Transformation of a PID Controller for Numerical Accuracy

N. Damouche[1,2] M. Martel[1,2] A. Chapoutot[3]

[1] *Université Montpellier II & CNRS, LIRMM, UMR 5506, Montpellier, France*
[2] *Université de Perpignan Via Domitia, DALI, Perpignan, France*
[3] *ENSTA ParisTech, Unité d'Informatique et d'Ingénierie des Systèmes, Palaiseau, France*

Abstract

Numerical programs performing floating-point computations are very sensible to the way formulas are written. Several techniques have been proposed concerning the transformation of expressions in order to improve their accuracy and now we aim at going a step further by automatically transforming larger pieces of code containing several assignments and control structures. This article presents a case study in this direction. We consider a PID controller and we transform its code in order to improve its accuracy. The experimental data obtained when we compare the different versions of the code (which are mathematically equivalent) show that those transformations have a significant impact on the accuracy of the computations.

Keywords: Numerical Accuracy, Semantics-Based Program Transformation, Floating-Point Arithmetic, Validation of Numerical Programs.

1 Introduction

Numerical programs performing floating-point computations are very sensible to the way formulas are written. Indeed, small syntactic changes in the arithmetic expressions which do not modify their mathematical meaning may lead to significant changes in the result of their evaluation. This sensibility to the way expressions are written is due to the particularities of the floating-point arithmetic in which, for example, addition is not associative or multiplication is not inversible [9,2,8]. In addition, it is very difficult to guess which writing of a formula gives the best accuracy when evaluated with floating-point numbers and abstract interpretation [10] have been developed to infer safe approximations of the round-off error on the result of a computation [4,5,1].

Our work concerns the automatic transformation of floating-point computations in order to improve their numerical accuracy. Several results have been obtained

[1] Email: nasrine.damouche@univ-perp.fr
[2] Email: matthieu.martel@univ-perp.fr
[3] Email: alexandre.chapoutot@ensta-paristech.fr

concerning the transformation of expressions [6,7,11] and now we aim at going a step further by automatically transforming larger pieces of code containing several assignments and control structures.

This article presents a case study in this direction. We consider a PID controller and we transform its code in order to improve its accuracy. More precisely, we take an initial PID code and we apply to it several processings in order to generate other PID programs which are mathematically equivalent to the initial one but more accurate in computing. The first transformation only rewrites the assignments while, in the second transformation, the loop is unfolded. While these transformations are made by hand, they are applied systematically, in a way which we aim at automatizing in future work. The experimental data obtained when we compare the executions of the three codes (which are mathematically equivalent) show that those rewritings have a significant impact on the accuracy of the computations.

The rest of this article is organized as follows. Section 2 introduce the original controller $\texttt{PID}_1$. The transformations are done in Sections 3 and 4, yielding $\texttt{PID}_2$ and $\texttt{PID}_3$. The experimental results are presented in Section 5 and Section 6 concludes.

2 Description of the PID Controller

In this section, we give a brief description of the original `PID` program of Listing 1. This kind of algorithm is used in embedded and critical systems to maintain a measure at a certain value named *setpoint*. The error being the difference between the setpoint and the measure, the controller computes a correction based on the integral i and derivative d of the error and also from a proportional error term p. A weighted sum of these terms is computed. The weights are used to improve the reactivity, the robustness and the speed of the program. The three terms are:

i) The proportional term p: the error e is multiplied by a proportional factor k_p,

$$p = k_p \times e \ .$$

ii) The integral term i: the error e is integrated and multiplied by an integral factor k_i,

$$i = i + (k_i \times e \times dt) \ .$$

iii) The derivative term d: The error e is derived with respect to time and is then multiplied by the derivative factor k_d,

$$d = kd \times (e - eold) \times \frac{1}{dt} \ .$$

3 How to Get a Better PID?

In this section, we detail the different steps needed to transform the original `PID`, named $\texttt{PID}_1$, into a new equivalent program named $\texttt{PID}_2$. The main idea consists of developing and simplifying the expressions of $\texttt{PID}_1$ and inlining them within the loop, in order to extract the constant expressions and to reduce the number of

