An Interactive PID Learning Module for Educational Purposes [⋆]

Alfred Theorin [*] **Charlotta Johnsson** [†]

[*] *Department of Automatic Control, Lund University, Lund, Sweden (e-mail: alfred.theorin@control.lth.se)*
[†] *Department of Automatic Control, Lund University, Lund, Sweden (e-mail: charlotta.johnsson@control.lth.se)*

Abstract: The PID controller is the most common controller and it is taught in most introductory automatic control courses. To develop an intuitive understanding of the basic concepts of the PID controller and PID parameter tuning, interactive and freely available tools are important. A PID module for educational purposes has been implemented in JGrafchart, a freely available development environment for the graphical programming language Grafchart. JGrafchart includes interactive graphical elements such as live plots and it is possibile to create animated graphics, for example of a simulated process. JGrafchart's variables, for example controller parameters and modes, can be changed interactively while executing. The PID module will be included in future releases of JGrafchart with sample applications which can be used for example to demonstrate a PID controller live in lectures or to let students interactively change controller parameters and modes to develop an intuitive understanding of the PID controller and PID parameter tuning. The sample applications are designed for users without any knowledge about JGrafchart and can be used to control both simulated and physical processes.

Keywords: Control education, Education, Grafchart, PID control, PID controllers, Teaching

1. INTRODUCTION

An important concept in automatic control is the concept of the PID controller. The PID controller is by far the most commonly used controller in industry. There are billions of control loops (Soltesz (2012)) and the PID controller is used for more than 95% of all control loops (Åström and Murray (2012)). The PID controller concept is also taught in most introductory automatic control courses and given its wide use it is important that the students obtain a deep understanding for PID control. The availability of interactive learning tools in these courses is important since it facilitates the possibility for students to develop an intuitive feel and understanding. Interactive tools provide students with an opportunity to experiment with PID controllers and give them hands on experience of how the PID controller works.

Examples of interactive learning tools in the field of automatic control are ICtools (Johansson et al. (1998)) and Interactive Learning Modules for PID (Guzman et al. (2008)). ICtools is an interactive MATLAB based tool for learning the basics of automatic control whereas Interactive Learning Modules for PID is a learning tool which covers many aspects of PID controller design and tuning in an intuitive and highly interactive way.

Students should also be familiarized with PID controllers in a realistic setting, that is, controlling a physical process. If a physical process is not available, an animated simulation of the process can be used instead. The idea is to let the students control the process, and by doing so find out the basic ideas of automatic control and get an intuitive feel for how the PID controller and its parameters work.

A PID module for educational purposes is presented in this paper. It has been implemented in JGrafchart, an integrated development environment for the graphical programming language Grafchart. JGrafchart includes interactive graphical elements such as live plots and the possibility to create animated graphics, for example of a simulated or physical process (Theorin (2013)). JGrafchart's variables, for example the controller parameters and modes, can also be changed interactively while executing. The learning module is designed for users without any knowledge about JGrafchart.

Unlike MATLAB, JGrafchart is free and based on an industrial control language. Unlike many other learning tools, JGrafchart can be used in industry-like environments as it can be connected to physical processes.

In this paper, the importance of interactive learning environments is discussed in Section 2. The PID controller itself and its implementation aspects are discussed in Section 3. Grafchart and JGrafchart are described in Section 4 and the PID module and information about usage and setup is presented in Section 5. The educational aspects with an interactive example and possible uses of the interactive tool in an introductory course are discussed in Section 6. Finally, in Section 7 conclusions are drawn and future work is discussed.

[⋆] Financial support from VINNOVA and the Swedish Research Council through the LCCC Linnaeus Center grant, the ELLIIT Excellence Center, and the LISA project are gratefully acknowledged.

2. LEARNING ENVIRONMENTS

To assure good learning situations and learning environments it is important to understand how knowledge is constructed in the mind.

Basically, there are two types of students:

A. The group that constructs new knowledge mainly from concrete experience.
B. The group that constructs new knowledge mainly by formation of abstract concepts and generalizations.

Students often prefer either experiments (group A) or abstract concepts (group B) as starting point for constructing new knowledge. Since this is individual, both starting points should be provided in a learning situation.

