Computational algorithms for ordinary differential equations

–Revised October 19, 1999–

Warwick Tucker
IMPA, Est. D. Castorina 110
Jardim Botânico, 22460-320
Rio de Janeiro, RJ, Brazil
warwick@impa.br

October 14, 1999

Abstract

We present an algorithm for computing rigorous solutions to a large class of ordinary differential equations. The main algorithm is based on a partitioning process and the use of interval arithmetic. We illustrate the presented method by computing solution sets for two explicit systems.

1 Introduction

In this paper, we will consider a general initial value problem:

$$\dot{x} = f(x); \qquad x(0) = x_0, \tag{1}$$

where $f \in C^1(\mathcal{D}, \mathbb{R}^n)$, and $\mathcal{D} \subseteq \mathbb{R}^n$. We will sometimes denote the solution of (1) by $\varphi(x,t)$, with $\varphi(x,0) = x(0)$. This setting is classical, and much studied in standard text books on ordinary differential equations. It is, however, not difficult to find situations where having a whole set of initial values is natural. Indeed, any model of a physical system always has some uncertainty concerning the measured initial values. Furthermore, we are seldom sure of exactly which vector field models our system. The natural thing do is to enclose the initial value x_0 in a box $[x_0]$ whose side lengths reflect the maximal error made in the measurements of the initial data, and to replace f in (1) by a function F, whose components are interval valued. The problem we then face is to find the solution of the following system:

$$\dot{x} \in F([x]); \qquad x(0) \in [x_0], \tag{2}$$

Our objective is to compute a set that is guaranteed to contain all the solutions of (2) at a given time T. The method we present is based on a partitioning process, which will be presented in more detail below.

2 Interval arithmetic

In this section, we will briefly describe the fundamentals of interval arithmetic. For a concise reference on this topic, see [2].

Let $\mathfrak{I}$ denote the set of closed intervals. For any element $[a] \in \mathfrak{I}$, we adapt the notation $[a] = [\underline{a}, \bar{a}]$. If $\odot$ is one of the operators $+, -, \times, \div$, we define arithmetic operations on elements of $\mathfrak{I}$ by

$$[a] \odot [b] = \{a \odot b \colon a \in [a], b \in [b]\},$$

except that $[a]/[b]$ is undefined if $0 \in [b]$. Working exclusively with closed intervals, we can describe the resulting interval in terms of the endpoints of the operands:

$$\begin{aligned}
[a] + [b] &= [\underline{a} + \underline{b}, \bar{a} + \bar{b}] \\
[a] - [b] &= [\underline{a} - \bar{b}, \bar{a} - \underline{b}] \\
[a] \times [b] &= [\min(\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}), \max(\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b})] \\
[a] \div [b] &= [a] \cdot [1/\bar{b}, 1/\underline{b}], \quad \text{if } 0 \notin [b].
\end{aligned}$$

To increase speed, it is customary to break the formula for multiplication into nine cases (depending of the signs of the endpoints), where only one case involves more than two multiplications. Moreover, the formula for division can be modified for improved accuracy. When computing with finite precision, directed rounding must also be taken into account, see e.g. [2], [3], [1].

It follows immediately from the definitions that addition and multiplication are both associative and commutative. The distributive law, however, does *not* always hold. As an example, we have

$$[-1, 1]([-1, 0] + [3, 4]) = [-1, 1][2, 4] = [-4, 4]$$

whereas

$$[-1, 1][-1, 0] + [-1, 1][3, 4] = [-1, 1] + [-4, 4] = [-5, 5].$$

This unusual property is important to keep in mind when representing functions as part of a program. Interval arithmetic satisfies a weaker rule than the distributive law, which we shall refer to as *sub-distributivity*:

$$[a]([b] + [c]) \subseteq [a][b] + [a][c].$$

Another key feature of interval arithmetic is that it is *inclusion monotonic*, i.e., if $[a] \subseteq [a']$, and $[b] \subseteq [b']$, then

$$[a] \odot [b] \subseteq [a'] \odot [b'],$$

where we demand that $0 \notin [b']$ for division.

