Formally Verifying an Algorithm Based on Interval Arithmetic for Checking Transversality

Marcio Gameiro and Panagiotis Manolios

School of Mathematics, Georgia Institute of Technology
Atlanta, Georgia, 30332, USA
gameiro@math.gatech.edu
http://www.math.gatech.edu/∼gameiro
Georgia Institute of Technology, College of Computing, CERCS Lab
801 Atlantic Drive, Atlanta, Georgia, 30332, USA
manolios@cc.gatech.edu
http://www.cc.gatech.edu/∼manolios

Abstract. In this paper we use ACL2 to formally verify the correctness of an algorithm used in the analysis of dynamical systems. The algorithm uses interval arithmetic to check that a given vector field is transverse (non-tangential) to an edge (line segment). Instead of operating on numbers, interval operations operate on intervals, and they are guaranteed to return an over-approximation of the actual answer, thereby allowing us to use floating point arithmetic in a safe way. In this paper we prove that if the algorithm identifies an edge as transverse, then it is in fact transverse, as long as the underlying interval arithmetic operations are correctly implemented.

1 Introduction

Differential equations are widely used in Physics, Engineering, Computer Science, and numerous other fields. Unfortunately, there are no general analytical methods for solving such equations. Instead, numerical algorithms are used to find approximate solutions. The development and application of these methods represent a significant portion of the fields referred to as Numerical Analysis and Scientific Computing. While in general these techniques are reliable, there are at least two situations in which special care concerning the reliability or accuracy of the numerical solution is necessary: Partial Differential Equations (PDEs) and chaotic dynamics.

Standard numerical methods for PDEs consist in finding a numerical approximation to the solution on a given grid or finding a projection of the solution on a finite dimensional space. Although there are techniques for stability and error analysis of such methods, determining how close the computed solution is to the actual solution is, in general, difficult. A general approach is to further refine the grid or to increase the number of nodes in the projection and compare the solutions thus obtained until the results stabilize. This technique can verify that the numerical methods being used are stable with respect to the grid size or number of nodes, but gives us no rigorous information about how close the result is to the actual solution.

Numerical methods also have difficulty identifying unstable objects, such as basin boundaries or chaotic dynamics. In chaotic dynamics nearby initial conditions give rise to solutions that drift apart exponentially fast. It means that very small numerical errors may quickly grow without bounds if care is not taken. This makes it difficult to accurately solve such systems.

To address such problems with standard numerical methods, Mischaikow and his group [19] have developed numerical methods based on a topological technique known as Conley Index Theory [20, 18]. The methods, while computer-intensive, involve interval arithmetic and produce proofs, not numerical approximations. In addition they can be used to detect several kinds of dynamics. They have been used to find chaotic dynamics in infinite dimensional maps [5], topological horseshoes [9], fixed points [22], and bifurcation diagrams [4, 10] for PDEs.

The purpose of this paper is to implement one of these algorithms in ACL2 [15, 16, 14] and to prove its correctness. Since we really want to prove theorems about the reals, our proof scripts can also be verified with ACL2(r) [7, 8]. The rest of the paper is structured as follows. In Section 2, we present a brief overview of the theory mentioned above. In Section 3, we present a detailed description of the algorithm we are interested in formalizing in ACL2. In Section 4, we describe the formalization of interval arithmetic in ACL2 and the implementation of the algorithm. Finally, we conclude in Section 5 and outline future work.

2 Analyzing the Dynamics of Differential Equations

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a differentiable function and consider the ordinary differential equation

$$\dot{x} = f(x), \quad x \in \mathbb{R}^n. \tag{2.1}$$

Let $\varphi : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ be the *flow* generated by the *vector field* f, that is, for a fixed $x_0 \in \mathbb{R}^n$ the solution of (2.1) with initial condition $x(0) = x_0$ is given by $\varphi(t, x_0)$. The *full trajectory* of a set $S \subseteq \mathbb{R}^n$ is defined by

$$\varphi(\mathbb{R}, S) := \bigcup_{t\in\mathbb{R}} \bigcup_{x\in S} \varphi(t, x).$$

We denote the full trajectory of a singleton set $\{x\} \subseteq \mathbb{R}^n$ by $\varphi(\mathbb{R}, x)$. A set $S \subseteq \mathbb{R}^n$ is an *invariant set* under φ if

$$S = \varphi(\mathbb{R}, S).$$

In words a set $S \subseteq \mathbb{R}^n$ is invariant if the full trajectory of any $x \in S$, which represents a curve through x in $\mathbb{R}^n$, is entirely contained in S. We also define the *maximal invariant set* in $N \subseteq \mathbb{R}^n$ under φ as

$$Inv(N, \varphi) := \{x \in N \mid \varphi(\mathbb{R}, x) \subseteq N\}.$$

Understanding the dynamics of (2.1) means understanding the structure of its invariant sets. In general we want to understand the structure of the invariant set within a given set $X \subseteq \mathbb{R}^n$, that is, we want to understand the structure of $Inv(X, \varphi)$. An example may help clarify these ideas. For the Van der Pol oscillator

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= -x + \mu(1 - x^2)y \end{aligned} \tag{2.2}$$

the typical solutions (trajectories) in the set $X = [-3, 3] \times [-3, 3] \subseteq \mathbb{R}^2$ are shown in Figure 1 for $\mu = 1$.

