Static Analysis of the Numerical Stability of Loops

Matthieu Martel

CEA - Recherche Technologique
LIST-DTSI-SLA
CEA F91191 Gif-Sur-Yvette Cedex, France
e-mail : mmartel@cea.fr

Abstract. We introduce a relational static analysis to determine the stability of the numerical errors arising inside a loop in which floating-point computations are carried out. This analysis is based on a stability test for non-linear functions and on a precise semantics for floating-point numbers that computes the propagation of the errors made at each operation. A major advantage of this approach is that higher-order error terms are not neglected. We introduce two algorithms for the analysis. The first one, less complex, only determines the global stability of the loop. The second algorithm determines which particular operation makes a loop unstable. Both algorithms have been implemented and we present some experimental results.
Keywords: Abstract Interpretation, Numerical Precision, Relational Analysis, Semantics of Floating-point Numbers.

1 Introduction

It is often hard for a programmer to understand how precise the result of a floating-point calculation is and to understand which operations introduce the most significant errors [7, 11]. Recent work has shown that abstract interpretation [6] is a good candidate for the validation and debugging of numerical codes [9, 13] and the first attempts to verify industrial programs with a prototype analyzer are promising [10]. This approach consists in defining a precise concrete semantics for floating-point operations, based on the IEEE 754 Norm [2, 7], and, next, in practice, using a more compact abstract semantics, among the many possible ones, to check properties of interest for a program [13].

To our knowledge, most of the alternative techniques aim at dynamically estimating a better approximation of the real numbers that the program would output if the machine were working in infinite precision [12, 16]. On the contrary, since we are interested in detecting the errors possibly introduced by floating-point numbers, we always work with the values used by the machine and we compute the errors attached to them. In addition, as our approach is based on abstract interpretation, our estimations are valid for a large class of executions.

[1] This work was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.

Concerning the abstract interpretation of numerical programs, the case of calculations carried out in loops, addressed in this article, requires particular attention because of the following reasons. First, loops introduce the most subtle and possibly large imprecision because the errors arising in one iteration are related to the ones of the previous iterations. Small errors arising in one iteration can be accumulated, magnified or made negligible in the following steps. Second, for many stable loops, i.e. loops in which the errors decrease at each iteration, the errors arising in the first iterations may be significant. A static analyzer must unfold such loops a finite but a priori unknown number of times in order to detect their stability. Otherwise, using widening operators, the analyzer might state that arbitrarily large errors can be made at this location and, consequently, might state that the loop is unstable. Third, the usual narrowing techniques cannot be used to improve the results obtained by widening. This is due to the fact that the information used to narrow the result only concerns the floating-point values of the variables and this cannot be used to reduce the error terms attached to them during the first phase.

We introduce a relational static analysis to detect whether the calculations carried out in a loop are stable, i.e. whether the errors due to the floating-point numbers increase or decrease as the loop is unrolled. An important feature of our analysis is that it does not neglect the higher-order error terms (error terms obtained by multiplying previous error terms), unlike most previous work [12, 16]. As shown in Section 3, the higher-order error terms may be large even if the first-order ones are negligible. The analysis is based on the semantics of floating-point numbers with errors [13] and on a stability test for non-linear maps obtained from the calculation of the Lyapunov exponents, a widely used technique to determine the stability of dynamic systems [1, 14]. Intuitively, this test consists of determining whether the semantic equations describing the errors at each iteration of a loop are contracting or expanding. The semantic equations are automatically generated from the syntax of the program and we introduce two algorithms to determine their stability. The first one involves fewer calculations but only computes the global stability of the loop by checking that no error term diverges. The second algorithm precisely computes the stability of each error term. So, if an instability is detected, the second algorithm indicates which operation in the loop is responsible for the loss of precision. Both algorithms have been implemented and we comment on some experimental results.

The semantics of floating-point numbers with errors is given in Section 2. The Lyapunov test used by the analysis is introduced in Section 3 and we show in Section 4 how to generate the semantic equations needed for the test. Finally, in Section 5 we introduce the two algorithms to determine the stability of a loop and we comment on some results obtained from our implementation.

2 Floating-Point Numbers with Errors

This section briefly introduces the semantics of floating-point numbers with errors which is detailed in ref. [9, 13]. As an introductory example, let us ex-

amine a simple calculation involving two values $a_{\mathbb{F}} = 621.3$ and $b_{\mathbb{F}} = 1.287$. For the sake of simplicity, we assume that $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$ belong to a simplified set of floating-point numbers composed of a mantissa of four digits written in base 10. We assume that initial errors are attached to $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$, and we write $a = 621.3\varepsilon + 0.05\varepsilon_1$ and $b = 1.287\varepsilon + 0.0005\varepsilon_2$ to indicate that the value of the initial error on $a_{\mathbb{F}}$ (resp. $b_{\mathbb{F}}$) is 0.05 (resp. 0.0005). ε is a formal variable attached to the floating-point coefficient of a and b and ε_1 and ε_2 are formal variables related to static control points. a and b are called *floating-point numbers with errors*. Let us focus on the product $a_{\mathbb{F}} \times b_{\mathbb{F}}$ whose exact result is $a_{\mathbb{F}} \times b_{\mathbb{F}} = 799.6131$. This calculation carried out with floating-point numbers with errors yields $a \times b = 799.6131\varepsilon\varepsilon + 0.06435\varepsilon\varepsilon_1 + 0.31065\varepsilon\varepsilon_2 + 0.000025\varepsilon_1\varepsilon_2$ which we rewrite as $a \times b = 799.6131\varepsilon + 0.06435\varepsilon_1 + 0.31065\varepsilon_2 + 0.000025\varepsilon_{hi}$. The rewriting step is made to keep only one formal variable per coefficient and obeys the following rule. The indices of the formal variables $\varepsilon_1, \varepsilon_2$ etc. are viewed as words of the alphabet of the control points and the product of two variables yields a new formal variable by concatenation of the index words. The empty word is the index of the variable ε related to the floating-point component of the number, a word of length one describes a first-order error and the special word hi is used to identify all the words of length greater than one. We assume $\text{length}(hi) > 1$. The product of formal variables is summed up by the rule:

$$\varepsilon_u \times \varepsilon_v = \begin{cases} \varepsilon_{uv} \text{ if } \text{length}(uv) \leq 1 \\ \varepsilon_{hi} \text{ otherwise} \end{cases} \tag{1}$$

The difference between $a_{\mathbb{F}} \times b_{\mathbb{F}}$ and 621.35×1.2875 is 0.375025 and this error stems from the fact that the initial error on a (resp. b) was multiplied by b (resp. a) and that a second-order error corresponding to the multiplication of both errors was introduced. So, at the end of the calculation, the contribution to the global error of the initial error on a (resp. b) is 0.06435 (resp. 0.31065) and corresponds to the coefficient attached to the formal variable ε_1 (resp. ε_2). Finally, the number 799.6131 has too many digits to be representable in our floating-point number system and we refer to the IEEE-754 norm for floating-point arithmetic to determine how the values are rounded [2]. Let $\mathbb{F}$ be either the set of simple or double precision floating-point numbers. The norm fully specifies the function $\uparrow_\circ \colon \mathbb{R} \to \mathbb{F}$ which returns the roundoff of a real number $r \in \mathbb{R}$ with respect to the current rounding mode $\circ$ [2, 7]. In addition, it ensures that the elementary operations are correctly rounded, i.e. for any operator $\diamond \in \{+, -, \times, \div\}$, we have:

$$\forall f_1, f_2 \in \mathbb{F}, \;\; f_1 \diamond_{\mathbb{F}} f_2 = \uparrow_\circ (f_1 \diamond_{\mathbb{R}} f_2) \tag{2}$$

In Equation (2), $\diamond_{\mathbb{F}}$ and $\diamond_{\mathbb{R}}$ denote the same operation over $\mathbb{F}$ and $\mathbb{R}$. For our needs, we also introduce the function $\downarrow_\circ \colon \mathbb{R} \to \mathbb{R}$ defined by $\downarrow_\circ r = r - \uparrow_\circ r$. Assuming that the machine we are using conforms to that standard, we may claim that the computed floating-point number for our example is $\uparrow_\circ (799.6131) = 799.6$ and that a new first error term $0.0131\varepsilon_\times$ is introduced by the multiplica-

tion. To sum up, we have

$$a \times b = 799.6\varepsilon + 0.06435\varepsilon_1 + 0.31065\varepsilon_2 + 0.000025\varepsilon_{12} + 0.0131\varepsilon_\times$$

At first sight, one would think that the precision of the calculation mainly depends on the initial error on a since it is 100 times larger than the one on b. However the above result indicates that the final error is mainly due to the initial error on b. Hence, to improve the precision of the final result, one should first try to increase the precision on b (whenever possible).

