Implementation of an Asymmetric Relay Autotuner in a Sequential Control Language

Theorin, Alfred; Berner, Josefin

Published in:
2015 IEEE International Conference on Automation Science and Engineering

DOI:
10.1109/CoASE.2015.7294191

2015

Document Version:
Peer reviewed version (aka post-print)

Implementation of an Asymmetric Relay Autotuner in a Sequential Control Language*

Alfred Theorin and Josefin Berner[1]

Abstract— **Control applications contain both logic, sequencing, and control algorithms. A holistic view of this is seldom presented in teaching and papers. One reason is the separation of communities – automation groups typically come from the Programmable Logic Controller (PLC) world while control engineers primarily come from the Distributed Control System (DCS) world. Both logic, sequencing, and control algorithms are, however, well integrated in today's control systems since DCS systems now contain logic and sequencing and PLCs now contain control algorithms. This paper shows that both logic, sequencing, and control algorithms can be integrated seamlessly in a sequential control language. The particular application considered is a PID controller with a relay autotuner. The presented autotuner implementation yields good controller parameters and is freely available.**

I. Introduction

The most common controller in the process and automation industry is the PID controller. Even though the PID controller is simple, many controllers operating in industry today are performing unsatisfactory due to bad tuning of the controller parameters. An automatic method of finding good controller parameters is therefore desirable. The relay autotuner [1] has been widely used since its introduction in the 1980s and it is included in many industrial control systems, such as DeltaV [2] by Emerson and 800xA [3] by ABB. The idea is to estimate the process dynamics from a relay oscillation experiment and to use this to calculate PID controller parameters using simple tuning rules. The relay autotuner's main advantages are that it does not need any a priori information, it keeps the experiment time short, and it does not disturb the process much. It also provides information in the interesting frequency interval close to the critical frequency, that is, where the phase is $-180°$.

Many researchers have contributed with developments and improvements of the original autotuner, see [4] for a thorough review. In recent research [5], [6], an asymmetric relay experiment is proposed, that finds and uses information about the normalized time delay of the process in the autotuning procedure. The asymmetry of the relay gives better process excitation [5], which makes it possible to also estimate the static gain of the process. Thus, low-order models can be obtained, instead of just a single point on the Nyquist curve, as is the case for the original relay autotuner.

*The authors are members of the LCCC Linnaeus Center and the ELLIIT Excellence center at Lund University. Financial support from VINNOVA and the Swedish Research Council through the LCCC Linnaeus Center grant and the ELLIIT Excellence Center are gratefully acknowledged.

[1]Both authors are with Department of Automatic Control, Lund University, Lund, Sweden `firstname.lastname@control.lth.se`

The relay experiment is primarily sequential, and well suited for implementation in a language especially developed for sequential control. Sequential Function Charts (SFC) is one of the IEC 61131-3 [7] PLC standard languages, which is widely used in industry to implement the sequencing parts of the control applications. Even though the IEC 61131-3 languages were originally developed for PLC systems, they are also supported by DCS systems, such as 800xA by ABB and SIMATIC PCS 7 [8] by Siemens.

A PID controller with a relay autotuner is a combination of logic, sequencing, and control algorithms. In this paper, an asymmetric relay autotuner is implemented in Grafchart [9], a language based on SFC. Grafchart is introduced in Section II. The asymmetric relay autotuner and how to estimate low-order model parameters from the experiment are described in Section III. The Grafchart implementation of the autotuner is described in Section IV, and a brief evaluation is presented in Section V.

II. Grafchart

Grafchart is a sequential control language that uses the same graphical syntax as SFC. The basic building blocks are steps and transitions. Steps represent the possible application states and transitions represent the possible changes of state [10]. Each transition has a Boolean guard condition, which specifies when the application state may change, that is, the condition for when to activate and deactivate steps. Each step may have actions which specify what to do, for example, when the step is activated or while the step is active.

A part of a running Grafchart application is shown in Fig. 1. Here, two steps are connected by a transition and there are two variables, `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. A token, shown as a black dot, indicates that the step is active. The upper step remains active until the guard condition of the transition becomes true, that is, until `cond` becomes 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.

