A primer on PCT for computer modelers

Unedited posts from archives of CSG-L (see INTROCSG.NET):

Date: 09 Nov 1992 10:25:31 -0700
Subject: Primer for modelers: draft

Experimenting with the control paradigm
A primer for computer modelers
DRAFT: William T. Powers

PART I: PROPERTIES OF A SIMPLE STABLE CONTROL SYSTEM

The following is intended to introduce programmers and control-system engineers to the terminology and architecture of control theory as used under the name PCT, or perceptual control theory. For those who have experience with modeling control systems, there are some readjustments to be made, because we will divide the system into functions in a way that is not standard in control engineering. For example, the output of the control system is not the controlled variable, but is an influence on the controlled variable. If the controlled variable were defined as the rotational speed of a motor, the output of the control system would be the torque applied to the motor armature, not the speed. The speed would be classified as an INPUT quantity, because it is this quantity that is sensed by a tachometer, and that can be disturbed by variables in the environment such as friction and loads. We define torque as the output because torque depends only on the output of the control system -- the current going through the motor -- and is not subject to disturbance by the environment. If you are an engineer it will take some effort to reorganize your thinking in this new way, but even in control engineering you might find that there are some considerable benefits in doing so. The normal way of presenting control processes to students is rather disorganized; the PCT organization brings in a standard approach that often makes control problems easier to solve.

The aim here is to develop some insights into the properties of control systems, not through complexity but through simple examples and hands-on experience. Watching computer simulations work is the next best thing to seeing a real control system work; in some ways it is superior because you have time to see the details of what is going on. Rather than exhort the reader to run these programs on a computer and examine the results, I have decided to make it necessary to do this by not presenting any numerical or graphical tables. You will have to run the program to see what this discussion is about. Perhaps frustration will prove to be an effective motive for actually experiencing this simulation in operation.

Suggestions for standard terminology

A function is a physical device with an output signal the magnitude of which can be computed from the state of its input magnitudes. All functions are true mathematical functions: that is, they may have multiple inputs (arguments) but they produce only one output (value of the function given those arguments). Thus the term function refers both to some physical element of the system and to the equivalent mathematical function that describes the dependence of its output on its input(s) in terms of magnitudes.

A generic control system consists of an input function, a comparator, and an output function. The output of one function generates a variable that is an input to another function. Such information- carrying variables inside the system are called signals. A signal not only represents the value of the function, but serves to carry that value to the input of another function in a different physical location. All signals have a single measure, magnitude. The name of a signal identifies a pathway; the value of the signal indicates the momentary magnitude of the signal carried unidirectionally by that pathway.

The environment model

In the environment of a control system the variables are called quantities. The output of the output function is measured in terms of an effect on a

physical variable called the output quantity. The output function in a model of a single control system interacting with an environment is therefore a transducer: its input is a signal while its output is a quantity. The output quantity is always defined so that its magnitude depends only on the output function's value: it is always a single variable. If it has multiple effects in the environment, each of those effects must be separately indicated in a model of the environment.

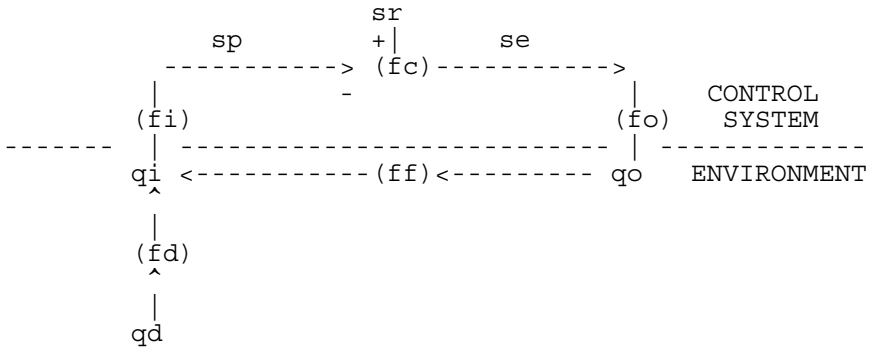
The input to the control system is another physical variable called the input quantity. The input function senses the state of the input quantity and converts it to a perceptual signal. The input function is also a transducer in a single system-environment model; its input is a physical quantity and its output is a signal. An input function may respond to multiple input quantities.

In the environment, there is a feedback link connecting the output quantity to the input quantity. This link is called the environmental feedback function, or simply the feedback function.

Also in the environment there is a link through which independent environmental variables called disturbing quantities act on the input quantity concurrently with the action of the output quantity on the input quantity. Because the number and kind of disturbing quantities is immaterial, it is customary, when modeling a single control system, to represent all disturbing quantities and their individual links to the input quantity as a single equivalent disturbance acting through a single equivalent disturbing function.