We can turn $\mathfrak{I}$ into a metric space by equipping it with the Hausdorff distance:

$$d([a], [b]) = \max\{|\underline{a} - \underline{b}|, |\bar{a} - \bar{b}|\}.$$

For dealing with higher dimensional problems, we define the arithmetic operations to be carried out component-wise. We then talk about an *interval vector* or, more simply, a *box*. The metric is then defined by

$$d([a], [b]) = \max_{1 \le i \le n} \{d([a_i], [b_i])\}.$$

Matrix operations are defined analogously to the real case.

3 Interval-valued functions

Consider a function $f \in C^1(\mathcal{D}, \mathbb{R}^n)$, where $\mathcal{D} \subseteq \mathbb{R}^n$. Given a box $[a]$ we define the *range* of f over $[a]$ by

$$R(f;[a]) = \{f(x) : x \in [a]\}.$$

If we fix a representation of f (which we also denote f), and evaluate it in interval arithmetic, we always have

$$R(f;[a]) \subseteq f([a]),$$

due to the inclusion monotonic property. From this property, it also follows that by splitting the box $[a]$ into smaller pieces $[a_0], \dots, [a_n]$, we have

$$R(f;[a]) \subseteq \bigcup_{i=0}^{n} f([a_i]) \subseteq f([a]).$$

It is clear that, by splitting $[a]$ into many small pieces, we can approximate the true range of f over $[a]$ with any desired accuracy. There are, however, better ways to approximate the range of f: let $m([a])$ denote the midpoint of $[a]$. By the Mean Value Theorem, we have the following relation:

$$R(f;[a]) \subseteq f_{MV}([a]) := f(m([a])) + [Df]([a])([a] - m([a])).$$

Let $\|[a]\|$ denote the maximal diameter of $[a]$. It is easy to show that

$$d(R(f;[a]), f([a])) = \mathcal{O}(\|[a]\|),$$

whereas

$$d(R(f;[a]), f_{MV}([a])) = \mathcal{O}(\|[a]\|^2).$$

It is obvious that the latter version is preferred, seeing that we have a quadratically small error. This assumes, however, that we only deal with intervals of small widths. The most fundamental part of our algorithm – the partitioning process – guarantees that this indeed will be the case, and thus allows us to attain a quadratic approximation of the vector field range $R(f;[a])$.

As mentioned earlier in the introduction, it is often desirable in applications to exchange the function f for its *interval extension* F. Given a finite representation of f, we define F to be any function having the same representation as f, except that all real coefficients are replaced by enclosing intervals. As an example, given $f(x,y) = 2x - \pi y$, we may take $F(x,y) = [1.99, 2.01]x - [3.14, 3.15]y$.

4 Algorithms

In this section, we will present some algorithms for rigorously solving an initial value problem. We will start with the most basic approach, using the Euler method.

The solution of (1) is formally given by

$$\varphi(x, t_{i+1}) = \varphi(x, t_i) + \int_{t_i}^{t_{i+1}} f(\varphi(x,s))ds, \tag{3}$$

where $\varphi(x, t_0) = x_0$. Approximating the integrand in (3) by $f(\varphi(x, t_i))$, we arrive at the classical Euler method, which gives the iterative scheme

$$x_{i+1} = x_i + \Delta t_i f(x_i) \qquad i \geq 0$$

for an approximate solution to (1), i.e., $x_i \approx \varphi(x, t_i)$. Here we have used the notation $\Delta t_i = t_{i+1} - t_i$. The error we are making is in assuming that the vector field f is constant over each time step. With interval arithmetic this can be overcome by using the following algorithm (in which we have omitted the stopping condition for clarity):

Algorithm 1. *For $i \geq 0$ do the following:*

1. *Enclose the computed solution at step i in a box: $[x_i] \subset [\tilde{x}_i]$;*
2. *Compute a time step Δt_i such that $[x_i] + \Delta t_i F([\tilde{x}_i]) \subseteq [\tilde{x}_i]$;*
3. *Set $[x_{i+1}] = [x_i] + \Delta t_i F([\tilde{x}_i])$.*

This algorithm produces a box-valued solution that is guaranteed to contain the true solution, i.e., $\varphi(x_0, t_i) \in [x_i]$. Moreover, it also covers the case when the initial value is a whole box. There is, however, one major flaw in this method: even if the true solution set is shrinking, the computed boxes $[x_i]$ are always non-decreasing in i. This is because we always have the equality $\|[a] + [b]\| = \|[a]\| + \|[b]\|$ for any two intervals $[a]$ and $[b]$.