We now introduce the formal semantics of floating-point numbers with errors assuming that the control points of the program are annotated by unique labels $\ell \in L$ and that $\mathcal{L}$ denotes the set of words on L of length at most 1 plus the special word hi used to denote the higher-order errors (and which corresponds to no particular control point). This comes from ref. [13] which introduces more general semantics for floating-point numbers with errors. Errors terms of order n correspond to words of length n and the empty word is related to the term for the floating-point number. The multiplication of terms of order m and n yields a new term of order $m+n$ denoted by a word of length $m+n$. In this article, hi identifies all the words of length greater that one and concatenation is defined by Equation (1). Words ℓ, ℓ_0, hi etc. are written as exponents of expressions, e.g. e^ℓ. A floating-point number r occurring at the control point ℓ_0 is represented by the series

$$r^{\ell_0} = f\varepsilon + \sum_{\ell \in \mathcal{L}} \omega^\ell \varepsilon_\ell \tag{3}$$

In Equation (3), f is the floating-point number approximating the value of r. f is always attached to the formal variable ε whose index is the empty word. A term $\omega^\ell \varepsilon_\ell$ denotes the contribution to the global error of the first-order error introduced by the operation labeled ℓ during the evaluation of r. $\omega^\ell \in \mathbb{R}$ is the scalar value of this error term and ε_ℓ is a formal variable.

The elementary operations are defined in Figure 1 for $r_1 = f_1\varepsilon_1 + \sum_{\ell \in \mathcal{L}^+} \omega_1^\ell \varepsilon_\ell$ and $r_2 = f_2\varepsilon_2 + \sum_{\ell \in \mathcal{L}^+} \omega_2^\ell \varepsilon_\ell$. $\mathcal{L}^+$ denotes the set $\mathcal{L}$ without the empty word. In addition, the symbols f and ω are used interchangeably to denote the coefficient of the variable ε. The formal series $\sum_{\ell \in \mathcal{L}} \omega^\ell \varepsilon_\ell$ related to the result of an operation $\diamond^{\ell_i}$ contains the combination of the errors on the operands plus a new error term $\downarrow_\circ (f_1 \diamond_{\mathbb{R}} f_2)\varepsilon_{\ell_i}$ corresponding to the error introduced by the operation $\diamond_{\mathbb{F}}$ occurring at point ℓ_i. The rules for addition and subtraction are natural. The elementary errors are added or subtracted componentwise in the formal series and the new error due to point ℓ_i corresponds to the roundoff of the result.

Multiplication requires more care because it introduces higher-order errors due to the multiplication of the first-order errors. For instance, let us consider the multiplication at point ℓ_3 of two initial data $r_1^{\ell_1} = (f_1\varepsilon + \omega_1^{\ell_1}\varepsilon_{\ell_1})$ and $r_2^{\ell_2} = (f_2\varepsilon + \omega_2^{\ell_2}\varepsilon_{\ell_2})$.

$$r_1^{\ell_1} \times^{\ell_3} r_2^{\ell_2} = \uparrow_\circ (f_1 f_2)\varepsilon + f_2\omega_1^{\ell_1}\varepsilon_{\ell_1} + f_1\omega_2^{\ell_2}\varepsilon_{\ell_2} + \omega_1^{\ell_1}\omega_2^{\ell_2}\varepsilon_{hi} + \downarrow_\circ (f_1 f_2)\varepsilon_{\ell_3} \tag{9}$$

As shown in Equation (9), the floating-point number computed by this multiplication is $\uparrow_\circ (f_1 f_2)$. The initial first-order errors $\omega_1^{\ell_1}\varepsilon_{\ell_1}$ and $\omega_2^{\ell_2}\varepsilon_{\ell_2}$ are multiplied

$$r_1 +^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 + f_2)\varepsilon + \sum_{\ell \in \mathcal{L}^+} (\omega_1^\ell + \omega_2^\ell)\varepsilon_\ell + \downarrow_\circ (f_1 + f_2)\varepsilon_{\ell_i} \quad (4)$$

$$r_1 -^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 - f_2)\varepsilon + \sum_{\ell \in \mathcal{L}^+} (\omega_1^\ell - \omega_2^\ell)\varepsilon_\ell + \downarrow_\circ (f_1 - f_2)\varepsilon_{\ell_i} \quad (5)$$

$$r_1 \times^{\ell_i} r_2 \stackrel{\text{def}}{=} \uparrow_\circ (f_1 f_2)\varepsilon + \sum_{\substack{\ell_1 \in \mathcal{L} \\ \ell_2 \in \mathcal{L} \\ \ell_1 \cdot \ell_2 \neq \epsilon}} \omega_1^{\ell_1}\omega_2^{\ell_2}\varepsilon_{\ell_1 \cdot \ell_2} + \downarrow_\circ (f_1 f_2)\varepsilon_{\ell_i} \quad (6)$$

$$(r_1)^{-1^{\ell_i}} \stackrel{\text{def}}{=} \uparrow_\circ (f_1^{-1})\varepsilon - \frac{1}{f_1}\sum_{\ell \in \mathcal{L}} \frac{\omega^\ell}{f_1}\varepsilon_\ell + \frac{1}{f_1}\sum_{n \geq 2}(-1)^n \left(\sum_{\ell \in \mathcal{L}} \frac{\omega^\ell}{f_1}\right)^n \varepsilon_{hi} + \downarrow_\circ (f_1^{-1})\varepsilon_{\ell_i} \quad (7)$$

$$r_1 \div^{\ell_i} r_2 \stackrel{\text{def}}{=} r_1 \times^{\ell_i} (r_2)^{-1^{\ell_i}} \quad (8)$$

Fig. 1. Elementary operations for floating-point numbers with errors.

by f_2 and f_1 respectively. In addition, the multiplication introduces a new first-order error $\downarrow_\circ (f_1 f_2)$ which is attached to the formal variable ε_{ℓ_3} in order to indicate that this error is due to the product occurring at the control point ℓ_3. Finally, this operation also introduces a second-order error that we attach to the formal variable ε_{hi}. In Figure 1, Equation (6) is a generalization of Equation (9).

The term for division is obtained by means of a power series development. The full semantics is defined by the domain of floating-point numbers with errors for values and by the usual operational semantics for the expressions [13].

This semantics details the contribution to the global error of the first-order error terms and globally computes the higher-order error arising during the calculation. In practice, higher-order errors are often negligible. So this semantics allows us to determine the sources of imprecision while checking that the higher-order errors are actually globally negligible. More precise semantics are introduced in ref. [13].

In the rest of this paper, we use an abstract semantics [6], based on the one defined in this section, in which the coefficients f and ω^ℓ are abstracted by intervals of floating-point numbers and intervals of multi-precision floating-point numbers, respectively.