The model Kolb Learning Cycle (Kolb (1984)), see Fig. 1, illustrates how an experience (left) results in reflection on observations (top), which results in new abstract thinking (right), which might cause conceptual changes in a student's mind (bottom), which might cause a need for obtaining more concrete experiences (left), and so on. The preferred starting point in the Kolb Learning Cycle is individual and depends on the student's preferred way of learning (compare to group A/B above).

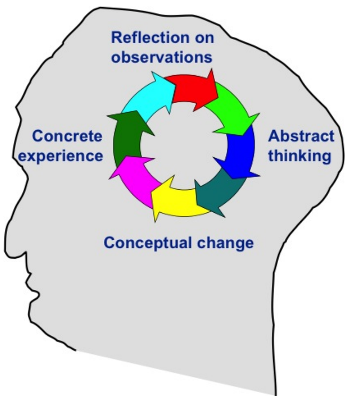


Fig. 1. The Kolb Learning Cycle.

The availability of interactive PID learning tools make it possible for students taking classes in automatic control to obtain concrete experiences on a PID controller.

3. PID CONTROLLERS

A controller in a feedback loop can be used to control a process (physical or simulated) to give it desired behavior, see Fig. 2. More than 95% of all control loops are PID controllers, or rather PI controllers as the derivative part is rarely used (Åström and Murray (2012)).

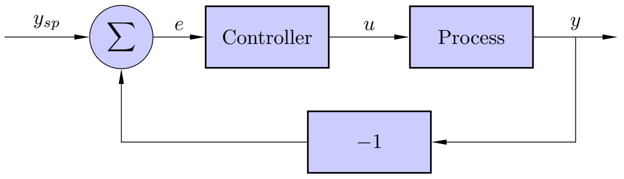


Fig. 2. A feedback loop where a controller is used to control the process by considering the control error.

3.1 Basic PID Controller

The input to the PID controller is the control error, $e(t)$, which is the difference between the desired (reference/setpoint) process state, $y_{sp}(t)$, and the measured process state, $y(t)$:

$$e(t) = y_{sp}(t) - y(t) \tag{1}$$

The output of the PID controller is the manipulated variable (control signal), $u(t)$. A basic PID controller in continuous time is described by

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{2}$$

where the PID controller parameters are the controller gain K, the integral time T_i, and the derivative time T_d (Åström and Hägglund (2006), Glad and Ljung (2006)). These parameters are used to tune the closed loop behavior.

3.2 PID Implementation

In (2) many details are ignored such as:

- The derivative for time t is unknown at time t.
- Physical control signal limits are not considered which will cause integrator windup.
- The controller parameters are assumed to be constant but the operator may change them at any time.

PID controllers are practically always implemented digitally on computers and thus (2) must be discretized. There are best practices for how to tackle these, and most other, PID implementation related issues.

For a PID learning tool to be complete it should include all common PID features:

- Anti-windup
- Auto/Manual mode
- Bumpless mode change (automatic/manual)
- Bumpless parameter change
- Feedforward
- Maximum derivative gain
- Setpoint weighting
- Tracking

3.3 PID Algorithms

Most PID controllers are implemented on positional form which means that the control signal is calculated directly. Each sample a new control signal is calculated based on the new and/or previous process and setpoint values as well as on the internal state of the PID controller. A drawback with positional form is that the internal states, primarily the integrator state, are sensitive to mode and parameter changes, and need many special cases to get proper behavior.

An alternative to positional form is velocity form which instead calculates increments to the control signal each sample. An advantage with velocity form is that it has fewer internal states. Most notably the integral term is not an internal state of the controller, it is instead included in the control signal itself, which means that several controller features like anti-windup, tracking, and bumpless mode changes practically come for free. On the other hand, without integral action a pure velocity form implementation has arbitrary stationary error. All in all, the velocity form implementation does not require as many special cases which makes it more compact and easier to implement correctly.

