PID Control-R

February 8, 2013
By Lee Pang

On a whim, I thought it might be fun to try to implement a PID control algorithm … in R.
Yes, I know I have a strange idea of fun.

Background

PID control is a heuristic method of automatically controlling processes as wide ranging as water levels in tanks to the direction of ships running against a current. What's important is that you don't need to know much about a process to control it with a PID controller. Hence, why it is so broadly used. I studied the theory behind it in college and even built a LabVIEW UI for use in a student teaching lab.

For more detailed information, the obligatory Wikipedia link should be a good starting point. In fact, that's where I got all the information I needed since my control theory text is currently buried in a closet somewhere.

A PID controller produces an output (aka a "Manipulated Variable", MV(t) or u(t)) – usually from 0 to 100% – based on a weighted sum of three terms:

- Error between where you want the process to be (a setpoint) and where the process is
- How much error has accumulated over time
- How rapidly the error is changing

which naturally leads to the P, I, and D in the name and the three weighting parameters (gains) that need to be tuned:

- Proportional
- Integral
- Derivative

The math for this looks like:

$$\mathrm{MV}(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)\, d\tau + T_d \frac{d}{dt} e(t) \right)$$

Don't be scared, it isn't as gnarly as it looks.
For the moment, let's ignore the Kp on the outside while we sort out the individual parts being added together.

The main variable in the equation above is e(t). This is the error between the process variable PV(t) – the thing you want to control – and the setpoint SP(t) – the value you want the process to be at. So this makes e(t):

e(t) = PV(t) – SP(t)

This happens to also be the first term – the Proportional response.
 This should make sense – move the control by about as much error that you currently see.

The second term – the Integral response – isn't as ugly as it looks.
 Integrals can be thought of as the area under a curve – in this case e(t).

Numerically, this simplifies to the sum of e(t)*dt, where dt is the amount of time between readings of the process variable. To complete the Integral response, this sum is divided by Ti, the integral time, which is an estimate of how long it will take to abolish all the accumulated error. In terms of the overall sum, this adds more control output to account for error that was previously "missed".

Last is the third and final term – the Derivative response. Here the derivative is of e(t) with respect to time – or how fast is e(t) changing
3
ɛ t. To compute this, take the difference between the current
of e(t) and its last value e(t-dt) and divide it by dt. To complete
ɛrivative response, multiply the derivative value by Td, the
tive time, which is an estimate of how far in the future the error
ː reliably predicted. This term adds control output to account for
hat might happen.

the above is multiplied by Kp, the proportional gain, which
the whole control response.

le

Process

First things first, we need a process variable.

```r
pv = function(pv.prev, u, tt) {
  out = pv.prev*1.1 + .5            # exponential growth + linea
  out = out - 0.1*u                 # the control response
  out = out + .5*runif(length(tt)) # a little noise, just for f

  if (out < 0) out = 0              # keep values positive
  return(out)
}
```

PIDControlR_ProcessVariable.R hosted with ❤ by GitHub view raw

The code above is a simple function that simulates a process that, uncontrolled, will grow exponentially over time. For added realism, I've included some uniform noise to the process growth. The value of the control variable (in this case `u`) decreases the process value.

Controller

To code the controller for the above process, we first need to initialize a few variables:

```r
# controller parameters
Kp = 10                 # proportional gain
Ti = 1                  # integral time
Td = 0.01               # derivative time

# simulation parameters
dt = .1                 # time step
tt = seq(0, 100, by=dt) # time vector

# initialize the following to a vector of zeros
# as long as the time variable tt
# - PV, process variable
# - U, control output
# - E, error
# - EI, error integral
# - ED, error derivative
PV = U = E = EI = ED = rep(0, length(tt))
PV[1] = 5 # initial state of the process variable
```

PIDControlR_Initialize.R hosted with ❤ by GitHub view raw

In the interest of book-keeping, I'm storing the values of the Integral and Derivative responses in vectors EI and ED, respectively. This isn't necessary, and indeed if this were a continuously operating controller, I wouldn't want to for the sake of conserving memory space.
Next a setpoint. Below is code to create a setpoint profile that has a few step changes in time.

```r
SP = rep(10, length(tt))
SP[which(tt >= 30)] = 5
```