3 Stability in Loops

This section introduces the mathematical tools used to determine the stability of the calculations in loops. Let $f : \mathbb{R}^m \to \mathbb{R}^m$ and let $\boldsymbol{x}_0 \in \mathbb{R}^m$. We want to study the behavior of the *orbit* of $\boldsymbol{x}_0$ under f, i.e. the sequence $(\boldsymbol{x}_n)_n$ of iterates defined by:

$$\boldsymbol{x}_n = f(\boldsymbol{x}_{n-1}) = f^n(\boldsymbol{x}_0) \quad (10)$$

Our aim is to determine whether the error on $\boldsymbol{x}_0$ as well as the errors arising during the calculation of $(\boldsymbol{x}_n)_n$ increase or decrease at each iteration. For example, let us consider the iterates of the function $g : x \mapsto x^2$ with x_0 in the vicinity of 1. Obviously, the sequence $(g^n(x_0))_n$ converges to 0 if $x_0 < 1$, is constant if $x = 1$ and goes to ∞ if $x > 1$. Moreover, for two distinct initial values $x_0 > 1$ and $x'_0 > 1$, $|g^n(x_0) - g^n(x'_0)| \to \infty$ as $n \to \infty$. Clearly, in a program computing $(g^n(x_0))_n$, a small error on the initial value x_0, possibly due to the use of floating-point numbers, may widely affect the final result. The map g is said to be *unstable* in the vicinity of 1.

Much work has been done on the theory of iterated function systems (IFS) [1, 3, 4, 14]. For multi-dimensional maps, the space is usually divided in two sets, the *Fatou set* and the *Julia set* respectively containing the points x_0 for which the sequences $(g^n(x_0))_n$ are stable and unstable [4]. For many maps, the limit between both zones is fractal. In this article, we do not directly study the stability of the map f corresponding to the body of a loop. Instead, we study the stability of the semantic equations generated from the loop with respect to the semantics of floating-point numbers with errors introduced in Section 2. In doing so, we not only compute the stability of the floating-point numbers returned by a program but also the stability of the errors made during the floating-point calculations. Let us consider again the function $g : x \mapsto x^2$ and the initial floating-point number with errors $x_0^\ell = 0.95\varepsilon + 0.1\varepsilon_\ell$. With the simple floating-point number $\hat{x}_0 = 0.95$, $(g^n(\hat{x}_0))_n \to 0$ and this sequence is stable in the immediate vicinity of this point. But if no error had been made in the calculation of $\hat{x}_0$, we would have $\hat{x}_0 = 1.05$ and $(g^n(\hat{x}_0))_n \to \infty$. So, the best information an analysis can give to the programmer is that if $x_0^\ell = 0.95\varepsilon + 0.1\varepsilon_\ell$ then the calculation of the iterates of g is stable (since the floating-point number goes to 0 in the vicinity of 0.95) but that the errors made during this calculation are unstable (they are arbitrarily large because of the initial 0.1 error).

The way this function f is generated from the syntax of the program is described in Section 4. For now on, we assume that f is given and we focus on determining its stability. In the particular case where $f : \mathbb{R}^m \to \mathbb{R}^m$ is a linear map (contrary to the example of g), the stability can be deduced from a calculation of its eigenvalues. If we assume that the matrix M of f has distinct real eigenvalues then M is similar to a diagonal matrix:

$$N = \begin{pmatrix} x_1 & & 0 \\ & \ddots & \\ 0 & & x_m \end{pmatrix} \tag{11}$$

and the n^{th} iterate f^n is $N^n = \begin{pmatrix} x_1^n & & 0 \\ & \ddots & \\ 0 & & x_m^n \end{pmatrix}$. Hence, the iterates of f are stable if $0 \leq |x_i| < 1$ for all $1 \leq i \leq m$. This last property still holds if f has repeated or complex eigenvalues. Here, a static analysis can use standard algorithms to compute an upper approximation of the eigenvalues (see Section 5.1). This approach was suggested by E. Goubault [9].

Unfortunately, the calculations carried out in programs are often non-linear (any multiplication between two non-constant terms is non-linear) and additional non-linearities are introduced in f by the functions modeling the propagation of errors. For instance, the higher-order terms are in general strongly non-linear. In addition, they may be non-negligible, even for simple examples:

$$\hat{g} \ : \ \begin{pmatrix} \hat{x} \\ e_x \\ e_{xh} \end{pmatrix} \mapsto \begin{pmatrix} \hat{x}^2 \\ 2xe_x \\ e_x^2 + e_{xh}^2 + 2\hat{x}e_{xh} + 2e_x e_{xh} \end{pmatrix} \tag{12}$$

$\hat{g}$ is a simplified version of the function generated by our method from a program implementing $g : x \mapsto x^2$. The real number x is divided into three parts, $\hat{x}$ for the floating-point number used instead of x, e_x and e_{xh} for the first-order and higher-order errors made on $\hat{x}$. $\hat{g}$ is obtained from g with $x = \hat{x} + e_x + e_{xh}$. The first dimension is related to the floating-point number, the second and third ones to the first and higher-order errors. Note that, for the sake of simplicity, some error terms are neglected in $\hat{g}$, e.g. the roundoff errors made during the calculations. The full function is introduced later on. Starting with $\hat{x} = 0.95$, $e_x = 0.1$ and $e_{xh} = 0$, one can easily check that the iterates go to $(0, 0, \infty)$. This result means that the floating-point calculations carried out inside the loop tend to 0, that the first-order errors also become null as n increases, but that the higher-order errors become arbitrarily large. Since the higher-order error only is significant, a similar analysis which neglected it would not detect any problem in this calculation.

As a consequence, we keep the non-linear model but there will no longer be a matrix N like the one of Equation (11). The role of the eigenvalues of N is taken by the Lyapunov exponents which measure the rates of separation of the iterates, starting from a point $x_0 \in \mathbb{R}^m$, in m orthogonal directions [1]. The intuition behind this concept is given in the next paragraph.

Let $f : \mathbb{R} \to \mathbb{R}$ and $x_0 \in \mathbb{R}$ such that $f(x_0) = x_0$. x_0 is an *attractive* (resp. a *repelling*) fixed point of f, if the first derivative satisfies $f'(x_0) < 1$ (resp. $f'(x_0) > 1$). If x_0 is repelling, an orbit of f starting from a point in the vicinity of x_0 will separate from x_0 with an approximate rate of $|f'(x_0)|$ by iteration. Otherwise the orbit is attracted by x_0. Now, if x_0 is a periodic point of f, i.e. $x_0 \neq f(x_0) \neq \ldots \neq f^k(x_0) = x_0$, x_0 is an attractive or a repelling periodic point of f depending on the value of $a = |f'(x_0)| \times |f'(x_1)| \times \ldots \times |f'(x_k)|$ with respect to 1. If x_0 is repelling then any orbit starting in a neighbor of x_0 will separate from the periodic orbit at each iteration with an average rate of $a^{\frac{1}{k}}$. Finally, for a non-periodic point x_0, the average rate of separation if given by $a = \lim_{k\to\infty}(|f'(x_0)| \times |f'(x_1)| \times \ldots \times |f'(x_k)|)^{\frac{1}{k}}$. The Lyapunov exponent of the orbit is defined as being the natural logarithm of the above quantity, i.e.

$$\lambda = \lim_{k\to\infty} \frac{\ln|f'(x_0)| + \ldots + \ln|f'(x_k)|}{k} \tag{13}$$

For a function $f : \mathbb{R}^m \to \mathbb{R}^m$, the eigenvalues of the Jacobian matrix Df are substituted to the values of the first derivative in the above notions and we obtain m exponents.