The control system model

The perceptual signal generated by the input quantity enters a comparator; also entering the comparator is a reference signal, an independent variable. Where possible, the signs of various system constants are chosen so that the reference signal has a positive effect on the comparator while the perceptual signal has a negative effect. The comparator is a function with two arguments and a single value. The output value is represented by an error signal, the magnitude of which is equal to the reference signal's magnitude minus the perceptual signal's magnitude. The error signal enters the output function. Often, as shorthand, we speak of subtracting one signal from another, or adding signals together. What is meant is that the magnitudes are subtracted or added. The system-environment diagram



DEFINITIONS:

Signals:
sp = perceptual signal
sr = reference signal
se = error signal

Functions:
fi = input function
fc = comparison function or comparator
fo = output function
ff = feedback function
fd = disturbance function

Quantities:
qi = input quantity
qo = output quantity
qd = disturbing quantity

THE CONTROL EQUATIONS:

System:
sp = fi(qi)
se = sr - sp

Interface transducers:
qo = fo(se)
sp = fi(qi)

Environment:
qi = ff(qo) + fd(qd)

Combined equations:
System: qo = fo(sr - fi(qi))
Env: qi = ff(qo) + fd(qd)

Setting up a working model

The following discussion assumes that you know a programming language like C, Fortran, Pascal, Modula, or Basic. The actual programming involved is elementary. The student is advised to write the simplest program possible in the most familiar language and experiment with it. The most important knowledge to be gained is a feel for the relationships among variables in a control system, and for the effects of changing various system parameters. There is a temptation to tackle some interesting and complex problem first, but without a strong intuitive foundation for designing more complex systems the most likely result will be confusion and failure. Control systems do many surprising things and the effects of changes in the parameters are seldom what you would initially guess.

The simplest control system to model on a digital computer is one in which all the functions are simple proportionalities except the output function, which is an integrator. Alternatively, the feedback function or the input function can be made into an integrator; however, only one function should be an integrator and the rest should be proportional multipliers. We will use a design with an integrator in the output function; you can experiment with the other possibilities.

In computer programs, integration is summation. Because this is a closed-loop system, integrations do not need to be precise, so advanced methods of numerical integration are not needed. If we make the output function into an integrator, the program step for computing output becomes (in C notation)

```
qo = qo + ko*se*dt;
```

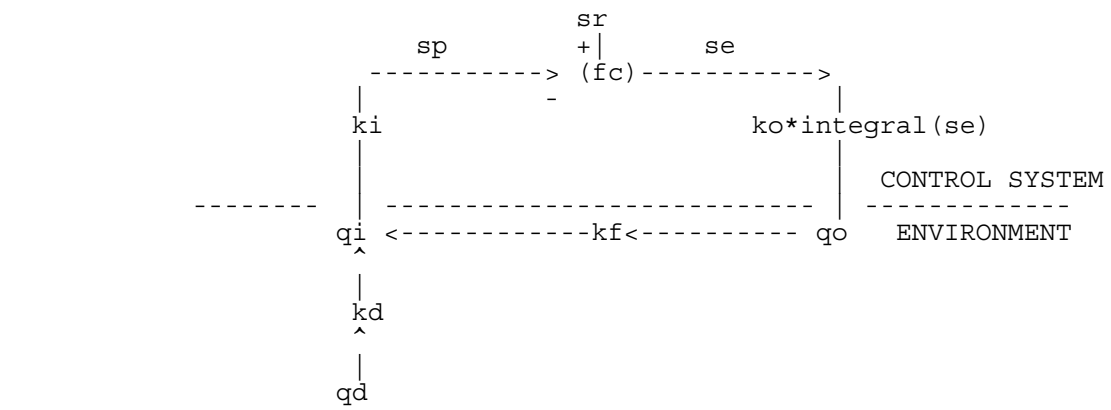
where ko is an integration factor determining how much the output will change on each iteration for a given magnitude of error signal. The constant dt defines the physical time represented by one iteration of the program -- it should be set to 0.1 or 0.01 initially, implying that you should use floating point variables. We will use 0.1 sec.

The central part of a C program for implementing a control system would then be (starting with the computation of the perceptual signal sp):

```
sp = ki*qi;
se = sr - sp;
qo = qo + ko*se*dt;
qi = kf*qo + kd*qd;
```

The disturbing, input, and feedback functions are replaced by constants kd, ki, and kf. For initial experimentation they can all be set to 1. Two variables have to be initialized before the first time this series of steps is used: qi and qo. Initializing to zero is sufficient. Two independent variables must also be set, sr and qd. Using these variables is described below.

The diagram with constants in place of the general functions:



The general flow chart of the program

1. Initialize variables qo and qi.

2. Input or set constants ki, kf, kd, and ko.

3. Input or set the values of the reference signal sr and the disturbing quantity qd.

4. Execute the four program steps above.

5. Plot or print the values of variables of interest.

6. Return to step 4 until the desired number of iterations is finished. If dt = 0.1, 25 iterations will show 2.5 seconds of behavior.

An alternative is to pre-record an array of values for the reference signal or the disturbing quantity or both, and step through this array as the iterations proceed. In that case step six would involve returning to step 3, and step 3 would advance pointers to the arrays of values for sr or qd or both. Below we will use still another way of showing the effects of changes in sr and qd.

