Theoretical Explanation of PID Control Simulator for Motor RPM with Kalman Filter: A Beginner's Guide

Maneesha Wickramasuriya
(for Team **Origins**)

***Abstract*—This document is designed for undergraduate students with no background in control systems or estimation techniques. It explains a simulator that uses a PID (Proportional-Integral-Derivative) controller to manage the speed (RPM) of a motor wheel, handling real-world issues like noisy sensor readings and disturbances (e.g., friction or external forces). A Kalman filter estimates the true speed from noisy measurements. The motor's behavior (dynamics) is simulated using numerical methods like Euler and Runge-Kutta. Each concept is broken down with continuous (smooth) and discrete (step-by-step) versions, Python code, and examples from the motor problem. By the end, you'll understand how to control a motor to reach a desired speed despite noise and disturbances.**

***Index Terms*—PID Control, Kalman Filter, Extended Kalman Filter, System Dynamics, Numerical Integration, Runge-Kutta, Euler Method, Motor Control, Sensor Noise**

I. Introduction

Imagine you're building a robot or a car, and you need to control how fast its wheels spin. That's what this simulator is about: controlling the "RPM" (revolutions per minute, or how many times the wheel turns in a minute) of a motor wheel. In real life, motors face challenges like friction (slowing the wheel), noisy sensors (inaccurate speed readings), and disturbances (like wind or bumps). This simulator uses a "PID controller" to adjust the motor's input (like voltage) and a "Kalman filter" to clean up noisy speed measurements, ensuring the wheel spins at the desired speed, such as 100 RPM.

Why is this useful? Precise speed control is critical in robotics (e.g., drone propellers), electric vehicles (wheel speed), or factories (conveyor belts). This guide is for beginners—no prior knowledge needed! We'll explain everything step-by-step with examples from the motor problem.

The document is organized as follows: Section II details the motor control problem, Section III explains PID control, Section IV covers Kalman filters (and why we don't need the Extended version here), Section V describes the motor's math and simulation methods, and Section VI summarizes. References include free online resources (PDFs and YouTube videos) for further learning.

II. The Problem: Controlling Motor Speed with Real-World Challenges

Picture a F1tenth car wheel you want to spin at 100 RPM. The simulator mimics real-world issues: - **Noisy Sensors**: The RPM sensor adds random errors (Gaussian noise, e.g., standard deviation $\sigma = 2$ RPM). If true RPM is 100, the sensor might read 102 or 98. The Kalman filter estimates the true RPM, reducing wobble to, say, 100.1. - **Disturbances**: External forces like wind or load changes affect speed. The simulator allows: - *Static*: Constant disturbance (e.g., $d = 0.5$ RPM/s, like steady friction). - *Dynamic*: Time-varying, like a sine wave $d(t) = A\sin(2\pi ft)$, with amplitude $A = 0.5$, frequency $f = 0.1$ Hz (slow wobble every 10s). - **Friction**: The wheel slows naturally, modeled as $-a\omega$, where $a = 0.1$ (1/s) is friction coefficient, ω is RPM. At 100 RPM, friction reduces speed by 10 RPM/s. - **Sensor Frequency**: The sensor measures at 20 Hz (every 0.05s). The simulator runs at 100 Hz ($\Delta t = 0.01$s), so updates happen every 5 steps. Slower rates (e.g., 1 Hz) delay reactions; faster (100 Hz) is more precise but computationally heavy. - **Negative RPM**: A checkbox allows reverse spinning (up to -5000 RPM); otherwise, RPM stays ≥ 0.

Goal: Use PID to compute control input u (e.g., voltage) based on error (desired RPM minus Kalman-estimated RPM). The simulator shows a rotating wheel, error plot (target - actual), and RPM plot (desired, estimated, true), with controls for gains, disturbances, and sensor settings.

This teaches control for drones, cars, or robots. Let's dive into the components.

III. PID Control: Fixing Speed Errors

A. What is PID? A Car Analogy

Think of driving: If you're 20 km/h below the speed limit (error), you press the gas (proportional). If you've been slow for a while, you press harder (integral). If you're speeding up too fast, you ease off (derivative). PID does this for the motor: If wheel spins at 80 RPM but target is 100, error=20, so PID increases voltage.

PID stands for: - **Proportional**: React to current error (bigger error, bigger fix). - **Integral**: Sum past errors to eliminate steady offset. - **Derivative**: Predict error changes to avoid overshooting.