A discretized PID algorithm on velocity form (Åström and Hägglund (2006)) is described by

$$
\begin{aligned}
\Delta u(t_k) &= u(t_k) - u(t_{k-1}) = && (3)\\
&= \Delta P(t_k) + \Delta I(t_k) + \Delta D(t_k) + \Delta u_{ff}(t_k) && (4)\\
\Delta P(t_k) &= P(t_k) - P(t_{k-1}) = \\
&= K(by_{sp}(t_k) - y(t_k)) - K(by_{sp}(t_{k-1}) - y(t_{k-1})) && (5)\\
\Delta I(t_k) &= I(t_k) - I(t_{k-1}) = b_{i1}e(t_k) + b_{i2}e(t_{k-1}) && (6)\\
\Delta D(t_k) &= D(t_k) - D(t_{k-1}) = \\
&= a_d\Delta D(t_{k-1}) - b_d(y(t_k) - 2y(t_{k-1}) + y(t_{k-2})) && (7)\\
\Delta u_{ff}(t_k) &= u_{ff}(t_k) - u_{ff}(t_{k-1}) && (8)
\end{aligned}
$$

where y is the process value, y_{sp} is the reference value, Δu is the control signal increment, and ΔP, ΔI, and ΔD are the increments for the P, I, and D part respectively. K is the controller gain and b is the scalar setpoint weight for the P-part. b_{i1} and b_{i2} are constants which depend on how the integral term is discretized. For no particular reason, backward difference was selected which gives $b_{i1} = \frac{Kh}{Ti}$ and $b_{i2} = 0$ where T_i is the integral time and h is the PID controller sample time. Similarly a_d and b_d are constants which depend on how the derivative term is discretized. Most derivative part discretizations except for backward difference either have issues with overflow or instability. Thus, backward difference was selected which gives $a_d = \frac{T_d}{T_d+Nh}$ and $b_d = \frac{KT_dN}{T_d+Nh}$ where T_d is the derivative time and N is the maximum derivative gain. Finally, u_{ff} is the feedforward signal.

Without integral action the velocity form has arbitrary stationary error. The reason is that the P part is assumed to be included in the control signal and only needs to be adjusted when the measurement value changes. This assumption does not hold for example when tracking or when the control signal saturates. To avoid this issue, the P-part is then instead

$$
\begin{aligned}
\Delta P(t_k) &= P(t_k) + u_b(t_k) - u(t_{k-1}) = \\
&= K(y_{sp}(t_k) - y(t_k)) + u_b(t_k) - u(t_{k-1}) && (9)
\end{aligned}
$$

where u_b is the bias term which will be set to u_{ff} to support feedforward for controllers without integral action. Setpoint weighting is also removed for this case.

The main reason to choose velocity form is that it requires fewer special cases which means less code, specifically considerably less special code which is rarely executed and thus more likely to contain errors.

4. GRAFCHART

Grafchart is a graphical programming language based on Sequential Function Charts (SFC), one of the IEC 61131-3 (IEC (1993)) PLC standard languages which is used to implement sequential, parallel, and general state-transition oriented applications. Grafchart uses the same graphical syntax with steps and transitions where steps represent the possible application states and transitions represent the change of state (Theorin (2013)). Associated with the steps are actions which specify what to do. Associated with each transition is a Boolean guard condition which specifies when the application state may change.

A part of a running Grafchart application is shown in Fig. 3. Here two steps are connected by a transition and there are two variables, namely `var` and `cond`. In the left part of the figure, the upper step has just been activated which involves executing its `S` action, thus setting `var` to 7. That the step is active is indicated by a black dot, known as a token. The upper step will remain active until the guard condition of the transition becomes true, that is, until `cond` gets the value 4. When the guard condition becomes true, shown in the right part of the figure, the upper step is deactivated and the lower step is activated which means that `var` is set to 12.

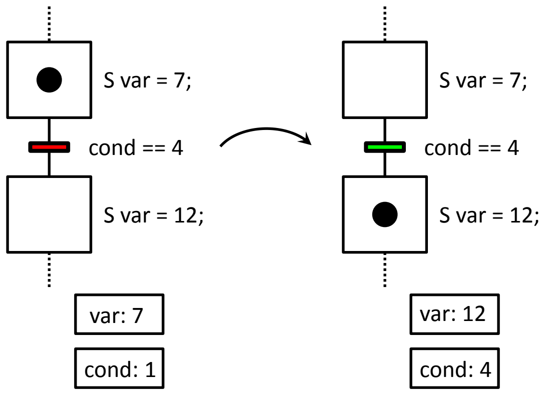


Fig. 3. A piece of a running Grafchart application. The left part shows one application state and the right part shows a later application state.