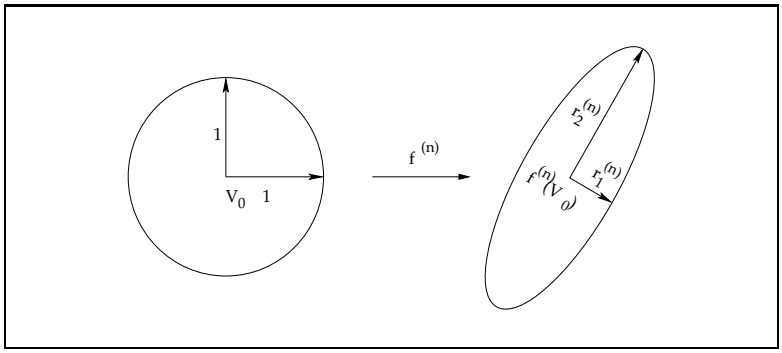

Fig. 2. Images of the unit sphere after n iterations.

Definition 1 *Let $f : \mathbb{R}^m \to \mathbb{R}^m$ and let e_i^n, $1 \leq i \leq m$ denote the i^{th} greatest eigenvalue of $Df^n(x_0)$. The i^{th} Lyapunov exponent is defined by*

$$\lambda_i = \lim_{n \to \infty} \frac{\ln e_i^n}{n}$$

f is *stable* in the vicinity of x_0 if the greatest Lyapunov exponent is a negative number. Otherwise the coefficients $\lambda_i > 0$ correspond to the unstable directions. $\lambda_i = 0$ indicates that there is a weak divergence (sub-exponential) in the i^{th} direction. In $\mathbb{R}^m$, to compute the Lyapunov exponents of $f : \mathbb{R}^m \to \mathbb{R}^m$ is equivalent to compute the images by f of an unit sphere U with center $\boldsymbol{x_0} \in \mathbb{R}^m$ [1]. As illustrated in Figure 2, we obtain a m-dimensional ellipsoid E. The lengths of the axes of E yield information on the stability of f. In practice, the images of U may evolve in three ways:

- The images of U tend towards the simple point p. f is contracting in any direction. p is an attracting fixed point of f, or a *sink*.
- The images of U are strictly growing in all directions. f is expansive in any direction and the related fixed point is a *source*.
- U becomes a very long and thin ellipsoid. f is contracting in some directions and expansive in others. The related fixed point is a *saddle*.

So, the Lyapunov exponents compute the rates of shrinking and stretching in the vicinity of the dynamics, starting from x_0. This leads us introduce an alternative definition of the Lyapunov exponents, equivalent to Definition 1.

Definition 2 *Let $f : \mathbb{R}^m \to \mathbb{R}^m$ and let r_i^n, $1 \leq i \leq m$, denote the length of the i^{th} longest orthogonal axis of the ellipsoid $Df^n(x_0)U$. The i^{th} Lyapunov exponent is defined by*

$$\lambda_i = \lim_{n \to \infty} \frac{\ln r_i^n}{n}$$

We have seen that the Lyapunov exponents can be defined in two equivalent ways, by means of the eigenvalues of the Jacobian matrix, or by means of the

images by f of the unit sphere. These two definitions yield two different algorithms, described in Section 5, to determine the stability of a loop. In the next section, we focus on the automatic generation of the Jacobian matrix needed by the stability test.

4 Generation of the Semantic Equations

As explained in Section 3, we determine the stability of the function f modeling the propagation of the errors inside a loop by means of the Lyapunov exponents. The exact definition of f stems from the semantics of Section 2. More precisely, the semantics is needed to build f from the syntax of the program and, next, we have to determine the Jacobian matrix Df. However, in practice, we directly build Df from the syntax of the program. The actual calculation, using Df, of the Lyapunov exponents is detailed in Section 5.

Let $r_1^{\ell_1} = f_1\varepsilon + \sum_{\ell\in\mathcal{L}} \omega_1^\ell\varepsilon_\ell$ and $r_2^{\ell_2} = f_2\varepsilon + \sum_{\ell\in\mathcal{L}} \omega_2^\ell\varepsilon_\ell$ be two floating-point numbers with errors occurring at the control points ℓ_1 and ℓ_2 of the program and let us focus on the operation $r_1^{\ell_1} \diamond^{\ell_3} r_2^{\ell_2}$. This operation sets the value $r_3^{\ell_3} = f_3\varepsilon + \sum_{\ell\in\mathcal{L}} \omega_3^\ell\varepsilon_\ell$ of point ℓ_3 to the result of $\diamond$ and can be viewed as a function of the variables $f_1, \omega_1^{\ell_1}, \ldots, \omega_1^{\ell_k}, \omega_1^{hi}$, $f_2, \omega_2^{\ell_2}, \ldots, \omega_2^{\ell_k}, \omega_2^{hi}$ and $f_3, \omega_3^{\ell_1}, \ldots, \omega_3^{\ell_k}, \omega_3^{hi}$, where $\mathrm{Card}(L) = k$ and $\mathcal{L}$ is the language defined in Section 2. For instance, from the equations of Figure 1, the function f_+ for an addition $r_1^{\ell_1} +^{\ell_3} r_2^{\ell_2}$ transforms the vector

$$\boldsymbol{v} = (f_1, \omega_1^{\ell_1}, \ldots, \omega_1^{\ell_k}, \omega_1^{hi}, f_2, \omega_2^{\ell_1}, \ldots, \omega_2^{\ell_k}, \omega_2^{hi}, f_3, \omega_3^{\ell_1}, \ldots, \omega_3^{\ell_k}, \omega_3^{hi})$$

into

$$\begin{array}{rl} f_+(\boldsymbol{v}) = (& f_1, \quad \omega_1^{\ell_1} \quad , \ldots, \quad \omega_1^{\ell_3} \quad , \ldots, \quad \omega_1^{hi}, \\ & f_2, \quad \omega_2^{\ell_1} \quad , \ldots, \quad \omega_2^{\ell_3} \quad , \ldots, \quad \omega_2^{hi}, \\ & \uparrow_\circ (f_1 + f_2), \omega_1^{\ell_1} + \omega_2^{\ell_1}, \ \ldots, \ \omega_1^{\ell_3} + \omega_2^{\ell_3} + \downarrow_\circ (f_1 + f_2), \ \ldots, \ \omega_1^{hi} + \omega_2^{hi}) \end{array}$$

The terms related to the operands are left unchanged and the coefficients of r_3 are assigned to the result of the addition. The Jacobian matrix of f_+ is denoted B^+ and is readily obtained from f_+. The derivative of the term $\downarrow_\circ (f_1 + f_2)$ with respect to f_1 or f_2 is $ulp(1)$[2], since for any floating-point number x we have $\downarrow_\circ (x) \approx x \cdot ulp(1)$. For instance we have:

$$B^+ = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ B^+_{\ell_3,\ell_1} & B^+_{\ell_3,\ell_2} & 0 \end{pmatrix} \tag{14}$$

[2] The unit in the last place (ulp) of a floating-point number is the magnitude of the least significant digit of the mantissa of a floating-point number [7].

with

$$B^{+}_{\ell_3,\ell_1} = \; ; \begin{array}{c} \\ f_3 \\ \omega_3^{\ell_1} \\ \vdots \\ \omega_3^{\ell_3} \\ \vdots \\ \omega_3^{\ell_n} \\ \omega_3^{\ell_{hi}} \end{array} \begin{array}{c} \begin{array}{ccccccc} \frac{\partial}{\partial f_1} & \frac{\partial}{\partial w_1^{\ell_1}} & \cdots & \frac{\partial}{\partial w_1^{\ell}} & \cdots & \frac{\partial}{\partial w_1^{\ell_k}} & \frac{\partial}{\partial w_1^{hi}} \end{array} \\ \begin{pmatrix} 1 & 0 & \ldots & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ddots & & & \vdots & 0 \\ 0 & 0 & \ddots & \ddots & & \vdots & \vdots \\ ulp(1) & \vdots & \ddots & 1 & \ddots & \vdots & 0 \\ 0 & \vdots & & \ddots & \ddots & 0 & \vdots \\ 0 & \vdots & & & \ddots & 1 & 0 \\ 0 & 0 & \ldots & 0 & \ldots & 0 & 1 \end{pmatrix} \end{array} \tag{15}$$