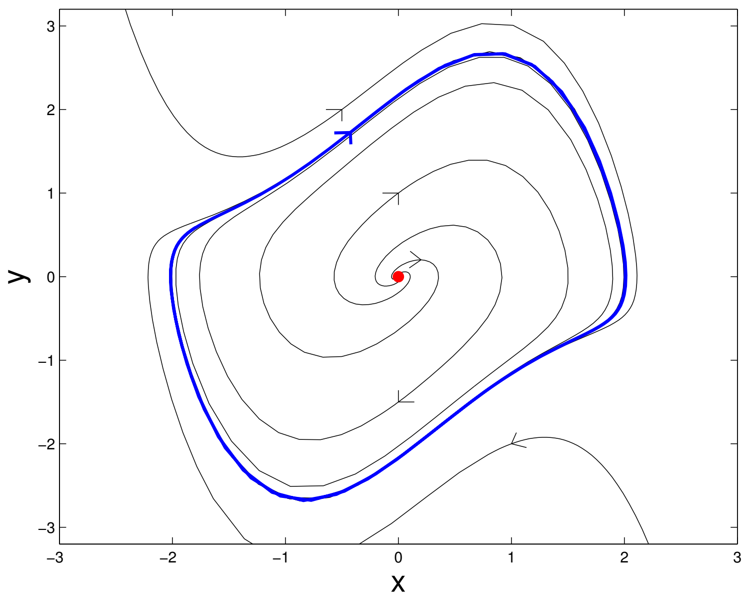

Fig. 1. Phase Portrait of (2.2) for $\mu = 1$.

As we can see in Figure 1 there are four kinds of solution in this region: a periodic solution (the thick closed curve), a fixed point (the dot in the middle), the curves that spiral from the fixed point toward the periodic solution, and the curves that come from outside and spiral toward the periodic solution. In this case we say that the periodic solution is *stable*, and the fixed point is *unstable*. The invariant set in this region (the set of curves that stay within the region for all times) consists of the periodic trajectory and all the curves enclosed by it. So in this case the invariant set consists of a stable periodic trajectory, a unstable fixed point, and connecting orbits between them. That is the kind of understanding we want to have from a given differential equation. We want to be able to decide whether there are fixed points, periodic trajectories, or even more complicated dynamics (like chaotic dynamics). As we mentioned in the introduction, standard numerical methods may not suffice or may not be reliable, and so we will use the topological techniques that we now describe. First we need some definitions.

A compact set $N \subseteq \mathbb{R}^n$ is an *isolating neighborhood* if

$$Inv(N, \varphi) \subseteq Int(N),$$

where $Int(N)$ denotes the interior of N. An invariant set S is called an *isolated invariant set* if there is an isolating neighborhood N such that $S = Inv(N, \varphi)$. An isolating neighborhood N is called and *isolating block* for $Inv(N, \varphi)$ if for every $x \in \partial N$, where ∂N denotes the *boundary* of N, and every $t > 0$ we have

$$\varphi((-t, t), x) \nsubseteq N. \tag{2.3}$$

It is easy to see that condition (2.3) is equivalent to saying that the flow is non-tangent to ∂N at the point x. More precisely given $x \in \partial N$, let $n(x)$ be a normal vector to ∂N at x. We say that the flow φ, or the vector field f, is *transverse (non-tangential)* to ∂N at x if

$$f(x) \cdot n(x) \neq 0,$$

where

$$x \cdot y := x_1 y_1 + \cdots + x_n y_n$$

denotes the *dot product* of the vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n) \in \mathbb{R}^n$. The dot product give us the following geometric information: The vectors x and y are *orthogonal (perpendicular)* iff $x \cdot y = 0$.

The *immediate exit set* for N is given by

$$N^- := \{x \in N \mid \forall t > 0, \varphi((0, t), x) \nsubseteq N\}.$$

If N is an isolating block for $Inv(N, \varphi)$, then the *Conley Index* [19, 20] of $Inv(N, \varphi)$ is given by

$$CH_*(N) := H_*(N, N^-),$$

where $H_*(N, N^-)$ denotes the *Relative Homology Groups* [12] of the sets N and N^-. Once we have the isolating blocks, the Conley Index is used to obtain a precise statement about the dynamics of (2.1) in N.

The two main steps of the algorithm are finding isolating blocks and then computing the Conley Index. Since we want to produce proofs rather than just numerics, every step of the algorithm needs to be rigorous in the sense that we need to control round-off errors and all other sources of numerical imprecision. Once we have the isolating blocks, the Conley Index can be computed using the software package `CHomP` [13]. All the computations in this package are combinatorial in nature, and therefore are free of numerical errors.

