Nonlinear PID Controller Tuning Guide

ABSTRACT

The C2000 Digital Control Library contains a nonlinear PID controller, denoted NLPID, which utilizes a power function to implement the control law. The NLPID is an adaptation of the linear PID in which a nonlinear law based on a power function is placed in series with each path. In high sample rate applications the control algorithm must be executed as quickly as possible so efficient computation of the power function is important. Typically, the power function is implemented in a standard C run-time support library and consumes several hundred CPU cycles. Two instructions have been added to the C2000 instruction set which enable the power function to be computed in only a dozen or so cycles and enable the use of nonlinear control in high sample rate applications; however the presence of nine tuning parameters (rather than three in the linear PID) significantly complicates the tuning process. This document contains a technical description of the NLPID and presents various methods for tuning such a controller.

Contents

Figures

1 Introduction

The C2000 Digital Control Library contains a nonlinear PID controller, denoted NLPID, which utilizes a power function to implement the control law. The NLPID is an adaptation of the linear PID in which a nonlinear law based on a power function is placed in series with each path. In high sample rate applications the control algorithm must be executed as quickly as possible so efficient computation of the power function is important. Typically, the power function is implemented in a standard C run-time support library and consumes several hundred CPU cycles. Two instructions have been added to the C2000 instruction set which enable the power function to be computed in only a dozen or so cycles and enable the use of nonlinear control in high sample rate applications; however the presence of nine tuning parameters (rather than three in the linear PID) significantly complicates the tuning process. This document contains a technical description of the NLPID and presents various methods for tuning such a controller. Although the document is concerned with the nonlinear PID controller, the material is also applicable to the nonlinear PI controller and to the nonlinear control law module in the DCL. Readers may find it helpful to refer to the *Linear PID Tuning Guide* document [4] which is included with the DCL package.

2 The Nonlinear PID Controller

2.1 Background

The structure of a typical parallel form PID controller is shown in Figure 1. In a typical feedback control loop, the controller is situated in the forward path immediately before the plant. The input to the controller is the servo error; the instantaneous difference between the commanded reference r and the measured output y.

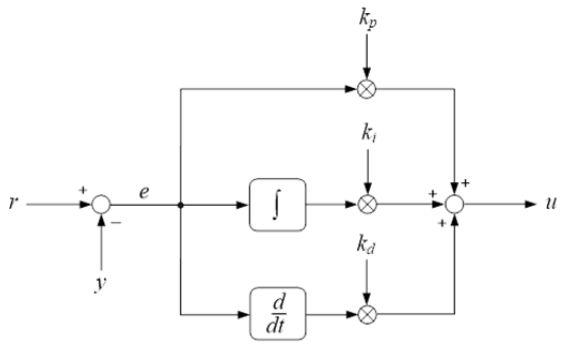


Figure 1. Parallel form linear PID controller

In the linear PID controller the servo error is connected to three parallel paths (proportional, integral, and derivative), the gain of each being independently adjustable. The influence of each path on the transient response is different, so tuning the controller is a matter of blending together the gains of the three paths to optimize the transient response. In some cases the gains might be chosen to place a pair of controller zeros in the complex plane, but this only applies to the linear case.

The nonlinear PID controller presented here is an extension of the basic design of Figure 1. A nonlinear block is introduced in series with each of the three controller paths. Each nonlinear block shapes the servo error according to a power function law in which the normalized input (the servo error) is raised to the power of an adjustable tuning parameter (α). Figure 2 shows a range of input-output curves for α between 0.2 and 2.

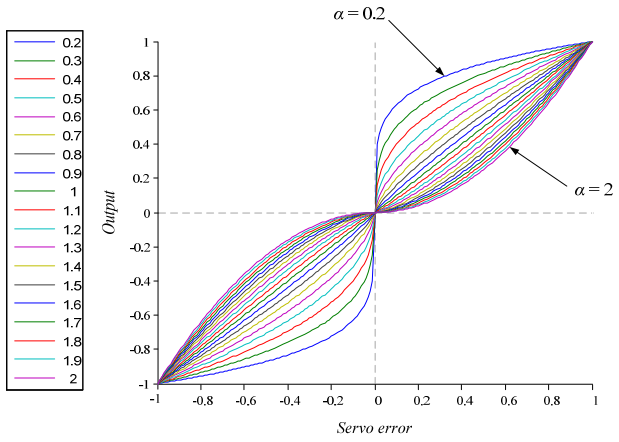


Figure 2. Input-output curves for varying tuning parameter (α)

The tuning parameter determines the degree and direction of the gain shape. Observe that when α is unity, the control law is linear with unity gain, and that all the curves intersect at the origin and at ±1.

An issue arises with the nonlinear law near the zero error point. When α is greater than one the gain of the law is zero when the loop error is zero. Since zero error is typically the equilibrium point in steady state, control action will be sluggish for small disturbances in each path for which alpha is greater than unity. When α is less than unity the path gain is infinite at the zero error point, potentially inducing sustained low amplitude oscillation in the steady state.

The solution is to define an input range covering the origin over which the gain is held constant. The gain in this region is chosen to ensure that linear and nonlinear curves intersect precisely at their boundaries, resulting in a smooth, glitch-free transition from one region to the other. Taking x as the input and y as the output, the complete nonlinear control law is:

$$y = \begin{cases} |x|^{\alpha} sgn(x) & : x \geq \delta \\ \delta^{\alpha-1} x & : x < \delta \end{cases} \quad (1)$$

Figure 3 shows the linear and nonlinear regions for α less than one. Notice that the linear gain is independent of the input x, so does not need to be computed each time the controller runs: the linear gain is fixed for each path and need only be recomputed when either of the nonlinear parameters in that path is adjusted.