$B^{+}_{\ell_3,\ell_2}$ is defined similarly and I denotes the identity block. The Jacobian matrix $B^{\times}$ for a multiplication is obtained in the same way from the function

$$\begin{array}{rl} f_{\times}(\boldsymbol{v}) = (& f_1, \quad \omega_1^{\ell_1} \quad , \ldots , \quad \omega_1^{\ell_3} \quad , \ldots , \omega_1^{hi}, \\ & f_2, \quad \omega_2^{\ell_1} \quad , \ldots , \quad \omega_2^{\ell_3} \quad , \ldots , \omega_2^{hi}, \\ & \uparrow_{\circ}(f_1 \times f_2),\ f_1\omega_2^{\ell_1} + f_2\omega_1^{\ell_1},\ \ldots,\ f_1\omega_2^{\ell_3} + f_2\omega_1^{\ell_3} + \downarrow_{\circ}(f_1 \times f_2),\ \ldots,\ \omega_3^{hi}) \end{array}$$

in which the higher-order term $\omega_3^{\ell_{hi}} = f_1\omega_2^{\ell_{hi}} + f_2\omega_1^{\ell_{hi}} + \sum_{\ell,\ell' \in \mathcal{L}^+} \omega_1^{\ell}\omega_2^{\ell'}$. The Jacobian matrix $B^{\times}$ is built is the same way to B^{+} in Equation (14). For example, we obtain the block:

$$B^{\times}_{\ell_3,\ell_1} = \; ; \begin{array}{c} \\ f_3 \\ \omega_3^{\ell_1} \\ \vdots \\ \omega_3^{\ell_3} \\ \vdots \\ \omega_3^{\ell_n} \\ \omega_3^{\ell_{hi}} \end{array} \begin{array}{c} \begin{array}{ccccccc} \frac{\partial}{\partial f_1} & \frac{\partial}{\partial w_1^{\ell_1}} & \cdots & \frac{\partial}{\partial w_1^{\ell}} & \cdots & \frac{\partial}{\partial w_1^{\ell_k}} & \frac{\partial}{\partial w_1^{\ell_{hi}}} \end{array} \\ \begin{pmatrix} f_2 & 0 & \ldots & \ldots & \ldots & 0 & 0 \\ \omega_2^{\ell_1} & f_2 & \ddots & & & \vdots & 0 \\ \vdots & 0 & \ddots & \ddots & & \vdots & \vdots \\ U & \vdots & \ddots & f_2 & \ddots & \vdots & 0 \\ \vdots & \vdots & & \ddots & \ddots & 0 & \vdots \\ \omega_2^{\ell_n} & 0 & \ldots & \ldots & 0 & f_2 & 0 \\ \omega_2^{\ell_{hi}} & S & \ldots & S & \ldots & S & S' \end{pmatrix} \end{array} \tag{16}$$

where

$$U = \omega_2^{\ell_2} + f_2 ulp(1) \qquad S = \sum_{\ell \in \mathcal{L}^+} \omega_2^{\ell} \qquad S' = S + f_2$$

Given a piece of code p corresponding to the body of a loop and the language $\mathcal{L}$ built on the labels of p, the Jacobian matrix M of the semantic function defined from p is built in blocks. For example, the operation $r_1^{\ell_1} \times^{\ell_3} r_2^{\ell_2}$ makes the system fill the blocks corresponding to the partial derivative of the variables of r_3 by the variables of r_1 and r_2. Roughly speaking, if $\frac{\partial r^{\ell_3}}{\partial r^{\ell_1}}$ and $\frac{\partial r^{\ell_3}}{\partial r^{\ell_2}}$ denote blocks of

the full matrix, the multiplication leads us to fill M as follows:

$$M = \begin{array}{c} \\ \vdots \\ r^{\ell_1} \\ \vdots \\ r^{\ell_2} \\ \vdots \\ r^{\ell_3} \\ \vdots \end{array} ; \begin{array}{c} \cdots \quad \frac{\partial}{\partial r^{\ell_1}} \quad \cdots \quad \frac{\partial}{\partial r^{\ell_2}} \quad \cdots \quad \frac{\partial}{\partial r^{\ell_3}} \quad \cdots \\ \left(\begin{array}{ccccccc} . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & B^{\times}_{\ell_3,\ell_1} & . & B^{\times}_{\ell_3,\ell_2} & . & . & . \\ . & . & . & . & . & . & . \end{array} \right) \end{array} \tag{17}$$

From a formal point of view, let $\sigma : \mathrm{Id} \to \mathcal{L}$ denote an environment $\sigma \in$Env mapping variables to labels and let $M \in$Mat denote a Jacobian matrix. For an expression e^{ℓ}, M is set to the new value $\mathcal{E}(e^{\ell}, M, \sigma)$ by the function $\mathcal{E} : \mathrm{Expr} \times \mathrm{Mat} \times \mathrm{Env} \to \mathrm{Mat}$ defined by:

$$\mathcal{E}(c^{\ell}, M, \sigma) = M$$

$$\mathcal{E}(x^{\ell}, M, \sigma) = M + I_{\ell,\sigma(x)}$$

$$\mathcal{E}\big((e_1^{\ell_1} \diamond^{\ell} \varepsilon_2^{\ell_2}), M, \sigma\big) = \mathcal{E}\big(e_2^{\ell_2}, \mathcal{E}(e_1^{\ell_1}, M, \sigma), \sigma\big) + B^{\diamond}_{\ell,\ell_1} + B^{\diamond}_{\ell,\ell_2}$$

A constant c does not modify M. I_{ℓ_1,ℓ_2} denotes the identity block set at position $\frac{\partial r^{\ell_1}}{\partial r^{\ell_2}}$. If $\sigma(x) = \ell_2$ then an occurrence of x^{ℓ_1} makes us insert the block I_{ℓ_1,ℓ_2} in the matrix M. For an operation $\diamond$, M is set as described in Equation (17). For a piece of code p related to the body of a loop, the matrix M is generated by the function $\mathcal{P} : \mathrm{Prg} \times \mathrm{Mat} \times \mathrm{Env} \to \mathrm{Mat} \times \mathrm{Env}$ (partly) defined below.

$$\mathcal{P}(x^{\ell_1} = e^{\ell_2}, M, \sigma) = \big(\mathcal{E}(e^{\ell_2}, M, \sigma) + I_{\ell_1,\ell_2} \ , \ \sigma[x \mapsto \ell_2]\big)$$

$$\mathcal{P}\ (p_1; p_2, M, \sigma) = \big(\lambda(M', \sigma').\mathcal{P}(p_2, M', \sigma')\big)\ \mathcal{P}(p_1, M, \sigma)$$

The full matrix for the body p of a loop is generated by $\mathcal{P}(p, O, [])$ where O denotes the null matrix and [] the empty environment. For example, the matrix obtained for the program below is:

```
while (x! = 0){
    p : x^{ℓ4} = x^{ℓ1} ×^{ℓ3} x^{ℓ2};
}
```

$$\mathcal{P}(p, O, []) = \begin{array}{c} \\ r^{\ell_1} \\ r^{\ell_2} \\ r^{\ell_3} \\ r^{\ell_4} \end{array} ; \begin{array}{c} \frac{\partial}{\partial r^{\ell_1}} \quad \frac{\partial}{\partial r^{\ell_2}} \quad \frac{\partial}{\partial r^{\ell_3}} \quad \frac{\partial}{\partial r^{\ell_4}} \\ \left(\begin{array}{cccc} 0 & 0 & 0 & I \\ 0 & 0 & 0 & I \\ B^{\times}_{\ell_3,\ell_1} & B^{\times}_{\ell_3,\ell_2} & 0 & 0 \\ 0 & 0 & I & 0 \end{array} \right) \end{array}$$

5 Abstract Calculation of the Lyapunov Exponents

In this section, we present two algorithms to prove the stability of the floating-point calculations carried out inside a loop. The algorithm of Section 5.1 is based on Definition 1. It involves less calculation than the second algorithm but only globally checks that no error term diverges. The algorithm of Section 5.2 is based on Definition 2 and precisely determines which error terms make the loop unstable.