To find isolating blocks we first use numerics to find a candidate block N. In this step our calculations do not need to be rigorous, as we next check that N actually is an isolating block. As mentioned above, this consists in checking that the flow is transverse to the boundary of N. This transversality check is the part of the algorithm we implemented and certified in ACL2.

Before we describe how the transversality check is done let us discuss a little how isolating blocks can be found in the context of ODEs. We will describe this by using our example (2.2). For more details see [6, 3, 9]. The first step of the algorithm is to make a polygonal decomposition of the region where one suspects the dynamics of interest is located. In our example the region of interest is $X = [-3, 3] \times [-3, 3]$. We use a triangulation of that region (a simplicial complex in higher dimensions) as a first approximation for this polygonal decomposition. We need this triangulation to give a good approximation for the flow, which means that the triangles need to be long and skinny and oriented according to the flow (Figure 2).

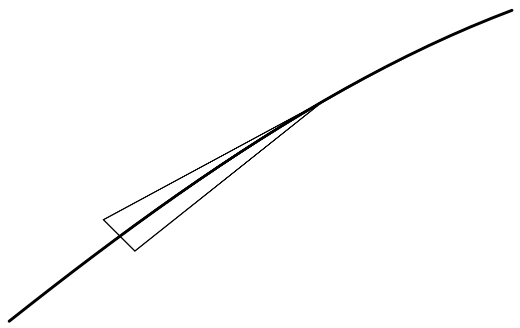

Fig. 2. Triangle Shape and Orientation.

Once we have such a triangulation we need to check that the flow is transverse to each one of the edges. If we can not verify transversality for a given edge, we remove this edge and make a polygon out of the corresponding neighbors. After we repeat this process for all edges we will have a polygonal decomposition of X such that the flow is transverse to each edge. Figure 3 shows a triangulation of X for (2.2) after the non-transverse edges have been removed.

We then construct a directed graph, where the nodes of the graph are the polygons, and there is a graph edge from polygon P_1 to polygon P_2 if P_1 and P_2 share a polygon edge and the flow goes from P_1 toward P_2 across this polygon edge. Using this directed graph we can capture the isolating blocks by finding its non-trivial strongly connected components. The fact that the flow is transverse to each edge of the polygonal decomposition guarantees that the strongly connected components are isolating blocks. For more details see [19, 3, 9].

In our example, after finding the isolating blocks shown in Figure 4, one can use Conley Index to prove the existence of a fixed point and a periodic orbit for (2.2) in X.

In the next section we describe in detail how the transversality of a given edge is checked. Even though in general we are interested in higher dimensions, we are going to restrict the rest of this paper to two dimensions for the sake of simplicity. However, the ideas are essentially the same in higher dimensions.

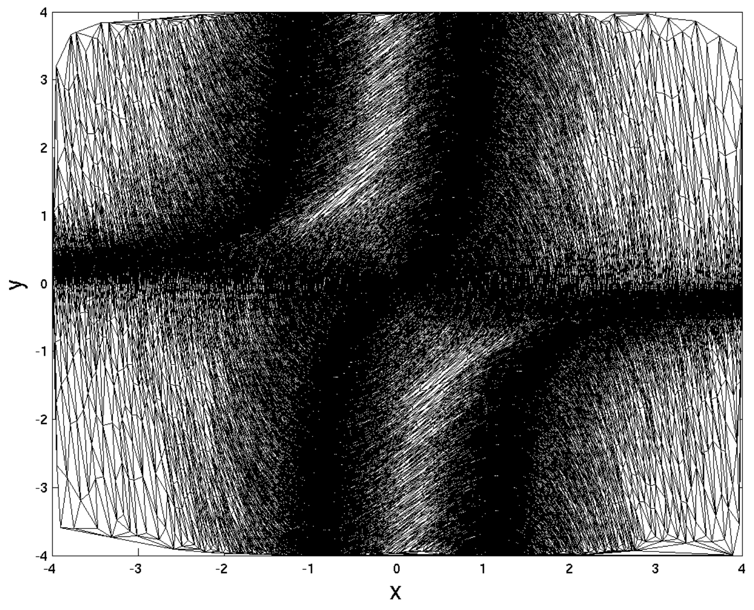

Fig. 3. Triangulation for (2.2).

