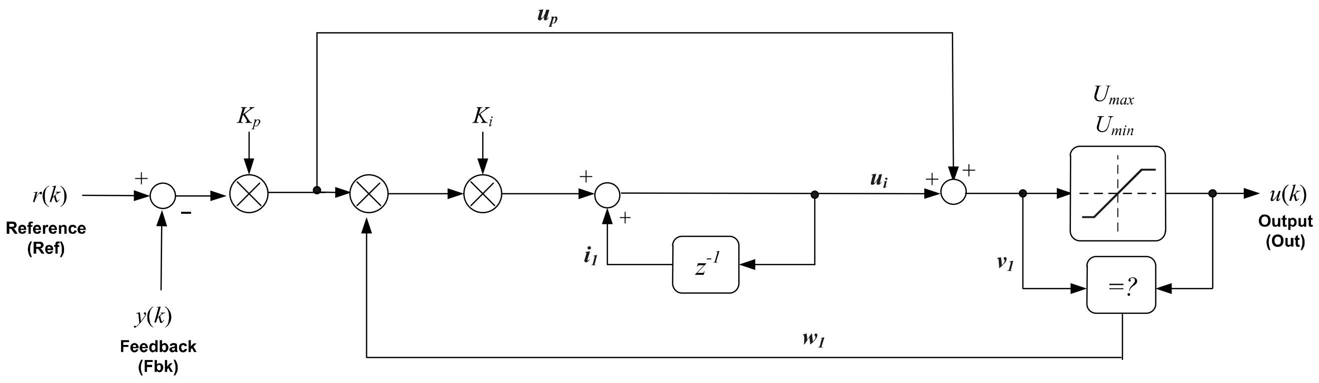
Technical Background

The PI_cntl module implements a basic summing junction and P+I control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Anti-windup integrator reset

The PI controller is a sub-set of the PID controller. All input, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.



a) Proportional path

The proportional path is a direct connection between the error term and a summing junction with the integral path. The error term is:

$$u_p(k) = e(k) = K_p[r(k) - y(k)] \quad \textbf{(1)}$$

b) Integral path

The integral path consists of a discrete integrator which is pre-multiplied by a term derived from the output module. The term w1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integral term from “winding up” and improves the response time on recovery from saturation. The integrator law used is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + K_i e(k) \quad \textbf{(2)}$$

c) Output path

The output path contains a summing block to sum the proportional and integral controller terms. The result is then saturated according to user programmable upper and lower limits to give the controller output.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero or one result is produced which is used to disable the integral path (see above). The output path law is defined as follows.

$$v_1(k) = u_p(k) + u_i(k) \qquad \textbf{(3)}$$

$$u(k) = \begin{cases} U_{\max} & : v_1(k) > U_{\max} \\ U_{\min} & : v_1(k) < U_{\min} \\ v_1(k) & : U_{\min} < v_1(k) < U_{\max} \end{cases} \qquad \textbf{(4)}$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \qquad \textbf{(5)}$$

Tuning the P+I controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable both integral and derivative paths. A suggested general technique for tuning the controller is now described.

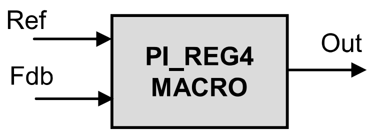
Step 1. Ensure integral is set to zero and proportional gain set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

PI_REG4 *PI Controller*

Description This module implements a simple 32-bit digital PI controller with anti-windup correction. Functionally, it is similar to PI module described above and uses the same object described above, the difference change in the path of P control such that K_p can be set to zero unlike the previous module. Refer to the previous section for object definitions and technical reference below for implementation details



