Coordinate-Independent Computations on Differential Equations

by

Kevin K. Lin

Last revised March 27, 1998

Coordinate-Independent Computations on Differential Equations

by

Kevin K. Lin

Abstract

This project investigates the computational representation of differentiable manifolds, with the primary goal of solving partial differential equations using multiple coordinate systems on general n-dimensional spaces. In the process, this abstraction is used to perform accurate integrations of ordinary differential equations using multiple coordinate systems. In the case of linear partial differential equations, however, unexpected difficulties arise even with the simplest equations.

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science in Septembr 1997 in partial fulfillment of the requirements for the degree of Master of Engineering.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-96-1-1228 and N00014-92-J-4097.

Acknowledgments

As is the case with any undertaking, there are far too many people I should thank. If I neglected anyone, it is purely out of failure of memory and I hope they won't hold a grudge.

To begin, I would like to thank Norman K. Yeh and Mariya Minkova for proof-reading my thesis proposal, and Tim McNerney for suggestions on how to present manifolds. Because of them, the presentation is much better than the rambling mess that was the first draft.

To the folk at Project Mathematics and Computation, especially Daniel Coore and Jim McBride: Thanks for the stimulating (and sometimes late-night) discussions on life, the universe, and everything. Thanks are also due to Eric Grimson, Thanos Siapas, and Ken Yip for crucial references, to Hardy Mayer and Michael Chechelnitsky for many helpful conversations on PDEs, and to Rebecca Bisbee and Anne Hunter for making sure that all the important things got done.

Portions of this work was supported by the Barry M. Goldwater Foundation, without whose help it would have been difficult to get this far.

I am forever indebted to my teachers throughout the years: To Mr. Reyerson at BCP, and Professors Flanigan and Morris at SJSU, for introducing me to *mathematics*; to Professors Munkres, Guillemin, and Wisdom for being such great teachers; and finally, to Gerald Jay Sussman, who has helped shape my interests, guided my work, and above all else encouraged me onward as I took the first faltering steps into a world of breathtaking wonders.

To my family and *all* my friends: You have made a great difference, and I could not have made it without you.

Still round the corner there may wait
A new road or a secret gate;
And though I oft have passed them by,
A day will come at last when I
Shall take the hidden paths that run
West of the Moon, East of the Sun.
—J. R. R. Tolkien

To Grandpa.

1908–1994

Contents

Chapter 1

Introduction

Partial differential equations[1] arise naturally in a large variety of physical problems. Like ordinary differential equations, the majority of partial differential equations cannot be solved analytically save in special cases. Thus, efficient and accurate numerical solutions of partial differential equations are essential in many applications. However, unlike ordinary differential equations, the solution of even linear partial differential equations can be a non-trivial task. There are no general methods that apply to all types of partial differential equations, and it is often necessary to exploit special structures in the problem at hand.

This project explores the the numerical solution of partial differential equations using coordinate-independent representations. It was hoped that this approach makes possible the use of whatever coordinate system that happens to simplify the problem locally, so that we can exploit the structure and locality of interaction inherent in many physical systems. This should provide more accurate solutions as well as insights into the physical and mathematical structure of problems. In addition, it may also help us reformulate such problems for distributed computers. Despite all these hopes, however, a large portion of this project is devoted to the study of ordinary differential equations. This is because the manifold abstraction arises naturally in the study of classical mechanics, which is described by ordinary differential equations. Furthermore, the formulation and solution of ODEs on manifolds is much more natural and straightforward than for PDEs.

The rest of this document, then, is divided into three chapters and two appendices. The first chapter develops the idea of differentiable manifolds and other basic concepts from modern differential geometry, and applies these concepts directly to the representation and solution of ordinary differential equations, particularly those arising from classical mechanics. Next, partial differential equations are discussed; for simplicity, the discussion is restricted to simple scalar linear equations, such as Laplace's equation over regions in the

[1]Often referred to as *PDEs* for short, just as ordinary differential equations are *ODEs*.

plane; the focus on low-order linear equations in low dimensions makes available analytical solutions, so that we can check our numerical results. Finally, the coordinate-independent solution of equations involving time, such as the linear wave equation, is investigated; in this context, the *spacetime* representation of equations (rather than the traditional "space + time") seems most natural.

While the manifold abstraction works beautifully with ordinary differential equations, some unexpected difficulties arise when dealing with even the simplest partial differential equations. Thus, most of the methods described herein, with one notable exception, actually do not work all that well, and in some cases completely fail. Thus, there is much work to be done. However, given the limited scope and time scale of this project, not all possible solutions to these problems can be adequately explored. It is hoped that these ideas can be explored more fully in the future.

Appendix A includes relevant background material on partial differential equations. In particular, it presents the numerical methods that form a basis for this project, as well as some important geometric and analytical properties of partial differential equations. Appendix B contains some material on the theory of manifolds that was not directly needed in this project. Complete program listings are *not* included in this report; interested readers should contact the author by electronic mail at `kkylin@alum.mit.edu` for more information on how to obtain the source code.

The entire document, including the material on abstract manifolds, suppose only a strong background in linear algebra and advanced calculus; little familiarity with more advanced mathematics is assumed. Also, it is helpful, though not necessary, to be acquainted with classical mechanics in Chapter 2, and §4.3.1 presumes some acquaintance with the basic concepts of relativity.

Finally, a note about the presentation: Throughout this document, programs implementing the main ideas will be presented alongside the mathematics. This serves a few different purposes: First, because this project is fundamentally about computational techniques, it would not be complete without actual programs. Second, it is often the case that seeing something presented in different ways aids in understanding, especially in subjects involving a significant amount of abstraction. Furthermore, programming languages, by their very nature, force one to be as careful with the details as with the main concepts, something that math and physics texts sometimes neglect. The language chosen for this project is Scheme, a dialect of Lisp. The choice is primarily based on the exceptional ease and flexibility with which Scheme expresses mathematical concepts; good references for the language are [9] and [2].

Chapter 2

Ordinary Differential Equations and Manifolds

This chapter describes the computational representation of manifolds, as well as their use in the formulation and numerical solution of ordinary differential equations. As motivating examples for the main definitions, problems in classical mechanics are presented using the manifold formalism. In following chapters, some ideas for integrating linear partial differential equations using multiple coordinate systems are treated.

For a good reference on advanced calculus as well as an elementary introduction to manifolds, Munkres [21] is excellent. Also, Guillemin and Pollack give a beautifully lucid exposition on the topology of manifolds [14]. A more technical and abstract treatment is given in Warner [28], and the classic by Arnold [4] presents manifolds in the context of classical mechanics—An approach followed closely in spirit in this chapter.

2.1 A brief introduction to manifolds

This section introduces the basic notions using a physical example, which will be revisited from time to time as new concepts are developed.

2.1.1 The spherical pendulum

A good starting point for the study of manifolds is a variation on the classical pendulum, the *spherical pendulum* (see Figure 2-1): Suppose a point mass of mass m is connected to a fixed point by a massless rod of length l. Furthermore, suppose that the point mass is allowed to move freely in any angle (not simply constrained to a vertical plane, as in the usual pendulum), and that it is subject only to a uniform gravitational field of constant magnitude g.

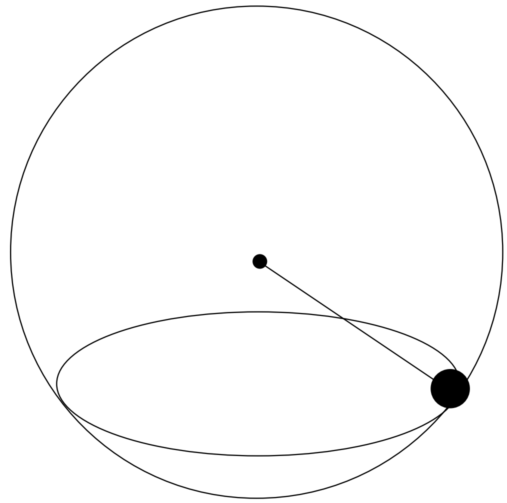


Figure 2-1: The spherical pendulum.

The equations of motion for this problem are easy to derive. However, instead of deriving the equations to analyze properties of the motion, let us focus on some of the more fundamental issues in a complete mathematical description of the problem. As we will see, this problem illustrates most of the basic ideas in the theory of manifolds.[1]

First, consider the problem of specifying the *configuration* of the system. What information do we need to specify the position of the pendulum? Since the point mass is constrained to move at a constant distance l from the fulcrum, the problem of specifying configurations of the spherical pendulum is equivalent to the problem of locating points on a sphere.

In order to specify points on a sphere, there are a couple of alternatives. One natural idea is to use the fact that the two-dimensional sphere sits inside three-dimensional Euclidean space, and to use the coordinates of R^3 to parametrize the sphere. Unfortunately, this approach is natural only for the sphere, and there are many important abstract spaces that cannot be easily visualized as subspaces of Euclidean space, such as the space of all orientations of a rigid body (which will be discussed later). Furthermore, in numerical integrations of ODEs, it will often happen that the trajectory "veers off" the sphere due to the accumulation of round-off error, and the constraint that the point mass lies at a constant distance l from the origin would no longer hold.

Another approach is to put coordinate systems on the sphere that require only two parameters. Formally, these are differentiable one-to-one mappings that map subsets of the sphere onto subsets of the plane and for which a differentiable inverse exists. This turns out to be a well-studied problem, since cartographers must deal with the fact that the surface of

[1]The ordinary pendulum, often used to illustrate important physical concepts, is not complicated enough geometrically to bring out the difficulties that manifolds were invented to handle.

the Earth is spherical (approximately) but maps are flat (Euclidean). The usual examples of map-making projections, such as the Mercator projection (cylindrical coordinates) or the system of longitudes and latitudes (spherical coordinates) are all examples of coordinate systems on the sphere. Note that there exists no two-parameter coordinate system that covers all of the sphere in a continuous fashion, but for every point on the sphere, we can always find a coordinate system that parametrizes a *neighborhood* of the point using a pair of parameters, so that the parametrization matches the dimension of the space.

In addition, in cartography, there is a natural solution to the problem that no coordinate system covers all of the Earth: We can simply use more than one map. We can simply switch to another map when one map becomes nearly useless. All that is required is some systematic way of figuring out when coordinates in two different maps are in fact the same point on the sphere, so that one could switch between maps without getting lost. This idea has been generalized beyond recognition to form the foundation of modern differential geometry, and spaces covered by maps (usually called *charts*) that make the space look "locally Euclidean" are called *manifolds*.

2.1.2 Differentiable manifolds

These ideas can be formulated mathematically as follows: Let M be a non-empty set of *points*,[2] and let n be some fixed positive integer. An n-dimensional chart on M is a triple (U, V, ϕ), where U is a subset of M, V an open subset of R^n, and ϕ a one-to-one map of U onto V (see Figure 2-2). ϕ is a *coordinate map*, and a chart (U, V, ϕ) is said to contain a point $p \in M$ if U contains p. Given two charts $C_1 = (U_1, V_1, \phi_1)$ and $C_2 = (U_2, V_2, \phi_2)$, suppose the intersection $U_1 \cap U_2$ is non-empty, and let W_i be the image $\phi_i(U_1 \cap U_2)$ for $i = 1, 2$. We can then form a *transition map*, $\phi_2 \circ \phi_1^{-1}$, which is a bijective mapping from W_1 to W_2. Note that the inverse of this transition map is $\phi_1 \circ \phi_2^{-1}$, which is represented by the same set of lines in Figure 2-2 with the arrows reversed. Now, if W_1 and W_2 are both open subsets of Euclidean n-space, then it makes sense to talk about the differentiability of the transition map $\phi_2 \circ \phi_1^{-1}$, and the charts C_1 and C_2 are said to be *compatible* if W_1 and W_2 are open and the corresponding transition map is *smooth*, i.e. has all orders of partial derivatives.[3] A collection $\mathcal{A}$ of charts on M is called an *atlas* if all its charts are mutually compatible, and if every point of M is contained in some chart in $\mathcal{A}$.[4] M, together with an

[2] In theory, points in an abstract space need not necessarily be points in a Euclidean space. They can also be classes of matrices or other abstract mathematical structures.

[3] In the present setting, the sets W_i are *required* to be open subsets of V_i (and hence of R^n). An alternative is to require the subsets U_i of the abstract space M to be open, but to define what that means requires some knowledge of general topology (which is not assumed here).

[4] It is easy to check that compatibility of charts is *transitive*, that is, if C_1 and C_2 are compatible charts, and C_2 and C_3 are also compatible, then so are C_1 and C_3.

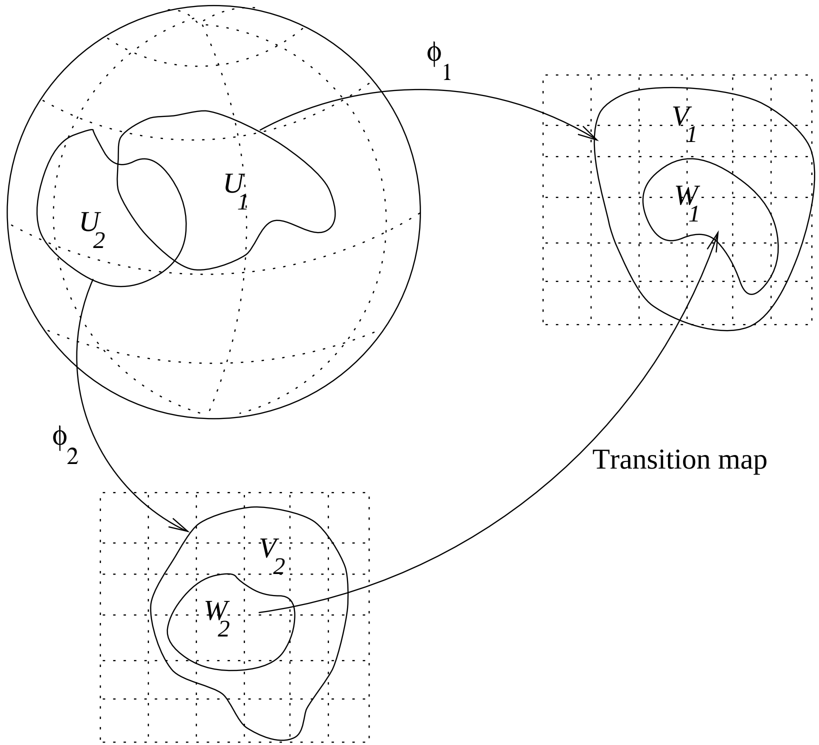


Figure 2-2: The sphere with generic charts and a transition map.

atlas $\mathcal{A}$, is called a *differentiable manifold of dimension* n.[5]

These formal definitions may take some time to absorb, but after some thought one should see that all that this says is that M is completely covered by a collection of maps, so that given any point $p \in M$, we can find a chart that makes a neighborhood of p look like an open subset of Euclidean n-space. Thus, the definition formalizes the idea of coordinate systems on abstract spaces, and transition maps allow one to switch between coordinate systems in a consistent way. This would, for example, allow us to define spheres in a way that solves the problem of locating points: One simply specifies a chart and a point, provided the appropriate charts have been constructed.