Listing 1: Source code of $\texttt{PID}_1$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2;
m = 8.0;      c = 5.0;       eold = 0.0;
while true do
     e = c - m;
     p = kp * e;
     i = i + ki * dt * e;
     d = kd * invdt * (e - eold);
     r = p + i + d;
     eold = e;          /* updating memory   */
     m = m + 0.01 * r;  /* computing measure: the plant */
```

operations. At the n^{th} iteration of the loop, we have:

$$\begin{aligned} p_n &= k_p \times e_n \ . \\ i_n &= i_{n-1} + k_i \times e_n \times dt \ . \\ d_n &= k_d \times (e_n - e_{n-1}) \times \frac{1}{dt} \ . \end{aligned}$$

If we inline the expressions of p_n, i_n and d_n in the formula of the result expression r_n and after extracting the common factors, we get:

$$r_n = e_n \times \left(k_p + k_d \times \frac{1}{dt} + k_i \times dt \right) + k_i \times dt \times \sum_{i=0}^{n-1} e_i - k_d \times e_{n-1} \times \frac{1}{dt} \ . \qquad (1)$$

Then we remark that there exists some constant sub-expressions in Equation (1). So, we compute them once before entering into the loop. We have:

$$c_1 = k_p + k_d \times \frac{1}{dt} + k_i \times d \ , \qquad c_2 = k_i \times dt \ , \qquad c_3 = k_d \times \frac{1}{dt} \ .$$

Next, we record in a variable s the sum $s = \sum_{i=0}^{n-1} e_i$ which adds the different errors from e_0 to e_{n-1}. Finally, we have:

$$r_n = R + c_2 \times s \qquad \text{with} \qquad R = c_1 \times e_n - c_3 \times e_{n-1} \ .$$

Our $\texttt{PID}_2$ algorithm is given in Listing 2.

Listing 2: source code of $\texttt{PID}_2$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2;
m = 8.0;      c = 5.0;       eold = 0.0;   R = 0.0;       s = 0.0;
c1 = kp + kd * invdt + ki * dt;  c2 = kd * invdt;  c3 = ki * dt;
while true do
     e = c - m;
     R = (c1 * e) - (c2 * eold);
     s = s + eold;
     r = R + (c3 * s);
     eold = e;
     m = m + 0.01 * r;
```

4 How to Get an Even Better PID?

The initial PID can be transformed even more drastically by unfolding the loop. In our case, we arbitrarily choose to unfold it four times in order to keep for each

execution the sum of the last four errors. Then, we change the order of the operations, either by summing the terms pairwise, or in increasing or decreasing order. The process applied to reach the third PID algorithm, named PID3, is given in the following. Let us begin by unfolding four times the integral term:

$$i_{n-1} = i_{n-2} + k_i \times dt \times e_{n-1} \qquad i_{n-2} = i_{n-3} + k_i \times dt \times e_{n-2}$$
$$i_{n-3} = i_{n-4} + k_i \times dt \times e_{n-3} \qquad i_{n-4} = i_{n-5} + k_i \times dt \times e_{n-4}$$

We inline the previous expressions in i_n. We obtain:

$$i_n = i_{n-5} + (k_i \times dt \times e_{n-4}) + (k_i \times dt \times e_{n-3}) + (k_i \times dt \times e_{n-2}) + (k_i \times dt \times e_{n-1}) + (k_i \times dt \times e_n) \quad (2)$$

with $i_{n-5} = i_0 + k_i \times dt \times \sum_{i=0}^{n-5} e_i$. Equation (2) can be even more simplified and assuming $i_0 = 0$, we have:

$$i_n = k_i \times dt \times \sum_{i=0}^{n-5} e_i + (k_i \times dt \times ((((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}) + e_n)) \quad .$$

Now, if we come back to the result expression after having done some manipulations, like developing the derivative and factorizing, we obtain as final expression:

$$r_n = e_n \times \left(k_p + k_d \times \frac{1}{dt}\right) + k_i \times dt \times \sum_{i=0}^{n-5} e_i - k_d \times \frac{1}{dt} \times e_{n-1} + k_i \times dt \times ((((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}) + e_n) \quad .$$

Denoting by $s = (((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1})$, $k_1 = k_p + \left(k_d \times \frac{1}{dt}\right) + (k_i \times dt)$ $k_2 = k_i \times dt$ and $k_3 = k_d \times \frac{1}{dt}$, the final expression of r_n is:

$$r_n = R + k_2 \times \left(s + \sum_{i=0}^{n-5} e_i\right) \qquad \text{with} \qquad R = (e_n \times k_1) - (k_3 \times e_{n-1})$$

The complete code of $\texttt{PID}_3$ is given in Listing 3.