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

Grafchart supports basic SFC features such as alternative and parallel paths. Additional constructs for hierarchical

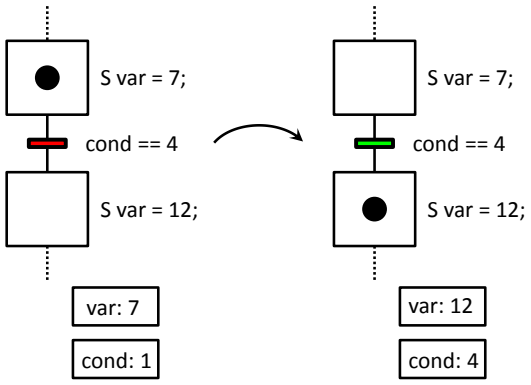

Fig. 1. A piece of a running Grafchart application. The left part shows one application state and the right part shows a later application state.

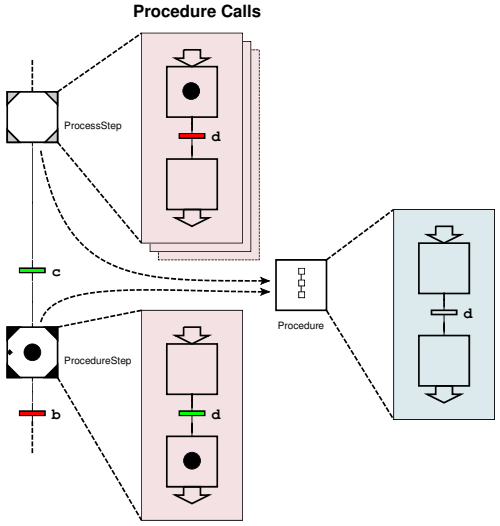

Fig. 2. A procedure can be called from procedure steps and process steps. Each procedure step and process step specify which procedure to call.

structuring, code reuse, and exception handling have been added to Grafchart to make it more convenient for large applications. The macro step is a common non-standard extension to SFC [11] which is also included in Grafchart. A generalization of the macro step, the procedure, is also included. A procedure is roughly a reusable macro step, which can be called from several places. Procedures may be parameterized with both input and output parameters. The main advantage of procedures is to avoid code duplication. Procedures can be called from procedure steps and process steps, see Fig. 2. The difference is that procedure steps wait for the call to complete, while process steps do not.

JGrafchart is a freely available integrated development environment for the Grafchart language [9], and it was used to implement the autotuner in this paper.

III. Relay Autotuning

The relay function used in this paper is defined as in [6],

$$u(t) = \begin{cases} u_{\text{on}}, & y(t) < -h \\ u(t^-), & -h \le y(t) \le h \\ u_{\text{off}}, & y(t) > h \end{cases} \quad (1)$$

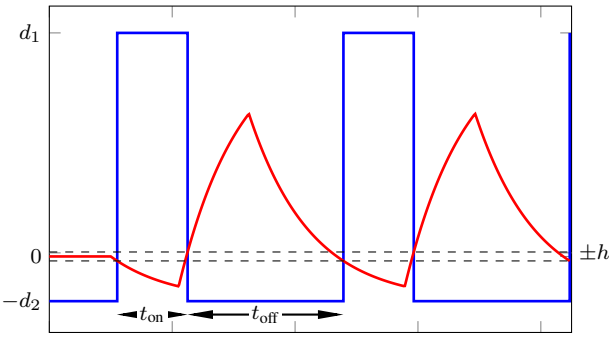

Fig. 3. An example of the signals from the asymmetric relay feedback experiment. The relay output u is shown in blue and the process output y is shown in red. The black dashed lines show the hysteresis levels, $\pm h$. The experiment is started when the system is in equilibrium at the point (u_0, y_0), here (0, 0). The asymmetric oscillation is caused by the different relay amplitudes d_1 and d_2. t_{on} and t_{off} are the durations that the relay output has been u_{on} and u_{off} respectively. Note that the relay output switches between u_{on} and u_{off} when the process output leaves the hysteresis band.

where h is the hysteresis of the relay and $u(t^-)$ is the value u had the moment just before time t. The output signals of the relay, u_{on} and u_{off}, are defined as

$$u_{\text{on}} = u_0 + \text{sign}(K_p)d_1, \quad u_{\text{off}} = u_0 - \text{sign}(K_p)d_2, \quad (2)$$

where the sign of the process gain K_p is determined during the startup of the experiment, see [6] for more details. It is assumed that the system is at equilibrium at the working point (u_0, y_0) before the relay experiment is started.