Implementation of manifolds in Scheme

The implementation of charts as computational objects is straightforward; it is accomplished through the procedure `make-simple-chart`. `Make-simple-chart` expects five arguments:

[5]Technically, manifolds must also satisfy the "second countability axiom" and be "Hausdorff spaces." These are rather technical condition and are not crucial for the purposes of our discussion, though some of the theorems quoted in later sections depend on manifolds being second-countable Hausdorff spaces.

`Dim`, the dimension of the chart; `in-domain?`, a procedure that takes a point as argument and returns `#t` or `#f` depending on whether the given point is in the chart; `in-range?`, the analogous procedure for coordinate vectors in the range of the coordinate map; `coord-map`, a computational representation of the function ϕ that maps points in the manifold to coordinate vectors in R^n; and its inverse, `inverse-map`. The constructor simply packages up these procedures and provides auxiliary procedures for accessing these methods.

Similarly, the procedure `make-manifold` constructs manifolds. It takes four arguments: `Dim`, the dimension of the manifold; `general-find-chart`, a procedure that takes a point p and a list of predicates, and returns a chart C containing p such that every predicate in the list returns `#t` when called with C; `find-minimizing-chart`, which takes a point p and a real-valued function f on *charts*, and returns the chart C that contains p and minimizes f;[6] and `get-local-atlas`, a function that takes a point p and returns the list of all charts containing p. Note that, since lists in Scheme must necessarily be finite, this means any atlas constructed this way is *locally finite*; that is, every point p is contained in only finitely many charts.[7] However, the fact that everything is implemented procedurally allows for the possibility that the atlas itself is potentially infinite. For convenience, there is also a constructor `charts->manifold` which takes a finite list of charts and constructs the procedures `general-find-chart`, etc., by searching through this finite list.

2.1.3 Some examples

One obvious class of examples of differentiable manifolds is the Euclidean space R^n. Here, the atlas consists of a single chart, (R^n, R^n, id_{R^n}), where id_{R^n} is the identity map on R^n.

We can express this example in Scheme as follows:

```
(define (make-euclidean-space dim)

  ;; Just need one big happy chart:
  ;; (test v) = #t iff v is a real vector of length dim:

  (let* ((test (make-euclidean-test dim))
         (chart (make-simple-chart dim test test identity identity)))
    (charts->manifold (list chart))))
```

Another example is the circle, where two charts are now required (see Figure 2-3): Removing the point $(1, 0)$ from the circle gives a smooth bijection between the rest of the

[6]For example, the function f can be a measure of how poorly the chart behaves at p, such as how close the *procedure* $\phi \circ \phi^{-1}$ comes to being the identity map at p and so on. This can be useful in integrating ODEs on manifolds.

[7]Note that this is only a restriction on our computational representations of manifolds, not on differentiable manifolds in general. A manifold, in theory, can have an infinite number of charts covering a given point. One should be careful to distinguish between differentiable manifolds, which are theoretical constructs, and their computational representations.

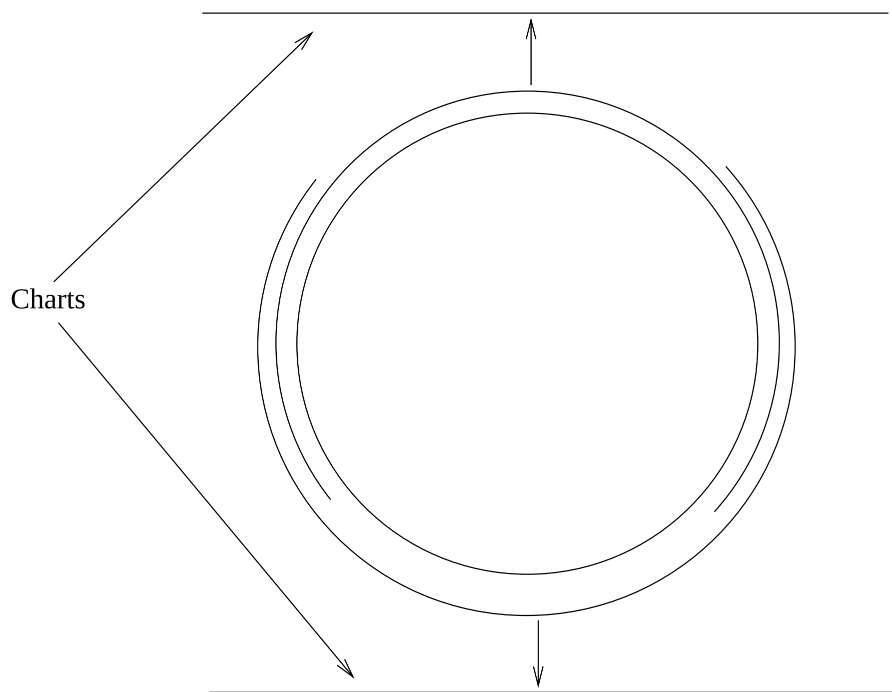


Figure 2-3: The circle as a manifold.

circle and the interval $(0, 2\pi)$, using the usual angular parametrization. Similarly, removing the point $(-1, 0)$ gives a correspondence between the rest of the circle and the interval $(-\pi, \pi)$. These two charts suffice to cover the circle.

We can, in fact, generalize such coordinate systems to higher-dimensional spheres. In dimensions higher than 1, though, no single choice of charts is completely natural. We could use cylindrical coordinates or spherical coordinates or some other coordinate system. Each choice has its advantage. However, it is not hard to see that we can always choose enough charts to cover all of the sphere. Rather than implementing the circle described above as a special case, here is some code that implements the n-dimensional sphere using *stereographic projection* (see Figure 2-4):

```
;;; Make a chart for the sphere using stereographic projection:

(define (make-stereographic-chart dim pole-dim pole-dir)
  (let* ((ubound 5)
         (dim+1 (+ dim 1))
         (pole (vector:basis dim+1 pole-dim pole-dir)))

    (letrec
        ((in-domain?
          (let ((sphere? (make-imbedded-sphere-test dim)))
```

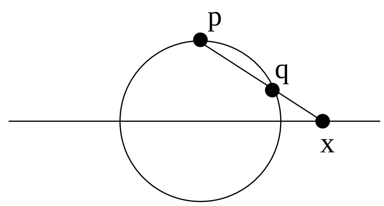


Figure 2-4: Stereographic projection.

```
           (lambda (v)
             (and (sphere? v)
                  (not (almost-equal? (vector:distance^2 v pole) 0))
                  (< (- (/ 4 (vector:magnitude^2 (vector:- v pole))) 1)
                     ubound)))))

        (in-range?
         (let ((euclidean? (make-euclidean-test dim)))
           (lambda (v)
             (and (euclidean? v)
                  (< (vector:magnitude^2 v) ubound)))))

        (map
         (lambda (x)
           (let* ((d (vector:- x pole))
                  (y (vector:* (/ 2 (vector:magnitude^2 d)) d)))
             (vector:drop-coord (vector:+ y pole) pole-dim))))

        (inverse
         (lambda (x)
           (let* ((d (vector:- (vector:add-coord x pole-dim) pole))
                  (y (vector:* (/ 2 (vector:magnitude^2 d)) d)))
             (vector:+ y pole)))))

     (let ((chart (make-simple-chart dim in-domain? in-range? map inverse)))
       (make-spherical-range chart (make-vector dim 0) (sqrt ubound))
       chart))))

;;; Construct the sphere:

(define (make-sphere dim)
  (charts->manifold (list (make-stereographic-chart dim 0 1.)
                          (make-stereographic-chart dim 0 -1.))))
```

Stereographic projection works as follows: Let i be an integer between 1 and n, and let p be a vector of the form $\pm e_i$, where e_i is the ith canonical basis vector of R^n. Then each point q on the sphere is mapped to the plane $\{x_i = 0\}$ by defining x to be the point where the straight line joining p and q intersects the plane. This creates a bijection between the set $S^n - \{p\}$ and the plane $\{x_i = 0\}$, which can be identified with R^{n-1} by dropping the ith coordinate. This defines a chart. The relevant formulae are easy to derive and are left as an exercise for the reader. In the program above, the variable `pole-dim` represents the

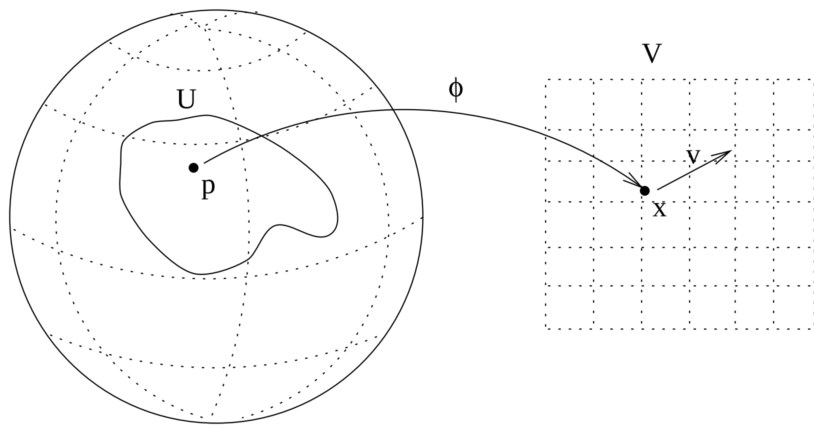


Figure 2-5: A local tangent vector.

index i; it is the dimension singled out for defining the point p (which is the vector `pole`). `Pole-dir` specifies whether p is $+e_i$ or $-e_i$.

Notice that it took quite a bit of work to define such a simple manifold; the implementation of spherical coordinates is even more involved. However, the manifold abstraction lets us separate the definition of the actual space from operations we would like to perform on the abstract space, such as integrating a differential equation. It makes these tasks independent of each other.

2.1.4 Tangent vectors

The manifold construction described above only provides a way for specifying positions of the pendulum. In order to completely capture the dynamical state of the problem, we also need a way to describe the velocity of the point mass.

Consider the evolution of the pendulum: As time goes on, the point mass traces a path on its configuration space, the 2-sphere. We can describe the position at each instant t by a 3-vector $\gamma(t)$ whose distance from the origin is the constant l. The velocity is then the derivative $\dot{\gamma}$. Since the path is imbedded in the 2-sphere, $\dot{\gamma}(t)$ must be tangent to the sphere itself. Conversely, if a vector v is tangent to the 2-sphere at some point p, then there exists a smooth path γ lying entirely in the sphere such that $\gamma(t) = p$ and $\dot{\gamma}(t) = v$ for some t, so that every vector tangent to the 2-sphere describes the velocity of some possible path of the pendulum. Velocities, then, are naturally described by vectors tangent to the configuration manifold, and we can define velocities for arbitrary configuration spaces by generalizing tangent vectors to manifolds.

We can arrive at a general definition of tangent vectors on manifolds as follows: First, note that if we are given a chart $C = (U, V, \phi)$, then *locally* a tangent vector at $p \in U$ can be represented by a vector v "anchored" to the coordinate vector $x = \phi(p)$ in the chart (see Figure 2-5). We call the object (C, p, v) a *local tangent vector*. Now, in order for tangent vectors to be *coordinate-independent*, there must be a consistent way of transforming them between charts, and locally they must always behave like the derivatives of paths. That is, if C_1, C_2, and C_3 are overlapping charts, and γ is a path on M, then there should be locally-defined transformations T_{ij} such that $\phi_j \circ \gamma = T_{ij} \circ \phi_i \circ \gamma$, and such that the derivative of $\phi_i \circ \gamma$ in C_i is carried to the derivative of $\phi_j \circ \gamma$ in C_j. This requires that applying T_{12} to some vector v, followed by T_{23}, yields the same result as applying T_{13} directly. In view of the chain rule, the transformation T_{ij} must be the transformation represented by the Jacobian matrix $D(\phi_j \circ \phi_i^{-1})$ of the transition map. Thus, we can say two local tangent vectors (C_1, p, v_1) and (C_2, p, v_2) at p are *equivalent* if $D(\phi_2 \circ \phi_1^{-1})(x) \cdot v_1 = v_2$, where $x = \phi_1(p)$. The *tangent vector* corresponding to a given local tangent vector (C, p, v) can then be defined as the set of all local tangent vectors equivalent to (C, p, v). The space of all tangent vectors at a given point p is the *tangent space of M at p*, denoted by T_pM. The union of all tangent spaces is denoted by TM and is called the *tangent bundle*.