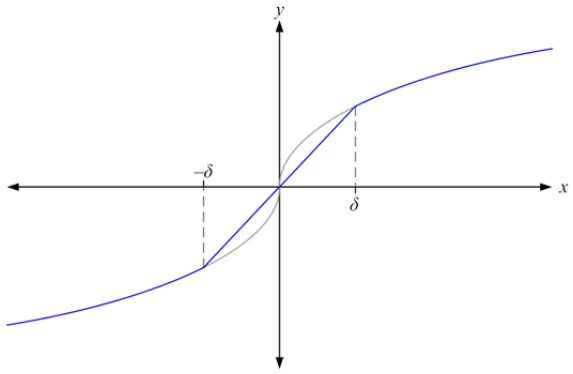


Figure 3. The linearized region

The full control law now comprises two nonlinear parameters (α and δ) and one series linear gain in each of the three paths, giving a total of nine tuneable parameters. Further information on this controller can be found in [1] and [2] .

2.2 Nonlinear Law Implementation

Figure 4 shows the NLPID_C2 controller in the DCL. The structure is similar to a typical parallel form linear PID controller except that a nonlinear block appears in series with each of the three paths. Each nonlinear block comprises the nonlinear law involving the α and δ parameters associated with that path. Terms common to both the proportional and integral paths are computed together in a 'pre-conditioning' block for cycle efficiency.

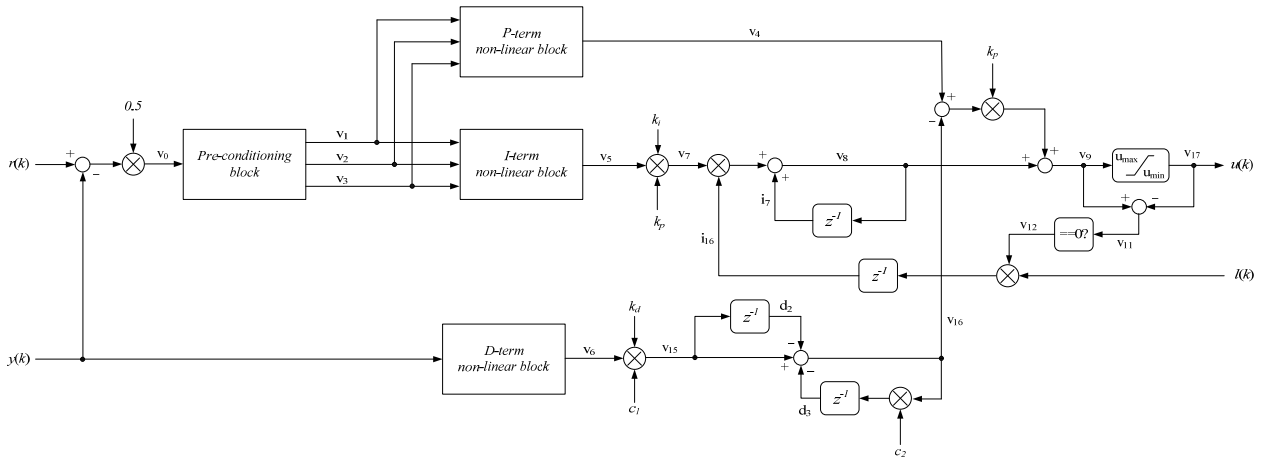


Figure 4. The NLPID_C2 controller

In the DCL, the nonlinear law for the proportional block in this controller is coded in an inline C function as follows.

```
v1 = rk - yk;                          // servo error (x)
v2 = (v1 < 0.0f) ? -1.0f : 1.0f;       // sgn(x)
v3 = fabsf(v1);                        // |x|
v4 = ((v3 > p->delta_p) ? (v2 * (float32_t) powf(v3, p->alpha_p)) : (v1 * p->gamma_p));
```

Figure 5. C code for the nonlinear control law

In the code of Figure 5 the variable ‘v1’ is the instantaneous loop error, ‘v2’ is the sign of the error, and ‘v3’ the absolute value of the error. The last line determines whether the error lies inside the semi-width of the linearized region defined by ‘delta_p’, and if so performs a simple multiplication of the loop error with the linear gain (‘gamma_p’) which is computed from ‘delta_p’ and ‘alpha_p’. In the nonlinear region the controller calls the powf() function with the loop error and the ‘alpha_p’ parameter as arguments, then restores the sign. Nonlinear terms for the integral and derivative paths are computed similarly.

In steady state magnitude of the loop error (v3 above) is near zero, so all three paths operate in linear mode. However, transiently all paths may be operating in the nonlinear region, so in the worst case the power function must be called three times each time the controller is run. This function is the most computationally expensive function present, so the efficiency with which it can be computed heavily influences the controller cycle count and therefore the maximum control rate at which it is feasible to use the controller.

3 The Power Function

The “powf” C function raises one floating-point number to the power of another floating-point number, and returns a floating-point result. All operands and results are in single precision. There are three ways of computing this function on C28x.

3.1 The RTS Library

The run-time support (RTS) library suppled with the C compiler contains support for powf() as defined in the header file “math.h”. This implementation is not optimized for cycle count and will execute the control law in Figure 5 in approximately 975 cycles.

