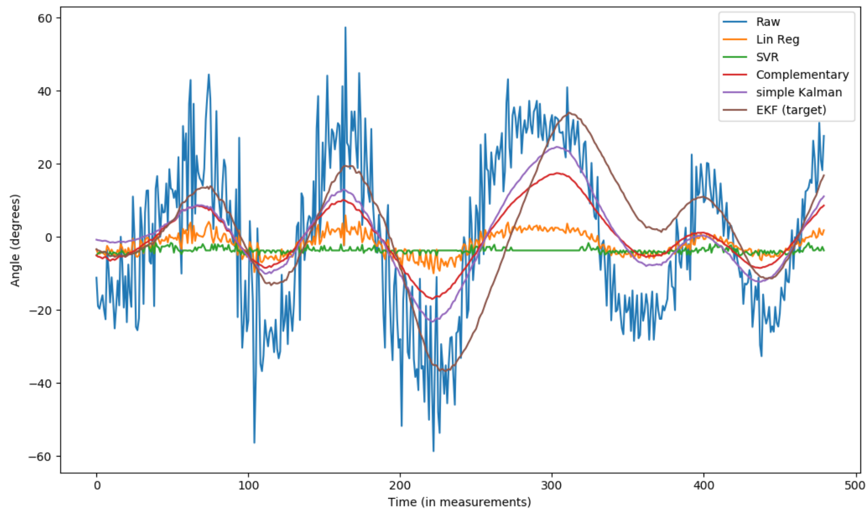


Figure 4: Constructed Balancing Robot

The filters were used in real-time for balancing the robot - in the above case, the simple Kalman filter was used to maintain balance, while the rest were used to simply record filtered angle measurements for comparison. The raw and filtered data was then exported and assessed through the two regression methods. As mentioned previously, the expected explanation for the poor performance of the regression methods compared to the success in the mouse-tracking experiment is that here there is no true angle to train the models. It depends entirely on the success of the filter measurements (in this case, the extended Kalman filter), however it is clearly visible from the charts that the EKF angle diverges significantly from the trend of the raw angle after the first two oscillations. The quantitative performance

comparison based on mean angle variation from the EKF angle is below:

Kalman filter performance:	6.90
Complementary filter performance:	7.54
Standard linear regression performance:	19.10
SVR performance:	18.73

From the plot and the performance values it is clear that the standard linear regression and SVR methods suffer from the lack of a true target value. From the plot, however, linear regression does seem to follow the oscillation in a better phase than the filters, however its amplitude is nowhere near the true angle. It is likely that with more time to adjust filter parameters and perhaps applying a gain to the linear regression estimates, the regression model could be a viable alternative.

Conclusion

Based on the two experiments, between regression and real-time noise filtering, each method has its own benefits and drawbacks. If scaled to higher computational power, the rate of measurement polling could be increased, and the regression models could be adjusted frequently enough for it to be viable. If this were the case, it would benefit in that it would be more self-correcting than the EKF, and more robust to local noise than the KF. This application of regression methods seem like a serious possibility in the future of nonlinear control.

Appendix

0.1 Literary Comparison

This practical implementation of these regression methods in nonlinear dynamic control seems to be a rare combination. Papers closely related to the subject were very sparse, with one notable exception; A group from Stanford's CS229 course published a paper titled 'Improving Accuracy of Intertial Measurement Units using Support Vector Regression' (Reference 11). In the paper the group discusses their experiment using SVR with a low-cost IMU along with a high-accuracy motion-capture system of a person's knee. Immediately the biggest difference between this experiment and the above is the presence of an accurate reference trajectory. The group used the accurate readings to train the model along with the noisy IMU readings, and saw a root-mean-squared error reduction of more that 50% versus the raw angle. The group did not compare these results to any traditional noise filtering methods, however, so it is difficult to compare their success to the balancing robot or mouse-tracking experiments.