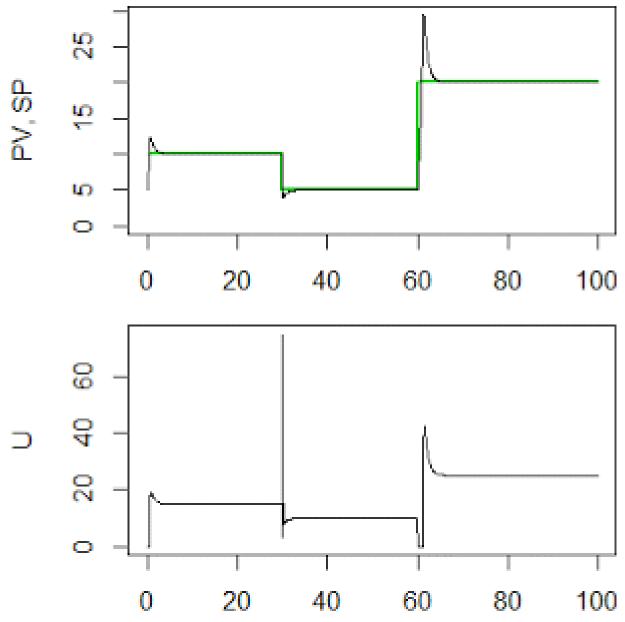
```r
SP[which(tt >= 60)] = 20
```

From here on, it’s a matter of running a simulation loop.

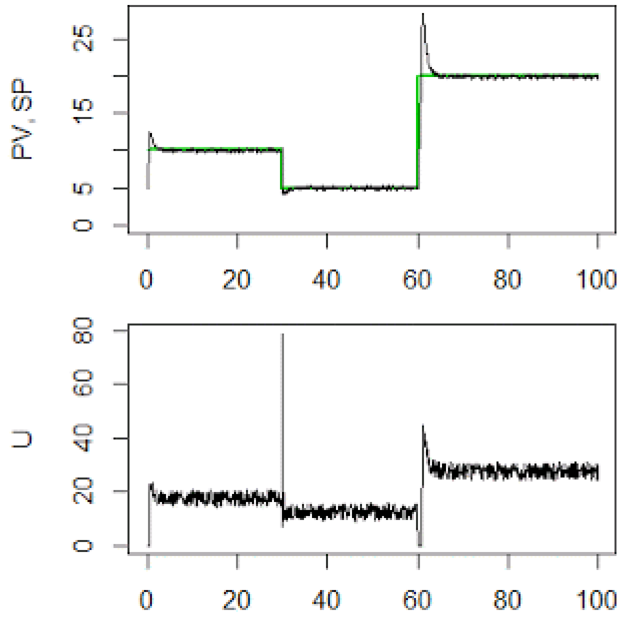
```r
for (k in 2:length(tt)) {
  PV[k] = pv(PV[k-1], U[k-1], tt[k])
  E[k] = PV[k] - SP[k]

  EI[k] = EI[k-1] + E[k]*dt  # integral
  ED[k] = (E[k] - E[k-1])/dt # derivative

  U[k] = Kp*(E[k] + (1/Ti)*sum(E*dt) + Td*ED[k])

  if (U[k] < 0) U[k] = 0
}
```

Folks savvy in calculus may note that my implementation of the integral isn’t entirely textbook – meaning, I’m not using the Trapezoid rule or Simpson’s rule to compute a more accurate area for each time step. This is on purpose, making the algorithm easier to implement. It also assumes that my time step, dt, is small enough that this doesn’t matter much.

Running the simulation and plotting the necessary variables, shows that the controller works reasonably well. Note, I disabled the noise in the process variable for this first run.



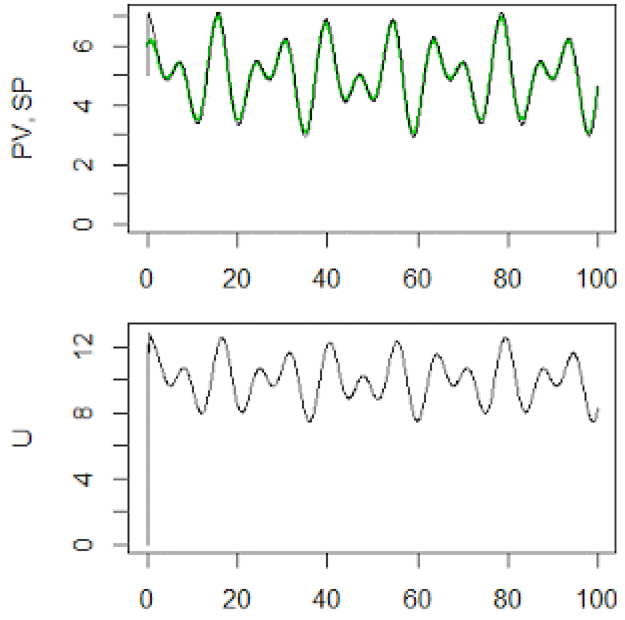
g noise to the process doesn’t affect the results much, just a
r output in U (the manipulated variable):



Using a more complicated setpoint profile:

```
SP = sin(0.5*tt) + cos(.8*tt) + 5
```

produces:



From here, one can have some fun (or do some work) playing with the parameters, process variable, and setpoint profile.

Summary

A PID controller is based on a reasonably simple algorithm that can be quickly implemented in R, or any other programming language for that matter.

Related

2 comments on this item