The name *asymmetric* relay reflects that the amplitudes d_1 and d_2 are different. This will cause asymmetric oscillations. The relay asymmetry level is denoted γ and is defined as

$$\gamma = \frac{\max(d_1, d_2)}{\min(d_1, d_2)}. \quad (3)$$

An illustrative example of asymmetric relay feedback, when the static gain of the process is positive, is shown in Fig. 3. The half-periods t_{on} and t_{off} are defined as the duration of the time intervals where $u(t) = u_{\text{on}}$ and $u(t) = u_{\text{off}}$ respectively.

A. Model Estimation

The low-order model structures used are either the first order time delayed (FOTD) model,

$$P(s) = \frac{K_p}{1 + sT} e^{-sL}, \quad (4)$$

or the integrating time delayed (ITD) model,

$$P(s) = \frac{k_v}{s} e^{-sL}. \quad (5)$$

The choice of which model structure to use is based on the normalized time delay, τ, defined as

$$\tau = \frac{L}{L + T}, \qquad 0 \le \tau \le 1. \quad (6)$$

A small τ implies that the time constant, T, is much larger than the time delay, L. The process is then close to an integrating process, and should therefore be modeled as an ITD model. Otherwise an FOTD model is estimated.

The only measurements required from the experiment to estimate these models are the half-periods t_{on} and t_{off} as well as the integrals of the process output, I_y, and the relay output (process input), I_u, defined as

$$I_y = \int_{t_p} (y(t) - y_0)dt, \qquad I_u = \int_{t_p} (u(t) - u_0)dt, \quad (7)$$

where integration is performed over one period of the oscillation $t_p = t_{\text{on}} + t_{\text{off}}$. All these measures are easily and robustly determined from the experiment.

From an asymmetric relay experiment, τ is estimated as

$$\tau(\rho, \gamma) = \frac{\gamma - \rho}{(\gamma - 1)(0.35\rho + 0.65)} \quad (8)$$

where γ is the asymmetry level and ρ is the half-period ratio, that is, the largest ratio between t_{on} and t_{off} [6]. The FOTD model parameters (K_p, T, L) are estimated as

$$K_p = \frac{I_y}{I_u}, \quad (9)$$

$$T = t_{\text{on}} \Big/ \ln\left(\frac{h/|K_p| - d_2 + (d_1 + d_2)e^{\tau/(1-\tau)}}{d_1 - h/|K_p|}\right), \quad (10)$$

$$L = T\frac{\tau}{1-\tau}. \quad (11)$$

The ITD model parameters (k_v, L) are estimated as

$$k_v = \frac{2I_y}{t_{\text{on}}t_{\text{off}}(u_{\text{on}} + u_{\text{off}})} + \frac{2h}{u_{\text{on}}t_{\text{on}}}, \quad (12)$$

$$L = \frac{u_{\text{on}}t_{\text{on}} - 2h/k_v}{u_{\text{on}} - u_{\text{off}}}. \quad (13)$$

For more details, see [6].

IV. The Grafchart Autotuner

The reusable PID procedure developed in [12] was used as a base for the autotuner implementation. It is a full-fledged PID controller implementation based on the velocity (incremental) algorithm. It supports e.g. tracking, feedforward, set-point weighting, auto/manual mode, and bumpless parameter and mode changes. In the base implementation, the control algorithm is implemented in a single Grafchart step. Even though this is not how PID controllers are typically implemented, this implementation is both powerful and concise. In this work, the base PID implementation was extended with autotune functionality, namely the asymmetric relay experiment and the AMIGO PID tuning rule [13].

The autotuner implementation supports features like automatic choice of hysteresis level and sign of process gain, soft startup, and adaptive relay amplitude. One decision for an asymmetric relay experiment is the direction of the large relay amplitude. In this implementation, the large amplitude is chosen to be directed towards the middle of the control signal range, to make it possible to use the autotuner close to either control signal saturation limit.