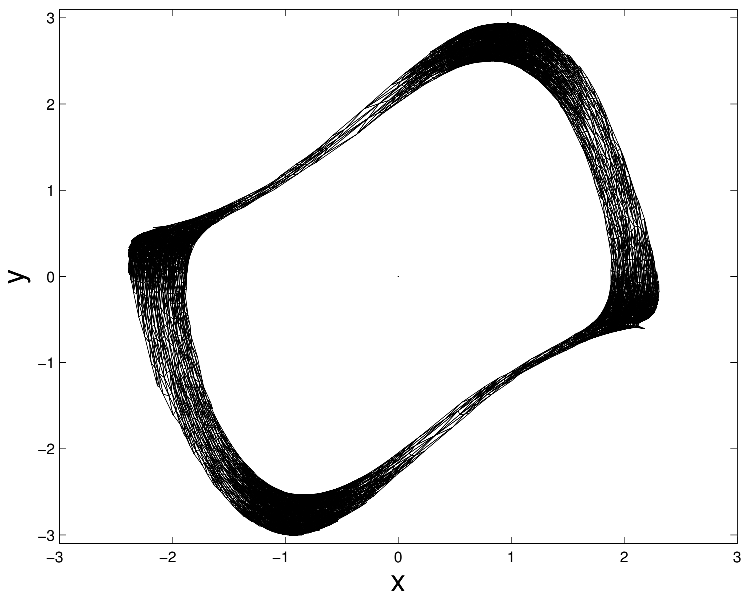

Fig. 4. Isolating Blocks for (2.2).

3 An Algorithm for Checking Transversality

In this section we describe how we check that the flow is transverse to an edge. Given and edge uv with endpoints $u = (u_1, u_2)$ and $v = (v_1, v_2)$, we define a *normal vector* to uv as

$$n = (u_2 - v_2, v_1 - u_1).$$

We can easily check that n is normal (orthogonal) to uv, by noting that $n \cdot (v - u) = 0$. We want f to be non-tangent to uv at every point p on this edge, this is equivalent to require that $f(p)$ be non orthogonal to n at p. This reduces the transversality check at p to the computation of a dot product. The problem is that we need to check transversality at all points in a given edge. The set of all the points in the edge uv is given by

$$\{u + \lambda(v - u) \mid \lambda \in [0, 1]\}.$$

Therefore to check that f is transverse to the edge uv at a given point is equivalent to check that

$$g(\lambda) := f(u + \lambda(v - u)) \cdot n \neq 0.$$

We want to check that f is transverse to the edge uv at all of its points. So we want to have $g(\lambda) \neq 0$ for all $\lambda \in [0, 1]$.

There are two points we need to address. One is that there are infinitely many $\lambda \in [0, 1]$. Thus, we cannot possibly try to directly compute $g(\lambda)$ for all $\lambda \in [0, 1]$. The second point is that, even for a single λ, we can not simply compute $g(\lambda)$ using floating point arithmetic, since, even if λ can be represented as a floating point number, floating point imprecision makes it difficult to determine with certainty that $g(\lambda) \neq 0$. As we will now describe, we can use *interval arithmetic* to solve both problems at the same time.

3.1 Interval Arithmetic

The main idea behind interval arithmetic is to work with intervals instead of single numbers. Let us first give a mathematical definition of the basic interval operations. Let $[x_1, x_2]$ and $[y_1, y_2]$ represent real-valued intervals. Then we can define the four basic arithmetic operations on these interval as follows:

$$[x_1, x_2] + [y_1, y_2] := [x_1 + y_1, x_2 + y_2] \tag{3.1a}$$

$$[x_1, x_2] - [y_1, y_2] := [x_1 - y_1, x_2 - y_2] \tag{3.1b}$$

$$[x_1, x_2] \times [y_1, y_2] := [\min(P), \max(P)] \tag{3.1c}$$

$$[x_1, x_2] \ / \ [y_1, y_2] := [x_1, x_2] \times [1/y_2, 1/y_1], \quad \text{if} \quad 0 \notin [y_1, y_2] \tag{3.1d}$$

where $P := \{x_1 y_1, x_1 y_2, x_2 y_1, x_2 y_2\}$. The fundamental property of these operations is:

$$[x_1, x_2] \circ [y_1, y_2] = \{x \circ y \mid x \in [x_1, x_2], y \in [y_1, y_2]\}, \tag{3.2}$$

where $\circ$ represents any one of the four operations defined above ($+$, $-$, $\times$, or $/$).

We can use interval arithmetic to reduce the computation of $g(\lambda)$ for all $\lambda \in [0,1]$ to one interval computation, or a finite set of interval computations if we divide the interval $[0,1]$ into a finite number of segments. However, to make this computationally feasible, we cannot compute exact answers. Instead, interval arithmetic implementations compute over-approximations that are guaranteed to contain (but are possibly larger than) the exact interval result. This is done by using floating point arithmetic and carefully setting the rounding modes to safe approximations of the intervals on the right hand side of (3.1).

Therefore, the fundamental property an interval library must satisfy is condition (3.2) with equality replaced by inclusion, that is,

$$[x_1, x_2] \circ [y_1, y_2] \subseteq [x_1, x_2] \boxdot [y_1, y_2]\,, \tag{3.3}$$

where $\boxdot$ denotes an implementation of the operation $\circ$. For more details see [21, 11, 2]. The only assumption we make regarding the interval libraries we use is that they satisfy the above property. For more details about interval arithmetic libraries see [2, 1].