B. The Math Behind PID

In continuous time (smooth, real-world):

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{de(t)}{dt} \tag{1}$$

- $u(t)$: Control output (e.g., voltage). - $e(t)$: Error = desired RPM - current RPM (e.g., 100 - 80 = 20). - K_p, K_i, K_d: Gains (e.g., $K_p = 0.2 \rightarrow 0.2 \times 20 = 4$).

In discrete time (computer, $\Delta t = 0.01$s):

$$u_k = K_p e_k + K_i \Delta t \sum_{i=0}^{k} e_i + K_d \frac{e_k - e_{k-1}}{\Delta t} \tag{2}$$

- Sum approximates integral (e.g., error=20 for 100 steps $\rightarrow$ sum=2000, $K_i = 0.1$, term=0.2). - Derivative: Error drops 20 to 18 $\rightarrow (18 - 20)/0.01 = -200$, $K_d = 0.05 \rightarrow$ -10.

Continuous: Ideal, no sampling errors. Discrete: Practical, but large Δt misses fast changes.

Example: Target=100, estimated=80, $K_p = 0.2$, $K_i = 0.1$, $K_d = 0$. First step: $u = 0.2 \times 20 + 0.1 \times 20 \times 0.01 = 4.02$. Next, error=19, integral grows, u adjusts.

C. How to Use and Tune PID

1. Set target (100 RPM) via GUI slider. 2. Get estimated RPM from Kalman (80). 3. Compute error (20). 4. Calculate u (4.02). 5. Feed u to motor model, repeat every 0.01s.

Tuning: Start $K_p = 0.2$ (reach 100 fast). Add $K_i = 0.1$ (fix 5 RPM offset from friction). Small $K_d = 0.05$ (stop wobbling). In simulator, adjust live—error plot should hit 0.

Disturbances: Static $d = 0.5$ slows 0.5 RPM/s; K_i counters by building u. Dynamic $d(t) = 0.5 \sin(0.2\pi t)$ wiggles—higher K_p tracks better.

D. Python Code for PID

Simulator's discrete PID:

```
class PID:
def __init__(self, kp, ki, kd)
    :
self.kp = kp
self.ki = ki
self.kd = kd
self.integral = 0
self.prev_error = 0

def compute(self, error, dt):
self.integral += error * dt
deriv = (error - self.
    prev_error) / dt if dt > 0
     else 0
u = self.kp * error + self.ki
    * self.integral + self.kd
    * deriv
self.prev_error = error
return u
```

Example: pid.compute(20, 0.01) $\rightarrow$ 4.02. GUI sets $K_p = 0.2$, $K_i = 0.1$, $K_d = 0$.

Learn: PID Control - A brief introduction (YouTube, 10 min).

IV. Kalman Filter: Cleaning Noisy Measurements

A. What is a Kalman Filter? GPS Analogy

Imagine tracking a car with a noisy GPS (off by ±2m). Kalman filter (KF) predicts position using motion (last position + speed) and corrects with GPS data. For our motor: Predict RPM from physics (friction slows it), measure noisy RPM (102 instead of 100), KF estimates 100.2—closer to truth.

KF works for linear systems with Gaussian noise (random, bell-shaped).

B. How Kalman Filter Works: Predict and Correct

KF cycles two steps:

1. **Predict**: Guess next RPM using model. - Physics: $\omega_k = F\omega_{k-1} + Gu_{k-1}$, where $F = e^{-a\Delta t}$, $G = g(1 - F)/a$. - Example: RPM=100, $u = 0$, $a = 0.1$, $\Delta t = 0.01$, $F \approx 0.999$, predict $\omega = 100 \times 0.999 \approx 99.9$. - Uncertainty P: Starts high (100, unsure), grows with $Q = 0.01$ (model errors, e.g., unmodeled vibration). $P_k = FP_{k-1}F + Q \approx 0.999^2 \times 100 + 0.01 \approx 99.81$.

2. **Correct (Update)**: At sensor rate (20 Hz, every 0.05s = 5 steps), blend prediction with measurement. - Prediction=99.9, measure=101 (true=100, noise=+1), $H = 1$ (direct RPM). - Kalman gain: $K = P/(P + R)$. With $P = 99.81$, $R = 4$ (std=2), $K \approx 99.81/(99.81 + 4) \approx 0.961$. - Update: $\hat{\omega} = 99.9 + 0.961 \times (101 - 99.9) \approx 99.9 + 1.056 \approx 100.956$. - New P: $P = (1 - K)P \approx (1 - 0.961) \times 99.81 \approx 3.89$ (more certain).