The autotuner functionality has been structured using macro steps to enable a good overview. The top level of the PID procedure is shown in Fig. 4. Compared to the base implementation [12], the `autotuner` macro step has been

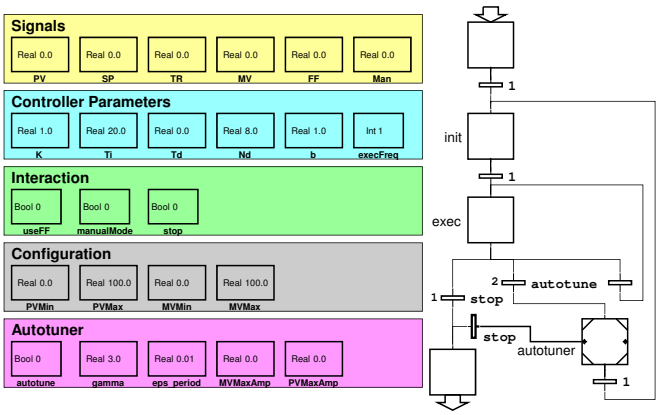

Fig. 4. The Grafchart PID procedure, extended with a relay autotuner.

added, and the initialization has been moved from the enter step to a separate step, `init`, to make it possible to re-initialize the PID algorithm when the autotuning is complete. One of Grafchart's exception handling features has been used to make it possible to abort the PID procedure, even when the autotuner is running, regardless of the current autotuner state. All step actions have been hidden in the figures. For example, the `exec` step contains the complete PID algorithm. Some new parameters have been added, namely the parameters in the Configuration and Autotuner boxes. The configuration parameters define the ranges for the control signal (Manipulated Variable, MV) and the measurement value (Process Value, PV). The `autotune` parameter is used to start and stop the autotuner, `gamma` is the asymmetry level, `eps_period` is used to determine when the relay experiment is finished, `MVMaxAmp` is the maximum control signal amplitude, and `PVMaxAmp` is the maximum allowed variations in the measurement value.

The PID sample time that is used in the control algorithm (given by `execFreq`), is not used by the autotuner. Instead, the autotuner executes as often as possible, that is, it has the same sample time as the PID procedure caller.

The next level of the autotuner is shown in Fig. 5. On this level, the autotuner parameters are checked and the autotuner is supervised and aborted if abnormal behavior is detected, for example, if the process value saturates or if the tracking signal differs too much from the requested control signal.

The next level, shown in Fig. 6, is the main implementation of the autotuner. It begins with an initialization, which will be described later, and is followed by the part which conducts the relay experiment. The two steps in this part have several responsibilities: They update the control signal, measure the half-period times, and integrate the control signal and measurement values. They also perform gain adaptation, prepare the graceful shutdown, and contain logic to detect when the relay experiment is finished. The graceful shutdown part is used to avoid a large process value disturbance when the relay experiment finishes. This is done by extending the experiment until the next process value peak, instead of turning it off when crossing the hysteresis band. In the next part, τ is estimated, and based on this, either

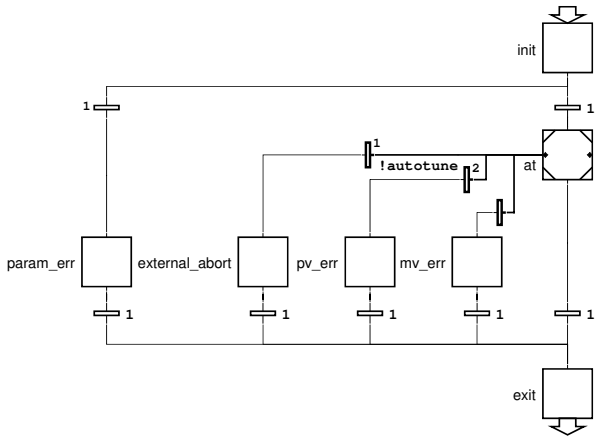

Fig. 5. The top level of the relay autotuner. It mainly contains experiment supervision and error detection.