0.2 Complementary Filter codes

The C++ code for the balancing robot:

```
float complementary_angle(float accel_angle, float gyro_av, float dt){
//C++ Complementary filter implementation

k1 = 0.98; //set k1 parameter here
k2 = 1 - k1;

comp_angle = ((k1)*(comp_angle + (gyro_av*dt))+(accel_angle*k2));
}
```

The Python code for the mouse-tracker:

```
def complementary_filter_x(x_measured):
    '''
    Calculation is of the form:
    x = k1 * (old_x + x_dot * dt) + k2 * (measured_x)
    Where parameters k1 and k2 are both positive and k1 + k2 = 1
    '''
    # Define the coefficients here:
    k1 = 0.45
    k2 = 1 - k1

    # This gives the time change, dt, in seconds
    global new_time_x
    old_time_x = new_time_x
    new_time_x = time.time()

    # This is set because new_time =/= time from the start of the program
    if old_time_x == 0:
        old_time_x = new_time_x

    dt = new_time_x - old_time_x

    global x_comp
    global x_old
    x_comp = k1 * (x_comp + (x_measured - x_old) * dt) + k2 * (x_measured)

    #print(x_comp)

    return x_comp
```

0.3 Kalman filter and EKF code

For the Kalman.h library, see: https://github.com/TKJElectronics/KalmanFilter

My extended Kalman filter C++ implementation based on the above library is shown below:

```
/*
Adapted from Kristian Lauszus' Kalman.h library under the GPL2 License: https://github.com/Lauszus
GPL2 License
Modified to be an Extended Kalman Filter by Jackson Banbury, 2017
*/
#ifndef _Ex_Kalman_h
#define _Ex_Kalman_h

class Ex_Kalman {
public:
    Ex_Kalman() {
        /* We will set the variables like so, these can also be tuned by the user */
        Q_angle = 0.001;
        Q_bias = 0.003;
        R_measure = 0.03;

        angle = 0; // Reset the angle
        bias = 0; // Reset bias

// Since we assume that the bias is 0 and we know the starting angle (use setAngle), the error
// covariance matrix is set like so:
        P[0][0] = 0;
        P[0][1] = 0;
        P[1][0] = 0;
        P[1][1] = 0;
    };
    // The angle should be in degrees and the rate should be in degrees per second
    // and the delta time in seconds
    double getAngle(double newAngle, double newRate, double dt) {

        // Discrete EXTENDED Kalman filter time update equations - Time Update ("Predict")
        // Update xhat - Project the state ahead
        /* Step 1 */
        rate = newRate - bias;
        //---Unlike the regular Kalman filter, where this was sum(dt * rate)---//
        //---hat{x}_(i) is sum(dt*angle)---//
        angle += dt*newAngle;
        // Update estimation error covariance - Project the error covariance ahead
        /* Step 2 */
        //---This is the same between KF and EKF---//
        P[0][0] += dt * (dt*P[1][1] - P[0][1] - P[1][0] + Q_angle);
        P[0][1] -= dt * P[1][1];
        P[1][0] -= dt * P[1][1];
        P[1][1] += Q_bias * dt;

        // Discrete Kalman filter measurement update equations - Measurement Update ("Correct")
        // Calculate Kalman gain - Compute the Kalman gain
        /* Step 4 */
        S = P[0][0] + R_measure;

        /* Step 5 */
        //---This is the same between KF and EKF---//
        K[0] = P[0][0] / S;
        K[1] = P[1][0] / S;

        // Calculate angle and bias - Update estimate with measurement zk (newAngle)
        /* Step 3 */
        y = newAngle - angle;
        /* Step 6 */

        angle += K[0] * newAngle;
        bias += K[1] * newAngle;

        // Calculate estimation error covariance - Update the error covariance
        /* Step 7 */
        //---These are the same between KF and EKF---//
        P[0][0] -= K[0] * P[0][0];
        P[0][1] -= K[0] * P[0][1];
        P[1][0] -= K[1] * P[0][0];
        P[1][1] -= K[1] * P[0][1];

        return angle;
    };
    void setAngle(double newAngle) { angle = newAngle; }; // Used to set angle, this should be
    // set as the starting angle
    double getRate() { return rate; }; // Return the unbiased rate

    /* These are used to tune the Kalman filter */
    void setQangle(double newQ_angle) { Q_angle = newQ_angle; };
    void setQbias(double newQ_bias) { Q_bias = newQ_bias; };
    void setRmeasure(double newR_measure) { R_measure = newR_measure; };

    double getQangle() { return Q_angle; };
    double getQbias() { return Q_bias; };
    double getRmeasure() { return R_measure; };
```

```
private:
    /* Kalman filter variables */
    double Q_angle; // Process noise variance for the accelerometer
    double Q_bias; // Process noise variance for the gyro bias
    double R_measure; // Measurement noise variance
    double angle; // The angle calculated by the Kalman filter - part of the 2x1 state vector
    double bias; // The gyro bias calculated by the Kalman filter - part of the 2x1 state vector
    double rate; // Unbiased rate calculated from the rate and the calculated bias

    double P[2][2]; // Error covariance matrix - This is a 2x2 matrix
    double K[2]; // Kalman gain - This is a 2x1 vector
    double y; // Angle difference
    double S; // Estimate error
};

#endif
```