Exploring a control system using the model

```
/* A SAMPLE PROGRAM IN C: */

#include "stdio.h"

void main()
 {
float sp = 0.0,sr = 20.0,se = 0.0,qo = 0.0,qi = 0.0, qd = 0.0;
float kd = 1.0,ki = 1.0,ko = 8.0, kf = 1.0, dt = 0.1;
int i;

/* you may put statements here to input values of the k-constants */
printf("\n");
 for(i=0;i<25;++i)
  {
   if(i > 12) qd = 10.0; else qd = 0.0;
   qi = kf*qo + kd*qd;
   sp = ki*qi;
   se = sr - sp;
   qo = qo + ko*se*dt;
   printf(
   "\x0d\x0a qd=%6.2f qi=%6.2f qo=%6.2f sp=%6.2f sr=%6.2f se=%6.2f",
   qd,qi,qo,sp,sr,se);
  }
 (void) getch(); /* pause to view; press key to exit */
}
-------------
```

In this program, the disturbance remains at zero for the first 12 iterations, and then jumps to 10.0 for the last 13. The resulting table will just fill the screen. You will see the input quantity and perceptual signal rise quickly to 20 units, to equal the reference signal's setting, and then briefly be disturbed when the disturbing quantity changes from 0 to 10. The perceptual signal sp will then return within half a second (5 iterations) to the reference value again.

Illuminating experiments with the program

There are several basic rules of thumb that can be demonstrated with this program. The first is the maxim that control systems control their own perceptual signals, not the input quantity and not their own outputs.

First, run the program and note that the perceptual signal sp, the input quantity qi, and the output quantity qo all rise to 20.0 just before the disturbance enters. Note that after the disturbance appears, the output drops from 20.0 to 10.0 units; this is because the disturbance is trying to make the perceptual signal too large; the output automatically drops by the amount needed to bring the perceptual signal sp back down to 20.0. The control system clearly does not control to produce a specific output. The output quantity changes as the disturbing quantity changes.

Because the input function is a multiplier of 1, the input quantity qi and the perceptual signal sp are numerically equal. Change the input constant ki from 1.0 to 0.5 and compile and run the program again (to save time you may want to insert some lines to read in ki, kf, ko, and kd from the keyboard). Intuitively one might expect that halving the input sensitivity would halve the perceptual signal. But this is a closed-loop system, and that is not what happens.

The perceptual signal still rises to match the reference signal's value of 20.0, although more slowly than before. The input quantity, however, rises nearly to 40. It must do this because we have reduced the effect of the input quantity on the reference signal -- the input quantity must be greater to produce the former amount of perceptual signal. This shows that changing the input function alters the input quantity, but does not alter the perceptual signal's final value. Note also that the output quantity has risen to 40.0 units, as it must do to bring the perceptual signal to the reference value of 20.0. The control system treats a change in the input function as just another disturbance, and alters its output to counteract the change in the perceptual signal. Both the output quantity qo and the input quantity qi are altered by this change in parameter, but the final value of the perceptual signal sp is not altered.

Now restore the input constant ki to 1.0, and change the feedback constant kf to 0.5. Note that the output quantity now becomes 40 instead of 20 as before, while the perceptual signal still rises to the same value as the reference signal, 20.0 units. This shows that changes in the feedback function will change the final value of the output quantity, but not the final value of the input quantity or perceptual signal.

Finally, if you alter the output integration factor ko, you will see a change in the speed with which errors are corrected, but the final states of all the variables are unaffected. CAUTION: keep the product ko*kf*ki*dt less than 1.0. If you make it equal to 1.0 or larger, a computer artifact will be introduced and the system will become unstable. You can try it to see what "unstable" means, but this is not true instability. It's caused by the fact that we're simulating a continous system on a digital computer. On an analogue computer this kind of instability would not happen, because "dt" is infinitesimal.

The integration factor is initialized to 8.0. If you use smaller values, the computation will remain stable.

To sum up, when we change the disturbance, the input function, the feedback function, or the output function, the perceptual signal always returns to a match with the reference signal as long as the control system is still working. The only variable that remains under control under all these changes

in conditions is the perceptual signal. That is why we say that control systems control their own perceptual signals and not their outputs. They can be said to control their input quantities only if the form of the input function does not change.

You may have noticed that when you alter a function, the resulting change in the final state of the system variables is not at the output of that function but at the input. Changing the input function ki does not alter the final value of the output of that function, sp, but the input to that function, qi. Changing the feedback function kf does not alter the quantity qi, but the quantity qo at the input of the feedback function. Causation appears to be working backward. Furthermore, the effect of a change is opposite to the direction of the change. By halving ki, we caused the final value of qi to double; by halving the feedback factor kf, we caused the final value of qo to double.

These strange reversals of causation are typical of control systems, and account for most of the difficulty people have in understanding how different control systems are from straight-through systems in which causation works in the direction we expect. Of course causation has not really been reversed here, but the feedback effects make it seem that it has.

Relationship of disturbance to output

Restore ki and kf to 1.0 and ko to 8.0, recompile, and run. When the disturbing quantity jumps from 0 to 10.0 in the middle of the run, look at the resulting change in the output quantity. It changes from 20.0 to 10.0. This relationship is not accidental. The effect of the disturbing quantity on the input quantity is just canceled by the change in the effect of the output quantity on the input quantity. 10 units of disturbance is canceled by -10 units of change in the output quantity.

Now double the constant representing the disturbance function (change kd from 1.0 to 2.0). This doubles the effect of a given change in the disturbing quantity on the input quantity. Recompile and run.