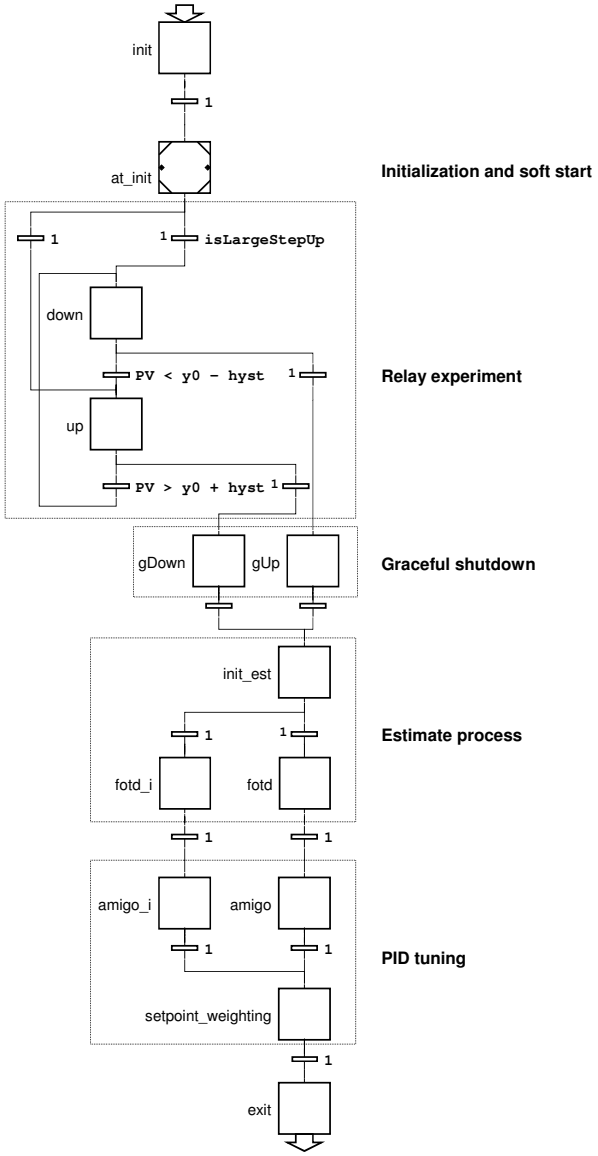

Fig. 6. The main part of the relay autotuner.

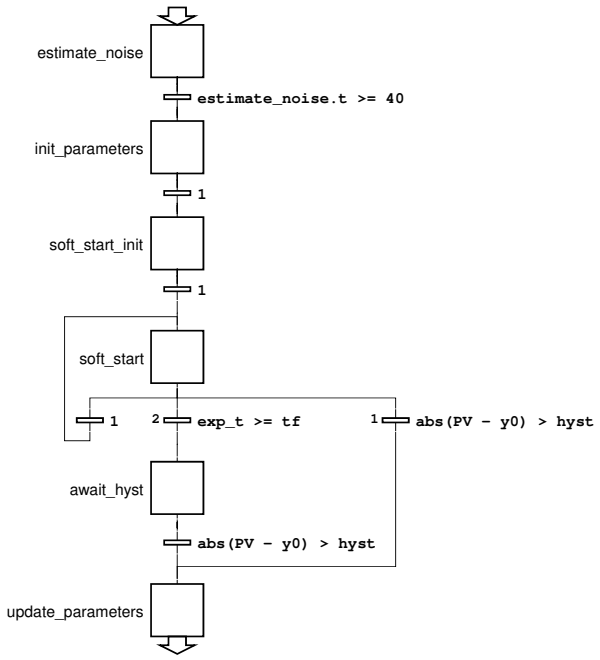

Fig. 7. The initialization part of the relay autotuner.

an FOTD model or ITD model is estimated according to (8)–(11) and (12)–(13). The equations use math functions such as `exp` and `log`, which were thus added to JGrafchart. Lastly, the PID parameters are estimated according to the AMIGO tuning method [13] and a fitted optimization formula for the set-point weight [14].

The principle for the gain adaptation is as follows: A minimum, a maximum, and a desired process value amplitude are calculated based on the noise estimate. The half-period with the large process value amplitude, that is, the half-period with the small control signal amplitude, is considered. At the end of that half-period, the largest process value deviation during the half-period is compared to the minimum and maximum process value amplitudes. If it is outside that range, the control signal amplitude is adapted to give the desired process value amplitude. The gain is not adapted if the half-period concludes the relay experiment, since the process estimate would then use the wrong amplitude.