This construction defines tangent vectors as *equivalence classes*. Now, each of these equivalence classes, and hence each tangent vector, can be in fact a rather large set of local tangent vectors.[8] While this may seem too abstract to be useful, one should realize here that any local tangent vector in the equivalence class can be used to represent the tangent vector, and the important thing is that there is a consistent rule for transforming local tangent vectors between charts. Similarly, the intrinsic structure of the manifold arises from the way charts overlap, and whether or not the manifold happens to be a subspace of Euclidean space is of secondary importance. In fact, as stated before, there are many important examples of manifolds that are most naturally defined in ways that make them hard to describe as subsets of Euclidean spaces, although in principle this can always be done.[9]

One last remark: A manifold as we have defined it has an intrinsic notion of smoothness, but has no intrinsic notions of distance or size. The property of smoothness is stronger than that of continuity, but not as strong as having a metric for measuring the distance between points. Thus, our constructions in this section have shown that the idea of tangent spaces

[8] *In theory*, these equivalence classes can potentially be uncountably infinite sets. However, the local finiteness requirement in §2.1.2 forces such equivalence classes to be finite, and hence they are representable computationally. These can still be rather large sets, though, if many charts cover a given point.

[9] The result that every abstract n-manifold can be imbedded as a subspace of some Euclidean space R^N is known as the *Whitney imbedding theorem*. Whitney also showed that there always exists an imbedding such that $N \leq 2n$. However, the proof of this theorem requires some rather complicated constructions and hence such imbeddings almost never provide much insight into how one could visualize manifolds.

is really a property of the differentiable structure of the manifold (i.e. its atlas), and not a metric property. A manifold where a particular metric is defined is called a *Riemannian manifold*; the definition of such a metric relies on defining inner products in a smooth way on the tangent spaces of a manifold, using the same methods that we have been using. They are important in applications of differential geometry to physics, but will not be needed in this chapter.

Tangent vectors in Scheme

The implementation of tangent vectors is easy. The constructor `make-tangent` simply packages up the structures for defining a local tangent vector into a convenient Scheme object:

```
(define (make-tangent chart p v)
  ;; p is the (abstract) point to which v is tangent, and v is the *coordinate
  ;; representation* of the tangent vector in the coordinates provided by the
  ;; given chart.
  (vector 'tangent chart p v))
```

Though it is not necessary for later work, it is instructive to consider the tangent space as a vector space. For example, how does one define addition on the tangent space T_pM? One can define vector addition for tangent vectors as follows:

```
;;; Add two tangent vectors:

(define (tangent:+ v w)
  (let ((p (tangent:get-anchor v))
        (q (tangent:get-anchor w)))
    (if (equal? p q)
        (let ((chart (tangent:get-chart v)))
          (make-tangent chart
                        p
                        (vector:+ (tangent:get-coords v)
                                  (chart:push-forward w chart))))
        (error "Cannot add vectors tangent to different points."))))

;;; Push a tangent vector along a chart:

(define (chart:push-forward tv chart)
  (let ((other (tangent:get-chart tv))
        (v (tangent:get-coords tv)))
    (if (eq? chart other)
        v
        (push-forward-in-coords
         (chart:make-transition-map other chart)
         (chart:point->coords (tangent:get-anchor tv) other)
         v))))

(define (push-forward-in-coords f x v)
  (((diff f) x) v))
```

The expression `(chart:push-forward w chart)` computes the image of the tangent vector `w` under the transition map $\phi_2 \circ \phi_1^{-1}$, and `(((diff f) x) v)` applies the Jacobian matrix of f at x to the vector v. The procedure `tangent:get-anchor` extracts the point p, which we call the *anchor* of the tangent vector, from the local tangent vector (C, p, v). Other operations on tangent vectors can be defined in a similar fashion, and scalar multiplication is even simpler:

```
(define (tangent* a v)
  (make-tangent (tangent:get-chart v)
                (tangent:get-anchor v)
                (vector:* a (tangent:get-coords v))))
```

2.1.5 Smooth maps and differentials

Having defined differentiable manifolds, the next natural step is to see how of the usual notions of the calculus carry over. For the sake of simplicity, only the concepts of differential calculus are discussed in this section; a discussion of integration on manifolds would take us too far afield and is thus postponed until the next chapter, where integration becomes a necessary tool.

Recall that in the definition of tangent vectors, charts were used to make the manifold look locally like Euclidean space, where tangent vectors are well-defined. We can define differentiable functions analogously. Let M and N be two differentiable manifolds, and let f be a function from M to N. Let p be any point of M and $q = f(p) \in N$. Then f is *smooth* or *differentiable* if for every chart (U, V, ϕ) containing p and every chart (U', V', ϕ') containing q, the function $\phi' \circ f \circ \phi^{-1}$ mapping V to V' is smooth; that is, if $\phi' \circ f \circ \phi^{-1}$, as a mapping from one subset of an Euclidean space into another, has all orders of partial derivatives. By extension, f is a *real-valued smooth function* on M if it is smooth as a map from M into the manifold R, where R is given the canonical atlas $\{(R, R, id_R)\}$, and smoothness is defined as above. It is easy to verify that when M is R^n, this definition of smoothness is equivalent to the usual one.

Let us now consider the idea of derivatives. As we saw in the discussion of tangent vectors in 2.1.4, derivatives of transition maps provide a natural way to transform tangent vectors from one coordinate system to another. Generalizing this observation, we can say that derivatives of smooth maps between manifolds should transport tangent vectors from one tangent space to another. This is, in fact, not that different from the use of gradients in vector calculus: The directional derivative of a real-valued function is the dot product of its gradient and a unit vector in some given direction. Furthermore, if v is the value of the gradient of a function at some point, and w is a vector, then mapping w by the linear transformation A to the vector Aw while keeping $v \cdot w$ an invariant quantity requires that v be mapped to vA^{-1}. Thus, coordinate representations of gradients actually change by a transformation *opposite* that of vectors. This shows that derivatives evaluated at a given point are not vectors, but are linear functionals. This is exactly the kind of duality captured by the use of row and column vectors in elementary calculus.

More formally, let M and N be differentiable manifolds, and let f be a smooth function from M to N. Let p be any point in M, and let $q = f(p)$. Consider the map that takes a local tangent vector (C, p, v), $C = (U, V, \phi)$, to (C', q, w), $C' = (U', V', \phi')$, where

$$w = D(\phi' \circ f \circ \phi^{-1})(\phi(p)) \cdot v. \tag{2.1}$$

One can easily check that if two local tangent vectors represent the same tangent vector in T_pM, then their images under this map also represent the same tangent vector in T_qN. Thus, the mapping can be used to define a map df_p from the tangent space T_pM to T_qN. Furthermore, one could see from the definition that the map is linear on local tangent vectors in the same chart, and hence df_p is a linear transformation between tangent spaces as well. The function df that assigns to each point p the linear transformation df_p is the *differential* of f.

Note that, in this notation, the chain rule can be stated very simply:

$$d(g \circ f)_p = dg_q \circ df_p, \tag{2.2}$$

where $q = f(p)$. This simply restates the usual chain rule while making the role of the differential as a mapping between tangent spaces explicit.

Computing differentials of smooth maps

The implementation of smooth maps is complicated by the implementation of differentiation in Scheme.[10] As a result, the constructor `make-smooth-map` takes four arguments: A

[10] The problem is that the differentiation of functions in our Scheme system depends not only on the values of the function over its domain, but also on the procedures that compute the function. In particular, the procedure to be differentiated must be a compsosition of elementary functions, such as `sin`, `cos`, and `exp`. Difficulties arise, then, in situations where a smooth map is "differentiated twice."

More precisely, let M and N be differentiable manifolds, and let f be a smooth map from M to N. We can define the function Tf from TM to TN by:

$$Tf(p, v) = (p, df_p(v)), \tag{2.3}$$

where we have used the short-hand (p, v) to denote a tangent vector v in T_pM and its anchor p.

Since tangent vectors are computationally represented by local tangent vectors, the procedure that computes $Tf(p, v)$ needs to first find a chart of N containing $f(p)$. When computing Tf, the function must *choose* a chart in the range of f before it could differentiate the transition function $\phi' \circ f \circ \phi^{-1}$ (where ϕ is a coordinate map on M and ϕ' a coordinate map on N). Thus, the procedure computing Tf is no longer a composition of primitive procedures because of this need to choose a chart in N, and the system encounters errors when attempting to compute $T(Tf)$ directly. One must therefore take care in forming transition functions using smooth maps.

manifold, `domain`; another manifold, `range`; a procedure that actually computes the function, `point-function`; and a procedure that constructs transition maps, `make-transition`. However, for most purposes, smooth maps can be constructed using `make-simple-map`, which only needs the first three arguments and requires that `point-function` is a composition of primitive Scheme functions. Another procedure, `make-real-map`, is also provided for convenience; it packages a real-valued function on a manifold into a `smooth-map` structure. Smooth maps cannot be called directly as functions, but may be applied using `apply-smooth-map`.

Here are some examples that will become useful when we discuss Lagrangian mechanics:

```
;;; Euclidean 3-space...

(define R^3 (make-euclidean-space 3))

;;; And its tangent bundle.

(define TR^3 (make-tangent-bundle R^3))


;;; The Lagrangian for a particle traveling in a uniform graviational field.
;;; It's just the difference between the kinetic energy, 1/2*|v|^2, and the
;;; potential energy, z, where v is the velocity and p = (x,y,z) is the
;;; position (in 3-space) of the point mass (assume m = l = 1).

(define falling-lagrangian
  (make-real-map
   TR^3 (lambda (p)
          (- (* 1/2 (vector:magnitude^2 (tangent:get-coords p)))
             (vector-third (tangent:get-anchor p))))))


;;; This restricts the Lagrangian above to the unit sphere, effectively forming
;;; a Lagrangian for the spherical pendulum.

(define S^2 (make-sphere 2))

;;; Define the identity map from the 2-sphere into 3-space, then differentiate
;;; it to extend the function to the tangent bundle.

(define spherical-inclusion
  (smooth-map:diff (make-simple-map S^2 R^3 identity)))

;;; This composition restricts the domain of the Lagrangian to the 2-sphere.

(define spherical-lagrangian
  (smooth-map:compose falling-lagrangian spherical-inclusion))
```

Here's an example of how the function can be used:

```
;;; The tangent bundle of the sphere is the state space of the spherical
;;; pendulum:

(define TS^2 (make-tangent-bundle S^2))
```

```
;;; Define the south pole of the sphere.

(define p (vector 0 0 -1))
;Value: p

;;; Find a chart.

(define chart (manifold:find-chart S^2 p))
;Value: chart

;;; Make a tangent vector.

(define v (make-tangent chart p (vector 0 1)))
;Value: v

;;; Compute the Lagrangian.  Note that, because Euclidean spaces are all
;;; constructed using a single procedure, elements of R^1 are actually vectors
;;; containing a single element, *not* real numbers (as is customary).

(apply-smooth-map spherical-lagrangian v)
;Value 61: #(1.5)

;;; Find a chart for the tangent vector itself in the tangent bundle.

(define another-chart (manifold:find-chart TS^2 v))
;Value: another-chart

;;; Make a tangent vector (this object lives in T(TS^2)).

(define w (make-tangent another-chart v (vector 0 0 1 0)))
;Value: w

;;; Apply the differential of the Lagrangian:

(define u (apply-smooth-map (smooth-map:diff spherical-lagrangian) w))
;Value: u

;;; u should be an object in TR.  Its anchor is the value of the Lagrangian at
;;; v.

(tangent:get-anchor u)
;Value 67: #(1.5)

(tangent:get-coords u)
;Value 68: #(0.)
```

2.1.6 Tangent bundles

A useful thing to notice, at this point, is that the tangent bundle is itself a differentiable manifold. More precisely, if M is an n-manifold, then TM is an $2n$-dimensional manifold. To see this, suppose $C = (U, V, \phi)$ is a chart on M. Then we can define the chart $TC = (TU, V \times R^n, \psi)$, where TU (by an abuse of notation) denotes the union of the tangent spaces T_pM for which $p \in U$, and is hence an open subset of TM, and ψ is the map defined by:

$$\psi(C, p, v) = (\phi(p), d\phi_p(v)), \tag{2.4}$$

where (C, p, v) is a local tangent vector in C. The expression $d\phi_p(v)$ makes sense because V is an open subset of R^n, and we can thus treat ϕ as a smooth map between manifolds and compute its differential. Furthermore, the tangent space of V is trivially equal to R^n at each point, so the dimension of TM is twice the dimension of M. The chart TC is called a *tangent chart*, and the tangent bundle TM is given the atlas consisting of the set of all tangent charts.

Implementation in Scheme

The construction of tangent bundles builds on tangent vectors, and the most important part is the construction of tangent charts:

```
;;; Construct a tangent chart:

(define (make-new-tangent-chart chart)

  ;; First, extract some useful information from CHART:

  (let* ((dim (chart:dimension chart))
         (2*dim (* 2 dim))

         (in-M-domain? (chart:get-membership-test chart))
         (in-M-range? (chart:get-range-test chart))

         (M-map (chart:get-coord-map chart))
         (M-inverse (chart:get-inverse-map chart))

         (dim-vector? (make-euclidean-test dim))
         (2*dim-vector? (make-euclidean-test 2*dim)))

    (letrec
        ((in-domain?
          (lambda (v)
            (and (in-M-domain? (tangent:get-anchor v))
                 (dim-vector? (tangent:get-coords v)))))

         (in-range?
          (lambda (v)
            (and (2*dim-vector? v)
                 (in-M-range? (vector-head v dim)))))

         (coord-map
          (lambda (v)
            (vector-append (M-map (tangent:get-anchor v))
                           (chart:push-forward v chart))))

         (inverse-map
          (lambda (x)
```

```
        (make-tangent chart
                      (M-inverse (vector-head x dim))
                      (vector-end x dim))))

     (transition
      (lambda (Tother)
        (let* ((other (chart:get-base-chart Tother))
               (f (chart:make-transition-map chart other)))
          (lambda (x)
            (let ((anchor (vector-head x dim))
                  (tangent (vector-end x dim)))
              (vector-append (f anchor)
                             (push-forward-in-coords
                              f anchor tangent))))))))

  (let ((new-chart (make-chart 2*dim in-domain? in-range?
                               coord-map inverse-map transition)))
    ;; Some auxiliary information:
    (chart:install-extra new-chart 'base-chart (delay chart))
    (chart:install-extra chart 'tangent-chart (delay new-chart))
    new-chart))))
```

This procedure can then be used to construct tangent bundles.