Steps also have additional properties, namely `x`, `t`, and `s`. `x` is true if the step is active and false if the step is inactive. `t` is how many scan cycles the step has been active since the previous activation if the step is active. For inactive steps `t` is 0. `s` works the same as `t` but counts seconds instead of scan cycles.

Grafchart supports basic SFC functionality such as several action types and alternative and parallel paths. Also, additional constructs such as hierarchical structuring, reusable procedures, and exception handling have been added which makes it convenient to implement large applications that are maintainable and possible to overview (Theorin (2013)).

With reusable components, code duplication is avoided. Reusable code can be put in a Grafchart Procedure which can then be called Procedure Steps and Process Steps, see Fig. 4. The difference is that Procedure Steps wait for the call to complete before the application can proceed while Process Steps do not.

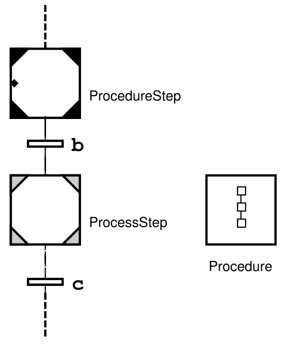


Fig. 4. A Procedure can be called from Procedure Steps and Process Steps. Each Procedure Step and Process Step specify which Procedure to call when activated.

4.1 Execution Model

Grafchart applications are, like SFC, executed periodically, one scan cycle at a time. Since Grafchart is a state oriented language and not a data flow language, ensuring the desired execution order requires knowledge about its execution model. A transition is *enabled* when all immediately preceding steps are active. An *enabled* transition is *fireable* if its condition is true. *Firing* a transition involves deactivating the immediately preceding steps and activating the immediately succeeding steps. The execution model of a scan cycle is described by the following sequence:

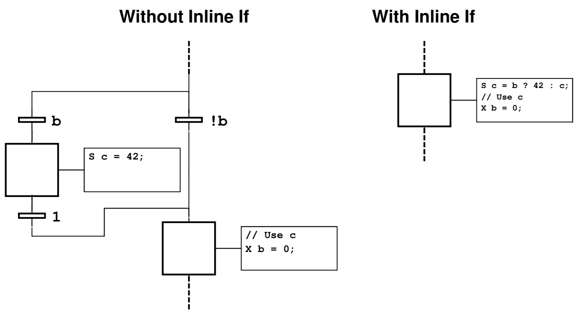
1. Read inputs.
2. Mark fireable transitions.
3. Remove mark for conflicting transitions of lower priority.
4. Fire marked transitions.
5. Update step properties `t` and `s`.
6. Execute `P` actions.
7. Mark variables subject to `N` actions.
8. Update marked variables.
9. Sleep until the start of the next scan cycle.

The execution model gives sufficiently deterministic behavior and has the property that a step which is activated always remains active for at least one scan cycle. The remaining non-determinism is for cases where the application should not depend on the execution order anyway, for example the firing order of transitions affects which step's `S` and `X` actions are executed first. Another example is which step's `P` actions are executed first.

4.2 JGrafchart

JGrafchart is a Java based integrated development environment for Grafchart with some extensions (Theorin (2013)). Most interesting for this paper are *inline if* and graphical elements.

Inline if (`?:`) is also known as the conditional operator and ternary if. It is used for conditional expression evaluation and can make applications easier by reducing the number of steps. The syntax is `<cond> ? <trueExpr> : <falseExpr>` which evaluates to `<trueExpr>` if the condition `<cond>` is true and to `<falseExpr>` otherwise. Fig. 5 shows a small JGrafchart application written without and with *inline if*. The implementations behave slightly different: The left part executes the initialization in one scan cycle and the rest in the next scan cycle. The right part executes everything in the same scan cycle and is more compact and less complicated.



```
S c = 42;
```

```
// Use c
X b = 0;
```

```
S c = b ? 42 : c;
// Use c
X b = 0;
```

Fig. 5. The variable `c` needs to be initialized to 42 if `b` is true. The left part shows how to implement this without *inline if*. The right part shows how it can be implemented with *inline if*.