As usual, the perceptual signal returns to 20.0 after the disturbance. But look at the output quantity: it drops from 20.0 to 0.0 when the disturbance turns on. Now a 10-unit change in the disturbing quantity results in a negative 20-unit change in the output quantity. The reason? One unit of disturbance now has twice as much effect on the input quantity as one unit of output from the control system. Result: twice as much output is now needed to counteract the same disturbance. That is what happens. It happens not because something "knows" that twice as much output is needed, but simply as the natural result of the operation of the control system.

The "behavioral illusion"

If we consider just the disturbing quantity and the output quantity, we can see that there is an apparent direct relationship between them. It looks as though changing the disturbing quantity causes the output quantity to change. If we didn't know about the input quantity qi, we might well think that the system was sensing the disturbing quantity directly, and responding by altering its output quantity. It looks as if a change in the disturbing quantity is a stimulus, which causes a response in the form of a change in the output quantity.

Of course we can see that this relationship holds only because of the input quantity and the fact that the perceptual signal is being maintained at a particular value. The apparent stimulus would alter the input quantity if it were the only influence. But it does not alter the input quantity (for long) because the output changes to have an equal and opposite effect on the input quantity. That is the true explanation of the relationship between the remote disturbing quantity and the output action of the system. The appearance of stimulus and response is an illusion. There can be, of course, true stimulus-response organizations. But an apparent stimulus-response relationship is an illusion when the behaving system is really a control system. Effect of the reference signal

Right after the bracket following the "for" statement, insert this line of code:

```
if(i>5 && i < 20) sr = 20.0 else sr = 5.0;
```

Recompile and run. Now the reference signal begins at 5.0, and rises to 20.0 halfway through the first part of the run. Then the disturbance is turned on. Halfway through the second part of the run, with the disturbance still present, the reference signal returns to 5.0.

Just follow the behavior of the perceptual signal. You will see that it rises quickly to 5.0, then rises again to 20.0 before the disturbance occurs. When the disturbance rises to 10.0 there is a brief excursion of the perceptual signal which immediately returns to 20.0. Then when the reference signal drops to 5.0 again, so does the perceptual signal.

In fact, the perceptual signal tracks the reference signal in terms of magnitude. The reference signal determines the value to which the perceptual signal will be brought initially, and at which it will be maintained, even if disturbances occur. By varying the reference signal, we can make the control system produce physical effects in the environment that result in corresponding variations in the perceptual signal. Even if disturbances come and go, and even if the feedback function and input function characteristics change (over some range), the system will still produce just the output needed to control the perception at the specified level.

This is why we identify the reference signal with the commonsense notions of intention and purpose.

Dynamic considerations

During the program run, the output quantity of the system becomes whatever it must be to keep sp matching sr, for all combinations of qd and sr. Immediately after sudden changes in these independent variables there is an error, but the error is soon corrected -- sooner if ko is larger.

In real environments, physical variables can't jump instantly from one state to another. Normally the changes are smooth; they are also slow enough to allow control processes to begin changing before the environmental changes have gone to completion. To illustrate this, change the line in the program

```
   if(i > 12) qd = 10.0; else qd = 0.0;
```

to

```
  if(i >= 5 && i < 15) qd = qd + 1;
```

In C this could be shortened but this will work in all languages. The effect is to make the disturbance change smoothly instead of in one jump.

Also, "comment out" the line that alters the reference signal sr, or delete it. Compile and run.

Now the perceptual signal sr rises quickly to the default reference level of 20.0. When the ramp disturbance begins, it rises slightly above 20.0 and remains there until the ramp levels out; then it returns quickly to 20.0 again. You will notice the output quantity changing during the change in the disturbance.

If you reduce ko to slow the system, you will find that the ramp disturbance has a greater effect. The amount of effect that a disturbance has depends on how rapidly it changes in comparison with the control system's speed of error-correction.

By reducing the time interval dt to 0.01 and raising ko to 80, you could make the control system work 10 times faster and reduce the effect of the disturbing ramp by a factor of 10. You would, however, need 10 times as many iterations to cover the same period of 2.5 seconds, and this would not fit on the screen. If you want to see this effect, change the print statement so it prints to the printer, and change the limit of the "for" statement to 250. If

you can program graphics, you could plot these variables on the screen and see the behavior in much more detail without using 5 pages of paper.

NOTE: I would appreciate it if programmers versed in BASIC, Pascal, and other languages would write versions of the above program, test them, and transmit them to me for inclusion in an appendix to this primer. Programs should be as generic as possible so as to run on as many computers as possible. I especially need versions for mainframes and workstations. In the future I will simplify the program and make it easier for the user to change parameters from the keyboard. For now, I hope that all programmers and would-be programmers will try the program as it stands and learn from it, even if you have to get some help from a 14-year-old. As this primer evolves I will post revisions and again ask for your help.

Best, Bill P.

Date: Wed Nov 11, 1992 8:32 am PST
Subject: Re: Primer for modelers: draft

William T. Powers writes:

Experimenting with the control paradigm
A primer for computer modelers
DRAFT: William T. Powers

[beginning part deleted ...]

> The simplest control system to model on a digital computer is one in which all the functions are simple proportionalities except the output function, which is an integrator. Alternatively, the feedback function or the input function can be made into an integrator; however, only one function should be an integrator and the rest should be proportional multipliers. We will use a design with an integrator in the output function; you can experiment with the other possibilities.

Why should there be only one integrator?

..stuff deleted...