Availability C interface version

Module Properties **Type:** Target Independent

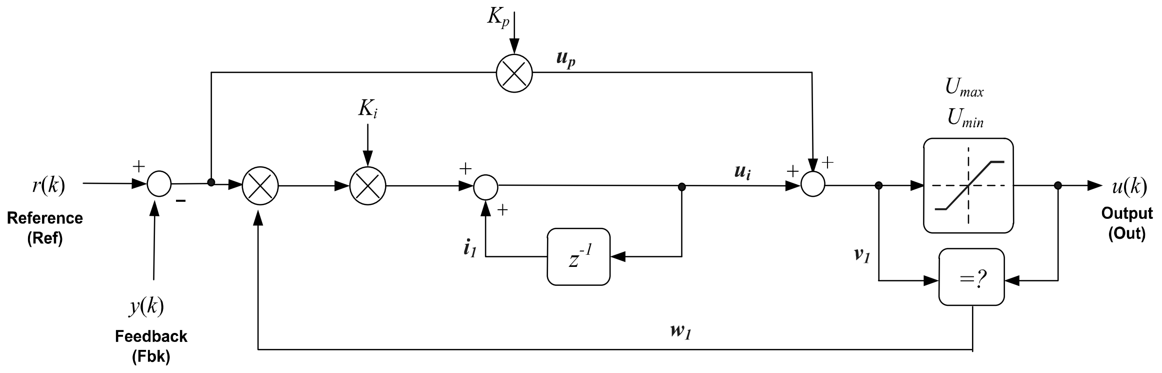
Target Devices: 28x Fixed or Floating Point

C Version File Names: pi_reg4.h

IQmath library files for C: IQmathLib.h, IQmath.lib

Technical Background

The PI_cntl module implementation resembles the previous one except that K_p is taken away from the forward path and is positioned in parallel to integral path as shown below.



a) Proportional path

The proportional path is a direct connection between the error term and a summing junction with the integral path. The error term is:

$$e(k) = r(k) - y(k) \quad \text{..........} \quad (2)$$

$$u_p(k) = K_p e(k) \quad \text{..........} \quad (2)$$

b) Integral path

The integral path consists of a discrete integrator which is pre-multiplied by a term derived from the output module. The term w1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integral term from “winding up” and improves the response time on recovery from saturation. The integrator law used is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + K_i[r(k) - y(k)] \quad \text{..........} \quad (3)$$

c) Output path

The output path contains a summing block to sum the proportional and integral controller terms. The result is then saturated according to user programmable upper and lower limits to give the controller output.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero or one result is produced which is used to disable the integral path (see above). The output path law is defined as follows.

$$v_1(k) = \lfloor u_p(k) + u_i(k) \rfloor \quad \textbf{(4)}$$

$$u(k) = \begin{cases} U_{\max} & : v_1(k) > U_{\max} \\ U_{\min} & : v_1(k) < U_{\min} \\ v_1(k) & : U_{\min} < v_1(k) < U_{\max} \end{cases} \quad \textbf{(5)}$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \quad \textbf{(6)}$$

Tuning the P+I controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable both integral and derivative paths. A suggested general technique for tuning the controller is now described.

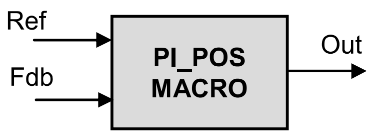
Step 1. Ensure integral is set to zero and proportional gain set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

PI_POS — *Position PI Controller*

Description This module implements a generic, simple 32-bit digital PI controller with anti-windup correction, exactly same as in the previous section on PI controller, except for the difference in error handling. Refer to the previous section for implementation details of PI controller, and technical literature below for error handling.



Availability C interface version

Module Properties **Type:** Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: pi.h