JGrafchart also includes a multitude of graphical elements such as rectangles, ellipses, lines, arrows, text fields, images, lists, and plots. The elements can be modified from actions in the Grafchart application which makes it possible to create animations, visualizations, or operator interfaces for physical or simulated processes. There are also buttons with associated *on click* actions. Another useful feature is that variables are interactive. They are shown graphically with their current value and can also be changed while executing. Here this is particularly useful for controller parameters and modes.

Finally, JGrafchart can also be connected to external environments through a multitude of customizable input/output (I/O) integration capabilities and can thus also be used to control physical processes.

5. IMPLEMENTATION

5.1 `PID` Procedure

A full-fledged PID module has been implemented as a reusable Grafchart Procedure which means that it can be used for any number of control loops by adding a Procedure Step or Process Step for each control loop. The Procedure is called `PID` and is shown in Fig. 6. The Procedure parameters are explained in Table 1.

A Procedure executes at the same rate as the application it is called from. The PID sample time is thus limited to a multiple of the application's scan cycle time. The PID algorithm is implemented in a single step and can thus execute at the same rate as the caller.

In Procedure calls, each Procedure parameters can be set as either *call-by-reference* (`R`), *call-by-value* (`V`), or *default* (omitted). For *call-by-value* and *default* the parameter is set once when the call is made. For *call-by-reference* the parameter will be a reference to a variable or I/O in the calling context. To be useful, `PV`, `SP`, `TR`, and `MV` should be *call-by-reference* and the caller should set and update all parameters except `MV` which is the `PID` Procedure's sole output.

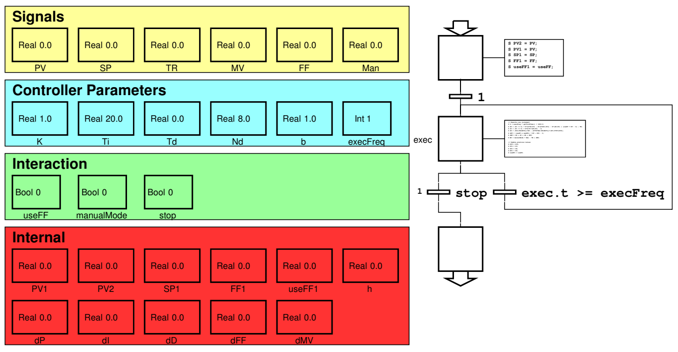
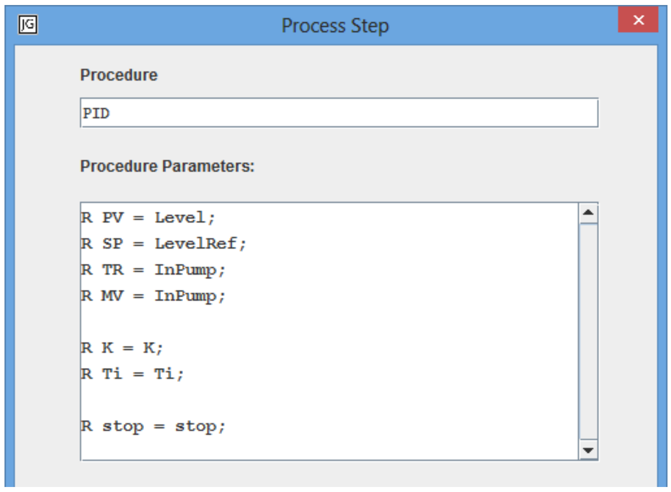


Fig. 6. The new PID Procedure. The parameter values here are the default values. See Table 1 for more detailed parameter descriptions. The step actions are not intended to be readable in this figure.

Table 1. Parameter descriptions for the parameters of the PID Procedure in Fig. 6.