The Python EKF library used in the mousetracking script can be found at:
https://github.com/simondlevy/TinyEKF

My simple Kalman filter Python implementation simply uses the OpenCV Kalman filter:

```
def kalman_tracker():
    '''
    Tracks with a simple Kalman (NOT EXTENDED) filter
    '''

    kalman = cv2.KalmanFilter(4,2)
    kalman.measurementMatrix = np.array([[1,0,0,0],[0,1,0,0]],np.float32)
    kalman.transitionMatrix = np.array([[1,0,1,0],[0,1,0,1],[0,0,1,0],[0,0,0,1]],np.float32)
    kalman.processNoiseCov = np.array([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]],np.float32) * 0.03

    return kalman
```

0.4 PD Controller

The equation for the PD controller for the robot is as follows:

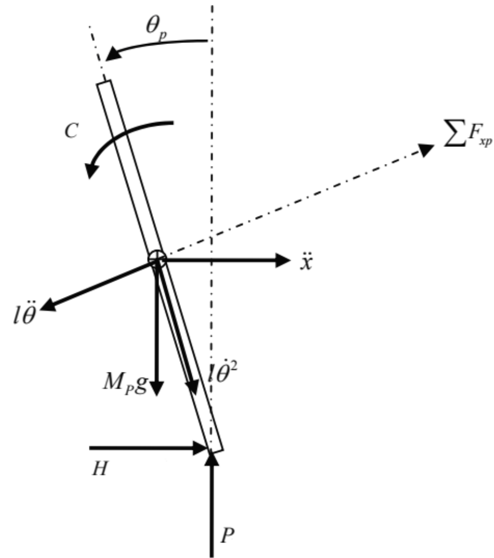
$$U = K_{p\theta}\theta + K_{d\theta}\dot{\theta} + K_{px}x + K_{dx}\dot{x}$$

Where the K parameters are the adjusted weights for the angular and translational velocities and accelerations.

0.5 Robot Parts List

Arduino Uno (Atmega328p):
MPU6050
Dual VNH2SP30 Motor Driver
2x 12v, 320RPM DC Motors with 330 Hall ratio encoders
3-cell Lithium Polymer battery (appx. 12v)
Chassis based on custom laser-cut 6mm acrylic

0.6 Inverted Pendulum Model



References

[1] Rich Chi Ooi. *Balancing a Two-Wheeled Autonomous Robot.* The University of Western Australia, 2003.

[2] Vaswani, Namrata *Kalman and Extended Kalman Filtering.* Iowa State University: https://pdfs.semanticscholar.org/1bab/607c38764ecdccd4f4589407ddd14a453923.pdf

[3] Abdul Gafar *Self-Balancing Robot.* The University of Manchester, 2016.

[4] Zhao-Qin Guo, Jian-Xin Xu, Tong Heng Lee. *Design and implementation of a new sliding mode controller on an underactuated wheeled inverted pendulum.* National University of Singapore, Franklin Institute, 2013.

[5] Gabriel Terejanu. *Extended Kalman Filter Tutorial.* The University at Buffalo: https://www.cse.sc.edu/ terejanu/files/tutorialEKF.pdf

[6] *Control of an Inverted Pendulum.* Linkopings University, Sweden https://www.control.isy.liu.se/student/tsrt19/ht2/file/invpendpmenglish.pdf

[7] Kalman, R.E., Bucy, R.S. *Forecasting, Structural Time Series Models and the Kalman Filter.* Cambridge University Press, 1961.

[8] Simonn D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches.* Wiley, 2006.

[9] Zhang, et al. *Nonlinear Noise Filtering with Support Vector Regression.* UEST of China, 2006. DOI: 10.1109/ISDA.2006.207

[10] Boots, Bryon. *Machine Learning for Modeling Real-World Dynamical Systems.* Georgia Tech. https://www.cc.gatech.edu/ lsong/teaching/CSE6740fall14/BBoots.pdf

[11] Ahuja, et al. *Improving Accuracy of Interial Measurement Units using Support Vector Regression.* Stanford. https://www.cc.gatech.edu/ lsong/teaching/CSE6740fall14/BBoots.pdf