> NOTE: I would appreciate it if programmers versed in BASIC, Pascal, and other languages would write versions of the above program, test them, and transmit them to me for inclusion in an appendix to this primer. Programs should be as generic as possible so as to run on as many computers as possible. I especially need versions for mainframes and workstations. In the future I will simplify the program and make it easier for the user to change parameters from the keyboard. For now, I hope that all programmers and would-be programmers will try the program as it stands and learn from it, even if you have to get some help from a 14-year-old. As this primer evolves I will post revisions and again ask for your help.

Here is a PASCAL-Version of your program. It should work on whatever platform ..

```
program control;
const kd = 1.0;
      ki = 1.0;
      ko = 8.0;
      kf = 1.0;
      dt = 0.1;

var   sp, sr, se : real;
      qi, qo, qd : real;
      i : integer;
```

```
begin
(* initialize variables *)
   sp := 0.0;
   sr := 20.0;
   se := 0.0;
   qo := 0.0;
   qi := 0.0;
   qd := 0.0;

   for i := 0 to 24 do begin
       if i > 12 then qd := 10.0 else qd := 0.0;
       qi := kf*qo + kd*qd;
       sp := ki*qi;
       se := sr - sp;
       qo := qo + ko*se*dt;

       write   ('qd = ',qd:6:2,' qi = ',qi:6:2,' qo = ',qo:6:2);
       writeln (' sp = ',sp:6:2,' sr = ',sr:6:2,' se = ',se:6:2);
   end;
   readln;
end.

--
Wolfgang Zocher
```

Date: Wed Nov 11, 1992 3:04 pm PST
Subject: primer program from Zocher

[From Bill Powers (921111.1230)] Wolfgang Zocher (921111)

RE: primer for computer modelers

Thanks for the Pascal version, Wolfgang. I will incorporate it, with credit. It runs fine under Turbo Pascal -- anybody else out there who can test these programs to make sure they work on your system?

I would welcome the efforts of anyone who would like to take my rather verbose version of the writeup and make it shorter and better organized.

Also, if someone wants to make the program more user-friendly (but still short), feel free: I'll check it out and revise the Primer accordingly. I can run C, Pascal, and Qbasic programs.

You ask why there shouldn't be a second integrator. This should go into the Primer. The best way to see why not is to put a second integrator into the program.

Try this: in place of

```
qi := kf*qo + kd*qd;
```

introduce a dummy variable x, and write

```
x := x + kf*qo;
qi := x + kf*qd;
```

You'll have to declare x and initialize it to 0. The variable x represents the contribution of the output via the feedback function to the state of the input quantity. The feedback function is now an integrator.

How about trying this, and writing back to tell the folks what happened?

(C version: same change, but use = instead of :=).

Best, Bill P.

Date: Sat Nov 14, 1992 2:13 pm PST
Subject: primer part II

Continuing with the draft version of the modeling primer:

Experimenting with the control paradigm
A primer for computer modelers

PART II: Finite gain and leaky integrators.

The model in Part I used a pure integrating output. In real physical systems, especially nervous systems, integrators are not perfect; they leak. The greater the output of the integrator, the faster it leaks. The result is to create the equivalent of an amplifier that multiplies the input by some finite number, but takes a while to follow changes in the input. For a step input, the output begins to rise as if a pure integrator were present, but instead of the output continuing to rise, it levels off at some multiple of the input magnitude.

We will look now at the control system of Part I with a leaky integrator for an output function. The program now is slightly more complex, in that we will plot the values of the variables against time, using a text-mode output on a graph that is 80 characters wide and 25 high.

This subject is discussed at some length because it has important implications for simulating continuous physical systems on a digital computer. After we have finished this section we will be ready to look at some new quantitative relationships found in control systems.

New program:

```
/* primer2.c */

#include "stdio.h"
#include "conio.h"

void main()
 {
float sp = 0.0,  /* initialize signals and quantities */
      sr = 20.0,
      se = 0.0,
      qo = 0.0,
      qi = 0.0,
      qd = 0.0;
float kd = 1.0, /* set constants */
      ki = 1.0,
      ko = 100.0,
      kf = 1.0,
      dt = 0.1
      ks = 50.0;  /* slowing factor */
int i;
 clrscr();  /* alternative: for(i=0;i<25;++i) putch(0x0d); */
 for(i=1;i<=80;++i)
  {
   if(i > 40) qd = 9.0; else qd = 0.0;
   qi = kf*qo + kd*qd;
   sp = ki*qi;
   se = sr - sp;   qo = qo + (ko*se*dt - qo)/ks;
    gotoxy(i,24 - sr/2.0 - 0.5); putch('r');
    gotoxy(i,24 - qd/2.0 - 0.5); putch('d');
    gotoxy(i,24 - sp/2.0 - 0.5); putch('p');
  }
 (void) getch();  /* wait for keystroke */
}
```

Notice the changes. We now iterate 80 times, with the index i starting at 1. The disturbance qd is zero until the 40th iteration when it rises to 9.0. It remains at 9.0 for the rest of the run. With dt = 0.1 sec, the horizontal dimension corresponds to 8 sec of real time.

The slowing factor

A leaky integrator could be made from the integrator of part I:

qo = qo + ko*se*dt

Simply by subtracting an amount proportional to qo on every integration:

qo = qo + ko*se*dt - qo*dt*(LeakFactor).