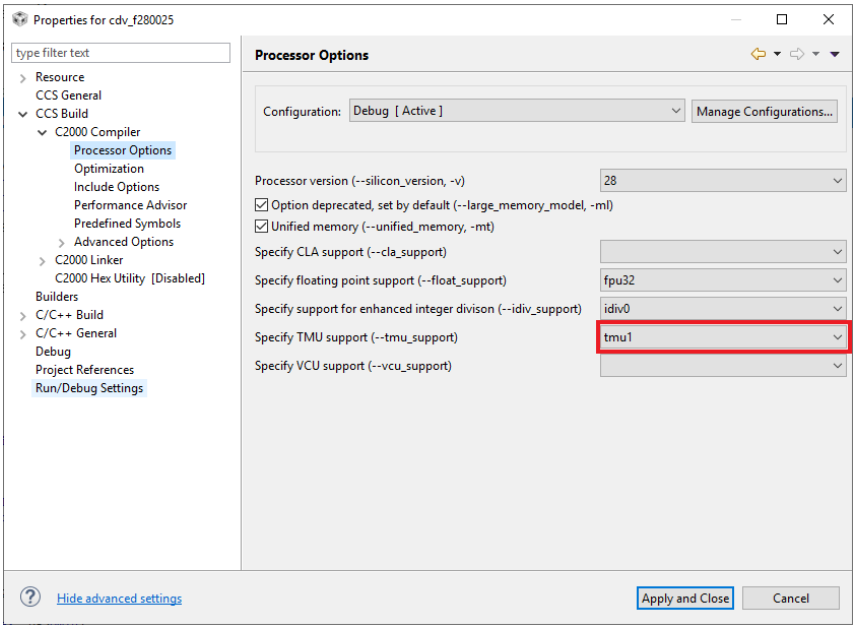
3.2 The FastRTS Library

The C2000 “FastRTS” library contains a substitute for the RTS powf function which executes in 109 cycles including calling overhead. For further information, refer to the Fast Run Time Support Library “User’s Guide in the “libraries -> math” sub-directory in C2000Ware.

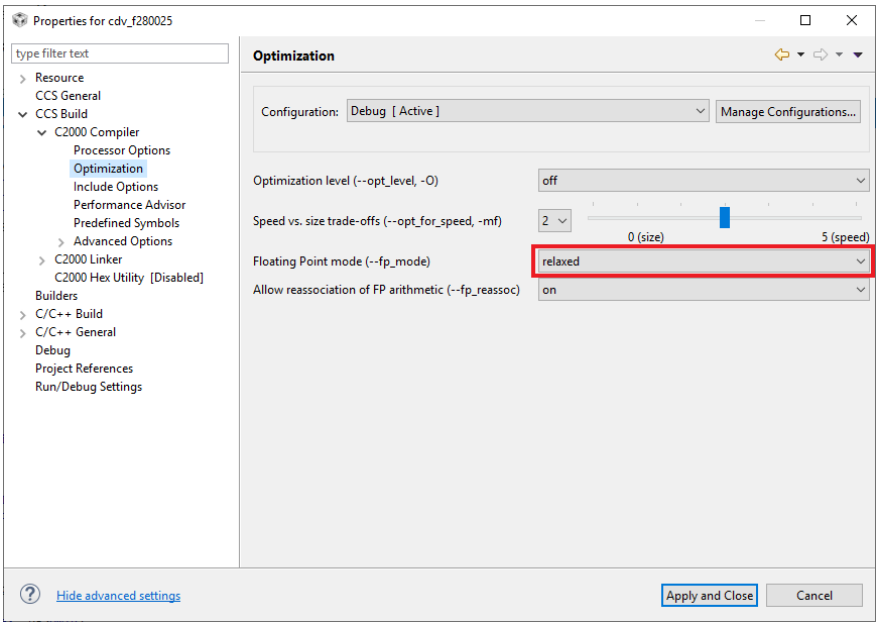
3.3 TMU Support

On devices equipped with TMU type 1 or higher, there are two instructions which support computation of the power function using a base-2 logarithm followed by an inverse base-2 exponent. To invoke C compiler support for these functions the following steps are required.

1. In the CCS Project Explorer window, right-click on the project name and select “Properties”.
2. Under “Processor Options, select “tmu1”:



3. Under “Optimization”, ensure the floating-point mode is set to “Relaxed”.



4. Recompile the project.

With these options selected, the four lines of C code in Figure 5 will execute in approximately 77 cycles.

It is possible to improve this figure by coding the lines in assembly. An assembly function is shown in Appendix A which computes the same nonlinear law in 33 cycles (including calling overhead). At the time or writing the DCL does not include an assembly coded NLPID controller, however a prototype for such a controller (NLPID_C3) is in the header file DCL_NLPID.h, allowing for implementation in a future release of the library.

4 Tuning

4.1 Manual Tuning

This section suggests a manual tuning procedure for the nonlinear PID controller. The approach requires a tuning index with which to measure the quality of the transient response. In what follows, the ITAE index is used. Information on the ITAE index applied can be found in [3] .

Initially, all nonlinear parameters are set to unity and the linear gains adjusted iteratively until an optimum tuning is obtained. This part of the process has been documented in the linear PID tuning guide which is packaged with the DCL. Once a satisfactory linear tuning has been obtained, the user introduces nonlinearity in an experimental fashion to improve the tuning index. Small adjustments are made to each parameter in turn, before moving on to the next.

The following list of steps constitutes a suggested procedure for manually tuning the nonlinear PID controller described in section 2.2.

Step 1. Configure the nonlinear parameters

Set all the nonlinear 'alpha' parameters to unity so that the controller operates in linear mode. Select an arbitrary value of 0.25 for all three linear region semi-widths (δ). These parameters will be adjusted later.

Step 2. Initialize the linear gains

Set the proportional gain (k_p) to a known safe initial value. Be sure to select a value significantly lower than the expected optimum value so that the gain can be safely increased without risk of the control loop becoming unstable. Ensure both the integral and derivative gains are set to zero.