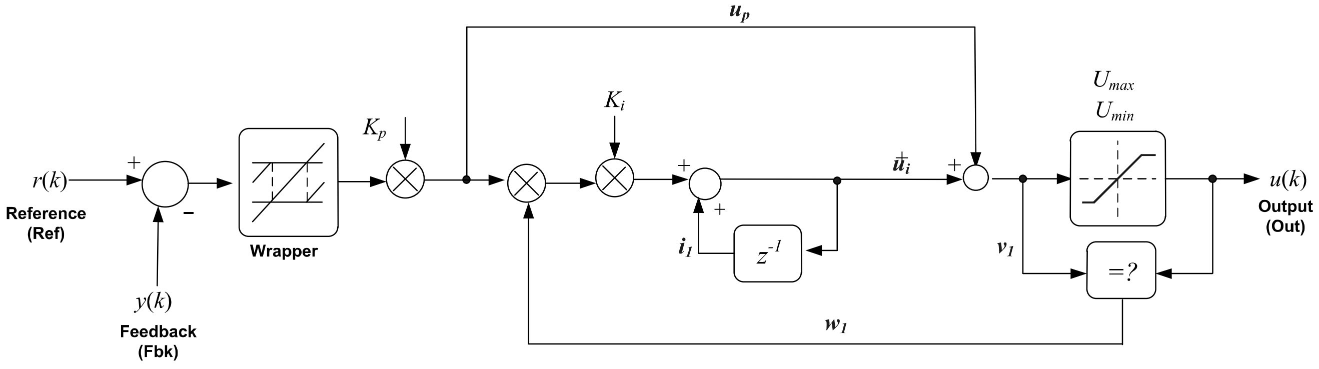
IQmath library files for C: IQmathLib.h, IQmath.lib

Technical Background

The PI_POS_cntl module implements a basic summing junction and P+I control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Anti-windup integrator reset
- Position error wrap around to fit within –π and +π

The PI controller is a sub-set of the PID controller. All input, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.

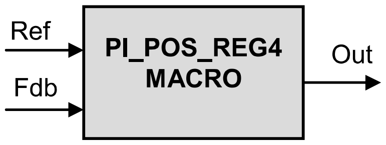


d) Position Error Wrapper

This block is the only difference between the original PI macro and the PI_POS macro. Since the reference and feedback for PI_POS macro are angles, it should be wrapped within - π and +π, otherwise, it will result in erratic behavior of the loop. Consider an error value of, say, 3π/2. This will pull the controller output in positive polarity. Actually this error should be treated as –π/2 which would have pulled it in negative polarity.

PI_POS_REG4 *Position PI Controller*

Description This module implements a generic, simple 32-bit digital PI controller with anti-windup correction, exactly same as in the previous section on PI_POS controller but treated as in PI_REG4. Refer to the previous section for implementation details of PI_POS controller, and technical literature below for block diagram.



Availability C interface version

Module Properties **Type:** Target Independent

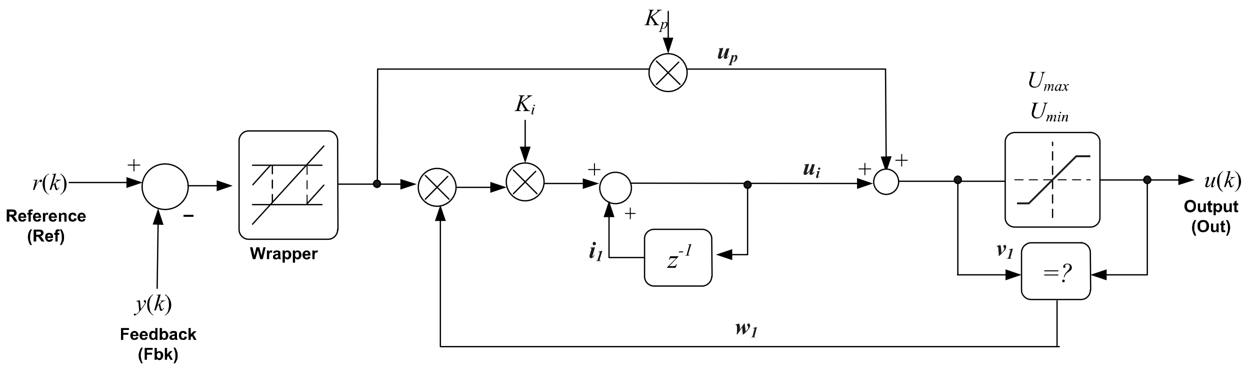
Target Devices: 28x Fixed or Floating Point

C Version File Names: pi_reg4.h

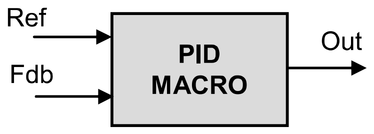
IQmath library files for C: IQmathLib.h, IQmath.lib

Technical Background

The block diagram of the internal controller structure is shown below. It is similar to the PI_POS module but for the K_p path which gives freedom to set K_p to zero.