2.1.7 Making new manifolds

As noted in the previous section, the tangent bundle of a manifold is also manifold. This gives us a way to construct new manifolds out of old ones. In this section, we will take a look at a few other ways of constructing new manifolds out of existing ones.

Product manifolds. First, consider two manifolds M and N. Let (U, V, ϕ) be a chart on M, and let (U', V', ϕ') be a chart on N. The *product chart* associated with the two charts is the chart $(U \times U', V \times V', \phi \times \phi')$, where $U \times U'$ is the *Cartesian product* $\{(x, y) : x \in U, y \in U'\}$, $V \times V'$ is similarly defined, and $\phi \times \phi'$ is the map taking $(x, y) \in U \times U'$ to $(\phi(x), \phi'(y)) \in V \times V'$. The *product manifold* $M \times N$, then, is the manifold whose space is the Cartesian product of the spaces M and N, and whose atlas is given by the set of all product charts. If the dimension of M is m and that of N is n, then the dimension of $M \times N$ is $m + n$. For example, the Euclidean space R^n, $n > 1$, may be constructed as a product manifold $R^{n-1} \times R$, and the *torus* can be thought of as the product manifold $S^1 \times S^1$ (where S^n denotes the n-dimensional sphere, and hence S^1 is the circle).

Cotangent bundles. Recall now that every vector space has a *dual space* of *linear functionals*. Thus, to every tangent space T_pM, we can find its dual vector space T_p^*M. It turns out that the union T^*M of all dual spaces T_p^*M is also a differentiable manifold, by using a construction similar to that of the tangent bundle. The space T^*M is called the *cotangent bundle* of the manifold, and is just as important geometrically as the tangent bundle, if not more so. In classical mechanics, the cotangent bundle of a configuration space is called its *phase space.* Whereas the state space describes a system by its position and velocity, the phase space describes a system by its position and *generalized momentum.*

The inverse function theorem. Finally, there is a method of constructing manifolds that is very useful theoretically, but practically useless for computation: *The inverse function theorem.* Briefly, it states that if f is a smooth map from M into N, the dimension of N is less than the dimension of M, and for some point $q \in N$, every $p \in M$ such that $f(p) = q$ has a surjective differential df_p, then the inverse image $f^{-1}(q) = \{p \in M : f(p) = q\}$ is a smooth manifold. Furthermore, if the dimension of M is m and that of N is n, then the dimension of this new manifold is $m - n$. For theoretical purposes, this is a very useful way of constructing manifolds, especially for describing constraints in mechanical systems. For example, the configuration space for a free particle is R^3, and if one were to enforce the constraint that the particle must stay at a constant distance l from the origin, this theorem immediately tells us that the resulting space (the sphere, in this case) is a differentiable manifold. However, the proof of this theorem involves some non-constructive arguments, and hence it cannot be used directly for computation. The efficient computation of general inverses of functions is, at the present, not possible.

Since most of these constructions (except the cotangent bundle) will not be used directly in later sections, their implementation will not be discussed here. The cotangent bundle will appear again when we discuss the Hamiltonian approach to mechanics.

2.1.8 Boundaries

Our definition of manifolds does not allow for spaces with *boundaries.* For example, notice that the *unit disc*

$$\{(x, y) \in R^2 : x^2 + y^2 \leq 1\} \tag{2.5}$$

is *not* a manifold by our definition, because the points (x, y) for which $x^2 + y^2 = 1$ (that is, those lying on the boundary of the disc) do not have neighborhoods that "look like" open subsets of R^2. However, it locally have the structure of an *Euclidean half-space* (see Figure 2-6). Since boundaries are often useful in applying partial differential equations to model physical systems, this section takes a closer look at this concept.

In order to describe manifolds with boundaries, a new type of chart is necessary. First, some definitions: Given an Euclidean space R^n, let R^n_+ be the *half-space*

$$R^n_+ = \{x \in R^n : x_n \geq 0\}, \tag{2.6}$$

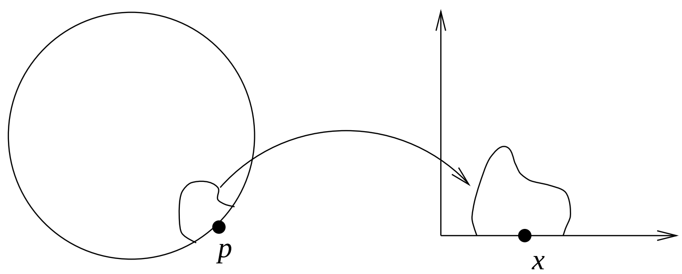


Figure 2-6: A boundary chart.

where x_n denotes the nth component of the n-vector x. A *boundary chart*, depicted in Figure 2-6, is then a triple (U, V, ϕ), where U is a subset of M, V is the (non-empty) intersection of some open subset V' of R^n with the half-space R^n_+, and ϕ is a bijection between U and V. The usual definition of compatibility between charts still applies to boundary charts, although what it means to be differentiable at the boundary $\{x_n = 0\}$ requires more careful analysis (omitted here).

Now suppose M is an arbitrary set, and extend the definition of atlases to allow boundary charts. A set M is a *manifold with boundary* if it has an atlas $\mathcal{A}$ with mutually compatible charts and boundary charts. If a point p has the property that for some boundary chart (U, V, ϕ), $x_n = 0$, where $x = \phi(p)$, then p is said to lie on the boundary of the manifold M.[11] The boundary of a manifold M is usually denoted by ∂M, and consists of the set of points that lie on the boundary of some boundary chart. It is easy to verify that the boundary of a manifold is itself a manifold without boundary.[12]

The computational implementation of boundaries is not used in the rest of this chapter, so its discussion is postponed until it is needed in the next chapter on partial differential equations and boundary value problems.

2.2 Vector fields and differential equations

2.2.1 Smooth vector fields

The tangent bundle construction actually facilitates the definition of smooth vector fields: Let π denote the projection map from TM into M, defined by:

[11] It is easy to verify that if p lies on the boundary according to one chart, then it must lie on the boundary according to all the charts.

[12] Manifolds with boundaries introduce some problems into the theory. For example, the class of differentiable manifolds with boundary is not closed under the product manifold construction: Consider the unit interval $I = [0, 1]$. It is a differentiable manifold with boundary, and yet the product manifold $I \times I$ is not a differentiable manifold with boundary—Transition maps will fail to be smooth at the corners of the square.

$$\pi(p, v) = p. \tag{2.7}$$

That is, the projection map π extracts the "anchor" of the tangent vector, much like the procedure `tangent:get-anchor`. A *smooth vector field on M* is then a smooth map v from M into TM, such that for every point $p \in M$, the equation $\pi(v(p)) = p$ holds.

It is easy to verify that, over each chart, a smooth vector field as defined here corresponds to what one usually means by a smooth vector field. Thus, the usual local existence and uniquness theorems apply. This abstraction lets one define systems of first-order ordinary differential equations, and higher-order equations are typically handled by using the tangent bundle construction. A second-order equation, for example, can be thought of as a vector field on the tangent bundle, and so on. This is why mathematical descriptions of mechanics problems involve vector fields (first-order equations) on tangent or cotangent bundles of manifolds.

2.2.2 Flows generated by smooth vector fields

How can we integrate ODEs on manifolds? Since within each chart (U, V, ϕ), the manifold "looks like" Euclidean space, the obvious thing to try is to use the coordinate map ϕ to "push" vector field onto the Euclidean subspace V. More precisely, suppose we are given a tangent vector that is represented by the local tangent vector (C', p, v'), and wish to map this local tangent vector over to the chart in which we are integrating the equations, $C = (U, V, \phi)$. Then we can simply apply the Jacobian of the transition map, to obtain (C, p, v), where v is defined by:

$$v = D(\phi \circ \phi'^{-1})(\phi'(p)) \cdot v'. \tag{2.8}$$

This consistently transforms the local tangent vector to the other chart. Thus, a smooth vector field on M can always be turned into a local vector field on the open subset V of Euclidean n-space, for which there exist numerous methods of integration.

The computational implementation of ODE integrators on manifolds, however, requires that we consider a few more issues. For example, for the sake of flexibility and efficiency, it is easier to implement vector fields directly as procedures which return local vector fields when given a chart, rather than a procedure that actually returns a local tangent vector representation of some tangent vector every time whenever it is given a point on the manifold. This is because, in some situations, it may be easier for procedures that compute vector fields to use internal representations that are not in the form of local tangent vectors. If the procedure must convert its internal representation to a local tangent vector,

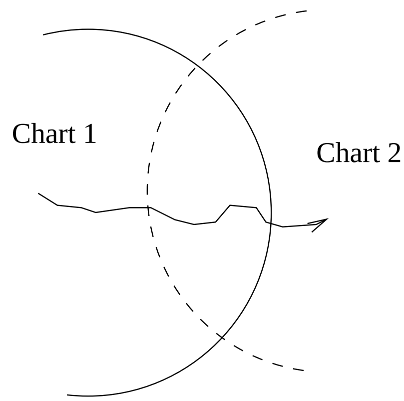


Figure 2-7: When should the ODE integrator switch charts?

as the integrator requires, it might as well directly convert it to the current chart. This structure gives procedures this flexibility, as will be demonstrated in later examples.

A more serious issue is that of switching between charts (see Figure 2-7): As the integrator moves along in one chart, taking discrete steps forward in time, it will eventually step off the chart. One solution is to always watch where the next step "lands" before actually committing to it, and to switch charts if the next step is outside the current chart. This approach has the problem that when switching charts, one needs to keep track of which charts have already been visited so that the integrator does not enter an infinite loop, idly switching from one chart to another without making progress. However, this introduces quite a bit of complexity into the integrator, and did not seem to be the best design for a first attempt.

There is, in fact, a more elegant solution to the problem of switching charts: Simply evolve the trajectory in all possible charts! This solution requires that the atlas be *locally finite*—For every point p, there must be only finitely many charts in $\mathcal{A}$ that contain p. This is not an overly restrictive requirement, and in general it is easy for the user to control the amount of overlap between charts when constructing them, so that there is not too much overhead in the multiple evaluation. This is the strategy finally chosen, and the main idea is expressed in the code below:[13]

```
;;; This is a simple description of the integration algorithm for ODEs on
;;; manifolds:

(define (v.field->flow manifold make-local-field next-step error-est)

  ;; Integrate the ODE starting at p0, with time index running from t0 to t1:
```

[13]This is not the final version of code used, but expresses the main ideas.

```
(lambda (p0 t0 t1)
  (let loop ((p p0) (t t0))
    (if (<= t t1)

        ;; Compute the possible next steps, then choose the one that
        ;; minimizes the error estimator, ERROR-EST.

        (let* ((charts (manifold:get-local-atlas manifold p))
               (p1 (minimize-function-over-list
                    (compose error-est integrator:get-new-x)
                    (map (lambda (chart)
                           (next-step (chart:point->coords p chart)
                                      (make-local-field chart)))
                         charts)
                    charts)))

          ;; If the local integrator can step forward in at least one chart,
          ;; then we can continue:

          (if p1
              (loop (integrator:get-new-x p1) (+ t (integrator:get-dt p1)))
              (error "Ran out of charts!")))))))
```

Notice a few things about this code: First, it takes four arguments: `Manifold` is just the domain of the ODE; `make-local-field` is the local vector field constructor, as described before; `next-step` is a *local ODE integrator*, a procedure that knows nothing about the manifold but can numerically solve a given ODE in Euclidean coordinates to produce a new coordinate vector; and `error-est`, a function for estimating the local numerical error.

Notice, first, that the integrator has no built-in notions of step size. It simply relies on the local integrator to supply both a new step *and* a step size. This facilitates the use of variable-step-size integrators, which can be more efficient and numerically robust. Second, it requires an error estimator that helps it choose from among the guesses supplied by the different charts. This is an advantage of this method: Because of truncation and round-off errors, numerical computations are not actually coordinate-independent. Thus, this integrator allows local error analysis, which improves accuracy greatly, especially in the presence of coordinate singularities (discussed in §2.3.3).

2.2.3 Manifolds and classical mechanics

There are several reasons why the manifold abstraction is especially suited to dealing with ordinary differential equations. First, notice that a classical n-particle system is described by the configuration space R^{3n}, since each particle has three coordinates. Nontrivial manifolds arise in classical mechanics from *constraints*, such as the constraint that a point mass lies at a constant distance l from the origin (which yields the spherical pendulum). Now, as noted in §2.1.1, in the traditional approach of modeling the manifold as a subset of a larger Euclidean space and integrating the ODEs in the larger space, trajectories can sometimes go off the manifold because of the accumulation of round-off and truncation errors. Thus, physical constraints are not enforced faithfully in this classical approach, whereas the manifold abstraction helps minimize this kind of error. Second, in generating local vector fields, it is useful to have explicit formulas. It is rather tedious, in general, to derive differential equations that describe complex physical systems in different coordinate systems. However, in classical mechanics, one could always use variational methods to derive the equations of motion in different coordinate systems with the aid of computer algebra, which is often easier than transforming second-order equations between coordinate systems.[14]

Furthermore, in classical mechanical systems, the `error-est` function above can be implemented rather easily: Instead of checking the local numerical properties of the chart, one can exploit the existence of *conserved quantities*, such as energy and momentum. This has the advantage that these quantities are often easy to compute, and systems in classical mechanics usually have a sufficient number of such conserved quantities that one could simply check their deviations from initial values as time marches forward to determine how well the integrator is doing.[15]

---

[14] Variational (or Lagrangian) mechanics differs from Newtonian mechanics in the following way: Instead of describing how systems change from moment to moment, as did Newton, one looks at the space of all possible paths through the configuration space that begin at some initial point x_1 at some time t_1 and ends up in some place x_2 at some time t_2. To every such possible path γ, one assigns to it a number (called the *action*) $S(\gamma)$. Then the path actually taken by a particle is the one that is a *stationary point* (in a sense that can be made mathematically precise) of the action S. This is known as the *principle of least action* because for many cases, S is actually minimized by the real path γ. $S(\gamma)$ is generally computed as the integral of some function L, called the *Lagrangian*, along paths; *Hamilton's principle of least action* then states that the "correct" Lagrangian for many situations is the difference between kinetic and potential energies.