Step 3. Initialize the control limits

Determine the physical range of the control output and set appropriate limits at the output of the controller. In some DCL controllers, it is possible to connect limits from remote parts of the control system. It is important to monitor the saturation variable inside the controller while conducting the tuning process to ensure the loop does not saturate. In DCL v3.x and later, each controller has a supporting structure (CSS) which contains a "testpoint" variable intended to bring out internal controller signals for monitoring purposes.

Step 4. Configure the derivative filter

Oscillatory effects in the transient response can sometimes be reduced by applying derivative gain. To do this, first decide on a bandwidth for the derivative low-pass filter. The choice of filter bandwidth should be low enough to remove most of any sensor and process noise present, but not too low that the operation of the differentiator is compromised. More general guidelines are difficult to give and some degree of trial-and-error may be necessary. If filtering is not required, use an infinite cut-off frequency (i.e. zero time constant).

Apply the equations in the DCL User's Guide [2] to find the derivative filter coefficients c1 & c2, and load these values into the NLPID controller structure.

Step 5. Apply a test stimulus

Apply a stimulus to the control loop in such a way as to induce an observable transient at the system output. In many cases the disturbance will be a sudden change in reference set-point; in others, a change in output load may be easier to apply. Record the transient part of the output response. The DCL data logger utility will be useful for this task and code examples can be found in the library illustrating its use.

It is important to ensure the output of the controller does not saturate during this step. Users should monitor the 'i16' saturation variable ('i10' for the PI controller) while executing the response test. A value of zero at any point indicates that the controller has saturated and the parameter in question should not be adjusted further. In general, a large controller output can be indicative of control problems and it is advisable to monitor the magnitude of the control effort during the tuning process using a separate data logger.

Step 6. Measure the response quality

Compute and record the ITAE tuning index using data captured in the data log. Refer to [2] for more information. It is important that a constant time interval is used for all ITAE measurements in order that a fair comparison of responses can be made. The user should also monitor the control effort to ensure the loop does not saturate during the test.

Step 7. Adjust the proportional gain

If the response does not meet specifications, adjust k_p and repeat the transient test. Ensure that the length of the data log remains the same in order that the quality of the measured responses can be fairly compared. Repeat this step until the optimum value of k_p is found which minimizes the tuning index.

In general, increasing k_p will reduce (but not eliminate) steady state error. Increasing k_p also tends to reduce the response rise time but can introduce over-shoot and oscillation. Keep in mind that like all tuneable parameters, changes to k_p should be made in small steps, and that some systems will become unstable if the proportional loop gain is excessive. There may also be a lower limiting value of k_p which yields a stable response.

If a value of k_p can be found which meets all the performance objectives the tuning procedure can be terminated at this point.

Step 8. Adjust the integral gain

Depending on the nature of the control loop, steady state error can sometimes be eliminated with the introduction of integral control action. The effect of integral control depends on the open loop transfer function and the type of test stimulus applied [3] . In industrial applications the test stimulus is often a step change of reference input, and zero steady state error is achieved when at least one integrator is present inside the loop. For illustrative purposes, this is the situation we will assume here.

If necessary, gradually increase the integral gain term (k_i) to reduce steady state output error. Increasing the k_i increases the rate at which the response converges on steady state, but may introduce or amplify overshoot and oscillation. If this happens, it may be useful to slightly decrease k_p and then re-adjust k_i. Repeat this step until an optimum value of k_i is found. Again, if all performance specifications are met, terminate the procedure here.

Depending on the plant dynamics, the response may be very sensitive to the integral term and may become unstable, so be sure to start with a very small gain. In general, the faster the response of the plant, the greater will be the sensitivity of the control loop to integral gain.

Step 9. Adjust the derivative gain

Apply a small amount of derivative gain (k_d) and repeat the transient test. As with the other gains, changes should be made in small steps. Depending on the nature of the plant, the control may be strongly or weakly dependent on this parameter. In general, a system which is sensitive to k_i is insensitive to k_d and vice-versa.

If small amount of derivative action makes no significant difference it may be necessary to use progressively larger increments until a difference is seen. In general, the faster the plant the less sensitive is the closed loop response to derivative action.

It may be necessary to re-adjust k_p and k_i gains at this point. Typically, tuning involves repeated adjustments to the gain terms to find the best achievable response.

Step 10. Repeat the procedure

At this point, the user will typically return to step 5 and re-adjust the proportional gain, again proceeding in small incremental steps. The introduction of integral or derivative action will change the optimum value of k_p and it is likely some improvement can be made. He or she will then move on to re-adjust k_i and k_d. This cyclic procedure continues until either all the performance specifications have been met, or the user judges that no further improvement to the performance index can be made.

Hitherto, we have proceeded in accordance with the guidelines for linear PID tuning. From this point onwards we introduce nonlinear control action.

Step 11. Introduce nonlinear action into the proportional path

Make a small change to the nonlinear ‘alpha’ parameter in the proportional path. Repeat the transient test and record the tuning index. Reverse the direction of the applied change to ‘alpha’ and re-test. If either change improved the tuning index, continue to adjust ‘alpha’ in that direction and repeat the test until the index is minimized.

Step 12. Adjust the linear region semi-width parameter

Make a small adjustment to the ‘delta’ parameter for the proportional path. This parameter was set to a suggested value of 0.25 in step 1. Test the response and record the tuning index. Reverse the direction of the change; re-test the response and compare the tuning index obtained. Continue to adjust delta until a minimum index has been obtained.

Step 13. Adjust the nonlinear terms in the integral path

Return to step 11 and proceed in a similar fashion for the nonlinear parameter in the integral path. Then, repeat step 12 to optimize the linear region semi-width in the integral path.

Step 14. Adjust the nonlinear terms in the derivative path

Return to step 11 and proceed in a similar fashion for the nonlinear parameter in the derivative path. Then, repeat step 12 to optimize the linear region semi-width in the derivative path.