The abovementioned problem can be overcome by replacing the Euler step by a higher order Taylor-method, see e.g. [2], [4]. This may increase the accuracy on a local level, but we are still left with a global problem: if the flow of the system under consideration is not contracting in all directions, the strongest expanding (or neutral) direction will generically *contaminate* all other directions. By this, we mean that the computed enclosures $[x_i]$ will expand in all directions, although the true solution may contract in several directions. This phenomena is often referred to as the *wrapping effect*, see Figure 1(a).

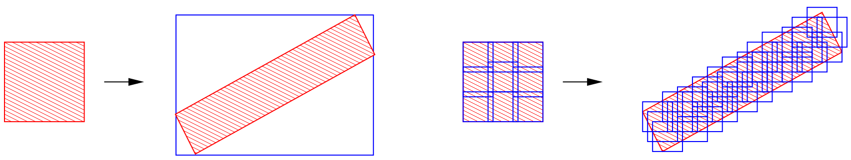

Figure 1: (a) The wrapping effect, and (b) how to overcome it.

Fortunately, we can reduce the wrapping effect by enforcing a *fixed scale*: if an element of any intermediate solution set (including the initial set) attains a width larger than a predetermined constant `MaxSize`, it is bisected along the directions that are too wide. Thus, the computed solution set will be made up of several small boxes, all having widths less than `MaxSize`. If the system has contracting directions, these will now show up in the solution set. This is due to the fact that elements squeeze together in the contracting directions, which results in an overlapping effect as illustrated in Figure 1(b). The global error is now of the same order as `MaxSize`, and the contamination is avoided. The following pseudo-code outlines an implementation of the algorithm just described:

Algorithm 2.

Initialize `Stack` *with a collection of boxes* $[x_1], \dots, [x_N]$
while `Stack` *is not empty*
{
  Get a box $[x]$ *from* `Stack`
  if $[x]$ *is too large*
    Bisect $[x]$ *in all directions that are wider than* `MaxSize`
    Put the partitioned boxes in `Stack`
  else
    Compute a time step Δt *and an enclosure* $[x']$ *containing*
    $\varphi([x], \Delta t)$*, using your favorite method (e.g. Algorithm 1)*
    if $[x']$ *satisfies the stopping condition*
      Put $[x']$ *in* `OutStack`
    else
      Put $[x']$ *in* `Stack`
}
Output `OutStack`

The partitioning process just described is self-adaptive: there is no need to know in advance where the expansion is strong, or in what directions it may act. Each element reports (by its current size) if a expanding region has been encountered, and the algorithm acts accordingly. Therefore, by just looking at the computed solution set, we can see which regions that have encountered a lot of expansion/contraction. Also, as mentioned earlier, we can attain quadratically close approximations of the interval-extended vector field F by choosing `MaxSize` small.

5 Examples

In this section, we will present a few simple examples illustrating the effectiveness of the partitioning process.

Example 1: pure rotation
Consider the system $(\dot{x}_1, \dot{x}_2) = (x_2, -x_1)$. The exact solution is given by

$$\begin{pmatrix}\varphi_1(x_1,x_2,t)\\ \varphi_2(x_1,x_2,t)\end{pmatrix} = \begin{pmatrix}\cos t & \sin t\\ -\sin t & \cos t\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}.$$

In [2], it is proved that Algorithm 1 produces extremely poor results for this system. If the time step is fixed, say $\Delta t = t/N$, then the widths of the computed enclosing boxes satisfy exponential growth. If w_0 denotes the width of the initial box, then the computed enclosing box at time t has width $w_t \approx e^t w_0$, whereas the enclosure width of the true solution is $(\cos t + \sin t)w_0$, which is bounded above by $\sqrt{2}w_0$.