Repeat: Predict every 0.01s (100 Hz), update every 0.05s (20 Hz). Over time, estimate tracks true RPM (e.g., error ¡0.5 RPM despite noise=±2).

Matrices: - $F = e^{-a\Delta t} \approx 0.999$ ($a = 0.1$, $\Delta t = 0.01$): Friction decay. - $G = g(1-F)/a \approx 10 \times (1-0.999)/0.1 = 0.1$ ($g = 10$): Control gain. - $H = 1$: Measure RPM directly. - $Q = 0.01$: Process noise (model uncertainty, e.g., friction varies). - $R = 4$ (std=2 squared): Sensor noise. - State $x = \omega$, measurement $z = \omega +$ noise.

Sensor Frequency: 20 Hz $\rightarrow$ update every 5 steps (0.05s/0.01s). Predict between updates, smoothing noise.

C. Continuous vs. Discrete Kalman Filter

Continuous (Kalman-Bucy): Solves differential equations for state and covariance (analog systems).

$$\dot{\hat{x}} = A\hat{x} + Bu + K(z - C\hat{x}), \quad K = PC^T R^{-1} \tag{3}$$

With Riccati equation for P. Example: $A = -a = -0.1$, $B = g = 10$.

Discrete: Algebraic, for computers (used here). Example: $F = e^{-0.1 \times 0.01} \approx 0.999$.

Simulator uses discrete for software, updates at sensor rate.

D. What is Extended Kalman Filter (EKF)?

KF assumes linear model. If nonlinear (e.g., $\dot{\omega} = -a\omega^2 + gu$, quadratic friction at high RPM), EKF linearizes using Jacobians: - Predict: $\hat{x}_{k|k-1} = f(\hat{x}_{k-1}, u_{k-1})$, $F_k = \partial f/\partial x$ (e.g., $-2a\omega$). - Update: Similar, $H_k = \partial h/\partial x$.

Our model is linear ($\dot{\omega} = -a\omega + gu - d$), so KF suffices. EKF needed for nonlinearities (e.g., motor saturation at 5000 RPM).

E. How to Use KF/EKF and Why Useful

Initialize: $\hat{x}_0 = 0$ RPM, $P_0 = 100$ (unsure). Tune: $Q = 0.01$ (trust model), $R = 4$ (noisy sensor). Run: Predict each step, update at 20 Hz. Useful: Cuts noise (std=2 to 0.5), keeps error low (¡0.1 RPM).

Example: True RPM=100, measure=101.5, predict=99.8, KF gives 100.2. GUI plots show estimated vs. true converging.

F. Python Code for KF

Simulator's linear KF:

```
import numpy as np

class KalmanFilter:
def __init__(self, a=0.1, g
    =10.0, dt=0.01, Q=0.01, R
    =4.0, x0=0.0, P0=100.0):
self.a = a
self.g = g
self.dt = dt
self.Q = Q
self.R = R
self.x = x0
self.P = P0
self.F = np.exp(-self.a * self
    .dt)
self.G = self.g * (1 - self.F)
     / self.a if self.a != 0
    else self.g * self.dt

def predict(self, u):
self.x = self.F * self.x +
    self.G * u
self.P = self.F * self.P *
    self.F + self.Q

def update(self, z):
K = self.P / (self.P + self.R)
self.x += K * (z - self.x)
self.P = (1 - K) * self.P
```

EKF would add Jacobians (e.g., $F = -2ax$ for ω^2).

Learn: Kalman Filter - 5 Minutes with Cyrill (YouTube), PDF: Welch & Bishop.

V. System Dynamics and Simulation: Modeling the Motor

A. The Motor Model Explained

The motor's speed changes via:

$$\dot{\omega}(t) = -a\omega(t) + gu(t) - d(t) \tag{4}$$

- $\dot{\omega}$: RPM change rate (e.g., +10 RPM/s). - $-a\omega$: Friction ($a = 0.1$, $\omega = 100 \rightarrow$ -10 RPM/s). - gu: Control ($g = 10$, $u = 1 \rightarrow$ +10 RPM/s). - $d(t)$: Disturbance (static: $d = 0.5$; dynamic: $0.5\sin(2\pi 0.1t)$).