Description This module implements a 32-bit digital PID controller with anti-windup correction.



Availability C interface version

Module Properties **Type:** Target Independent

Target Devices: 28x Fixed or Floating Point

C Version File Names: pid_grando.h

IQmath library files for C: IQmathLib.h, IQmath.lib

C Interface

Object Definition

The structure of PI object is defined by following structure definitions

```
typedef struct {
                        _iq Ref          // Input: reference set-point
                        _iq Fbk          // Input: feedback
                        _iq Out          // Output: controller output
                        _iq c1           // Internal: derivative filter coefficient
                        _iq c2           // Internal: derivative filter coefficient
                        _iq Iae          // Output: performance index
                        _iq Err          // Internal: servo error

                      } PID_TERMINALS;

typedef struct {
                          _iq  Kr;       // Parameter: reference set-point weighting
                          _iq  Kp;       // Parameter: proportional loop gain
                          _iq  Ki;       // Parameter: integral gain
                          _iq  Kd;       // Parameter: derivative gain
                          _iq  Km;       // Parameter: derivative weighting
                          _iq  Umax;     // Parameter: upper saturation limit
                          _iq  Umin;     // Parameter: lower saturation limit
                          _iq Kiae       // Parameter: IAE scaling
                        } PID_PARAMETERS;

typedef struct {
                          _iq  up;       // Data: proportional term
                          _iq  ui;       // Data: integral term
                          _iq  ud;       // Data: derivative term
                          _iq  v1;       // Data: pre-saturated controller output
                          _iq  i1;       // Data: integrator storage: ui(k-1)
                          _iq  d1;       // Data: differentiator storage: ud(k-1)
                          _iq  d2;       // Data: differentiator storage: d2(k-1)
                          _iq  w1;       // Data: saturation record: [u(k-1) - v(k-1)]
                        } PID_DATA;


typedef struct {
                        PID_TERMINALS        term;
                        PID_PARAMETERS       param;
                        PID_DATA             data;
                      } PID_CONTROLLER;
```

Special Constants and Data types

PID
The module definition is created as a data type. This makes it convenient to instance an interface to the PID module. To create multiple instances of the module simply declare variables of type PID.

PID_DEFAULTS
Structure symbolic constant to initialize PID module. This provides the initial values to the terminal variables as well as method pointers.

Module Usage

Instantiation
The following example instances PID object

```
PID pid1;
```

Initialization
To Instance pre-initialized objects

```
PID pid1 = { PID_TERM_DEFAULTS, PID_PARAM_DEFAULTS, PID_DATA_DEFAULTS };
```

Invoking the computation macro

```
PID_MACRO(pid1);
```

Example

The following pseudo code provides the information about the module usage.