The autotuner initialization is shown in Fig. 7. First, the noise level is estimated, then all autotuner parameters are initialized. Next, soft start is used to avoid a potentially large process value disturbance due to a control signal step on a process with unknown gain. The soft start ramps up the control signal exponentially to at most the maximum allowed amplitude. The process value is monitored during the ramp-up. When the hysteresis band is left, the relay amplitude is re-initialized so that the large relay amplitude is the current ramp-up amplitude. The sign of the process gain is determined by the change of the process value.

V. Evaluation

The autotuner has been run on several different processes, both simulated and physical, with and without noise. This

section presents noise-free simulations for three processes with different dynamics. The processes are the same as the ones used in [5], that is,

$$\begin{aligned} P_1(s) &= \frac{1}{(s+1)(0.1s+1)(0.01s+1)(0.001s+1)}, \\ P_2(s) &= \frac{1}{(s+1)^4}, \\ P_3(s) &= \frac{1}{(0.05s+1)^2}e^{-s}, \end{aligned} \tag{14}$$

where P_1 is lag dominated, P_2 is balanced, and P_3 is delay dominated. The reason to use the same processes is to make it possible to verify the implementation. Note, however, that the results should not be identical since there are some significant differences between the implementations. First, in [5], the autotuner is run in continuous time while the autotuner presented in this paper runs in discrete time with a sample time of 5 ms. Second, the autotuner in [5] uses fixed relay experiment parameters, while the autotuner in this paper derives some parameters. However, the non-derived parameters have the same values, namely $\gamma = 1.5$ and $\epsilon = 0.01$. The method used to find the normalized time delay also differs, since in [5] a fitted second order polynomial was used instead of (8). Lastly, the algorithm in [5] does not have an adaptive relay amplitude or soft startup.

The experiments presented in this section were run from JGrafchart, using a newly written simulation library to simulate the processes. The simulation library includes standalone procedure for basic building blocks such as first order filters, time delays, additive white noise, and quantization. Coding conventions and knowledge about the internal execution model were used to obtain the desired simulation data flow in JGrafchart, which is not a data flow language. The experiments were logged to a Matlab .m-file using JGrafchart's Socket I/O and a custom socket server application called SockLog.

The relay experiment for P_1 is shown in Fig. 8. The experiment begins with a phase where the measurement noise is estimated, which is then used to set the hysteresis level for the experiment. Since there is no noise in these experiments, the hysteresis is set to a default minimum value, which is a fraction of the process value range. Next, at time 0.2, the soft startup begins to slowly ramp up the control signal to its maximum allowed amplitude. A default value, which is a fraction of the control signal range, was used here, but it could also be specified by the operator. Since P_1 has slow dynamics, the ramp-up completes before the process value leaves the hysteresis band. The control signal then stays at the maximum allowed relay amplitude until the process value leaves the hysteresis band at time ≈ 0.5. At this time, it is concluded that the process gain is positive, since the process value moves in the same direction as the applied control signal change. The relay then switches to u_{off} and waits for the process value to cross the hysteresis band again, which happens at time ≈ 1. This continues for a few half-periods until time ≈ 1.9 where the algorithm concludes that the oscillation has stabilized. The half-periods t_{on} and t_{off}

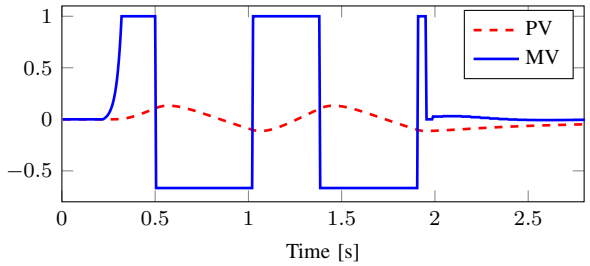

Fig. 8. Relay experiment for $P_1(s)$.