3.2 Using Interval Arithmetic to Check Transversality

Using interval arithmetic we can then rewrite the condition $g(\lambda) \neq 0$ for all $\lambda \in [0,1]$ as follows.

$$0 \notin g([0,1]). \tag{3.4}$$

The problem is that we can only compute an over-approximation of $g([0,1])$; thus, even if $g(\lambda) \neq 0$ for all $\lambda \in [0,1]$ we may still have that 0 is in the computed interval. The key idea to overcoming this problem is that if the length of the interval $[a,b]$ is very small then the computed interval for $g([a,b])$ tends to be close to the exact interval. Therefore, we subdivide the interval $[0,1]$ into N subintervals, $0 = \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_{N-1} \leq \lambda_N = 1$, and evaluate g for each of these subintervals; the resulting condition we have to check is:

$$0 \notin g([\lambda_{k-1}, \lambda_k]) \tag{3.5}$$

for each $k \in \{1, \cdots, N\}$. If (3.5) is satisfied for all $k \in \{1, \cdots, N\}$, then we have that $g(\lambda) \neq 0$ for all $\lambda \in [0,1]$, since the fundamental property of interval arithmetic guarantees that the actual value is always contained in the computed interval. We therefore have the algorithm of Figure 5 for checking whether the vector field f is transverse to the edge with endpoints $u = (u_1, u_2)$ and $v = (v_1, v_2)$, where `l` represents the partition $0 = \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_{N-1} \leq \lambda_N = 1$.

We implemented the algorithm of Figure 5 and showed its correctness in ACL2. The ACL2 implementation is described in the next section.

4 ACL2 Formalization

In this section we describe the implementation and verification of the transversality algorithm in ACL2. We start, in Section 4.1, by formalizing interval arithmetic in

```
function CheckTransversality(f, u1, u2, v1, v2, l)

for k=1 to N do
  evaluate g([l(k-1), l(k)])

  if g([l(k-1), l(k)]) contains 0
    return non-transverse
  end if
end for

return transverse
```

Fig. 5. Algorithm for Transversality Check.

ACL2. Then, in Section 4.2, we formalize and analyze the transversality algorithm. What follows is a tour through the supporting materials, but note that we omit several definitions and most of the details of the proofs, as they are readily available online.

4.1 Interval Arithmetic Library

We formalize in ACL2 what it means for an interval arithmetic library to be correct, with respect to all the arithmetic operations we are interested in. We use encapsulation for this purpose. Notice that it does not matter how we define the interval operations, since our intent is not to implement an interval library, but, rather, to prove theorems that hold about any library that correctly implements interval operations, as per (3.3).

We start by defining the auxiliary functions `intervalp` and `in` that check if two values form an interval and if a value is in a given interval, respectively. We use the predicate `real/rationalp` in order to be able to certify our books in both ACL2 and ACL2(r).

```
(defun intervalp (x1 x2)
  (and (real/rationalp x1)
       (real/rationalp x2)
       (<= x1 x2)))


(defun in (x x1 x2)
  (and (real/rationalp x)
       (intervalp x1 x2)
       (<= x1 x)
       (<= x x2)))
```

Next we define the basic interval operations. The function `i+`, `i-`, and `i*` take four arguments representing two intervals and perform the respective operations on them. They are constrained functions satisfying the condition (3.3).

```
(encapsulate
 (((i+ * * * *) => (mv * *)))

 (local (defun i+ (x1 x2 y1 y2)
          (mv (+ x1 y1)
              (+ x2 y2))))

 (defthm i+_ok
   (implies (and (in x x1 x2)
                 (in y y1 y2))
            (mv-let (z1 z2)
                    (i+ x1 x2 y1 y2)
                    (in (+ x y) z1 z2)))))


(encapsulate
 (((i- * * * *) => (mv * *)))

 (local (defun i- (x1 x2 y1 y2)
          (mv (- x1 y2)
              (- x2 y1))))

 (defthm i-_ok
   (implies (and (in x x1 x2)
                 (in y y1 y2))
            (mv-let (z1 z2)
                    (i- x1 x2 y1 y2)
                    (in (- x y) z1 z2)))))


(encapsulate
 (((i* * * * *) => (mv * *)))

 (local (defun max4 (x1 x2 x3 x4)
          (max (max x1 x2)
               (max x3 x4))))

 (local (defun i* (x1 x2 y1 y2)
          (let ((m0 (max4 (abs x1) (abs x2) (abs y1) (abs y2))))
            (mv (- (* m0 m0)) (* m0 m0)))))

 (defthm i*_ok
   (implies (and (in x x1 x2)
                 (in y y1 y2))
            (mv-let (z1 z2) (i* x1 x2 y1 y2)
                    (in (* x y) z1 z2))))
```

Notice that interval multiplication, `i*`, is not defined using (3.1), in contrast with the definitions of `i+` and `i-`, because it is easier to prove `i*_ok` with the above definition, and all we require is `i*_ok`. Notice also that we do not formalize the quotient operator on intervals; we did not need it in this application.