Algorithm 2, however, does not display this irregular feature. We can make the error as small as desired by choosing `MaxSize` sufficiently small. In Figure 2, we illustrate a typical computed solution set at various times. Here, Algorithm 2 was implemented (using the Euler step) on the initial set $[-\frac{1}{2},\frac{1}{2}]\times[-\frac{1}{2},\frac{1}{2}]$ with `MaxSize` $=1/5$. No computation required more than one second on an ordinary laptop computer.

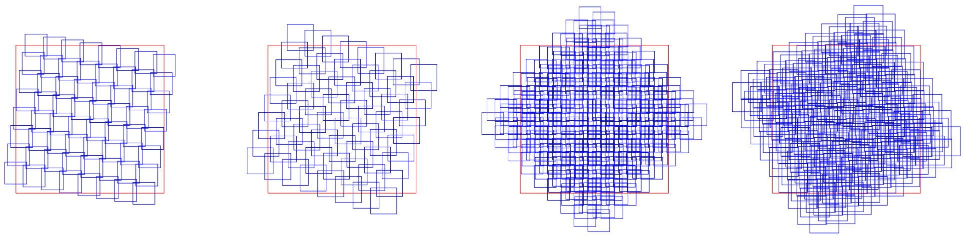

Figure 2: The solution set at times (a) $\pi/20$, (b) $\pi/10$, (c) $\pi/4$, and (d) $\pi/3$.

Example 2: skew hyperbolicity
Consider the system $(\dot{x}_1,\dot{x}_2)=(x_2,x_1)$. This is simply the uncoupled system $(\dot{x}_1,\dot{x}_2)=(x_1,-x_2)$ rotated by the angle $\frac{\pi}{4}$. Here is a situation where one typically would expect contamination, much like the case illustrated in Figure 1(a). In Figure 3, we again illustrate a computed solution set at various times. Here, Algorithm 2 was implemented (using the Euler step) on the initial set $[-\frac{1}{4},\frac{1}{4}]\times[-\frac{1}{4},\frac{1}{4}]$ with `MaxSize` $=1/16$. Again, no computation required more than one second on an ordinary laptop computer.

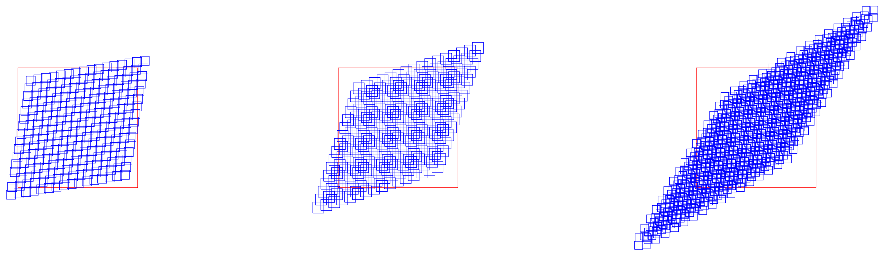

Figure 3: The solution set at times (a) $\sqrt{2}/8$, (b) $\sqrt{2}/4$, and (c) $\sqrt{2}/2$.

References

[1] L. H. de Figueiredo, J. Stolfi, Métodos numéricos auto-validados e aplicações, Braz. Math. Colloq. **21**, IMPA, 1997

[2] R. E. Moore, Interval Analysis, Prentice-Hall Series in Automatic Computation, Englewood Cliffs, N. J., 1966

[3] R. E. Moore, Methods and Applications of Interval Analysis, SIAM Studies in Applied Mathematics, Philadelphia, 1979

[4] N. S. Nedialkov and K. R. Jackson, *ODE Software that Computes Guaranteed Bounds on the Solution*, to appear in Advances in Software Tools for Scientific Computing, (ed. H. P. Langtangen, A. M. Bruaset and E. Quak), Springer-Verlag, 1999