At this point, the user may decide to return to one of the earlier steps and re-adjust one or more of the parameters. Continued adjustments can potentially improve control performance; however there is no guarantee that adjustment of one parameter will not be detrimental to the optimum setting of another and many repetitions through the adjustment cycle may not yield significant improvements. Other techniques for optimizing the controller settings are described later in this document.

By way of an example, consider the plots in Figure 6, which illustrate the effects of introducing nonlinear control action to a well-tuned linear PID. From left to right, the plots in each show respectively the input reference and feedback, controller output, loop error, and controller saturation. Controller gains and the corresponding ITAE index are shown on the left.

The top row is the best which could be achieved using only the linear parameters. Notice that the nonlinear gains are all set to unity. At this point, the user has completed step 10 in the above procedure.

The plots in the second row show the effect of adjusting the alpha term in the proportional part. In this case, it was necessary to reduce this parameter in order to improve the index. The user would have experimented with alpha values above and below unity to establish this. This completes step 11.

The third row plots show the result of slightly adjusting the delta parameter in the proportional path, as descripted in step 12. Only a small adjustment was made, and only a small improvement in the index was seen. This is likely due to the fact that the servo error was quite small in this case.

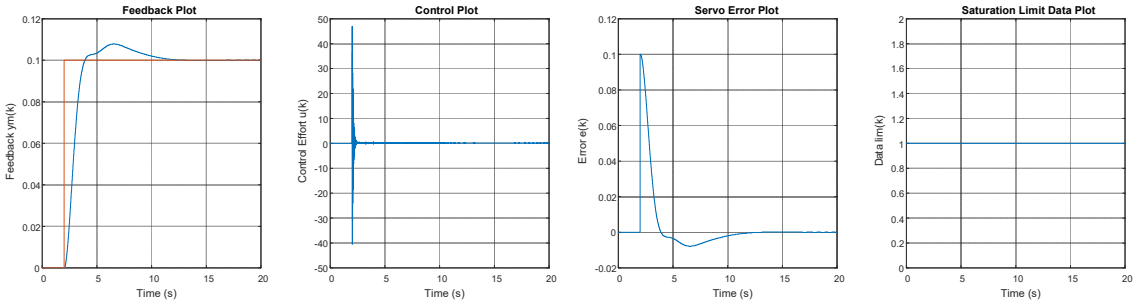
The fourth and fifth rows show the results of similar adjustments to the integral and derivative nonlinear parameters.