From these basic definitions we can develop a theory of vector operations on intervals. We define the vector field `vec_fld`, which take the vector `(x,y)` as an argument and returns another vector. We again use encapsulation because we want our proof to be valid for any vector field. We also define the interval version of the vector field `i_vec_fld`, which takes four arguments representing an interval vector `([x1,x2], [y1,y2])`, and returns an interval vector.

```
(encapsulate
 (((vec_fld * *) => (mv * *))
  ((i_vec_fld * * * *) => (mv * * * *)))

 (local (defun vec_fld (x y)
          (let ((u (+ x y))
                (w (- x y)))
            (mv u w))))

 (local (defun i_vec_fld (x1 x2 y1 y2)
          (mv-let (u1 u2)
                  (i+ x1 x2 y1 y2)
                  (mv-let (w1 w2)
                          (i- x1 x2 y1 y2)
                          (mv u1 u2 w1 w2)))))

(defthm vec_fld_ok
  (implies (and (in x x1 x2)
                (in y y1 y2))
           (mv-let (u1 u2 w1 w2) (i_vec_fld x1 x2 y1 y2)
                   (mv-let (u w) (vec_fld x y)
                           (and (in u u1 u2)
                                (in w w1 w2))))))
```

Next, we define the dot product `dot`, and its interval version `i_dot`. The `dot` function takes two vectors `(x,y)` and `(u,v)` as input and returns a number. The function `i_dot` takes two interval vectors `([x1,x2], [y1,y2])` and `([u1,u2], [v1,v2])` as input and returns an interval.

```
(defun dot (x y u v)
  (+ (* x u) (* y v)))

(defun i_dot (x1 x2 y1 y2 u1 u2 v1 v2)
  (mv-let (p11 p12)
          (i* x1 x2 u1 u2)
          (mv-let (p21 p22)
                  (i* y1 y2 v1 v2)
                  (mv-let (d1 d2)
                          (i+ p11 p12 p21 p22)
                          (mv d1 d2)))))
```

```
(defthm i_dot_ok
  (let ((idot (i_dot x1 x2 y1 y2 u1 u2 v1 v2)))
    (implies (and (in x x1 x2)
                  (in y y1 y2)
                  (in u u1 u2)
                  (in v v1 v2))
             (in (dot x y u v) (nth 0 idot) (nth 1 idot)))))
```

To reason about functions which return multiple arguments, we (1) prove a rewrite rule that replaces `mv-nth` with `nth` (they are equal), (2) disable `mv-nth`, and (3) use a library for reasoning about `nth` from [17]. This should explain the `nth`'s in `i_dot_ok`.

We now define the perpendicular of a vector, `perp`, and prove that the dot product of a vector with its perpendicular is zero. We also define the interval version of the perpendicular, `i_perp`. The function `perp` takes a vector `(x,y)` as input and returns a vector orthogonal to it, and `i_perp` takes an interval vector `([x1,x2], [y1,y2])` as input and returns another vector.

```
(defun perp (x y)
  (mv (* -1 y) x))

(defthm perp_correct
  (let ((perp (perp x y)))
    (equal (dot x y (nth 0 perp) (nth 1 perp)) 0)))

(defun i_perp (x1 x2 y1 y2)
  (mv-let (ny1 ny2)
          (i* -1 -1 y1 y2)
          (mv ny1 ny2 x1 x2)))

(defthm i_perp_ok
  (let ((perp (perp x y))
        (iperp (i_perp x1 x2 y1 y2)))
    (implies (and (in x x1 x2)
                  (in y y1 y2))
             (and (in (nth 0 perp) (nth 0 iperp) (nth 1 iperp))
                  (in (nth 1 perp) (nth 2 iperp) (nth 3 iperp))))))
```

Next, we define a normal vector, `normal_vec`, to a given edge and its interval version, `i_normal_vec`. These functions take two and four arguments as input respectively.

```
(defun normal_vec (u1 u2 v1 v2)
  (mv-let (w1 w2)
          (perp (- v1 u1) (- v2 u2))
          (mv w1 w2)))
```

```
(defun i_normal_vec (u1 u2 v1 v2)
  (mv-let (x1 x2)
          (i- v1 v1 u1 u1)
          (mv-let (y1 y2)
                  (i- v2 v2 u2 u2)
                  (mv-let (w11 w12 w21 w22)
                          (i_perp x1 x2 y1 y2)
                          (mv w11 w12 w21 w22)))))
```

```
(defthm i_normal_vec_ok
  (let ((nvec (normal_vec u1 u2 v1 v2))
        (invec (i_normal_vec u1 u2 v1 v2)))
    (implies (and (real/rationalp u1)
                  (real/rationalp u2)
                  (real/rationalp v1)
                  (real/rationalp v2))
             (and (in (nth 0 nvec) (nth 0 invec) (nth 1 invec))
                  (in (nth 1 nvec) (nth 2 invec) (nth 3 invec))))))
```