Listing 3: source code of $\texttt{PID}_3$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2;
R = 0.0;  S = 0.0;  s = 0.0;  m = 8.0;  c = 5.0;  eold= 0.0;
e1 = e2 = e3 = e4 = 0.0; k1 = kp + kd * invdt; k2 = kd * invdt; k3 = ki * dt;
while true do
     e = c - m;
     R = (k1 * e) - (k2 * eold);
     S = s + (e4 + (e3 + (e2 + e1)));
     r = R + (k3 * S);
     eold= e;  e4 = e3;  e3 = e2;  e2 = e1;  e1 = e;
     m = m + 0.01 * r;
```

5 Experimentations

Let us focus now on the execution of the three `PID` programs. In our Python implementation using the *GMPY2* library for multiple precision computations, the results obtained show that there is a significant difference between $\texttt{PID}_1$, $\texttt{PID}_2$ and $\texttt{PID}_3$ on the first digit of the decimal values of the result and sometimes less. To better visualize these results, the curves corresponding to the three `PID` algorithms are given in Figure 1. We can observe a significant difference between the curves corresponding to the three `PID`, mainly between 0 and 150 of the x-axis.

Figure 2 shows the difference between the three `PID`. This difference, which is important, is computed with many precisions. So, the same behavior was observed by using 24, 53 and 50000 bits of the mantissa. The error between $\texttt{PID}_1$ and $\texttt{PID}_3$ oscillates between 0.03 and 0.06 while the value ranges between 5 and 8.

We also observe that the differences between $\texttt{PID}_1$ and $\texttt{PID}_2$ are significant. Indeed, we can remark that a small syntactic change in the code may yield an important difference in term of accuracy. For example, let us take the following expression r of $\texttt{PID}_1$ and let us just inline the three terms p, i and d and factorize

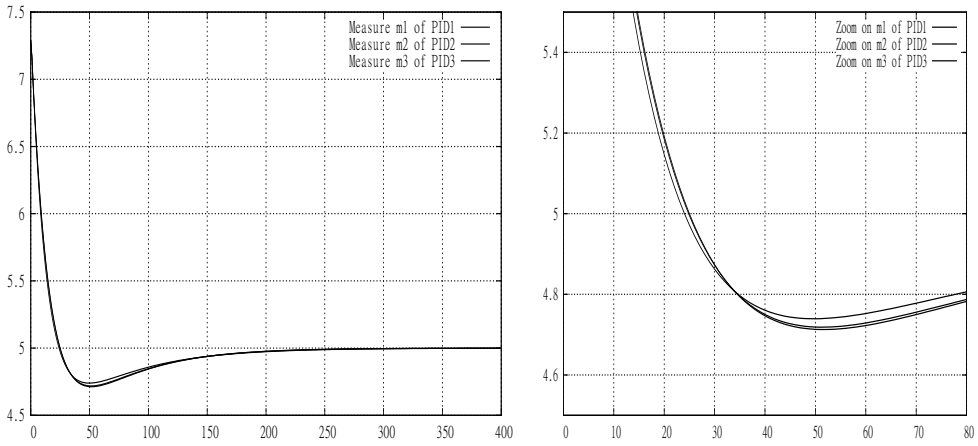

Fig. 1. Value of m in the three `PID` algorithms.

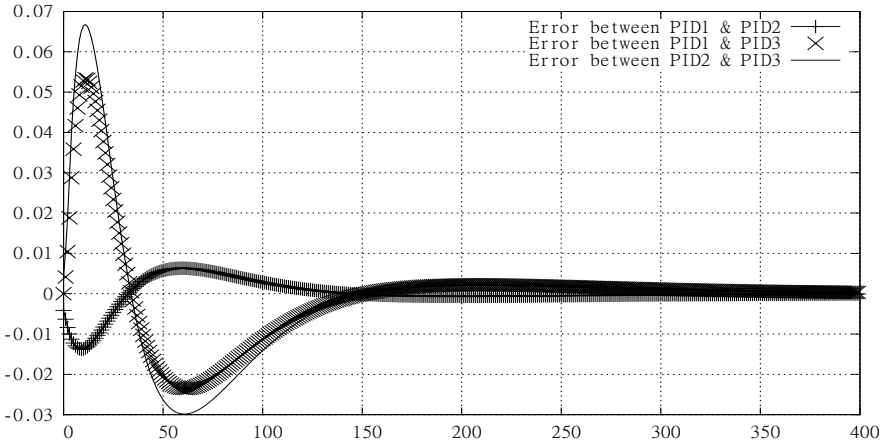

Fig. 2. Error between the values of run in the three PID.

e in it. Initially, $r = p + i + d$ and we obtain after factorizing:

$$r' = e \times \left(k_p + k_i \times dt + k_d \times \frac{1}{dt}\right) + i_0 - \left(k_d \times \frac{1}{dt} \times eold\right) \quad .$$

With this simple modification, the difference in accuracy is already important, as shown in Figure 3 which gives the difference between r and r' and between m and m' for the first iterations of the loop.

It	r	r'	m	m'
1	−71.449235999999999	−71.863271999999995	7.2855076399999996	7.2813672799999996
2	−12.165627583387671	−12.38244654421103	7.1638513641661232	7.157542814557889
3	−19.748721881874111	−19.956017221473264	6.9663641453473817	6.9579826423431568
4	−17.074726204512665	−17.236242459488839	6.7956168833022552	6.7856202177482681
5	−16.089171574335889	−16.21814718941534	6.6347251675588961	6.6234387458541146
6	−14.934346159219487	−15.033378340554384	6.4853817059667014	6.4731049624485708
7	−13.892138878139511	−13.965323318091185	6.346460317185306	6.3334517292676589
8	−12.913239775385652	−12.963910797328566	6.2173279194314492	6.2038126212943734
9	−12.000031667420215	−12.031224799805722	6.0973276027572467	6.0835003732963164