Kp = 4.5
Ki = 0.004
Kd = 0.55

alpha_p = 1.0
alpha_i = 1.0
alpha_d = 1.0

delta_p = 0.15
delta_i = 0.15
delta_d = 0.15

ITAE = **0.4586**

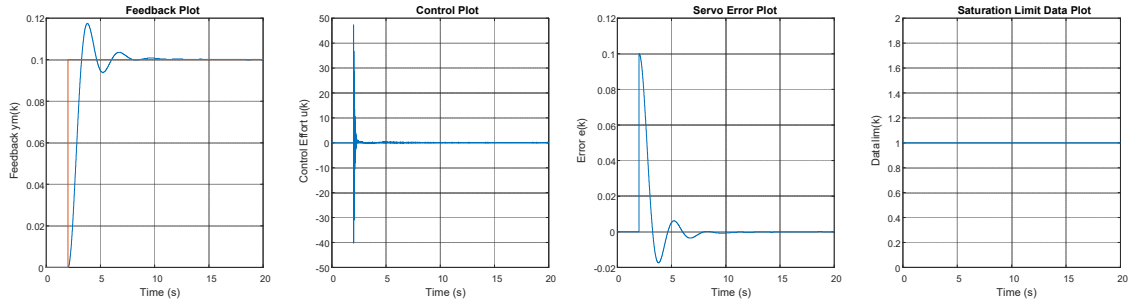


Kp = 4.5
Ki = 0.004
Kd = 0.55

alpha_p = 0.75
alpha_i = 1.0
alpha_d = 1.0

delta_p = 0.15
delta_i = 0.15
delta_d = 0.15

ITAE = **0.3181**

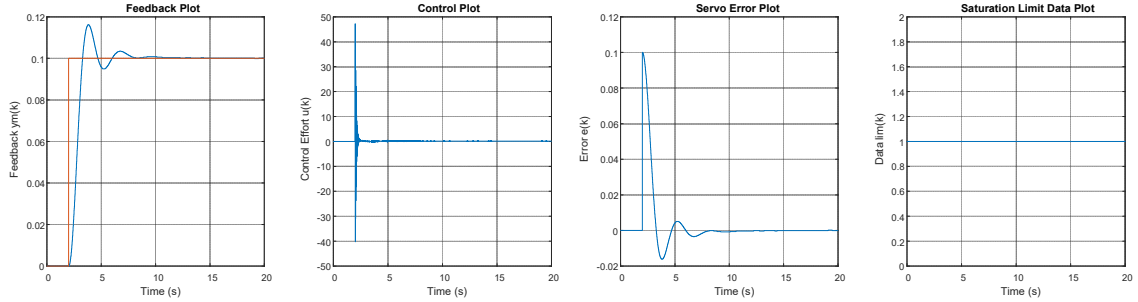


Kp = 4.5
Ki = 0.004
Kd = 0.55

alpha_p = 0.75
alpha_i = 1.0
alpha_d = 1.0

delta_p = 0.17
delta_i = 0.15
delta_d = 0.15

ITAE = **0.3165**

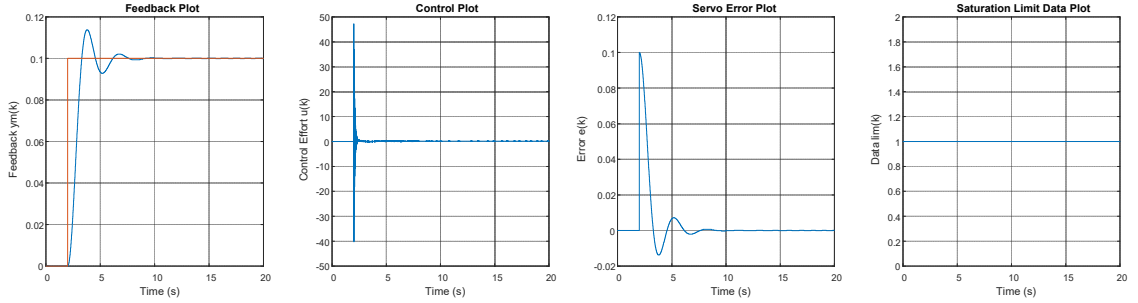


Kp = 4.5
Ki = 0.004
Kd = 0.55

alpha_p = 0.75
alpha_i = 1.1
alpha_d = 1.0

delta_p = 0.17
delta_i = 0.14
delta_d = 0.15

ITAE = **0.2940**



Kp = 4.5
Ki = 0.004
Kd = 0.55

alpha_p = 0.75
alpha_i = 1.1
alpha_d = 0.7

delta_p = 0.17
delta_i = 0.14
delta_d = 0.15

ITAE = **0.2262**

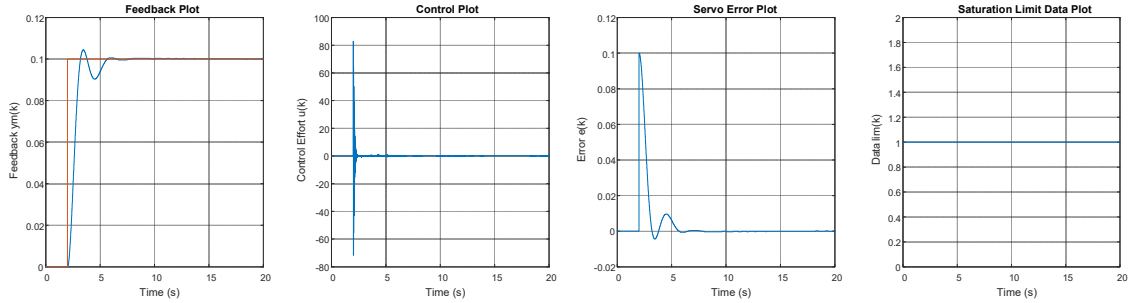


Figure 6. Manual tuning example plots

The ITAE achieved with nonlinear control is significantly better than that obtained using linear control alone; however it must be understood that the nonlinear parameters are not necessarily optimal in a global sense.

This concludes the suggested procedure for manually tuning the nonlinear PID controller. In summary, the process of manual tuning is nothing more than a series of adjustments to the three linear controller gains, followed by iterative adjustments to the nonlinear parameters. The key points are that all adjustments must be made in small steps, and that the use of a performance index provides a useful, non-subjective performance metric.

4.2 Gradient Descent

The gradient descent method is a formalization of the manual tuning process in which each parameter adjustment is automated according to whether the change improves or degrades the performance index. It is described here to illustrate the difficulties in finding optimal controller parameters. No supporting code or materials are provided.

Consider a parallel form PI controller in which ITAE index results have been compiled for proportional and integral gains over a given range. Since there are only two parameters, we could present the results in the form of a surface in which the parameter values occupy the floor of the diagram, and the vertical axis is the ITAE value. An example of such a surface is shown in Figure 7.

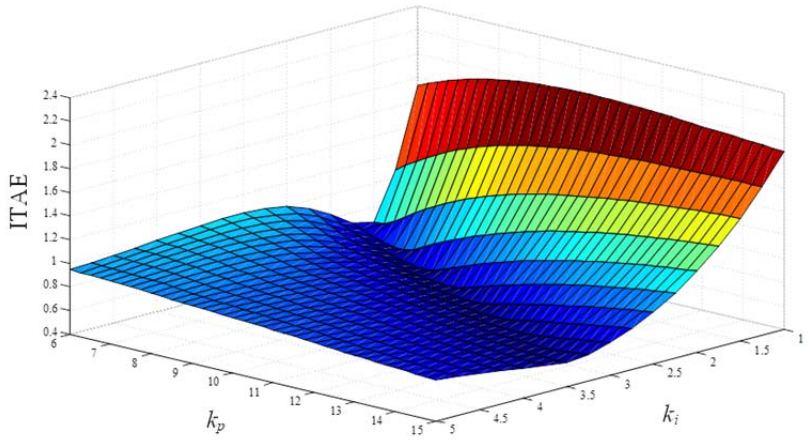


Figure 7. Typical PI controller ITAE surface

Determination of the optimum controller gains is now a matter of finding the lowest point on the surface, which is easily done using Matlab, for example. Clearly much effort must be expended in conducting all the response tests before the search can begin, so in practice this method is rarely feasible unless a plant model is available and the work can be done off-line.

The gradient search process involves evaluation of the index at pairs of points corresponding to two slightly different values of a control parameter. The change in parameter value and the corresponding difference in ITAE are used to calculate the gradient of the surface and to compute further adjustment in the parameter. In this way, consecutive adjustments 'walk' the parameter choice downhill across the surface towards the lowest point. Adjustments are made for one parameter at a time while the other is held constant. Attention is then transferred to a different parameter, and the cycle repeats until all the parameters have been treated in this way, or satisfactory tuning achieved.