4.2 Algorithm Formalization

Using the interval and interval vector operations defined above we can now implement the algorithm for checking transversality. Given points $u = (u_1, u_2)$, $v = (v_1, v_2)$, $\lambda \in [0, 1]$, and $0 \leq \lambda_1 \leq \lambda_2 \leq 1$ we compute the point $uv_\lambda := u + \lambda(v - u)$ on the edge uv using the function `edge_lbda`, and the points corresponding to the interval computation $uv_{[\lambda_1,\lambda_2]} := u + [\lambda_1, \lambda_2](v - u)$ using the function `i_edge_lbda`. We then prove that $uv_\lambda \in uv_{[\lambda_1,\lambda_2]}$ if $\lambda \in [\lambda_1, \lambda_2]$.

```
(defun edge_lbda (u1 u2 v1 v2 lbda)
  (mv-let (y1 y2)
          (vect_lbda u1 u2 v1 v2 lbda)
          (mv (+ u1 y1) (+ u2 y2))))
```

```
(defun i_edge_lbda (u1 u2 v1 v2 l1 l2)
  (mv-let (z11 z12 z21 z22)
          (i_vect_lbda u1 u2 v1 v2 l1 l2)
          (mv-let (y11 y12)
                  (i+ u1 u1 z11 z12)
                  (mv-let (y21 y22)
                          (i+ u2 u2 z21 z22)
                          (mv y11 y12 y21 y22)))))
```

```
(defthm i_edge_lbda_ok
  (let ((iuv (i_edge_lbda u1 u2 v1 v2 l1 l2))
        (uv  (edge_lbda u1 u2 v1 v2 lbda)))
    (let ((y11 (nth 0 iuv))
          (y12 (nth 1 iuv))
          (y21 (nth 2 iuv))
          (y22 (nth 3 iuv))
          (y1 (nth 0 uv))
          (y2 (nth 1 uv)))
      (implies (and (real/rationalp u1)
                    (real/rationalp u2)
                    (real/rationalp v1)
                    (real/rationalp v2)
                    (unitsubintervalp l1 l2)
                    (in lbda l1 l2))
               (and (in y1 y11 y12)
                    (in y2 y21 y22))))))
```

The function `check_trans_lbda` returns true if the flow is transverse to the edge uv at the point uv_λ, and the function `i_check_trans_lbda` returns true if the interval computations indicate that flow is transverse to $uv_{[\lambda_1,\lambda_2]}$.

```
(defun check_trans_lbda (u1 u2 v1 v2 lbda)
  (and (real/rationalp u1)
       (real/rationalp u2)
       (real/rationalp v1)
       (real/rationalp v2)
       (in lbda 0 1)
       (mv-let (n1 n2)
               (normal_vec u1 u2 v1 v2)
               (mv-let (y1 y2)
                       (edge_lbda u1 u2 v1 v2 lbda)
                       (mv-let (f1 f2)
                               (vec_fld y1 y2)
                               (not (equal (dot f1 f2 n1 n2) 0)))))))

(defun i_check_trans_lbda (u1 u2 v1 v2 l1 l2)
  (and (real/rationalp u1)
       (real/rationalp u2)
       (real/rationalp v1)
       (real/rationalp v2)
       (unitsubintervalp l1 l2)
       (mv-let (n11 n12 n21 n22)
               (i_normal_vec u1 u2 v1 v2)
               (mv-let (y11 y12 y21 y22)
                       (i_edge_lbda u1 u2 v1 v2 l1 l2)
                       (mv-let (f11 f12 f21 f22)
                               (i_vec_fld y11 y12 y21 y22)
                               (mv-let (dot1 dot2)
                                       (i_dot f11 f12 f21 f22 n11 n12 n21 n22)
                                       (not (in 0 dot1 dot2))))))))
```

We now prove that if `i_check_trans_lbda` returns true and $\lambda \in [\lambda_1, \lambda_2]$ then `check_trans_lbda` is also true.

```
(defthm edge_trans_l1_l2
  (implies (and (i_check_trans_lbda u1 u2 v1 v2 l1 l2)
                (in lbda l1 l2))
           (check_trans_lbda u1 u2 v1 v2 lbda)))
```

The function `(i_check_trans u1 u2 v1 v2 l)` is used to check if the flow is transverse to $uv_{[\lambda_{k-1},\lambda_k]}$ for all $[\lambda_{k-1}, \lambda_k]$ in the `unit-partition l`, $0 = \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_{N-1} \leq \lambda_N = 1$.

```
(defun i_check_trans (u1 u2 v1 v2 l)
  (if (endp (cddr l))
      (i_check_trans_lbda u1 u2 v1 v2 (car l) (cadr l))
    (and (i_check_trans_lbda u1 u2 v1 v2 (car l) (cadr l))
         (i_check_trans u1 u2 v1 v2 (cdr l)))))
```