There is another form equivalent to this that makes it possible to adjust the steady-state amplification factor and time constant independently. We will use this alternate form for the output function, as follows:

```
  qo = qo + (ko*se - qo)*dt/ks;  /* dt/ks = slowing factor */
```

To understand this function, first consider just the part in the parentheses. ko*se is the value that the output qo would have if the error signal se were simply multiplied by an amplification factor ko. Subtracting qo from this value calculates how far from the current value of qo the final value of the output is: this is the change necessary to reach a final state of qo defined as ko*se. If we just added (ko*se - qo) to qo, the result would be ko*se -- the value of qo on the left would simply be equal to ko*se.

This amount of change, however, is multiplied by dt/ks. If ks, a "slowing factor," is set equal to dt, then the entire difference between the current value of qo and the final value is added to qo, so that qo (on the left of the = signal) becomes equal to the final value on the first iteration. If ko*se does not change, on the next iteration qo will be equal to ko*se, so the difference ko*se - qo will be zero and nothing more will be added to qo during later iterations.

On the other hand, if ks is made larger than dt, so dt/ks is less than 1, only part of the difference will be added on each iteration and qo will approach the value ko*se in a series of diminishing steps. If, for example, ks were made equal to 2*dt, then only half the difference would be added on each iteration. The output qo would rise toward the limiting value ko*se by going half the distance, then half the remaining distance, and so on, on each successive iteration. With a one-unit final value, the steps would be 1/2, 3/4, 7/8, 15/16 ... and so on. To a first approximation, then, the slowing factor ks sets the time constant of the output function. If ks is set to 1.0, the time constant will be about 10 iterations, because dt is 0.1. In 10 iterations, the output will go about 2/3 of the way to the final value after a step- change in se.

Loop gain and the optimum slowing factor

In this closed-loop system, the error signal does not remain constant on successive iterations. Thus we do not see the actual time constant of the output function in the overall behavior of the control system. The control system as a whole will show a much shorter time constant than ks would imply. In fact, the overall time constant can be reduced to a single iteration of the program no matter how long is the time constant in the output function.

Let us first define the loop gain of this system. The loop gain is the product -ko*kf*ki: the product of all multiplication constants encountered in one trip around the closed loop (starting anywhere). The negative sign is introduced by the comparator, where an increase of 1 unit in the perceptual signal produces a change in the error signal of - 1 unit (reference signal constant). For calculating loop gain we are concerned only with the effects of small changes, not the absolute magnitudes of signals and quantities.

It can be shown that the optimum value of dt/ks is simply 1/(1 - loop gain), where the loop gain itself is always a negative number. With this value of dt/ks, the control system will, after a step-disturbance, reach equilibrium on the first iteration of the program. Conversely, for any value of ks there is a loop gain that will give the same result.

The above program is set up with a loop gain of -100 (ko = 100.0, kf = 1.0, ki = 1.0). The optimum value of dt/ks would be 1/101. With dt set to 0.1, the optimum ks is 10.1. The program is initialized with ks = 50, or about 5 times the optimum value. This should produce a time constant of the whole control system of about 5 times the optimum value for reaching equilibrium in one iteration; it should therefore take the control system about 5 iterations to reach 2/3 of the final value. You can compile the program and run it now to see that this is the case. Notice that with an inherent time constant of 50 iterations in the output function, the overall system has a time constant of only 5 iterations.

On the plot, r indicates the reference value, p the perceptual signal, and d the disturbance.

Now reduce ks to the optimum value of 10.1, recompile and run, and verify that the final value is reached after the first iteration.

Note what happens when the disturbance turns on (where the row of d's suddenly jumps upward). The perceptual signal is disturbed upward, above the reference signal, for one iteration. Then it returns to the reference value and stays there even though the disturbance is still present. If you increase the value of ks, you will see that the error correction becomes slower.

A digital artifact

If you make ks smaller than dt, you will begin to see a computer artifact. For ks in the range between 0.5*dt and dt, the approach to a final state will be oscillatory. If ks is even smaller, the oscillations will begin to increase in amplitude exponentially; the system will run away. Try a value of ks = 5.5 to see the oscillatory approach to a steady state. The nearer you get to 5.0, the longer the oscillations will persist. If you reduce ks to or below 5.0, all you will see will be a few scattered p's -- the growing oscillations quickly carry the trace off the screen.

This is an important phenomenon that shows the limitations of modeling a continuous system on a digital computer. The clue that tells you this is a computer artifact is simple: the oscillations (when they occur) always have a reversal period of one iteration of the program, regardless of the setting of ks or dt. The constant dt expresses the meaning of one iteration in terms of real time. As long as ks is greater than dt, the time constants of changes have physical meaning. But as soon as the oscillations begin, they occur at a frequency of exactly one reversal per iteration, however you change ks. That frequency has no physical significance.

If you leave the value of ks at 5.0 and change dt to 0.01, you will now see a normal exponential rise of sp just as before. It won't be evident from the plot, but now the width of the plot represents only 0.8 seconds instead of 8.0 as before because one iteration now corresponds to 0.01 sec instead of 0.1 sec. The rise time of the p's is still about 5 iterations (to the 2/3 value), but now that corresponds to 0.05 sec or half an iteration with the old value of dt. This proves that the model system still behaves normally with small values of the time constant (or slowing factor), even though when the same model with the same physical parameters is run with a coarser time scale, it seems to become unstable. Control systems CAN become unstable. What we see here, however, is not instability in the physical system, but only in the digital representation of it.