5.1 Global Stability of a Loop

We show how to use the Jacobian matrices generated in Section 4 to determine the stability of the calculations made inside a loop. The analysis is based on an abstract interpretation of the semantics of Section 2, in which the floating-point numbers and the error terms are abstracted by intervals of floating-point numbers. Hence, if the stability of a loop is assessed, we can ensure that any actual execution based on a concretization of the abstract values is stable.

As explained in Section 3, the Lyapunov exponents are used to determine the stability of an orbit defined by the initial point $x_0 \in \mathbb{R}^m$ and the iterates of $f : \mathbb{R}^m \to \mathbb{R}^m$. From Definition 1, the exponents are computed from the eigenvalues of the Jacobian matrix $Df^n(x_0)$ of the n^{th} iterate of f. Recall that the calculation is stable is all the exponents are negative. Our test is based on an abstract interpretation of the semantics of Section 2 in which the coefficients of the error terms are intervals. $Df^n(x_0)$ is computed as follows. Since $Df^n(x_0) = D(f(f^{n-1}(x_0)))$, we have

$$\begin{aligned} Df^n(x_0) &= Df^{n-1}(x_0) \times Df(f^{n-1}(x_0)) \\ &= Df^{n-1}(x_0) \times Df(x_{n-1}) \end{aligned}$$

Finally, by recurrence, $Df^n(x_0) = Df(x_{n-1}) \times \ldots \times Df(x_0)$. So, we focus on the product of interval matrices $Df^n(x_0) = Df(x_{n-1}) \times \ldots \times Df(x_0)$ whose eigenvalues indicate the stability of the first n iterates. This approach leads us to evaluate (by abstract interpretation) the first n iterates of the loop in order to compute $x_0,\ x_1 = f(x_0), \ldots, x_n$. At each step k, the matrix Df, generated from the syntax of the program by the method of Section 4 and which is composed of formal expressions, is valued, yielding $Df(x_k)$, and the new product $Df^{k+1}(x_0) = Df(x_k) \times \ldots \times Df(x_0)$ is computed.

$Df^k(x_0)$ is a matrix of intervals and we use the Gerschgorin disks [5, 9] to find an upper bound of the greatest module of its eigenvalues, as explained below. For a matrix A of size $m \times m$, the eigenvalues are contained in $D_1 \cap D_2$, where D_1 and D_2 are the disks of the complex plane $\mathbb{C}$ defined by:

$$D_1 = \cup_{1 \leq i \leq m} D_{1,i} \qquad\qquad D_2 = \cup_{1 \leq j \leq m} D_{2,j}$$

$D_{1,i}$ is the disk of center a_{ii} and radius $r_{1,i} = \sum_{1 \leq j \leq m, j \neq i} |a_{ij}|$. Similarly, $D_{2,j}$ is the disk of center a_{jj} and radius $r_{2,j} = \sum_{1 \leq i \leq m, i \neq j} |a_{ij}|$. So, an upper bound of the module of the greatest eigenvalues of A is given by:

$$G(A) = \max\big(|a_{ii}| + \max(r_{1,i}, r_{2,i}) \ : \ 1 \leq i \leq m\big)$$

From Definition 1, the greatest Lyapunov exponent after k iterations is bounded by the abstract Lyapunov exponent:

$$\lambda_k^\sharp = \frac{\ln\big(G(Df^k(x_0))\big)}{k} \tag{18}$$

Let k_0 be the first iteration of the algorithm such that $\lambda^\sharp < 0$, let f denote the semantic function of the body of the loop and let f^k be the function f

iterated k times. The algorithm stops after k_0 iterations, i.e. as soon as $\lambda^\sharp < 0$. This ensures that f^{k_0} is a contracting function. Consequently, $f^{k_0}(x_0) \prec x_0$, where $\prec$ is the componentwise ordering used for the abstract domain of Section 2 [13]. So, any error term is decreasing after k_0 iterations and $f^{k_0}(f^{k_0}(x_0)) \prec f^{k_0}(x_0) \prec x_{k_0}$, etc. Note that this condition does not imply that the error terms are always decreasing between the iterations nk_0 and $(n+1)k_0$ for any integer n. However, it implies that no error term significantly diverges since the function becomes contracting after some more iterations. Obviously, one can also force the algorithm to stop after a predefined number of iterations k_{max} and return the conservative result that the loop is unstable if the termination condition is not satisfied. To sum up, the algorithm we use is given below.

while $\lambda^\sharp >= 0$ do

1. Unfold the loop once: $\rho^\sharp = [\![p]\!]^\sharp\ \rho^\sharp$; $k = k + 1$
2. Valuate Df: $\Delta = Df(\rho^\sharp)$
3. Compute the new Jacobian Df^k: $Df^k = \Delta \times Df^{k-1}$
4. Approximate the greatest eigenvalue: $e = \mathrm{G}(Df^k)$
5. Compute the Lyapunov exponent after k iterations: $\lambda^\sharp = \frac{\ln(e)}{k}$

We end this section by showing the results obtained with an implementation of this method. The top two curves of Figure 3 correspond to the loop whose body is `x = x*x`. The x-axis corresponds to the iterations k, while the y-axis corresponds to the upper approximation $\lambda_k^\sharp$ of the Lyapunov exponents. The first case is stable, since the initial value of `x` is $[0.0, 0.95] + [0.0, 0.01]\varepsilon_x$. $\lambda_k^\sharp$ becomes negative after a number of iterations, indicating that the loop is stable. In the second case we have `x`$= [0.0, 0.95] + [0.0, 0.1]\varepsilon_x$. Even if the calculation carried out with floating-point numbers is stable and converges to 0, the initial real number can be as great as 1.05 and the errors made during the execution of the loop may be unbounded. We can see on the top right curve of Figure 3 that $\lambda_k^\sharp$ is positive and increasing. Our second example concerns a classical numerical algorithm, Jacobi's iterative method to solve a system of linear equations. We consider the systems:

$$(S_1)\ :\ \begin{cases} 2x + 3y = 3 \\ 4x + \frac{3}{2}y = 3 \end{cases} \qquad (S_2)\ :\ \begin{cases} 2x + y = \frac{5}{3} \\ x + 3y = \frac{5}{2} \end{cases}$$

To solve (S_1) and (S_2) by Jacobi's method [8], the following sequences must be computed:

$$(S_1)\ :\ \begin{cases} x_{n+1} = \frac{3}{2} - \frac{3}{2}y_n \\ y_{n+1} = 2 - \frac{8}{3}x_n \end{cases} \qquad (S_2)\ :\ \begin{cases} x_{n+1} = \frac{5}{6} - \frac{1}{2}y_n \\ y_{n+1} = \frac{5}{6} - \frac{1}{3}x_n \end{cases}$$

The solution to both (S_1) and (S_2) is $x = \frac{1}{2}$ and $y = \frac{2}{3}$, but (S_1) is unstable for Jacobi's method while (S_2) is stable[3]. Indeed, a C program implementing (S_1) in double precision, compiled with GCC and executed on a Pentium III processor

[3] A sufficient condition to the stability of Jacobi's method for a linear system of equations is that $|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|$, where the a_{ij} are the coefficients of the variables [8].