Finally we prove that if `(in lbda 0 1)`, `l` is a unit partition, all of `u1`, `u2`, `v1`, and `v2` are `real/rationalp`'s, and `(i_check_trans u1 u2 v1 v2 l)`, then we have `(check_trans_lbda u1 u2 v1 v2 lbda)`, that is, the flow is transverse at every point uv_λ for $\lambda \in [0, 1]$, which is our main theorem.

```
(defthm edge_trans_f
  (implies (and (in lbda 0 1)
                (unit-partition l)
                (real/rationalp-hyps u1 u2 v1 v2)
                (i_check_trans u1 u2 v1 v2 l))
           (check_trans_lbda u1 u2 v1 v2 lbda)))
```

5 Conclusions

We have described an algorithm that can be used to analyze differential equations. The algorithm is interesting in that it uses floating point operations, yet it can still be used to give precise results, *i.e.*, proofs. The reason is that instead of operating on numbers, it operates on intervals, using interval arithmetic. A correct implementation of the interval arithmetic operations is guaranteed to return an over-approximation of the actual answer, which is what saves us from precision errors. We used ACL2 to formalize interval arithmetic and the algorithm. Furthermore, we proved that if the algorithm identifies an edge as transverse, then it is in fact transverse.

There are several ways in which we plan to extend this work. First, we intend to integrate an interval arithmetic library into ACL2 and ACL2(r), which would allow us to compute with the reals and to prove (simple) theorems via computation. Second, we plan to verify a generalization of the algorithm presented here that works for arbitrary dimensions. Finally, we propose to implement and verify an executable version of the generalized algorithm in ACL2, so that the code we verify is the code we run.

References

1. C-XSC - A C++ class library, 2004. See URL `http://www.xsc.de/`.
2. Interval computations, 2004. See URL `http://www.cs.utep.edu/interval-comp/`.
3. E. Boczko, W. Kalies, and K. Mischaikow. Polygonal approximation of flows. *In preparation*, 2004.
4. S. Day, Y. Hiraoka, K. Mischaikow, and T. Ogawa. Rigorous numerics for global dynamics: a study of the swift-hohenberg equation. *Submitted to SIAM Appl. Dyn. Sys.*, 2004.
5. S. Day, O. Junge, and K. Mischaikow. A rigorous numerical method for the global analysis of infinite dimensional discrete dynamical systems. *SIAM J. Appl. Dyn. Syst.*, 3(2):117–160 (electronic), 2004.
6. M. Eidenschink. *Exploring Global Dynamics: A numerical Algorithm Based on the Conley Index Theory.* PhD thesis, School of Mathematics, Georgia Institute of Technology, August 1995.
7. R. Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2.* PhD thesis, The University of Texas at Austin, 1999.
8. R. Gamboa and M. Kaufmann. Non-standard analysis in ACL2. *Journal of Automated Reasoning*, 27(4):323–351, 2001.
9. M. Gameiro, T. Gedeon, W. Kalies, H. Kokubu, K. Mischaikow, and H. Oka. Topological horseshoes of travelling waves for a fast-slow predator-prey system. *In preparation*, 2004.
10. Y. Hiraoka, O. Toshi, and K. Mischaikow. Conley index based numerical verification method for global bifurcations of the stationary solutions to the swift-hohenberg equation. *Trans. Jpn. Soc. Ind. Appl. Math.*, 13(2):191, 2003.
11. L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied interval analysis.* Springer-Verlag London Ltd., London, 2001. With examples in parameter and state estimation, robust control and robotics.
12. T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational homology*, volume 157 of *Applied Mathematical Sciences.* Springer-Verlag, New York, 2004.
13. W. Kalies and P. Pilarczyk. Computational homology program, 2004. See URL `http://-www.math.gatech.edu/~chom`.
14. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, June 2000.
15. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, July 2000.
16. M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/-users/moore/acl2`.
17. P. Manolios. Verification of pipelined machines in ACL2. In M. Kaufmann and J. S. Moore, editors, *Proceedings of the ACL2 Workshop 2000.* The University of Texas at Austin, Technical Report TR-00-29, November 2000.
18. K. Mischaikow. The Conley index theory: a brief introduction. In *Conley index theory (Warsaw, 1997)*, volume 47 of *Banach Center Publ.*, pages 9–19. Polish Acad. Sci., Warsaw, 1999.
19. K. Mischaikow. Topological techniques for efficient rigorous computation in dynamics. *Acta Numer.*, 11:435–477, 2002.
20. K. Mischaikow and M. Mrozek. Conley index. In *Handbook of dynamical systems, Vol. 2*, pages 393–460. North-Holland, Amsterdam, 2002.
21. R. E. Moore. *Methods and applications of interval analysis*, volume 2 of *SIAM Studies in Applied Mathematics.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pa., 1979.

22. P. Zgliczyński and K. Mischaikow. Rigorous numerics for partial differential equations: the Kuramoto-Sivashinsky equation. *Found. Comput. Math.*, 1(3):255–288, 2001.