In principle, the method finds the optimum parameter combination providing the change in surface gradient is monotonic with respect to both parameters: i.e. that a downhill gradient always leads towards the lowest point on the entire surface. However the surface shape is dependent on the plant as well as the controller, and may exhibit topological features such as 'ridges' and 'isolated depressions' which steer parameter adjustments towards a choice which is sub-optimal in a global sense. This drawback is also true of the manual tuning method. Figure 8 shows an example of a more complicated surface which exhibits both of these features.

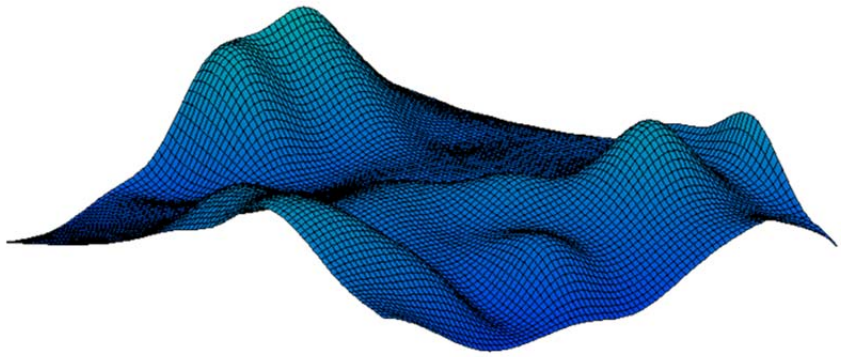


Figure 8. ITAE surface example with local minima

Clearly the gradient search method cannot be simply applied to controllers with more than two adjustable parameters; however the concept of local and global minima is valid for control problems of any dimension.

4.3 Genetic Algorithm

A method which in principle can identify a globally optimal parameter selection is the genetic algorithm (GA). This method mimics the process of natural selection to breed desirable qualities into a population consisting of randomly selected controller parameters. Crucially, the GA allows for random mutations to appear in the population which ensures that the entire parameter space is covered as the population evolves and enables a global minimum to be identified. Like the gradient descent method above, the GA method is described with no supporting code or materials. Further information on this method can be found in [5] .

The first step is to define a population consisting of N 'chromosomes', each of which comprises nine controller parameters. All parameters are initialized with random values, in which state the population is said to be made up of 'phenotypes'.

Each chromosome is tested and its ITAE computed. The population is then ranked from 1 to N, where rank 1 has the lowest ITAE score (i.e. performs best) and rank N the highest ITAE score (performs worst).

The breeding process begins by selecting pairs of parents. In this case, four different chromosomes are selected at random from the entire population, and 'tournament selection' used to select one parent from each pair. This involves selecting from each pair the chromosome with the highest ranking, thereby ensuring that the two lowest ranked chromosomes do not breed, and reducing the breeding chances of the poorest performing chromosomes.

Each pair of parents produces two children and then immediately disappears from the process, so that the population size remains constant. The attributes of the parents are passed on to the children according to the formulae below, in which β is a random number between zero and one which is fixed for each parent pair. The symbols p_1 and p_2 denote some controller parameter in the two parents, while c_1 and c_2 are the same parameter inherited by the two children.

$$\begin{aligned} c_1 &= \beta p_1 + (1-\beta)p_2 \\ c_2 &= (1-\beta)p_1 + \beta p_2 \end{aligned} \tag{3}$$

At this point we could re-test and rank the population, however doing so means that within three to four generations the chromosomes will all acquire similar parameters because the solutions converge on a (possibly) local minimum. In order to ensure the entire parameter space is adequately explored, mutations are introduced into the population at the rate of approximately 50%. In each randomly selected chromosome, one parameter is selected at random and replaced with a random number.

Elitism is introduced by omitting a small number of the highest ranked chromosomes from the elimination and mutation phase, such that they survive intact for more than one generation. The size of the elite group is typically only two or three.

The process now returns to the test stage and repeats for the specified number of generations. A flow diagram of the genetic algorithm is shown in Figure 9.

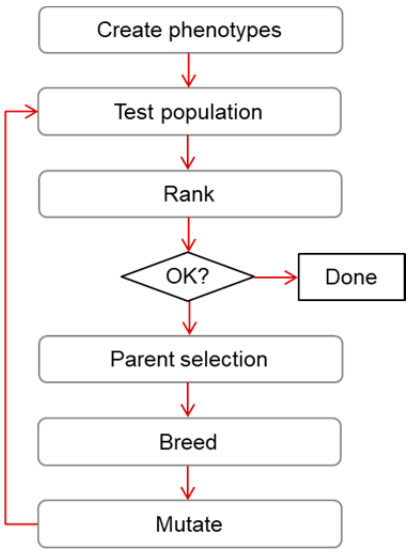


Figure 9. Genetic algorithm flow diagram

The Matlab/Simulink test model used to develop the nonlinear PID in the DCL was adapted for the GA experiment. This uses a third order plant model with adjustable delay time. The GA test parameters were a population size of 20, elite (i.e. non breeding) group size of 3, mutation rate of 55%, and generation count of 19.

The figure below shows the ITAE evolution of the highest ranked chromosome in each generation. Observe that the evolution proceeds in a series of downward jumps as better parameter settings are discovered. The best ITAE achieved after 19 generations was 0.1616. The step response for this controller is shown below.