The moral of this story is that when you employ slowing factors you must always make sure that ks is greater than or equal to dt. If you need to represent a system with a faster rise to asymptote, you must reduce dt. Then you can reduce the time constant further by reducing ks as required to match the behavior of the real system. We will see later how this kind of analysis works when the real system has some actual time delays in it.

Things to experiment with

Loop gain is affected by not only by ko, but by kf and ki. You can try varying these other constants to see the effect. You will always find that the fastest physically meaningful speed of error correction is a single iteration, and

that to get this speed you must set kd/ks to 1/(1 + ko*kf*ki). If you want the system to have a time constant of x iterations, you simply multiply the denominator by x. When you have created a certain time constant in the correction of errors due to disturbances, you can then verify that you get the same time constant in response to step-changes in the reference signal sr. To do this you can crib some lines from primer1.c in Part I. The reference signal value is already displayed. If you want to display the error signal, just add the line

```
gotoxy(i,24 - se/2.0 - 0.5); putch('e');
```

... in the appropriate place, and so on for other variables you would like to plot. Notice that when two or more variables plot to the same position, only the last one to be plotted can be seen.

Best to all, Bill P.

Date: Sun Nov 15, 1992 9:53 am PST
Subject: Primer part III

Continuing with the draft version of the modeling primer:

Experimenting with the control paradigm
A primer for computer modelers

PART III: The effects of delays.

It is often said among the ill-informed that control systems will not work well in organisms because of reaction-time and other delays. In this section we will set up a control system with delays in it, first in the output function and then in the input function, to see what is required for such a system to maintain tight and stable control.

Output delay

The program is an extension of the one in Part II:

```
/* primer3.c */

#include "stdio.h"
#include "conio.h"

void main()
 {
float sp = 0.0,  /* initialize signals and quantities */
      sr = 20.0,
      se = 0.0,
      qo = 0.0,
      qi = 0.0,
      qd = 0.0;
float kd = 1.0, /* set constants */
      ki = 1.0,
      ko = 100.0,
      kf = 1.0,
      ks = 70.0,
      dt = 0.1;
float qo3 = 0.0, /* initialize variables for output delay */
      qo2 = 0.0,
      qo1 = 0.0;
int i;
 clrscr();  /* alternative: for(i=0;i<25;++i) putch(0x0d); */
 for(i=1;i<=80;++i)
  {
   if(i > 40) qd = 9.0; else qd = 0.0;
   qi = kf*qo + kd*qd;
   sp = ki*qi;
   se = sr - sp;
   qo = qo1;
```

```
   qo1 = qo2;
   qo2 = qo3;
   qo3 = qo3 + (ko*se - qo3)*dt/ks;
    gotoxy(i,24 - sr/2.0 - 0.5); putch('r');
    gotoxy(i,24 - qd/2.0 - 0.5); putch('d');
    gotoxy(i,24 - qo/2.0 - 0.5); putch('o');
    gotoxy(i,24 - sp/2.0 - 0.5); putch('p');  }
 (void) getch();
}
```

The delay is put into the output function by the following lines:

```
   qo = qo1;
   qo1 = qo2;
   qo2 = qo3;
   qo3 = qo3 + (ko*se - qo3)*dt/ks;
```

The three dummy variables qo1 through qo3 provide a delay of 3 iterations. On the current iteration, the value of qo3 is computed just as it was in Part II. But this value is passed, on the next iteration, to qo2, then to qo1, and only on the third iteration to qo, the actual output quantity. So the output is always based on the error signal that existed three iterations, or 3*dt seconds, ago.

This is called a "transport lag." The variations in qo follow the variations in qo3, but three iterations later. The effect is much like shouting commands from one end-zone of a football field to a marching band in the other end zone. The marchers follow all the commands in properly-spaced sequence, but with a delay of about 0.3 seconds.

It may not be obvious, but in the program of Part II there was already a transport lag of dt, one iteration. The effect of a disturbance of the input quantity during one iteration did not make its way around the loop to affect the input quantity until one iteration later. We saw that it was possible, with a delay of 0.1 second, to adjust the system for tight control despite this lag.

Now the situation is slightly different. Effects still propagate around the loop in one iteration, but the transport lag inserts a delay of 3 iterations in the output function alone. In ALL the functions, including the output function, there is a new value of input and output calculated on every iteration; the output quantity and input quantity can change during the transport lag period. Previously nothing at all could happen during the lag of one iteration. Now, with a lag of three iterations, the error signal can change on every iteration even though the effect on the output quantity is delayed by 3 iterations.

Compile and run the program. The appearance is much the same as in Part II, but now a "reaction time" is visible. After the initial rise in reference signal, the output quantity, represented as an 'o', does not start to rise until the fourth iteration, and the perceptual signal starts to rise one iteration later. The perceptual signal, a 'p', then rises smoothly until it matches the reference signal. When the disturbance jumps to 9.0 units, the perceptual signal is affected immediately. But the output signal, which opposes the disturbance, begins to change on the fourth iteration after the start of the disturbance. Then the perceptual signal begins to return toward the reference signal (some apparent lags are simply due to the coarseness of the resolution of the plot).