Parameter	Description
PV	Process Value, the measurement value for the process.
SP	SetPoint, the reference value.
TR	TRacking signal, the actually actuated control signal in the previous scan cycle.
MV	Manipulated Variable, the control signal.
FF	FeedForward, an external signal added to the control signal.
Man	Manual, the control signal when in manual mode.
K	Controller gain.
Ti	Integral time.
Td	Derivative time.
Nd	Maximum derivative gain, usually called N but that is a reserved word in JGrafchart.
b	Setpoint weight for the P-part.
execFreq	Execution Frequency, the PID controller sample time is `execFreq` times the calling Grafchart application's scan cycle time.
useFF	Signal to turn feedforward on/off.
manualMode	Signal to turn manual mode on/off.
stop	Signal to terminate the Procedure call and thus stop executing the PID controller.



PID

Procedure Parameters:

```
R PV = Level;
R SP = LevelRef;
R TR = InPump;
R MV = InPump;

R K = K;
R Ti = Ti;

R stop = stop;
```

Fig. 7. A Process Step calling the PID Procedure.

A call to the `PID` Procedure is shown in Fig. 7. Here the parameters for feedforward, manual mode, setpoint weighting, and the D-part are omitted and will thus get their default values, which means that these features are not used. Call-by-reference is used for all parameters, for example the parameter `PV` in the call will be a reference to the variable `Level` in the calling context. Changing the value of the variable also updates the parameter and vice versa.

5.2 *Code*

The code for the main step (`exec`) of Fig. 6 is shown in Fig. 8. It is a straightforward implementation of the discrete equations in Section 3.

```
// Execute one increment
S h = execFreq * getTickTime() / 1000.0;
S dP = (Ti != 0) ?
          K*(b*SP-PV) - K*(b*SP1-PV1) :
          K*(SP-PV) + (useFF ? FF : 0) - TR;
S dI = (Ti != 0) ? K*h/Ti*(SP-PV) : 0;
S dD = (Td/(Td+Nd*h))*dD -
          (K*Td*Nd/(Td+Nd*h))*(PV-2*PV1+PV2);
S dFF = (useFF & useFF1) ? FF - FF1 : 0;
S dMV = dP + dI + dD + dFF;
S MV = manualMode ? Man : TR + dMV;

// Update previous values
S PV2 = PV1;
S PV1 = PV;
S SP1 = SP;
S FF1 = FF;
S useFF1 = useFF;
```

Fig. 8. The main step code of the `PID` Procedure in Fig. 6.

First the current PID controller sample time is calculated. Then the P, I, D, and feedforward increments are calculated. `dP` and `dI` have special handling if the integrator is turned off as discussed in Section 3. The feedforward calculation has special handling to avoid bumps when feedforward is enabled at the same time as the feedforward value is changed. After this the whole control signal increment `dMV` is calculated and then the control signal is selected depending on the current mode. Note that `TR` is typically the limited `MV` from the previous scan cycle and the caller is responsible to update `TR`. Finally, previous values are stored for the next sample in internal variables suffixed with how many scan cycles old they are, for example PV1 is PV from the previous scan cycle.

5.3 *Simulation Setup*

A tricky part of setting up a simulated process that uses the `PID` Procedure is to ensure proper execution order:

1. Execute the PID controller
2. Limit the control signal
3. Update the simulated process

The `PID` Procedure uses `S` actions to make it execute as early as possible (step 4 in the execution model). Hence `P` actions can be used for the simulated process (step 6 in the execution model). To ensure that the limited control signal is used to update the simulated process, the limiting should be a preceding `P` action in the same step.

6. EDUCATIONAL ASPECTS

For educational tools it is paramount that they are easily available and easy to run for everyone. The interactive PID module presented in this paper only requires JGrafchart, a freely available software. JGrafchart is written in Java and is platform independent. It has been verified to run on Windows, Linux, and Mac. The interactive PID module will be made readily available by including sample applications with the JGrafchart installation as well as having the `PID` Procedure as a JGrafchart library component.

6.1 Evaluation

The implementation was evaluated on a simulated tank process similar to the upper tank of the tank process used in the introductory automatic control course at Lund University, Sweden. The process consists of a water tank and the objective is to control the tank's water level. The inflow to the tank is controlled with a pump and there is an outflow through a hole in the bottom of the tank.