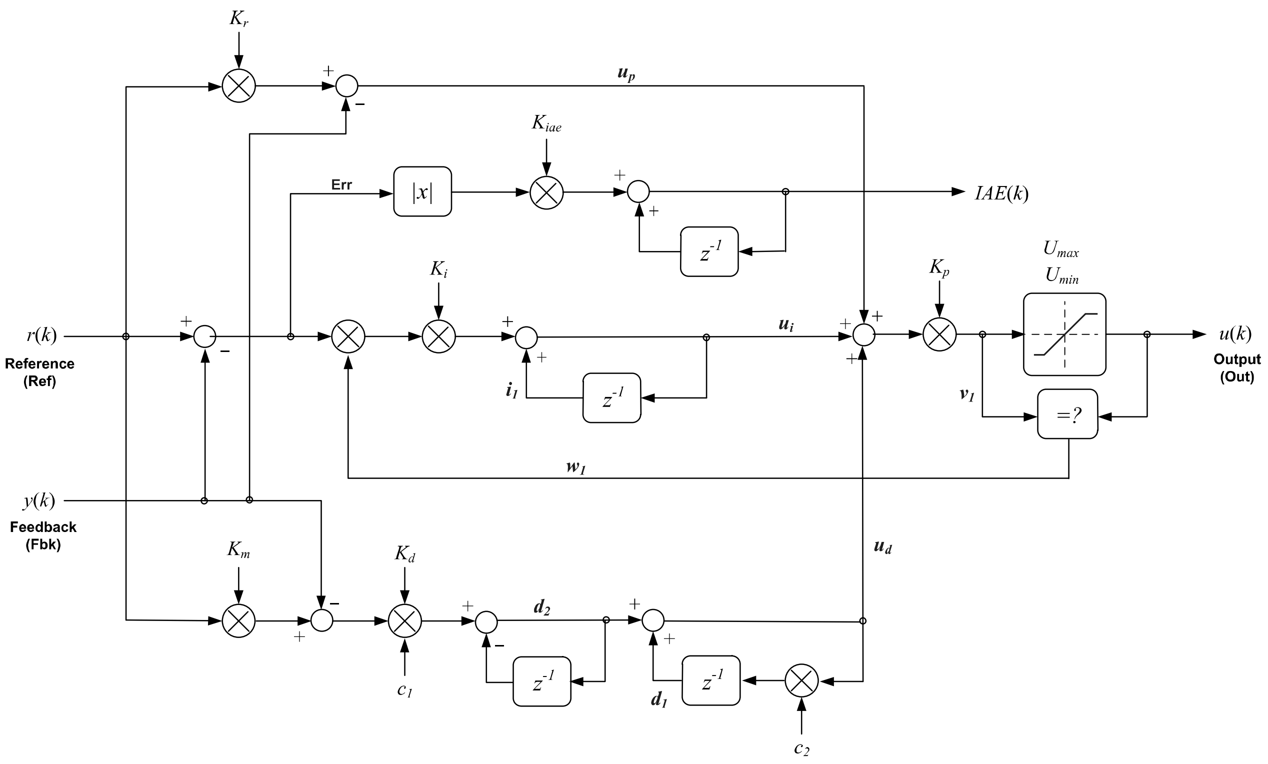
```
 /* Instance the PID module */

PID   pid1={ PID_TERM_DEFAULTS, PID_PARAM_DEFAULTS, PID_DATA_DEFAULTS };

main()
{

        pid1.param.Kp = _IQ(0.5);
        pid1.param.Ki  = _IQ(0.005);
        pid1.param.Kd = _IQ(0);
        pid1.param.Kr  = _IQ(1.0);
        pid1.param.Km =_IQ(1.0);
        pid1.param.Umax= _IQ(1.0);
        pid1.param.Umin= _IQ(-1.0);

}

void interrupt periodic_interrupt_isr()
{
        pid1.Ref = input1_1;            // Pass _iq inputs to pid1
        pid1.Fbk = input1_2;            // Pass _iq inputs to pid1

        PID_MACRO(pid1);                // Call compute macro for pid1

        output1 = pid1.Out;             // Access the output of pid1

}
```

Technical Background

The PID Grando module implements a basic summing junction and PID control law with the following features:

- Programmable output saturation
- Independent reference weighting on proportional path
- Independent reference weighting on derivative path
- Anti-windup integrator reset
- Programmable derivative filter
- Transient performance measurement

PID Grando is an example of a PID structure often called "standard" form, in which proportional gain is applied after the three controller paths have been summed. This contrasts with the "parallel" PID form, in which P, I, and D gains are applied in separate paths. All input, output and internal data is in I8Q24 fixed-point format. A block diagram of the internal controller structure is shown below.



Block diagram of the internal controller structure

a) Proportional path

The servo error is the difference between the reference input and the feedback input. Proportional gain is usually applied directly to servo error, however a feature of the Grando controller is that sensitivity of the proportional path to the reference input can be weighted differently to that of the feedback input. This provides an additional degree of freedom when tuning the controller. The proportional control law is:

$$u_p(k) = K_r r(k) - y(k) \quad \text{.......} \quad (3)$$

Note that "proportional" gain is applied to the sum of all three terms and will be described in section d).

b) Integral path

The integral path consists of a discrete integrator which is pre-multiplied by a scalar gain (K_i) and a term derived from the output saturation module. The term w_1 is either zero or one, and provides a means to disable the integrator path when output saturation occurs. This prevents the integrator from "winding up" and improves the recovery time following saturation in the control loop. The integrator law used in Grando is based on a backwards approximation.

$$u_i(k) = u_i(k-1) + w_1(k) K_i \left[r(k) - y(k) \right] \quad \text{.......} \quad (2)$$

c) Derivative path

The derivative term is a backwards approximation of the difference between the current and previous servo error. The input is the difference between the reference and feedback terms, and like the proportional term, the reference path can be weighted independently to provide an additional variable for tuning.