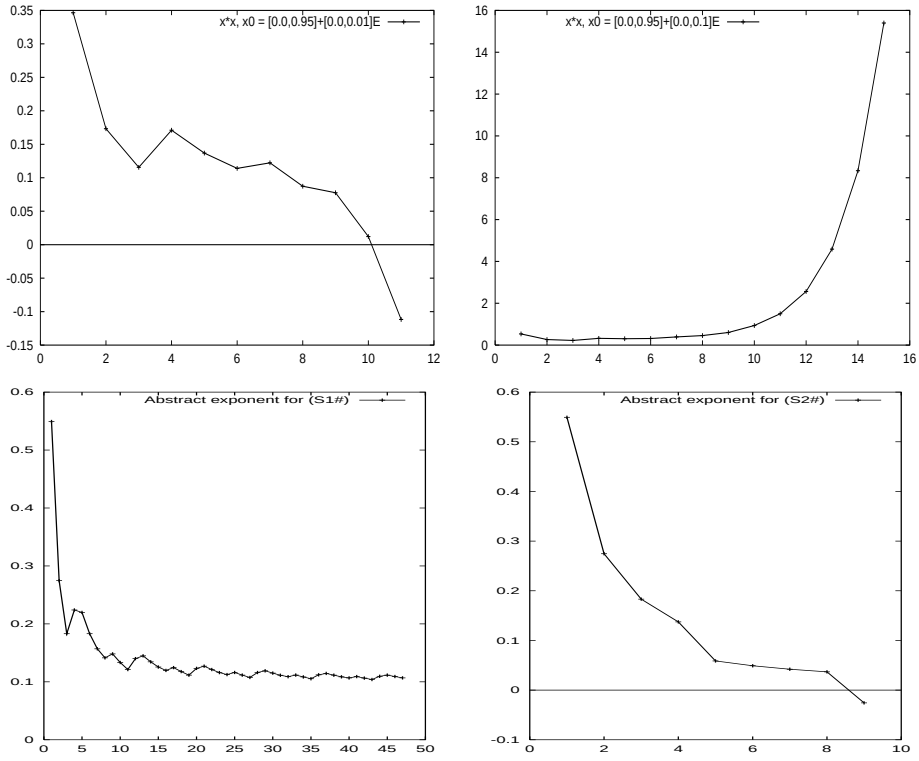

Fig. 3. Exponents $\lambda^\sharp$ computed by our implementation of the first algorithm for the examples described in the text.

yields after 15 iterations $x = 19408738461538.50$, $y = 13275241025642.0$ if $x_0 = y_0 = 3$. Under the same assumptions, our implementation of (S_2) yields after 15 iterations $x = 0.500001$ and $y = 0.666668$. The bottom curves of Figure 3 show the values of the abstract Lyapunov exponents computed by our implementation for the systems:

$$(S_1^\sharp) \; : \; \begin{cases} x_{n+1} = [1.2, 1.6] - [1.2, 1.6]y_n \\ y_{n+1} = [2.0, 2.5] - [2.5, 3.0]x_n \end{cases} \qquad (S_2^\sharp) \; : \; \begin{cases} x_{n+1} = [0.80, 0.85] - [0.4, 0.6]y_n \\ y_{n+1} = [0.80, 0.85] - [0.30, 0.35]x_n \end{cases}$$

The initial values are $x_0^\sharp = y_0^\sharp = [2.0, 3.0]$. The curve corresponding to $(S_2^\sharp)$ becomes negative after a number of iterations, indicating that any actual execution of this program using concrete values in the given intervals is stable. The curve related to the unstable system $(S_1^\sharp)$ remains positive, which means that the calculations carried out in the program are unstable. Finally, let us remark that, for (S_2), the function evaluated in the loop is a linear map defined by the matrix $M = \begin{pmatrix} \frac{5}{6} & \frac{-1}{2} \\ \frac{5}{6} & \frac{-1}{3} \end{pmatrix}$ and is stable. However, $G(M) = \frac{5}{3}$ and the technique introduced in ref. [9] to determine the stability of a linear loop by the test $G(M) < 1$ also

requires the unfolding of the loop. The abstract Lyapunov exponent test $\lambda^\sharp < 0$ is also verified after some iterations.

5.2 Calculation of Every Exponent

The algorithm of Section 5.1 enables the global determination of the stability of the calculations carried out inside a loop. However, if the loop is unstable, it does not help to understand the source of the imprecision, i.e. the error terms which do not converge. In this section, we introduce a second algorithm, more complex than the previous one, to compute the full sequence of Lyapunov exponents.

Let f be the semantic function related to the body of a loop. By definition 2, we directly compute the images by f of the unit sphere U [1]. Again, the calculation are based on an abstract version of the semantics in which the coefficients of the series are intervals. After k iterations, we obtain an ellipsoid E_k whose axes indicate the contracting or expanding nature of f in a given direction. More precisely, we aim at computing $Df^k(x_0)U = Df(x_{k-1}) \ldots Df(x_0)U$. Let $(e_1^0, e_2^0, \ldots, e_m^0)$ be an orthonormal basis of $\mathbb{R}^m$, we compute

$$z_1 = Df(x_0)e_1^0, \ldots, z_m = Df(x_0)e_m^0 \tag{19}$$

The vectors $(z_1, \ldots, z_m)$ are axes of the ellipsoid $Df(x_0)U$ but they are not necessarily orthogonal. So, the next step consists of computing a new set $(y_1, \ldots, y_m)$ of orthogonal vectors which define an ellipsoid with the same volume as $Df(x_0)U$. In the current implementation, this is done by the Gram-Schmidt procedure [15] but more sophisticated algorithms shXould be used. The next iteration is carried out with the new set of vectors

$$e_1^1 = y_1, e_2^1 = y_2, \ldots, e_m^1 = y_m \tag{20}$$

and, after k iterations, we obtain an ellipsoid whose axes are $(e_1^k, e_2^k, \ldots, e_m^k)$. Following the definitions of Section 3, the total expansion r_i^k in the i^{th} direction is given by e_i^k and the related abstract Lyapunov exponent is $\lambda_i^\sharp = \frac{\ln r_i^k}{k}$. Finally, since the ellipsoid may have long and short axes after some iterations, we normalize the basis at each step. Equation (20) becomes

$$e_1^k = \frac{y_1^k}{\|y_1^k\|}, \ldots, e_m^k = \frac{y_m^k}{\|y_m^k\|} \tag{21}$$

where y_i^k denotes the vector y_i at iteration k. The i^{th} axis r_i^k of the ellipsoid after k iterations has length $r_i^k = \|y_i^k\| \ldots \|y_i^1\|$, so the i^{th} abstract Lyapunov exponent after k iterations is

$$\lambda_i^{\sharp k} = \frac{\ln\|y_i^k\| + \ldots + \ln\|y_i^1\|}{k} \tag{22}$$

The algorithm is summarized below. The termination is managed in the same way as for the first algorithm of Section 5.1.

let E be an orthonormal basis of $\mathbb{R}^m$
while $\exists i,\ 1 \leq i \leq m\ :\ \lambda_i >= 0$ do

1. Unfold the loop once: $\rho^\sharp = [\![p]\!]^\sharp\ \rho^\sharp$; $k = k + 1$
2. Valuate Df: $\Delta = Df(\rho^\sharp)$
3. Compute the axes Z of the new ellipsoid: $Z = \Delta \times E$
4. Orthogonalize Z: $Y = (y_1, \ldots, y_m) = \text{Gram-Schmidt}(Z)$
5. Normalize Y: $E = (\frac{y_1}{\|y_1\|}, \ldots, \frac{y_m}{\|y_m\|})$
6. For $1 \leq i \leq m$, compute the new exponents by Equation (22), using Y