Since the principle of least action is formulated in terms of integrals of real-valued functions over time intervals, it is coordinate-independent. Furthermore, one can derive the equations of motion in terms of the Lagrangian:

$$D(\partial_{\dot{x}} L \circ \gamma) = \partial_x L \circ \gamma, \tag{2.9}$$

where $\partial_{\dot{x}}$ denotes differentiation with respect to the velocity part of the Lagrangian, ∂_x denotes differentiation with respect to the position part of the Lagrangian, and D is the operator that differentiates real-valued functions of one real variable (in this case time). Equation (2.9) is known as the *Euler-Lagrange equation*, and gives a system of second-order equations that determine the stationary path. It can be deduced by using the same technique as in the derivation of Equation (4.19).

This provides an easy way to change coordinate systems: Simply substitute the new coordinates into the Lagrangian, simplify the resulting expression, and derive the Euler-Lagrange equations for the new coordinate system. For more information on this topic, see Arnold [4].

[15] It is also possible to enforce the conservation laws as *constraints*, so that one integrates the equations of motion on *submanifolds* of the state space. While this would ensure that the conservation laws are satisfied exactly (up to round-off error), it also makes checking the accuracy of solutions a little harder—Because the conservation laws were "used up" as constraints, one would now need to perform numerical error analysis to estimate the accuracy of the numerical integration.

Lagrangian mechanics

Although it is extremely inefficient, one can in fact implement Lagrangian mechanics directly using our Scheme system:

```
;;; The Lagrangian should be a smooth map from the tangent bundle of some
;;; manifold into the real line.

;;; This is very slow, as every evaluation of the field involves a matrix
;;; inversion.  Which is why Hamiltonians are *better*, even for comuptational
;;; purposes!

(define (lagrangian->v.field L)
  (let ((TM (smooth-map:get-domain L))
        (R (smooth-map:get-range L)))
    (lambda (p)
      (let ((U
             (if (tangent? p)
                 (make-tangent-chart (tangent:get-chart p))
                 (manifold:find-best-chart TM p))))
        (let ((f (smooth-map:make-transition
                  L U (car (manifold:get-finite-atlas R))))
              (x (chart:point->coords p U)))
          (let ((v (vector-tail x (/ (vector-length x) 2))))
            (let ((E-L (euler-lagrange-in-coords f x)))
              (let ((A (car E-L))
                    (B (cadr E-L))
                    (c (caddr E-L)))
                (let ((accel (matrix:solve-linear-system
                              A
                              (vector:+ (apply-linear-transformation B v) c))))
                  (make-tangent U p (vector-append v accel))))))))))


;;; Derive the Euler-Lagrange equations for f at x (in coordinates) in the form
;;; A*xdotdot = B*xdot + c.

(define (euler-lagrange-in-coords f x)
  (let* ((n (/ (vector-length x) 2))
         (A (make-matrix n n))
         (B (make-matrix n n))
         (c (make-vector n 0)))

    (do ((i n (+ i 1))
         (p 0 (+ p 1)))
        ((>= p n))

      ;; First, compute the hessian of f with respect to the velocity part of
      ;; the independent variable:

      (matrix-set! A p p (vector-first (((pdiff i) ((pdiff i) f)) x)))

      (do ((j (+ i 1) (+ j 1))
           (q (+ p 1) (+ q 1)))
          ((>= q n))
        (let ((val (vector-first (((pdiff j) ((pdiff i) f)) x))))
          (matrix-set! A p q val)
          (matrix-set! A q p val)))
```

```
      ;; Next, compute the rest of the terms involving the partials of the
      ;; Lagrangian with respect to the positions (note the minus sign):

      (do ((j 0 (+ j 1)))
          ((>= j n))
        (let ((val (- (vector-first (((pdiff j) ((pdiff i) f)) x)))))
          (matrix-set! B p j val)))

      ;; And then there's the term due to the derivative of the Lagrangian with
      ;; respect to the position variables:

      (vector-set! c p (vector-first (((pdiff p) f) x))))

    (list A B c)))


;;; In many mechanics problems, it's natural to check conservation laws:

(define (check-vector-conservation-law quantity ref-point)
  (let ((ref (quantity ref-point)))
    (lambda (chart tangent)
      (vector:distance (quantity (tangent:get-anchor tangent)) ref))))
```

The cost of inverting the matrix (when `matrix:solve-linear-system` is called) makes this a prohibitively slow way to compute vector fields, but it does work.

Hamiltonian mechanics

A slightly more efficient form of automatically generating vector fields is provided by the Hamiltonian point of view.[16] It can be implemented much more directly:

```
;;; The Hamiltonian should be a smooth map from the cotangent bundle of some
;;; manifold into the real line.

(define (hamiltonian->v.field H)
  (let ((T*M (smooth-map:get-domain H))
        (R (smooth-map:get-range H)))
    (lambda (p)
      (let ((U (manifold:find-best-chart T*M p)))
        (make-tangent U p
                      (hamilton-in-coords
                       (smooth-map:make-transition
```

[16]The Hamiltonian formulation describes mechanics using position and momenta, instead of position and velocity. The space of states here is the cotangent bundle of the configuration space, not its tangent bundle. And, finally, the dynamics is described by the *Hamiltonian*, which is a function that in many cases agrees with the energy function. As with Lagrangian mechanics, Hamiltonian mechanics also lets us change coordinates easily; the analogous equations of motion for a given Hamiltonian H are:

$$\begin{aligned} \dot{q} &= \partial_p H, \\ \dot{p} &= -\partial_q H, \end{aligned} \tag{2.10}$$

where q denotes position, p denotes momentum, and ∂_p and ∂q denote the corresponding differential operators. These are *Hamilton's equations.* Notice that they are antisymmetric, and do not require a matrix inversion to isolate the highest-order derivatives.

```
                     H U (car (manifold:get-finite-atlas R)))
                    (chart:point->coords p U)))))))


;;; Derive Hamilton's equations for f at x (in coordinates):

(define (hamilton-in-coords f x)
  (let* ((2n (vector-length x))
         (v (make-vector 2n))
         (n (/ 2n 2)))

    (do ((i n (+ i 1))
         (j 0 (+ j 1)))
        ((>= j n) v)

      (vector-set! v i (- (vector-first (((pdiff j) f) x))))
      (vector-set! v j (vector-first (((pdiff i) f) x))))))
```

However, this is still rather inefficient due to the evaluation of the partial derivatives. In the numerical experiments that follow, the appropriate vector fields are pre-computed for each chart in the relevant manifold.

2.3 Numerical experiments

Finally, this section presents the results of three numerical experiments.

2.3.1 The circle field

The first example is a simple integration around a circle. The vector field simply consists of unit vectors going counter-clockwise around the circle, and the trajectories of this system of equations are simply unit-velocity curves around the circle:

$$\gamma(\theta) = (\cos(\theta - \theta_0), \sin(\theta - \theta_0)), \tag{2.11}$$

where the phase shift θ_0 comes from the initial condition.

This can be implemented easily as follows:

```
;;; First, construct the circle:

(define circle (make-sphere 1))


;;; Here's a trivial vector field on the circle:

(define (circle-field p)
  (let ((x (vector-ref p 0))
        (y (vector-ref p 1)))

    ;; IMBEDDING->TANGENT takes an imbedded tangent vector to the tangent
    ;; bundle of the given (imbedded) manifold.
```

```
    (imbedding->tangent circle p (vector (- y) x))))


;;; Integrate the ODE:

(define circle-path
  (v.field->flow circle
                 (v.field->local-field-maker circle-field)
                 (make-rk4-integrator (* 2 pi 1e-3))
                 ;; LOCAL-DISTORTION checks the numerical error in the current
                 ;; chart.
                 local-distortion))


;;; The real answer (with no phase shift):

(define (real-circ t)
  (vector (cos t) (sin t)))


;;; Here is a test run: After 2*pi seconds, the path should end up where it
;;; started.  Let's compare the results of using the manifold and using the
;;; tranditional approach:

(define result (circle-path (vector 1 0) (* 2 pi)))
;Value: result

;;; RESULT is a list of pairs of the form (time-index position), sorted in
;;; *descending* order by time index.  Thus, (CAAR RESULT) returns the final
;;; time index, and (CADAR RESULT) returns the final position.

;;; The difference in time index:

(abs (- (caar result) (* 2 pi)))
;Value: 1.127986593019159e-13

;;; The difference in position:

(vector:distance (cadar result) (vector 1 0))
;Value: 4.447015332496363e-14


;;; Here is the more tranditional approach: Simply embed the circle in the
;;; plane, and integrate in two real variables (and hope the trajectory
;;; actually stays on the circle):

(define (traditional-circle-field p)
  (let ((x (vector-ref p 0))
        (y (vector-ref p 1)))
    (vector (- y) x))
;Value: traditional-circle-field

(define traditional-result
  (let ((next-step (make-rk4-integrator (* 2pi 1e-3))))
    (let loop ((t 0) (x (vector 1 0)) (result '()))
      (if (<= t 2pi)
          (let* ((new (next-step x traditional-circle-field (lambda () #f)))
                 (dt (integrator:get-dt new))
```

```
               (new-x (integrator:get-new-x new)))
          (loop (+ t dt) new-x (cons (list t x) result)))
        result))))
;Value: traditional-result


;;; The error in time index is the same:

(abs (- (caar traditional-result) 2pi))
;Value: 1.127986593019159e-13


;;; The error in position is actually larger: This is because, as stated
;;; before, the traditional method allows the trajectory to veer off the
;;; circle, whereas the manifold approach enforces the constraint strictly.

(vector:distance (cadar traditional-result) (vector 1 0))
;Value: 8.16059276567945e-11
```

Notice that the manifold approach actually produced a more accurate "walk" around the circle!

2.3.2 The spherical pendulum

The next example is the one we started out with: The spherical pendulum. As opposed to our previous example, this one actually comes from a physical problem. Furthermore, this particular problem can be understood analytically, so that the motion generated by the integrator can be checked closely for consistency with the actual physical situation.

For this integration, the integration is done on the phase space (the cotangent bundle of the sphere). The vector field could very well have been generated using the following Hamiltonian:

```
;;; The phase space:

(define T*R^3 (make-cotangent-bundle R^3))


;;; The Hamiltonian for a point mass in a uniform gravitational field:

(define falling-hamiltonian
  (make-real-map
   T*R^3 (lambda (p)
          (+ (* 1/2 (vector:magnitude^2 (cotangent:get-coords p)))
             (vector-third (cotangent:get-anchor p))))))

;;; Define the Hamiltonian:

(define T*S^2 (make-cotangent-bundle S^2))

(define spherical-inclusion*
  (let* ((chart (car (manifold:get-finite-atlas R^3)))
         (f (lambda (v)
              (apply make-cotangent
```

```
                    (cons chart (cotangent->imbedding S^2 v))))))
    (make-simple-map T*S^2 T*R^3 f)))

(define spherical-hamiltonian
  (smooth-map:compose falling-hamiltonian spherical-inclusion*))


;;; We can even generate the vector field from the Hamiltonian directly:

(define spherical-field
  (hamiltonian->v.field spherical-hamiltonian))

(define spherical-init
  (imbedding->cotangent S^2 (vector 1 0 0) (vector 0 1 .5)))

(define spherical-path
  (v.field->flow T*S^2
                 (v.field->local-field-maker spherical-field)
                 (make-rk4-integrator 1e-3)
                 (check-vector-conservation-law
                  (smooth-map:get-point-function spherical-hamiltonian)
                  spherical-init)))


;;; Try to integrate a few time steps:

(define result
  (show-time
   (lambda ()
     (spherical-path spherical-init .01))))
process time: 122020 (95550 RUN + 26470 GC); real time: 135198
;Value: result


(/ 135198 1000. 60)            ;; 135198 msec. = 2.25 minutes.
;Value: 2.2533000000000003

(length result)
;Value: 10

(for-each
 (compose write-line
          (smooth-map:get-point-function spherical-hamiltonian)
          cadr)
 result)
#(.6250000000009046)
#(.6250000000008399)
#(.6250000000008247)
#(.6250000000008603)
#(.6249999999981802)
#(.6249999999053644)
#(.6249999999049425)
#(.6249999999039165)
#(.6249999999048136)
#(.625)
;No value
```

As seen above, this approach produces reasonable answers: For a short integration, the Hamiltonian (which equals energy, in this case) is conserved, as expected. However, this approach is very inefficient. Instead, one could derive Hamilton's equations for this Hamiltonian over some atlas of the 2-sphere, and carry these local vector fields to other charts.

```
(define make-spherical-pendulum
  (let* ((C1 (make-cotangent-chart (make-spherical-chart 2 '(2 0 1) 0)))
         (C2 (make-cotangent-chart (make-spherical-chart 2 '(1 0 2) pi)))
         (T*S^2 (charts->manifold (list C1 C2))))
    (lambda (g mass length)
      (let ((k1 (/ (* mass (square length))))
            (k2 (* mass g length)))
        (lambda (p)
          (let* ((chart (manifold:find-best-chart T*S^2 p))
                 (x (chart:point->coords p chart))
                 (phi (vector-ref x 0))
                 (theta (vector-ref x 1))
                 (p_phi (vector-ref x 2))
                 (p_theta (vector-ref x 3)))
            (make-tangent chart
                          p
                          (if (eq? chart C1)
                              (vector (* k1 p_phi)
                                      (* (/ k1 (square (sin phi))) p_theta)
                                      (+ (* k1 (square p_theta)
                                            (/ (* (square (sin phi))
                                                  (tan phi))))
                                         (* k2 (sin phi)))
                                      0)
                              (vector (* k1 p_phi)
                                      (* (/ k1 (square (sin phi))) p_theta)
                                      (+ (* k1 (square p_theta)
                                            (/ (* (square (sin phi))
                                                  (tan phi))))
                                         (* k2 (cos phi) (sin theta)))
                                      (* k2 (sin phi) (cos theta))))))))))))
```

This way of defining vector fields requires a bit more work, and tends to produce rather unreadable programs. However, it is sufficiently fast to generate some real data. The local integrator used is a simple 4th-order Runge-Kutta with a fixed step size of 1×10^{-3}, and the constants are normalized so that $l = g = m = 1$. The initial condition, in these units, is $q = (1, 0, 0), p = (0, 1, 0.5)$.