A first order digital filter is applied to the derivative term to reduce nose amplification at high frequencies. Filter cutoff frequency is determined by two coefficients (c_1 & c_2). The derivative law is shown below.

$$e(k) = K_m r(k) - y(k) \quad \text{.......} \quad (3)$$

$$u_d(k) = K_d \left[c_2 u_i(k-1) + c_1 e(k) - c_1 e(k-1) \right] \quad \text{.......} \quad (4)$$

Filter coefficients are based on the cut-off frequency (a) in Hz and sample period (T) in seconds as follows:

$$c_1 = a \quad \text{.......} \quad (5)$$

$$c_2 = 1 - c_1 T \quad \text{.......} \quad (6)$$

d) Output path

The output path contains a multiplying term (K_p) which acts on the sum of the three controller parts. The result is then saturated according to user programmable upper and lower limits to give the output term.

The pre-and post-saturated terms are compared to determine whether saturation has occurred, and if so, a zero value is assigned to the variable $w_1(k)$ which is used to disable the integral path (see above). The output path law is defined as follows.

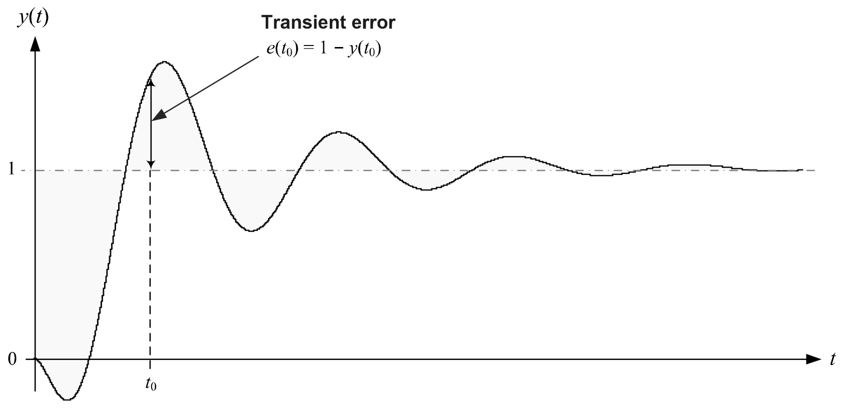
$$v_1(k) = K_p\left[u_p(k) + u_i(k) + u_d(k)\right] \quad (7)$$

$$u(k) = \begin{cases} U_{\max} : v_1(k) > U_{\max} \\ U_{\min} : v_1(k) < U_{\min} \\ v_1(k) : U_{\min} < v_1(k) < U_{\max} \end{cases} \quad (8)$$

$$w_1(k) = \begin{cases} 0 : v_1(k) \neq u(k) \\ 1 : v_1(k) = u(k) \end{cases} \quad (9)$$

e) Performance measurement

The Grando controller contains a single line of code which implements an IAE algorithm (Integral of Absolute Error). This integrates the absolute difference between the input reference and feedback (i.e. servo error), and can be used to tune transient performance by subjecting the loop to a step change of input reference and allowing the IAE term to integrate over a fixed time.



A small value of IAE indicates small absolute difference between the input reference and the measured output. Controller parameters may be adjusted iteratively to minimise measured IAE.

The IAE measurement is implemented as follows.

$$IAE(k) = IAE(k-1) + K_{iae}\left| r(k) - y(k) \right| \quad (10)$$

It is the task of the user code to reset and enable the IAE measurement (see below). The feature may be enabled and disabled by setting the K_{iae} term to one or zero respectively. The IAE output may be reset by setting the Iae term to zero. The K_{iae} term may be set to a value smaller than one if integrator overflow is an issue. The user may comment out the last line of the PID Grando macro to reduce cycle count if IAE is not required.