10	−11.147227169037631	−11.161638699085488	5.9858553310668707	5.9718839863054614

Fig. 3. Comparaison between r and r' and between the corresponding measures m and m'.

6 Conclusion

In this paper, our attention focused on the transformation of the PID Controller. From the $\texttt{PID}_1$, we believe that is possible to move to the $\texttt{PID}_2$ and to the $\texttt{PID}_3$ (see Section 3 and 4). The results obtained when running the three PID are promising and drive us to go further more by developing a prototype which can provide automatically more equivalents programs.

Currently, we are developing a software, based on a set of rules (associativity, commutativity, etc.) and commands (while do, if then else, etc.) in order to infer new equivalents arithmetic expressions. This tool takes as input an initial program, the $\texttt{PID}_1$, and generates automatically other PID programs that are equivalents mathematically and more precise. Although, we have to be careful in front of the combinatory explosion problem which occurs when transforming the different statement of the PID.

In the future, after we finish developing our prototype, we aim to implement it on Lustre.

References

[1] Alexandre Chapoutot, *Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables*, 2010, 184-200, Static Analysis, 2010. Springer, LNCS, 6337.

[2] ANSI/IEEE, IEEE Standard for Binary Floating-point Arithmetic, Std 754-2008, ANSI/IEEE, 2008.

[3] Arnault Ioualalen, Matthieu Martel. *Synthesis of arithmetic expressions for the fixed-point arithmetic: The Sardana approach.* DASIP. 2012. pages 1-8.

[4] Eric Goubault, *Static Analysis by Abstract Interpretation of Numerical Programs and Systems, and FLUCTUAT*, 2013, 1–3, Static Analysis, 2013. Springer, LNCS, 7935.

[5] Eric Goubault, Sylvie Putot, *Static Analysis of Finite Precision Computations*, 2011, 232–247, Verification, Model Checking, and Abstract Interpretation, 2011. Springer, LNCS, 6538.

[6] M. Martel.*Semantics-Based Transformation of Arithmetic Expressions*. InStatic Analysis Symposium, SAS'07, number 4634 of Lecture Notes in Computer Science, pages 298-314, 2007. Springer-Verlag.

[7] Matthieu Martel. *Accurate Evaluation of Arithmetic Expressions (Invited Talk)*, Electr. Notes Theor. Comput. Sci. v 287. 2012. 3–16. website http://dx.doi.org/10.1016/j.entcs.2012.09.002

[8] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. ISBN 0-89871-482-6.

[9] Muller, Jean-Michel and Brisebarre, Nicolas and de Dinechin, Florent and Jeannerod, Claude-Pierre and Lefèvre, Vincent and Melquiond, Guillaume and Revol, Nathalie and Stehlé, Damien and Torres, Serge, *Handbook of Floating-Point Arithmetic*, Birkhäuser Boston, 978-0-8176-4704-9, 2010.

[10] Patrick Cousot, Radhia Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, 238–252, Principles of Programming Languages, ACM Press, 1977.

[11] Xitong Gao, Samuel Bayliss, George A. Constantinides. *SOAP: Structural optimization of arithmetic expressions for high-level synthesis*. Pacific J. Math. International Conference on Field-Programmable Technology, 2013.