Figure 2-8 shows the relative error in energy conservation, and Figure 2-9 shows the relative error in angular momentum conservation.

Notice that in the code for the integration, `check-vector-conservation-law` was only asked to minimize the error in energy conservation. Hence, in Figure 2-8, the relative error in energy conservation has been kept rather constant. However, the error in angular momentum makes a few large jumps, probably at the occasions when the integrator decides to switch charts. This indicates that in order to obtain the most accuracy, perhaps one

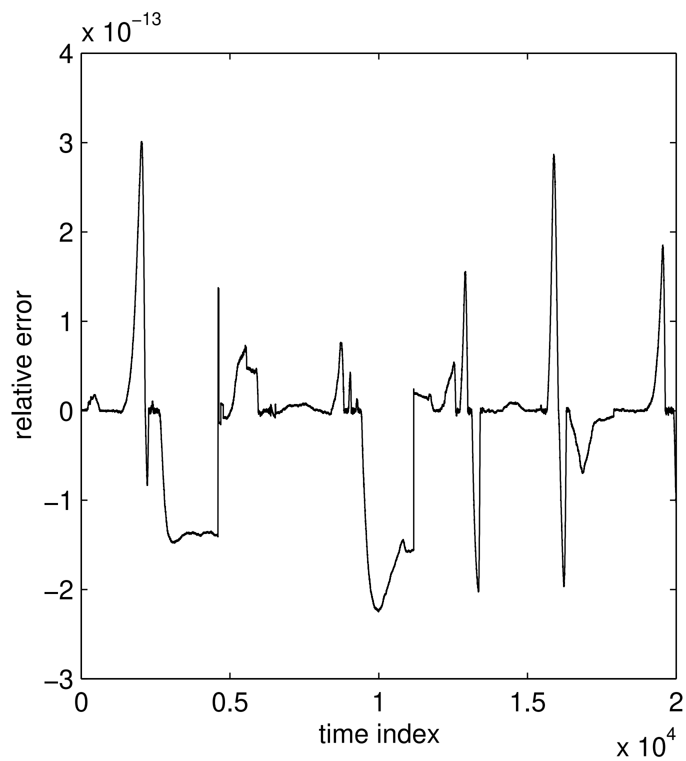


Figure 2-8: Relative error in energy conservation for the spherical pendulum.

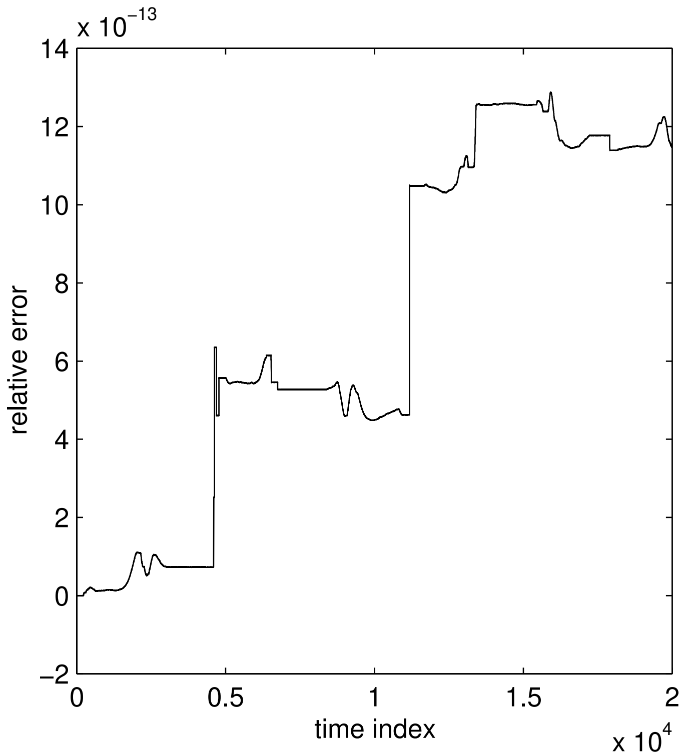


Figure 2-9: Relative error in angular momentum conservation for the spherical pendulum.

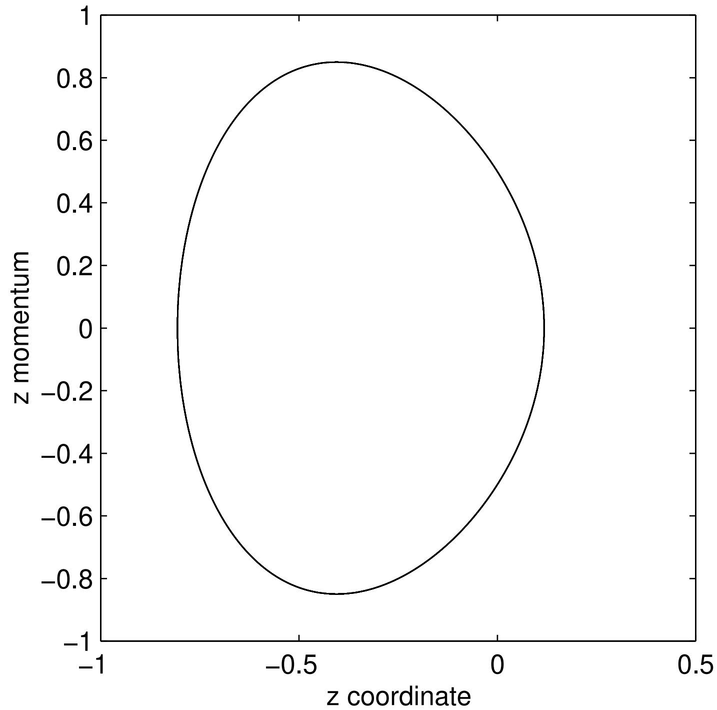


Figure 2-10: A contour of the reduced Hamiltonian for the spherical pendulum.

should try to minimize the error in a number of conservation laws. This is what is done with the example of rigid body motion.

Finally, Figure 2-10 shows more evidence that this integrator has found the correct solution: Since angular momentum is conserved for the spherical pendulum, we know that the angular motion (about the vertical axis) of the pendulum may be decoupled from its vertical motion, and the system may be reduced to one with a lower degree of freedom. In this figure, the z vs. p_z plot shows that the trajectory of the *reduced system* is a closed curve. This is because energy is also conserved in the reduced system, and hence trajectories of the reduced system must follow equipotential curves of the reduced Hamiltonian.

2.3.3 Rigid body motion and coordinate singularities

Our last example, and the most important, is rigid body motion. Its importance stems from the fact that, although the vector fields describing its motion are perfectly smooth, the coordinate systems traditionally used to describe it contain *coordinate singularities*, so that usual integrations of rigid body motion can produce inaccuracies near those coordinate singularities.

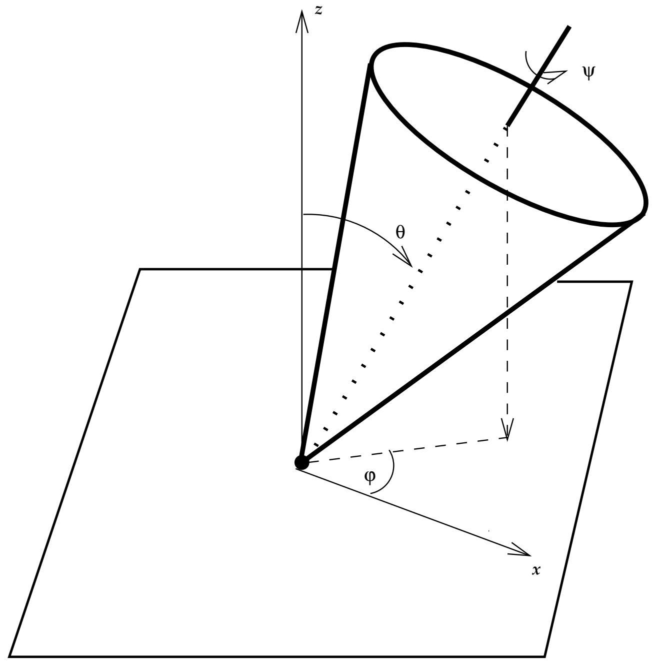


Figure 2-11: Euler angles for a rigid body.

Furthermore, the configuration space for the rotational motion of rigid bodies is the space of all orientations of a rigid body, or equivalently the space of all rotation matrices in three dimensions.[17] The manifold structure of this space is rather abstract, and since it is really a 3-manifold imbedded in the 9-dimensional space of all 3×3 matrices, we can no longer rely on our geometric intuition to approach this problem. This is one of the most important examples of an abstract manifold.

Traditionally, orientations of rigid bodies are described by *Euler angles*, depicted in Figure 2-11. As hinted at earlier, this coordinate system has the problem that the coordinates "blow up" (the Jacobian of the coordinate map becomes singular) when the rigid body is standing vertically, as a bit of analysis will show. This is known as a *coordinate singularity* because the singularity is part of the coordinate system, not a feature of the dynamics.

The traditional approach to this problem is to work entirely in Euler angles. This works well so long as the trajectory does not come near the coordinate singularity. But when it does, the singularity can have a serious effect on numerical accuracy, which is often reflected in fluctuations in the conserved quantities. In this example, the results of a numerical integration of rigid body motion is presented using the traditional and the

[17]This space is commonly denoted as SO_3, the *special orthogonal group*. It is an example of a *Lie group*, which are manifolds that also happen to be groups, and where the group operations are smooth as maps on manifolds.

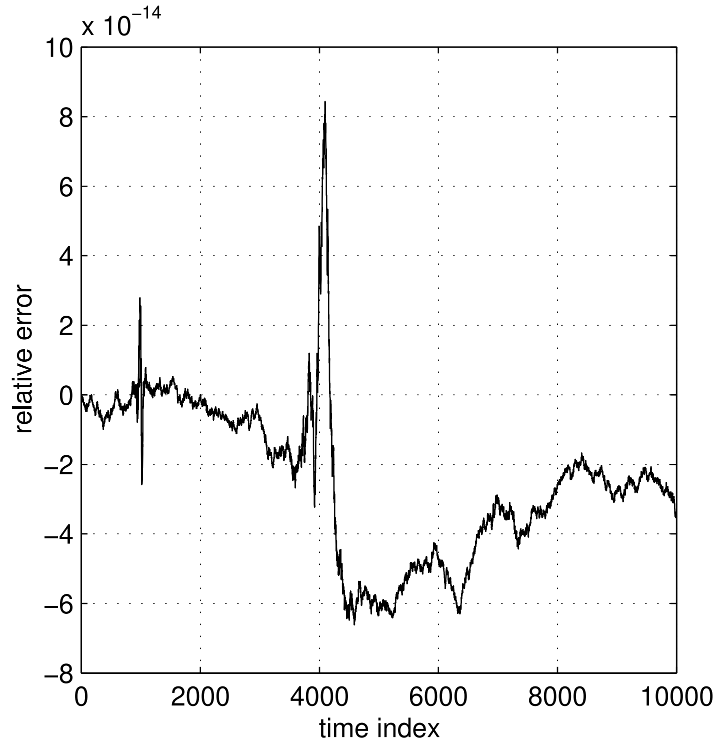


Figure 2-12: Relative error in energy conservation for rigid body motion in Euler angles.

manifold method. The principal moments of inertia of the rigid body are 1, $\sqrt{2}$, and 2, with mass set to $m = 1$. The initial conditions, in Euler angles, are $\phi = 0$, $\theta = 1$, $\psi = 0$, $\dot{\phi} = -0.01$, $\dot{\theta} = -0.1$, and $\dot{\psi} = -0.01$; these initial conditions have been chosen to take the trajectory close to the coordinate singularity in Euler angles, so that the effects of the singularity on conserved quantities can be observed. The integration was performed using a time step of 0.01, for 100.0 time units (which equals 10,000 time steps). The integration in Euler angles used a Bulirsch-Stoer integrator, which the manifold integrator also used as its local integrator.

Figure 2-12 shows the relative error in energy conservation for a trajectory that comes relatively near the singularity. Figure 2-13 shows the analogous plot for the manifold method.