Fig. 9 shows the application in action. There is a live animation showing the current state of the (simulated) process and both setpoint and basic control parameters can be changed while executing. The figure shows a suitable setup for beginners. More of the `PID` Procedure features can be included in subsequent exercises.

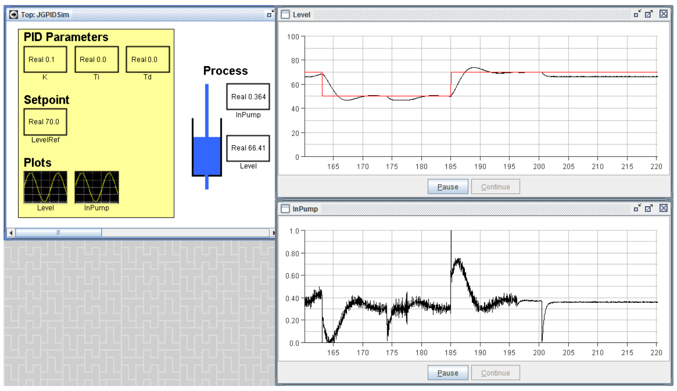


Fig. 9. In the left part, the setpoint and controller parameters can be changed and an animation of the simulated tank is shown. The upper plot shows the measurement value (black) and the setpoint (red). The lower plot shows the control signal. In this screenshot, the setpoint and controller parameters were recently changed several times.

6.2 Possible Uses

It is possible to create and use simulated processes for hands on or laboratory exercises. With JGrafchart's capability to connect to external environments it is also possible to use the same implementation for controlling the actual process. One possibility is to control a physical process in a laboratory exercise and provide a simulated process for preparatory or followup work.

Another possible use is live demonstrations in lectures and there are certainly many other possibilities.

7. CONCLUSIONS

The PID controller is the most common controller and is taught in most introductory automatic control courses. To aid learning of new concepts interactive learning tools play an important role. The interactive PID module presented in this paper aims to aid obtaining both an understanding of the PID controller concept and an intuitive feel for PID parameter tuning in an industry-like setting. Unlike many other learning tools JGrafchart is free, based on an industrial control language, and can be used in industry-like environments. The module was evaluated on a simulated process and works very well.

The PID module and samples will be included with future releases of JGrafchart. The samples are designed for users without any knowledge about JGrafchart. Download link: `http://www.control.lth.se/Research/tools/grafchart.html`

7.1 Future Work

The next step is to evaluate the tool further by including it in introductory course education.

Issues with the PID algorithm have been noticed by running it in JGrafchart, specifically issues with the D-part. Further reaserch on the PID algorithm is thus desired.

There are several possible extensions for the current `PID` Procedure. For example it would be useful to include an optional process value filter or more advanced features, such as an auto-tuner.

REFERENCES

Åström, K. and Hägglund, T. (2006). *Advanced PID Control*. ISA-The Instrumentation, Systems, and Automation Society.

Åström, K. and Murray, R. (2012). *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton and Oxford. URL `http://www.cds.caltech.edu/~murray/amwiki`.

Glad, T. and Ljung, L. (2006). *Reglerteknik: grundläggande teori*. Studentlitteratur.

Guzman, J.L., Åstrom, K., Dormido, S., Hägglund, T., Berenguel, M., and Piguet, Y. (2008). Interactive learning modules for PID control [lecture notes]. *Control Systems, IEEE*, 28(5), 118–134.

IEC (1993). IEC 61131-3: Programmable controllers – part 3: Programming languages. Technical report, International Electrotechnical Commission.

Johansson, M., Åström, K.J., and Gäfvert, M. (1998). Interactive tools for education in automatic control. *IEEE Control Systems*, 18(3), 33–40.

Kolb, D. (1984). *Experiential learning: experience as the source of learning and development*. Prentice Hall, Englewood Cliffs, NJ.

Soltesz, K. (2012). On automation of the PID tuning procedure. Licentiate Thesis ISRN LUTFD2/TFRT--3254--SE, Department of Automatic Control, Lund University, Sweden.

Theorin, A. (2013). Adapting Grafchart for industrial automation. Licentiate Thesis ISRN LUTFD2/TFRT--3260--SE, Department of Automatic Control, Lund University, Sweden.