Tuning the controller

Default values for the controller coefficients are defined in the macro header file which apply unity gain through the proportional path, and disable IAE, integral and derivative paths. A suggested general technique for tuning the controller is now described.

Steps 1-4 are based on tuning a transient produced either by a step change in either load or reference set-point.

Step 1. Ensure integral and derivative gains are set to zero. Ensure also the reference weighting coefficients (K_r & K_m) are set to one.

Step 2. Gradually adjust proportional gain variable (K_p) while observing the step response to achieve optimum rise time and overshoot compromise.

Step 3. If necessary, gradually increase integral gain (K_i) to optimize the return of the steady state output to nominal. The controller will be very sensitive to this term and may become unstable so be sure to start with a very small number. Integral gain will result in an increase in overshoot and oscillation, so it may be necessary to slightly decrease the K_p term again to find the best balance. Note that if the integral gain is used then set to zero, a small residual term may persist in u_i.

Step 4. If the transient response exhibits excessive oscillation, this can sometimes be reduced by applying a small amount of derivative gain. To do this, first ensure the coefficients c_1 & c_2 are set to one and zero respectively. Next, slowly add a small amount of derivative gain (K_d).

Steps 5 & 6 only apply in the case of tuning a transient set-point. In the regulator case, or where the set-point is fixed and tuning is conducted against changing load conditions, they are not useful.

Step 5. Overshoot and oscillation following a set-point transient can sometimes be improved by lowering the reference weighting in the proportional path. To do this, gradually reduce the K_r term from its nominal unity value to optimize the transient. Note that this will change the loop sensitivity to the input reference, so the steady state condition will change unless integral gain is used.

Step 6. If derivative gain has been applied, transient response can often be improved by changing the reference weighting, in the same way as step 6 except that in the derivative case steady state should not be affected. Slowly reduce the K_m variable from it's nominal unity value to optimize overshoot and oscillation. Note that in many cases optimal performance is achieved with a reference weight of zero in the derivative path, meaning that the differential term acts on purely the feedback, with no contribution from the input reference. This can help to avoid derivative "kick" which occurs following a sudden change in input reference.

The derivative path introduces a term which has a frequency dependent gain. At higher frequencies, this can cause noise amplification in the loop which may degrade servo performance. If this is the case, it is possible to filter the derivative term using a first order digital filter in the derivative path. Steps 7 & 8 describe the derivative filter.

Step 7. Select a filter roll-off frequency in radians/second. Use this in conjunction with the system sample period (T) to calculate the filter coefficients c_1 & c_2 (see equations 5 & 6).

Step 8. Note that the c_1 coefficient will change the derivative path gain, so adjust the value of K_d to compensate for the filter gain. Repeat steps 5 & 6 to optimize derivative path gain.

Tuning with IAE

The IAE measurement feature may be useful in determining optimal controller settings based on transient response. The user code should reset and enable IAE just before applying a transient change as follows:

```
// reset and enable IAE
pid1.term.Iae = _IQ(0.0);
pid1.Param.Kiae = _IQ(1.0);
```

After a fixed time, the IAE integrator should be disabled and the measurement read as follows:

```
// disable IAE integrator
pid1.param.Kiae = _IQ(0.0);
IAE = pid1.term.Iae;
```

The user should ensure that the integration period is sufficiently long to allow transient effects to dissipate and should remain fixed between iterative tests for comparisons to be valid. Controller settings may be adjusted using steps 1 to 8 above to minimise measured IAE.

Saturation & Integrator Wind-up

The Grando controller includes a saturation block to limit the range of the control effort, $u(k)$. If the output saturates, the integrator is disabled to prevent a phenomenon known as "wind-up". In cases where saturation may occur in other parts of the control loop, user code should disable integral action by temporarily setting the integrator gain (K_i) to zero when saturation occurs, and restoring it once saturation has been cleared.