We end this section by presenting some experimental results provided by an implementation of this algorithm. The top two curves of Figure 4 show the values of the abstract Lyapunov exponents at each iteration for a loop computing `x=x*x`. The top left curve corresponds to a stable case in which the initial abstract value of `x` is $[0.0, 0.95] + [0.0, 0.01]\varepsilon_x$. The values of the abstract exponents related to the first-order error on `x` and to the higher-order error on `x` are plotted and we can observe that both become negative after a number of iterations, indicating that the program is stable in this case. The top right curve corresponds to an unstable case for the same program since the initial value of `x` is $[0.90, 0.95] + [0.0, 0.1]\varepsilon_x$. Here, the only value of the abstract exponent related to the first-order error term becomes negative after a number of iterations. The algorithm detects that there is an instability due to the the higher-order error on `x`, as predicted by the theory (see Section 3). Our second example concerns the system of linear differential equations with initial conditions:

$$(S)\ :\ \begin{cases} \dot{x} &= -2x \\ \dot{y} &= x + 3y \end{cases} \qquad x(0) = 1.0, \quad y(0) = 2.0 \tag{23}$$

The exact solution is $x(t) = x(0)e^{-2t}$, $y(t) = -\frac{1}{5}x(0)e^{-2t} + (y(0) - x(0))e^{3t}$. The solution $x(t)$ is insensitive to a small variation of $x(0)$ since this term is multiplied by e^{-2t} but $y(t)$ is sensitive to $x(0)$ and $y(0)$. The algorithm introduced in this section enables to detect this phenomenon in a program computing a numerical solution to (S) by Euler's method [15]. Let us consider the sequence:

$$\begin{cases} x_{n+1} &= x_n - 2hx_n \\ y_{n+1} &= y_n + hx_n + 3hy_n \end{cases} \qquad x_0 = 1.0, \quad y_0 = 2.0 \tag{24}$$

where the step $h = 0.1$. For the abstract initial condition $x_0 = [1.0, 1.0] + [0.0, 0.1]\varepsilon_{x_0}$ and $y_0 = [2.0, 2.0] + [0.0, 0.1]\varepsilon_{y_0}$, the abstract Lyapunov exponents computed by our algorithm are given in Figure 4. The left bottom curve shows the values of the exponent related to the first-order error on x_n. Since it is negative, we conclude that the sequence $(x_n)_n$ is stable. Conversely, the values of the exponent related to the first-order error on y_n, given in the right bottom curve, are positive, indicating that this calculation is unstable.

6 Conclusion

In this article, we introduced a static test to determine whether the calculations carried out in a loop are stable. This test is based on an abstract calculation of

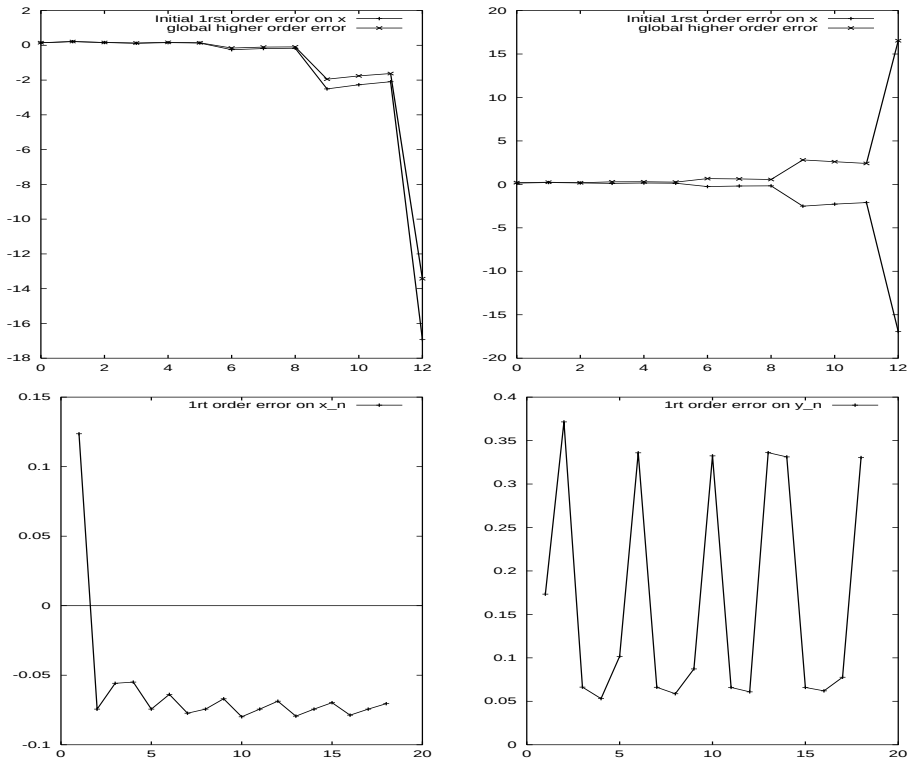

Fig. 4. Exponents $\lambda^\sharp$ computed by our implementation of the second algorithm for the examples described in the text.

the Lyapunov exponents for the semantic function defined by the body of a loop and the semantics of floating-point numbers with errors [9, 13]. In Section 5 we introduced two algorithms, to test that a loop is globally stable and to determine which operations of a loop are unstable. We used classic numerical methods to demonstrate the usefulness of both algorithms.

A first advantage of our approach is that non-linear terms as well as higher-order error terms are fully analyzed. Not to accept non-linear terms, as proposed by some (dynamic) approaches [12, 16], is an important restriction on the accepted codes and to neglect higher-order errors may lead to erroneous conclusions. A second advantage is that the stability test does not depend on parameters of the analysis. The point is that, with the usual widening operators, to precisely analyze even a stable calculation (for example the system (S_2) of Section 5.1) the loop must be unfolded a finite but unknown number of times and the unfolded code must be executed another finite but unknown number of times before widening [10]. So, if a loop is declared unstable by the analyzer, there is no way of knowing whether this is due to a bad parameter value or to

an actual numerical problem. The evolution of the abstract Lyapunov exponent, as shown in the figures 3 and 4 yields far more information.

The current implementation, described in Section 5 could be significantly improved if more elaborate numerical algorithms were used. The matrices generated by the method of Section 4 contain few non-zero elements and a sparse representation would be well-suited. For the second algorithms, a more accurate orthogonalization procedure should be used. Finally, we are investigating other ways to improve the treatment of loops in the analyzer described in ref. [10]. We believe that by mixing classic loop unfolding with a dynamic repartitioning of the control points, we could assess the stability of many new programs. This technique should complement the one introduced in this article, each being better-suited to some kinds of codes.

References

1. K. T. Alligood, T. D. Sauer, and J. A. Yorke. *Chaos, an Introduction to Dynamical Systems.* Springer-Verlag, 1996.
2. ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-1985 edition, 1985.
3. M. F. Barnsley. *Fractals Everywhere, Second Edition.* Academic Press, 1993.
4. A. F. Beardon. *Iteration of Rational Functions.* Number 132 in Graduate Texts in Mathematics. Springer-Verlag, 1991.
5. F. Chatelin. *Valeurs propres des matrices.* Masson, 1988.
6. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Symbolic Computation*, 2(4):511–547, 1992.
7. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.
8. G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 2d edition, 1990.
9. E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
10. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
11. D. Knuth. *The Art of Computer Programming - Seminumerical Algorithms.* Addison Wesley, 1973.
12. P. Langlois and F. Nativel. Improving automatic reduction of round-off errors. In *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 2, 1997.
13. M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
14. F. C. Moon. *Chaotic and Fractal Dynamics.* Wiley-Interscience, 1992.
15. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.
16. J. Vignes. A survey of the CESTAC method. In *Proceedings of Real Numbers and Computer Conference*, 1996.