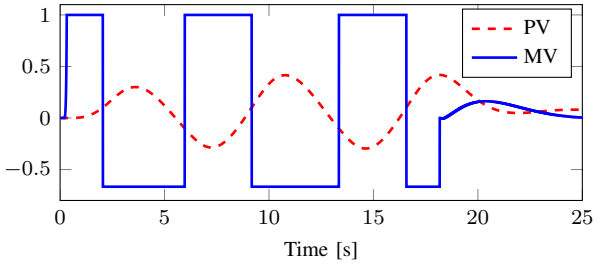

Fig. 9. Relay experiment for $P_2(s)$.

and the integrals I_y and I_u are then measured from the last oscillation. A final step is then applied until the time when the process value reaches its maximum amplitude. Finally, the model is estimated, controller parameters are calculated, and the PID procedure returns to normal control.

The relay experiment for P_2 is shown in Fig. 9. It looks similar to the experiment for P_1, except that it is considerably slower. The disturbance on the process value is also larger, but not large enough to cause the gain adaption to act.

The relay experiment for P_3 is shown in Fig. 10. The disturbance on the process value is considerably larger than for both P_1 and P_2. During the second half-period, the process value amplitude is considered too large, and the relay amplitude is decreased.

Table I shows the estimated model parameters for P_1, P_2, and P_3 for both the Matlab implementation in [5] and the JGrafchart implementation presented in this paper. Since τ is small for P_1, it is close to an integrating process. Hence, both an ITD and an FOTD model are presented.

Based on the process estimates in Table I, PI parameters are calculated, see Table II. For each set of PI parameters, a unit step load disturbance is applied in JGrafchart to obtain the integrated absolute error (IAE) value, the classic

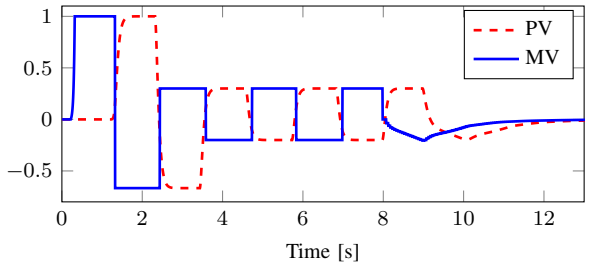

Fig. 10. Relay experiment for $P_3(s)$.

TABLE I

THE ESTIMATED FOTD MODELS. FOR P_1, PARAMETERS FOR AN ITD MODEL ARE ALSO PRESENTED.

		K_p/k_v	T	L	τ
P_1	Matlab	0.999	1.115	0.073	0.062
	JGrafchart ITD	0.862	-	0.077	0.072
	JGrafchart	0.994	1.132	0.088	0.072
	Step	1.000	1.040	0.075	0.067
P_2	Matlab	0.996	2.993	1.770	0.372
	JGrafchart	0.976	3.148	1.899	0.376
	Step	1.000	2.900	1.42	0.330
P_3	Matlab	1.006	0.078	1.030	0.929
	JGrafchart	1.000	0.115	0.980	0.895
	Step	1.000	0.093	1.010	0.920

TABLE II

THE PID PARAMETERS AND CORRESPONDING UNIT STEP LOAD DISTURBANCE IAE VALUES.

		K	T_i	M_{ST}	IAE
P_1	Matlab	4.590	0.549	1.398	0.120
	JGrafchart ITD	5.281	1.030	1.350	0.195
	JGrafchart	3.835	0.609	1.325	0.159
	Optimal	4.200	0.494	1.398	0.118
P_2	Matlab	0.360	2.769	1.252	7.690
	JGrafchart	0.349	2.953	1.230	8.452
	Optimal	0.432	2.250	1.397	5.208
P_3	Matlab	0.170	0.370	1.410	2.177
	JGrafchart	0.180	0.364	1.453	2.020
	Optimal	0.164	0.371	1.389	2.262

controller performance measure defined as

$$\text{IAE} = \int_0^\infty |e(t)|\, dt \tag{15}$$

where $e(t)$ is the control error during a step load disturbance at the process input. Note that a lower IAE value means better control performance. The optimal controller parameters are obtained by minimizing the IAE value for the true process model under the constraint that the maximum of the sensitivity functions, $M_{ST} = \max(M_S, M_T) \leq 1.4$.