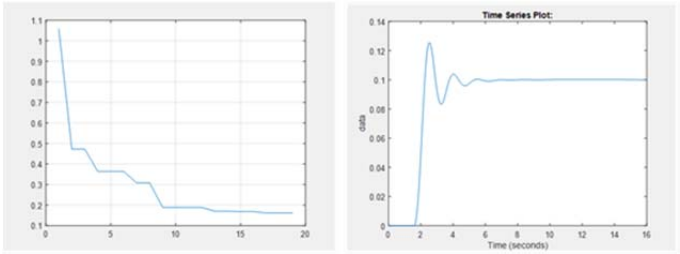


Figure 10. ITAE evolution and step response of top ranked chromosome

To place this in context, the optimum linear PID settings evolved in the same way gave an ITAE of 0.3806. This is already an impressive result for a system in which an ITAE of around 0.5 is known to be good. With the mutation rate set to zero for the nonlinear case, the algorithm stabilized on an ITAE of 0.4888 – a local minimum for this plant.

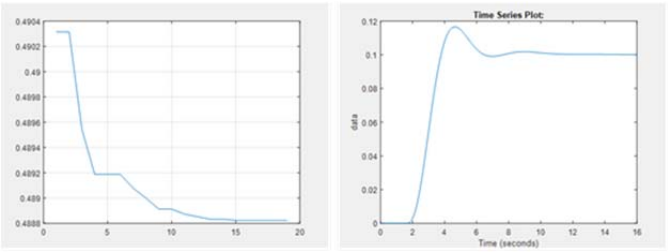


Figure 11. ITAE evolution and step response of top ranked chromosome without mutation

The GA method is amenable to techniques for shaping the transient response using weights applied to the performance index. The ITAE index comprises a time weighting which penalizes failure to achieve zero steady state error, while remaining tolerant of (unavoidable) large errors early in the response.

Other weights could be applied to the loop error signal to accentuate those characteristics we wish to constrain in the optimized response. For example, application of a gain of 2 to the negative oscillations in the error is equivalent to amplifying any over-shoots in the response, which in turn are penalized by the GA.

The step response below shows the top ranked chromosome after 19 generations with this constraint applied. The ITAE of 0.3016 is slightly worse than the unconstrained case, as would be expected. Notice also that in this case the additional constraint meant that no better solution could be found beyond the sixth generation.

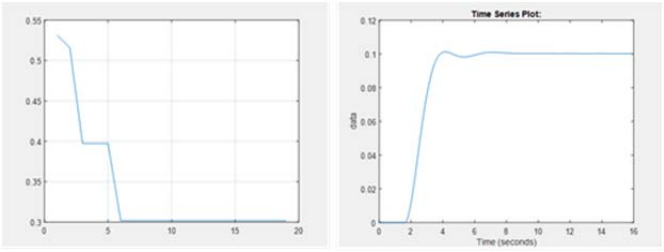


Figure 12. ITAE evolution and step response of top ranked chromosome with over-shoot weighting

As with all tuning strategies, the user should consider that the results are only optimal for the particular input used in the tests. In the above case, a step input of normalized height 0.1 was applied. For nonlinear systems the superposition principle does not apply, and in particular it does not follow that a scaling of the input always results in a corresponding scaling at the output. Therefore, if the nonlinear PID tuning process is repeated with a different input step height the results will certainly be different. The user has to exercise good judgement in selecting a test stimulus and amplitude which represents the kind of disturbance seen by the system in normal use.

5 References

[1] Jingqing Han, "*From PID to Active Disturbance Rejection Control*", Jingqing Han, IEEE Transactions on Industrial Electronics, Vol. 56, No. 3, March 2009

[2] *Digital Control Library User's Guide*, v3.2, C2000Ware

[3] R. Poley, "*Control Theory Fundamentals*", 3rd Ed., 2015

[4] *Linear PID Tuning Guide*, C2000 Digital Control Library v3.2

[5] K.D. Wilkie, M.P. Foster, D.A. Stone, and C.M. Bingham, "*Hardware-in-the-loop tuning of a feedback controller for a buck converter using a GA*", International Symposium on Power Electronics, 2008

Appendix A

```
; asm_powf.asm - assembly function to compute sgn(a)*|a|^x using TMU1
;
; Copyright (C) 2019 Texas Instruments Incorporated - http://www.ti.com/
; ALL RIGHTS RESERVED

          .if $defined(__TI_EABI__)
        .if __TI_EABI__
        .asg    asm_powf, _asm_powf
        .endif
      .endif

        .global _asm_powf

        .sect   "dclfuncs"

; C prototype: float32_t asm_powf(float32_t a, float32_t x)
; argument 1 = a : 32-bit floating-point mantissa [R0H]
; argument 2 = x : 32-bit floating-point exponent [R1H]
; return = c = a^x : 32-bit floating-point [R0H]
; note: if no TMU1 present, returns 'a'
;
; 'a' range: -1.0f < a < 1.0f
; 'x' range: 0.0f < x < 2.0f

        .align  2

_asm_powf:
        .asmfunc
 .if $defined(.TMS320C2800_TMU1)
        MOV32           *SP++, R2H
        ZERO            R2H
        ADDF32          R2H, #1.0, R2H
        CMPF32          R0H, #0.0                       ; set flags on sgn(a)
        NEGF32          R2H, R2H, LT                    ; R2H = (a < 0.0f) ? -1.0f : 1.0f
        ABSF32          R0H, R0H                        ; R0H = |a|
        LOG2F32         R0H, R0H                        ; R0H = log2(|a|)
        NOP
        NOP
        NOP
        MPYF32          R0H, R1H, R0H                   ; R0H = x*log2(|a|)
        NOP
        NEGF32          R1H, R0H                        ; R1H = -(x*log2(|a|))
        IEXP2F32        R0H, R1H                        ; R0H = 2^(x*log2(|a|))
        NOP
        NOP
        NOP
        MPYF32          R0H, R2H, R0H                   ; R0H = sgn(a)*|a|^x
        MOV32           R2H, *--SP
 .endif
        LRETR
        .endasmfunc
        .end

; end of file
```