We have paid one penalty for this transport lag. The slowing factor, which had an optimum value of 10.1 in Part II, now must have a value of at least 70 to avoid overshoots. This added slowing is required in order to compensate for the transport lag and keep the control system stable. If you reduce ks to speed up the transitions, the perceptual signal will begin to overshoot and undershoot. Try reducing the slowing factor ks in steps of 10. The instability will get worse and worse until a runaway oscillation starts and control is totally lost.

Input delay

Now we restore the output lag to zero (or one iteration) and use the same mechanism to put a lag into the input function.

```
/* primer4.c */

#include "stdio.h"
#include "conio.h"

void main()
 {
float sp = 0.0,  /* initialize signals and quantities */
      sr = 20.0,
      se = 0.0,
      qo = 0.0,
      qi = 0.0,
      qd = 0.0;
float kd = 1.0, /* set constants */
      ki = 1.0,
      ko = 100.0,
      kf = 1.0,
      ks = 70.0,
      dt = 0.1;
float sp3 = 0.0, /* initialize variables for perceptual delay */
      sp2 = 0.0,
      sp1 = 0.0;
int i;
 clrscr();  /* alternative: for(i=0;i<25;++i) putch(0x0d); */
 for(i=1;i<=80;++i)
  {
   if(i > 40) qd = 9.0; else qd = 0.0;
   qi = kf*qo + kd*qd;
   sp = sp1;
   sp1 = sp2;
   sp2 = sp3;
   sp3 = ki*qi;
   se = sr - sp;
   qo = qo + (ko*se - qo)*dt/ks;
    gotoxy(i,24 - sr/2.0 - 0.5); putch('r');
    gotoxy(i,24 - qd/2.0 - 0.5); putch('d');
    gotoxy(i,24 - qo/2.0 - 0.5); putch('o');
    gotoxy(i,24 - sp/2.0 - 0.5); putch('p');
  }
 (void) getch();
}
```

Compile and run this program.

At the start of the run, when the reference signal is set to 20.0, the output immediately begins to rise. Examining the block diagram of the system (Part I), you can see that a change in the reference signal shows up immediately as a change in the error signal, which drives the output. This change in the output, however, is not reflected in the perceptual signal until the fifth iteration (it is probably affected on the fourth iteration, but not enough to show on the plot). When the disturbance switches to 9.0, the perceptual signal does not show the effect for three iterations even though the input quantity is immediately affected; then, when this effect does appear in the perceptual signal, the output begins to change at the same time.

So there is some difference when the delay is shifted to the input side. But control remains just as good as before, and with the same slowing factor of 70.

Sampled Control

Control systems can be designed so that they sample the state of the input at intervals instead of continuously. In the final program in this section, we use a timer variable t to count iterations. On every fourth iteration, the

perceptual signal is set to the appropriate level on the basis of the input quantity. Between samples, the perceptual signal simply holds the last sampled value. All the other variables in the loop change on every iteration as usual. Here is the program:

```
/* primer5.c */

#include "stdio.h"
#include "conio.h"

void main()
 {
float sp = 0.0,  /* initialize signals and quantities */
      sr = 20.0,
      se = 0.0,
      qo = 0.0,
      qi = 0.0,
      qd = 0.0;
float kd = 1.0, /* set constants */
      ki = 1.0,
      ko = 100.0,
      kf = 1.0,
      ks = 70.0,
      dt = 0.1;
int i,t = 0;
 clrscr();  /* alternative: for(i=0;i<25;++i) putch(0x0d); */
 for(i=1;i<=80;++i)
  {
   if(i > 40) qd = 9.0; else qd = 0.0;
   if(t == 0)
    {
     qi = kf*qo + kd*qd;
     t = 4;    }
   t = t - 1;
   sp = ki*qi;
   se = sr - sp;
   qo = qo + (ko*se - qo)*dt/ks;
    gotoxy(i,24); putch('-');
    gotoxy(i,24 - sr/2.0 - 0.5); putch('r');
    gotoxy(i,24 - qd/2.0 - 0.5); putch('d');
    gotoxy(i,24 - qo/2.0 - 0.5); putch('o');
    gotoxy(i,24 - sp/2.0 - 0.5); putch('p');
  }
 (void) getch();
}
```

Once again, smooth control is achieved with a slowing factor of 70. Notice that while the output quantity changes smoothly, the perceptual signal changes in steps, one step on every fourth iteration. Obviously no disturbance can be resisted if it occurs during the hold period; the opposing output can only begin to change when the next sample occurs. The average delay would be half the sampling period. As with all control systems, disturbances can come and go too rapidly to oppose. But natural control systems are adapted to the environment that exists; most natural disturbances, therefore, can be opposed before their effects become important.

Best, Bill P.

[From Bill Powers (921120.0830)] Wolfgang Zocher (direct)

I am VERY interested in your program for assembling control systems using modules. If you will send me the source code I'll see if I can get it running, and offer any suggestions that seem useful. If we can work out something with the required properties, I'll switch to using it in my Primer series.

Bill P.

[For fruits of this collaboration, see PCTdemos: SIMCON directory].