The results of the JGrafchart implementation and the Matlab implementation from [5] are comparable. The results are also similar to the ones achieved from an open-loop step response experiment. For P_1 and P_3, the IAE values of the autotuners are about the same as the optimal values. For P_3 they are slightly better, which is possible since their maximum sensitivity is greater than 1.4. For P_2 on the other hand, the results are considerably worse than the optimal. This is due to the fact that this process is not satisfactorily described by an FOTD model, because of its multiple identical poles.

VI. SUMMARY

In this paper, an asymmetric relay autotuner has been integrated to an existing PID controller in Grafchart. The relay experiment is completely automatic, and includes features such as soft startup, adaptive relay amplitude, and detection of the sign of the process gain. The only requirement is that the process is in steady-state when the experiment is requested to start, a limitation shared with the original autotuner. The use of an asymmetric relay has the disadvantage that it disturbs the process more than a symmetric relay. However, it gives better excitation, which makes it possible to obtain a low-order model from a single relay experiment. Hence, the process is disturbed for a shorter time than for autotuners that must perform two or more experiments to obtain comparable models.

The relay experiment is primarily sequential and this work confirms that it is suitable to implement it in a language dedicated to sequential control. Using the high level sequential programming language Grafchart gives a clear overview and is still efficient for implementations. Grafchart's support for hierarchical structuring, exception handling, and code reuse are all utilized. Without these features, the autotuner implementation would be much more complicated.

Since the functions of the autotuner are independent of the controller functions, it could be moved to a separate procedure. The relay experiment would then become a standalone, reusable, process estimation tool, which would be used from the PID procedure.

The results of the implemented autotuner are comparable to other tuning methods, and the autotuner will be included in future public releases of JGrafchart.

REFERENCES

[1] K. J. Åström and T. Hägglund, "Automatic tuning of simple regulators with specifications on phase and amplitude margins," *Automatica*, vol. 20, no. 5, pp. 645–651, Sept. 1984.

[2] Emerson, "Emerson process management – DeltaV DCS." [Online]. Available: http://www2.emersonprocess.com/en-us/brands/deltav/pages/index.aspx

[3] ABB, "ABB 800xA DCS." [Online]. Available: http://new.abb.com/control-systems/system-800xa/800xa-dcs

[4] T. Liu, Q.-G. Wang, and H.-P. Huang, "A tutorial review on process identification from step or relay feedback test," *Journal of Process Control*, vol. 23, no. 10, pp. 1597–1623, 2013.

[5] J. Berner, K. J. Åström, and T. Hägglund, "Towards a new generation of relay autotuners," in *Proceedings of the 19th IFAC World Congress (IFAC'14), Cape Town, South Africa*, Aug. 2014.

[6] J. Berner, "Automatic Tuning of PID Controllers based on Asymmetric Relay Feedback," Dept. Automatic Control, Lund University, Sweden, Licentiate Thesis ISRN LUTFD2/TFRT--3267--SE, May 2015.

[7] IEC, "IEC 61131-3: Programmable controllers – part 3: Programming languages ed2.0," International Electrotechnical Commission, Tech. Rep., Jan. 2003.

[8] Siemens, "DCS SIMATIC PCS 7." [Online]. Available: http://w3.siemens.com/mcms/process-control-systems/en/distributed-control-system-simatic-pcs-7/pages/distributed-control-system-simatic-pcs-7.aspx

[9] Department of Automatic Control, Lund University, "Grafchart." [Online]. Available: http://control.lth.se/Research/tools/grafchart.html

[10] A. Theorin, "A sequential control language for industrial automation," Ph.D. dissertation, Department of Automatic Control, Lund University, Sweden, Nov. 2014.

[11] R. W. Lewis, *Programming Industrial Control Systems Using IEC 1131-1*, 2nd ed. The Institution of Engineering and Technology, Dec. 1998.

[12] A. Theorin and C. Johnsson, "An interactive PID learning module for educational purposes," in *Proceedings of the 19th IFAC World Congress (IFAC'14), Cape Town, South Africa*, Aug. 2014.

[13] K. Åström and T. Hägglund, *Advanced PID Control*. ISA – The Instrumentation, Systems, and Automation Society, 2006.

[14] M. Hast, "Design of low-order controllers using optimization techniques," Ph.D. dissertation, Department of Automatic Control, Lund University, Sweden, June 2015.