In Figure 2-12, the maximum absolute value is $8.43194301271212 \times 10^{-14}$, and the corresponding average is $2.6428202894715013 \times 10^{-14}$. In contrast, in Figure 2-13, the maximum absolute value of the error is $1.394387463191693 \times 10^{-14}$, and the average absolute value of the error is $4.31070783106112 \times 10^{-15}$. Thus, the manifold approach actually conserves energy better: In terms of relative error, it outperforms the traditional approach

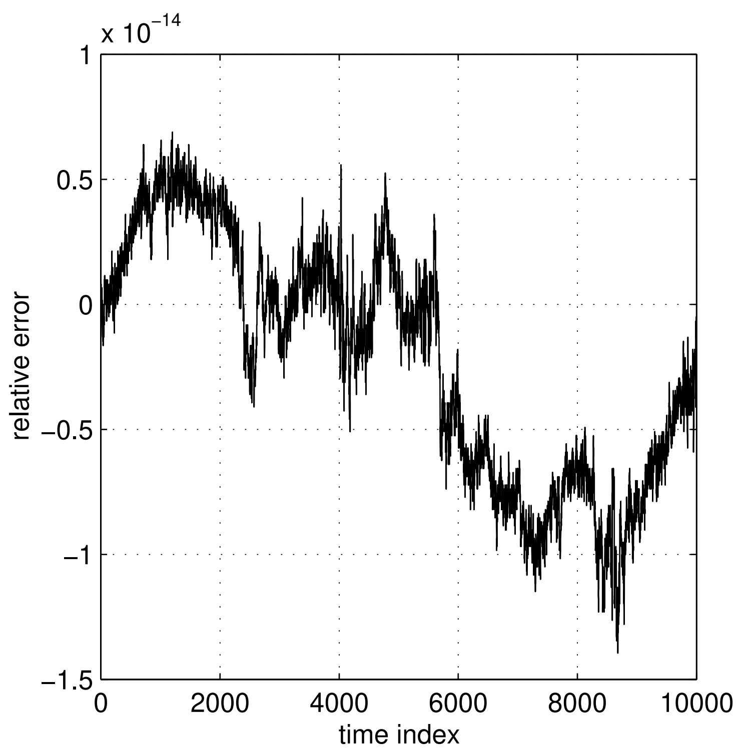


Figure 2-13: Relative error in energy conservation for rigid body motion using the manifold approach.

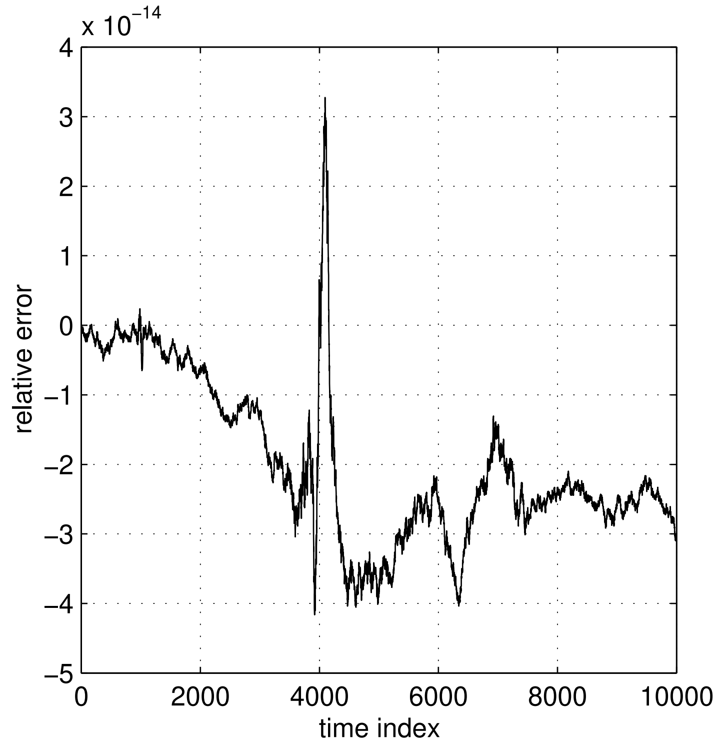


Figure 2-14: Relative error in conserving the x component of the angular momentum for rigid body motion using Euler angles. The maximum absolute value of the error is $4.163336342344337 \times 10^{-14}$, while the average absolute value of the error is $1.9975479603751012 \times 10^{-14}$.

by about six times.

Note that in Figure 2-12, the curve has a rather sharp peak at time index 4000. That is a consequence of a close encounter between the trajectory and the coordinate singularity. Such a peak can be seen in all of the following plots that were generated using the Euler angles (Figures 2-14, 2-16, and 2-18), and are absent from the plots generated by using the manifold integrator (Figures 2-13, 2-15, 2-17, and 2-19).

Similar comparisons can be made using the components of the angular momentum, as shown in Figures 2-14 through 2-19.

In contrast to the spherical pendulum, in this example all the components of angular momentum (as computed from the inertial frame), as well as the energy function, are used in the integration. Thus, the manifold integrator attempts to minimize deviations from initial values of conserved quantities, which improves their conservation at the cost of making it harder to check how well the system does.

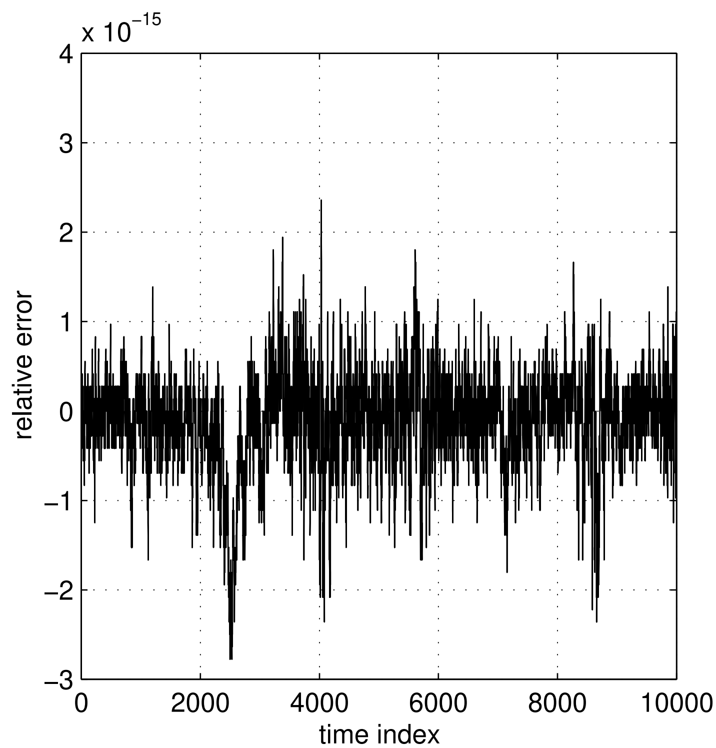


Figure 2-15: Relative error in conserving the x component of the angular momentum for rigid body motion using the manifold approach. The maximum absolute value of the error is $2.7755575615628914 \times 10^{-15}$, while the average absolute value of the error is $3.8748171338198744 \times 10^{-16}$.

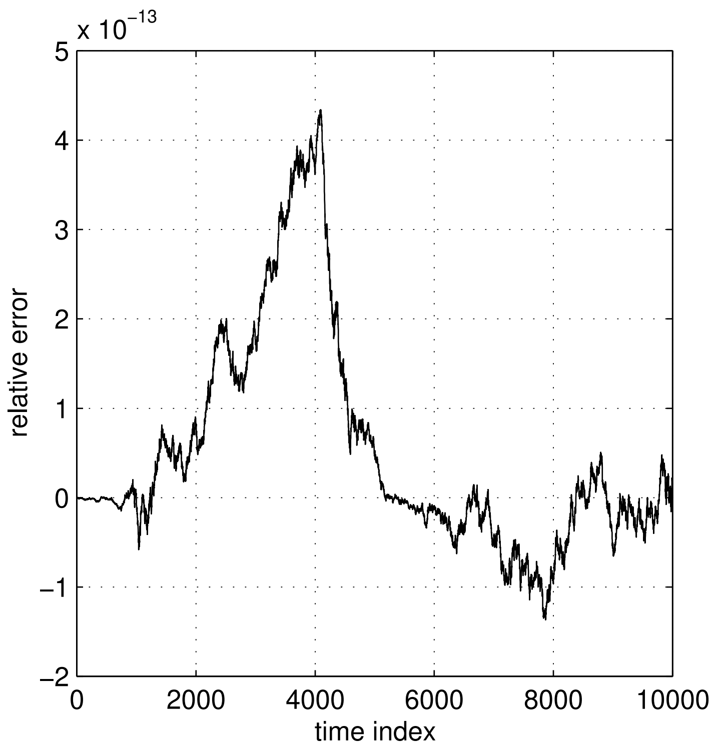


Figure 2-16: Relative error in conserving the y component of the angular momentum for rigid body motion using Euler angles. The maximum absolute value of the error is $4.3375450673823944 \times 10^{-13}$, while the average absolute value of the error is $8.348734810181229 \times 10^{-14}$.

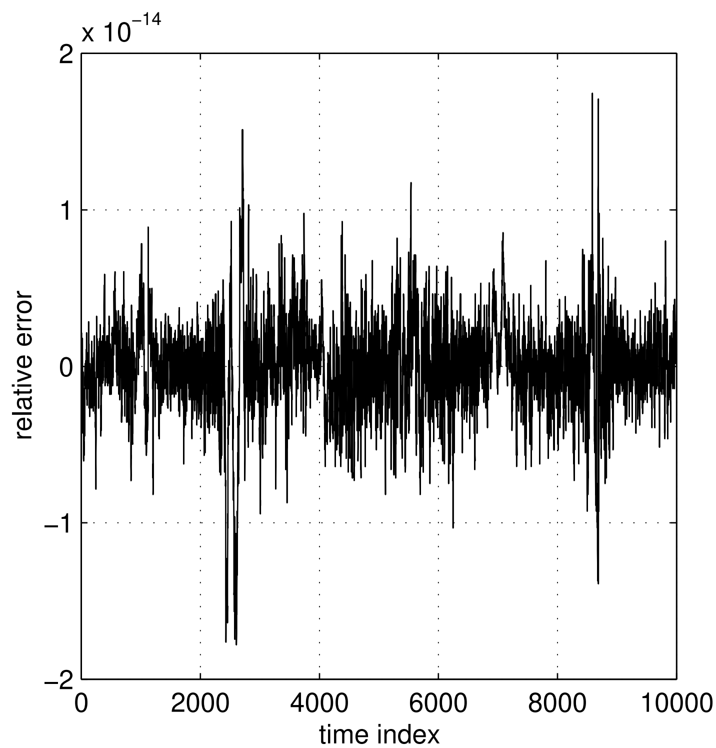


Figure 2-17: Relative error in conserving the y component of the angular momentum for rigid body motion using the manifold approach. The maximum absolute value of the error is $1.7798707703661857 \times 10^{-14}$, while the average absolute value of the error is $2.1083459210375576 \times 10^{-15}$.

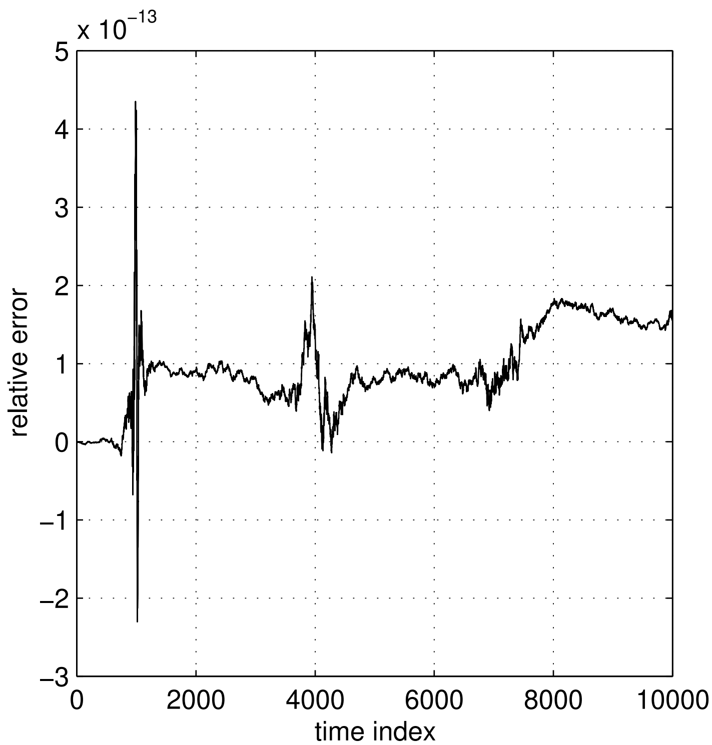


Figure 2-18: Relative error in conserving the z component of the angular momentum for rigid body motion using Euler angles. The maximum absolute value of the error is $4.352060412667006 \times 10^{-13}$, while the average absolute value of the error is $9.479992724704404 \times 10^{-14}$.

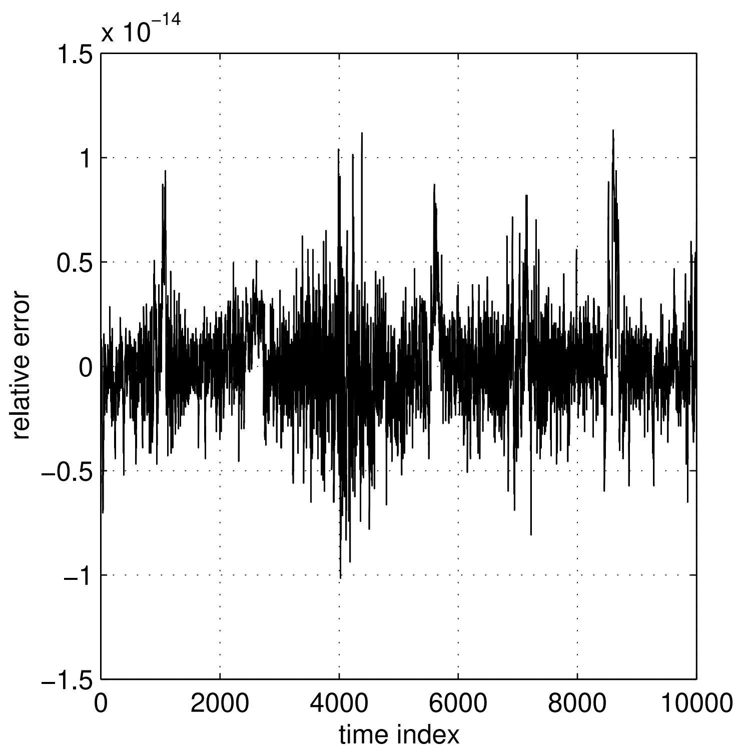


Figure 2-19: Relative error in conserving the z component of the angular momentum for rigid body motion using the manifold approach. The maximum absolute value of the error is $1.1322645212381266 \times 10^{-14}$, while the average absolute value of the error is $1.5603516120091776 \times 10^{-15}$.