Continuous solution: $\omega(t) = (\omega_0 - \frac{gu-d}{a})e^{-at} + \frac{gu-d}{a}$ (steady at $(gu-d)/a$, e.g., 95 for $u = 1$, $d = 0.5$, $a = 0.1$).

Discrete: Approximate per step (e.g., $\omega_k = 0.999\omega_{k-1} + 0.1u_{k-1} - 0.01d$).

Negative RPM: Allowed if checkbox enabled (up to -5000), else $\omega \geq 0$.

B. Creating the Simulator: Numerical Methods

Simulator: Every 0.01s, compute ω, add noise at 20 Hz, update plots.

1) Euler Method: Simple but Basic: Forward Euler:

$$\omega_{k+1} = \omega_k + \Delta t\,(-a\omega_k + gu_k - d_k) \tag{5}$$

Example: $\omega = 100$, $u = 0$, $d = 0$, $a = 0.1$, $\Delta t = 0.01 \rightarrow \omega = 100 - 1 = 99$. (Exact: 99.005.) Pros: Simple. Cons: Errors grow (0.5% off here).

Continuous: Exact for constant slope.

2) Runge-Kutta Methods: Accurate: RK4 averages four slopes:

$$k_1 = f(t_k, \omega_k) = -a\omega_k + gu_k - d(t_k) \tag{6}$$

$$k_2 = f(t_k + \frac{\Delta t}{2}, \omega_k + \frac{\Delta t}{2}k_1) \tag{7}$$

$$k_3 = f(t_k + \frac{\Delta t}{2}, \omega_k + \frac{\Delta t}{2}k_2) \tag{8}$$

$$k_4 = f(t_k + \Delta t, \omega_k + \Delta t k_3) \tag{9}$$

$$\omega_{k+1} = \omega_k + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{10}$$

Example: Above, RK4 gets 99.005 (exact). RK45: Adaptive, adjusts step.

Simulator uses scipy.solve_ivp with RK45 for precision with $d(t)$.

Continuous: Analytical solution. Discrete: Numerical, risks aliasing if Δt too big.

C. Python Code for Simulation

RK45 in update_sim:

```
from scipy.integrate import
    solve_ivp

def plant_fun(t_rel, y, u, a,
    g, dist_func, t_start):
t_abs = t_start + t_rel
dist = dist_func(t_abs)
return [-a * y[0] + g * u -
    dist]

sol = solve_ivp(self.plant_fun
    , (0, self.dt), [self.
    rpm_true], method='RK45',
args=(self.u, self.kf.a, self.
    kf.g, self.get_dist, self.
    t))
self.rpm_true = sol.y[0, -1]
```

Euler alternative:

```
self.rpm_true += self.dt * (-
    self.kf.a * self.rpm_true
    + self.kf.g * self.u -
    self.get_dist(self.t))
```

Clamp $\omega \geq 0$ if negative RPM disabled.

Learn: Runge-Kutta Method Introduction (YouTube), PDF: Wang.

VI. Conclusion

This guide explained PID, Kalman filter, and numerical simulation for motor RPM control, with examples like tracking 100 RPM despite 2 RPM noise or 0.5 disturbances. PID adjusts voltage, KF cleans measurements, RK45 simulates accurately. Try the simulator—tweak gains ($K_p = 0.2$), noise (std=5), or reverse RPM (-5000). Watch error plot hit 0, estimated RPM track true. References offer more.

References

References

[1] S. Brunton, "PID Control - A brief introduction," YouTube Video, https://www.youtube.com/watch?v=UR0hOmjaHp0, 2016.

[2] C. Stachniss, "Kalman Filter - 5 Minutes with Cyrill," YouTube Video, https://www.youtube.com/watch?v=o_HW6GnLqvg, 2018.

[3] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," UNC-Chapel Hill, 2006. https://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf.

[4] LearnChemE, "Runge-Kutta Method Introduction," YouTube Video, https://www.youtube.com/watch?v=kUcc8vAgoQ0, 2015.

[5] Y. Wang, "Notes on Runge-Kutta Methods," Oklahoma State University, 2011. https://math.okstate.edu/people/yqwang/teaching/math4513_fall11/Notes/rungekutta.pdf.

[6] K. J. Åström and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press, 2008. Free PDF: https://www.cds.caltech.edu/~murray/books/AM08/pdf/am08-complete_22Feb09.pdf.

[7] D. Simon, *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*, Wiley, 2006. (Partial free previews available online).