2.4 Directions for future work

Clearly, in order for this to be useful, several improvements are required. Among these, the most important is probably efficiency: While the manifold integrator is, in many cases, more accurate than traditional methods, the cost of integrating in several charts simultaneously can make such integrators prohibitively slow. One solution is to integrate in one chart at a time, and to have much more sophisticated methods for when and how to switch from one chart to another. While not nearly as elegant as the current approach, this would probably be much more efficient.

Another problem is the difficulty in constructing manifolds. As shown by the example of covering the sphere using stereographic projection, constructing a manifold can take quite a bit of work (especially without the aid of the inverse function theorem). Thus, there need to be better tools, or at any rate larger libraries, for constructing and combining manifolds.

It would be interesting to compare the efficiency and accuracy of this approach to more sophisticated techniques, such as symplectic integrators [29].

Chapter 3

Linear partial differential equations

This chapter describes the application of the manifold abstraction to the numerical solution of linear partial differential equations. For simplicity, the discussion is restricted to scalar equations over two-dimensional manifolds. This is because some of the algorithms described here depend on efficient mesh generators, which are most easily constructed for two dimensions.[1] However, it should be noted that there exist much more powerful mesh generators than the one used here, and hence the programs developed in this section should generalize to higher dimensions without too much difficulty [6].

Appendix A briefly describes some background material on partial differential equations, including a brief treatment of finite element methods and an even less complete description of iterative solution methods for sparse linear systems of equations. Readers unfamiliar with these topics may wish to take a look at Appendix A first, and to use Vichnevetsky [27] as a more in-depth reference. Petersson [23] describes the solution of PDEs using multiple coordinate systems in a more specialized and less abstract context, as do Chesshire and Henshaw [7].

Note that this chapter focuses on elliptic boundary-value problems, although many of the ideas extend to more general problems. Hyperbolic initial-value problems are considered in the next chapter. The rest of this chapter begins with an exploration of theoretical representations of partial differential operators on manifolds. Then §3.2 discusses different approaches to the discretization of PDEs on manifolds. These approaches are developed and analyzed in more detail in later sections.

[1]It is even easier to do in one dimension, but such cases are too simple.

3.1 Partial differential operators on manifolds

In Chapter 2, first-order ordinary differential equations were redefined as smooth vector fields on differentiable manifolds. By using the tangent bundle construction, higher-order ODEs also became representable in a coordinate-independent fashion. This approach provided a natural framework for representing ODEs using multiple coordinate systems, and for developing these ideas into functional programs that improved the accuracy of numerical integrations. The questions that naturally follow are: How can PDEs be represented in a coordinate-independent fashion? And can similar improvements in accuracy be made?

We begin with a simple observation: Let M be a differentiable manifold, and let f be a smooth real-valued function on M. Given a point p, df_p is a linear transformation from T_pM into $T_{f(p)}R$, by definition. But the tangent space to R at $f(p)$ is just another copy of R, so for any tangent vector $v \in T_pM$, the value of the differential of f at p on v, $df_p(v)$, is just another real number. By definition, this corresponds to the directional derivative of f in the direction v in local coordinates, scaled by the length of v.[2] Since this gives us a way to define the directional derivative of f in the direction v in a coordinate-independent way, we can turn the argument around and say that the vector v *operates* on the function f.

More precisely, let v be a smooth vector field on M, and define $v_p(f)$ to be $df_p(v_p)$, where v_p is the value of the vector field at p. Furthermore, define the function $v[f]$ by the equation:

$$v[f](p) = v_p(f) = df_p(v_p). \tag{3.1}$$

Since v and f are smooth, so is $v[f]$. Furthermore, v as an operator on functions is *linear*, and satisfies the *product rule*:

$$v[f \cdot g] = v[f] \cdot g + f \cdot v[g]. \tag{3.2}$$

As an operator, then, v has the properties of a differential operator. In fact, one can easily check that, in local coordinates, this turns the vector field v into a *linear first-order partial differential operator*. Conversely, let (U, V, ϕ) be a chart. Then it is not difficult to verify that every first-order differential operator of the form

$$Lf(x) = \sum_{i=1}^{n} a_i(x) D_i f(x), \tag{3.3}$$

where f is a smooth function on the open subset V of R^n, uniquely generates a "local vector

[2]It makes sense to speak of the length of v because this is a directional derivative in a given chart. The length of v is its magnitude according to the dot product with respect to the chart's coordinate maps.

field" on the corresponding subset U of the manifold via the mapping ϕ^{-1}. Hence, we can *define* first-order partial differential operators on manifolds to be smooth vector fields. Since vector fields are already coordinate-independent objects, this means first-order operators are also coordinate-independent. Furthermore, higher-order operators may be produced by linear combinations and compositions of first-order operators, so linear partial differential operators can be defined in a nicely coordinate-independent way on manifolds. A linear PDE on a manifold then takes the form

$$Lf = g, \tag{3.4}$$

where f and g are smooth functions on the domain M, and L is a linear partial differential operator, as described above. Furthermore, if M is a manifold with boundary and h is a smooth function on the boundary ∂M of M, then a function f is said to satisfy the boundary value problem with boundary data h if $Lf = g$ and $f = h$ on ∂M.

Unfortunately, this definition of partial differential operators is too abstract to be useful for practical implementations. In fact, it is very difficult to develop a general representation of differential operators that is efficient for all numerical methods. Thus, each method in this chapter uses a different representation of operators, and programs are structured to provide flexibility with respect to the choice of representation. However, this theoretical definition is still important for the logical framework it provides, and for demonstrating a different way to view vector fields on manifolds. In practice, though, it is Equation (3.3) and its higher-order generalizations that play a more important role in computation.

3.2 Approaches to discretization

General comments. Differential equations determine unknown functions. Thus, to facilitate numerical computation, it is often necessary to parametrize the set of possible solutions using finitely many variables, and to reduce the PDE itself to a system of algebraic equations that determine the values of these variables. This process of reducing a PDE into a system of algebraic equations is called *discretization*.

In general, one can describe discretization in terms of two separate but interdependent steps: First, one must choose a representation for the approximate solution, so that a finite set of variables can be mapped to a *function* approximating the true solution. This often involves series expansions, such as Fourier series, power series, or expansion in terms of finite element basis functions. For these cases, the finite set of variables to which the unknown function has been reduced are, respectively, the Fourier coefficients, the Taylor coefficients, or values of the given function at specified sample points. The choice of a representation, informally, corresponds to the *geometric* part of discretization: In choosing a representation for approximate solutions, one often needs to first discretize (i.e. represent using a finite number of parameters) the *domain* of the PDE. Of course, as finite element methods show, there is more to choosing representations than simply discretizing (or triangulating) the domain—One must also choose the order of the basis functions and various other parameters.

In contrast, the derivation of discrete algebraic equations can be said to *discretize the PDE itself*. This step often involves either replacing the differential operator with finite difference operators, as in standard finite difference schemes, or by invoking some other formulation of physical problems, such as variational principles or Galerkin's orthogonality condition.[3] To some extent, this component of the discretization process can be performed independently of the domain discretization in that one can often use the same discretized domain to discretize different PDEs that are defined over the same domain. However, the *method* of discretizing the PDE, be it finite elements or finite differences, must work very closely with the discretized domain. Thus, the two components are not truly independent, although it is important to recognize the flexibility and modularity in the structure of PDE solvers.

In this report, the focus will be on finite difference and finite element methods, so the domain discretization will involve choosing a discrete set of sample points and, for finite elements, generating the appropriate mesh.

Global methods versus local ones. The discussion above on discretization applies unambiguously to the discretization of PDEs whose domains are regions in Euclidean spaces. However, in the case of manifolds, we have a choice in the *order* in which the various steps are carried out because of the existence of multiple coordinate systems: One choice is to discretize the *entire manifold* first, and then discretize the PDE. For example, using finite elements, we would first triangulate the entire manifold before invoking variational principles to derive the discretized equations. In this report, this type of discretization is called *global discretization*.

On the other hand, we can first discretize the PDE locally, so that for each chart there exists a set of discretized equations. These sets of discretized equations must then somehow be combined to form a global system of equations that determine the approximate solution everywhere. This is called *local discretization*.

Since finite difference methods are inherently local, the distinction between global and local discretization is very little when one uses finite difference techniques. However, finite

[3]In the case of spectral decomposition methods, the PDE discretization involves the Fourier transform.

element methods require triangulations, and the general problem of triangulating manifolds is a rather difficult one in computational geometry. There appears to be no well-documented way of performing such triangulations except for low dimensions.[4] With improvements in computational-geometric algorithms, the global discretization approach may become tractable someday, but it is too difficult to use in general with currently available tools. In contrast, local discretization methods do not suffer from such handicaps because we can always choose charts with simple images in R^n, which simplifies the local triangulation process.[5]

Consequently, this chapter concentrates on local methods: Each chart is independently discretized in the *local discretization phase*, and the resulting local equations are then combined to form a global set of equations in the *combination phase*.[6] §3.3 considers local discretization using finite difference techniques, where the primary problem is the formulation of local equations and their solution. §3.4 then discusses the use of finite element methods, which require special attention to the combination phase; some simple ideas are proposed and tested first, followed by a somewhat more efficient and accurate algorithm. Finally, §3.5 revisits the topic of mesh generation on manifolds and discusses some of the difficulties involved.

There is much more work to be done in the application of the manifold abstraction to the numerical solution of PDEs, and §3.6 suggests some of these possible directions.

3.3 Finite differences on manifolds

Recall that finite difference techniques generally involve the use of difference quotients to replace derivatives, thus transforming partial differential equations into linear algebraic equations which can then be solved using a variety of numerical techniques. Approximate solutions are represented by their values at some set of chosen *sample points*, often referred to as *nodes* in this document,[7] and algebraic equations are derived to relate the values at these discrete sample points to each other.

[4] In particular, the triangulation of surfaces and solids in R^3 has been extensively studied because of their extensive engineering applications.

[5] One might well imagine triangulating each chart first, and then somehow combining these local meshes to form a global mesh. This is, in fact, the strategy employed in proving that every manifold has a triangulation. However, there are technical difficulties with a direct implementation of this idea, as discussed in §3.5.

[6] Please do not confuse the local discretization *phase* with local discretization *methods*: The former is part of the latter. Since global discretization is not the focus of this report, this terminology should not be *too* confusing.

[7] This terminology comes from imagining the use of these algorithms on massively-parallel computers, where each processor, or node, represents a sample point. For example, Abelson, et. al., describe a novel new approach to computing that may be able to exploit the locality inherent in finite difference and finite element approximations to perform computations in parallel [1].

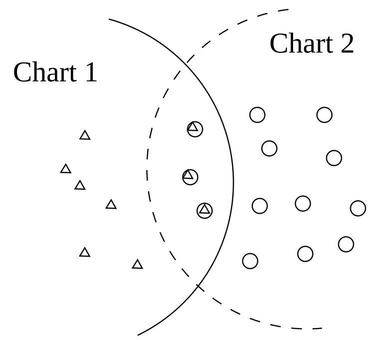


Figure 3-1: Copying nodes in the overlap between two charts to enforce constraints on unknown values, thus combining local equations into a global system. In this figure, the triangular nodes belong to chart 1, while the circular nodes belong to chart 2.

There are several possibilities for applying finite difference techniques to manifolds. What follows is the pseudocode for one of the simplest methods:

```
;;; This is the pseudocode for a finite differences algorithm on manifolds.
;;; Actually, this can easily be turned into a working program, but since most
;;; of the following material has already been implemented in C for speed, the
;;; Scheme versions were never implemented.

(define (finite-difference-on-manifold M L g h)

  ;; M should be a manifold, L a linear differential operator, and G and H
  ;; should be smooth functions on M.  The solution U is a function such that
  ;; (L U) = F over M, and where U = G on the boundary of M.

  (let ((charts (manifold:get-finite-atlas M)))

    ;; Based on the local geometry of each chart, construct a collection of
    ;; sample points.  Then for each node, compute its finite difference
    ;; coefficients with respect to its neighbors in each chart:

    (for-each

     (lambda (nodes)
       (for-each
        (lambda (node)
          (node:set-fd-coefficients! (compute-fd-coefficients node nodes)))
        nodes))

     (process-node-lists (map make-nodes charts) charts))))
```

This program contains a number of auxiliary procedures: `Make-nodes` takes a chart and constructs a list of nodes for that chart. `Manifold:get-finite-atlas` returns a finite atlas (i.e. a finite list of charts) for the manifold, if such a thing exists. `Process-node-lists` is a procedure that copies nodes between charts in overlapped regions (see Figure 3-1), so that nodes that lie in the overlap of two charts will exist in both charts and agree on the value of the approximate solution at that point. Finally, the procedure `compute-fd-coefficients` locallly discretizes the PDE, and can use any method it prefers to derive the finite difference coefficients of `node` with respect to its neighbors in `chart`.

Note that by copying nodes between lists in `process-node-lists`, we have implicitly constrained the system of equations to be consistent with each other on overlapped regions between charts. Thus, two sample points $x_1 \in C_1$ and $x_2 \in C_2$ are guaranteed to have the same value if x_1 and x_2 really correspond to the same point p in M. `Process-node-lists` thus performs all the necessary work for the combination phase. On the other hand, `compute-fd-coefficients` is the part of the program that controls how information flows between different parts of the discretized domain. For example, since many physical systems arise from local interactions, this procedure can be written to consider only those nodes in the list `nodes` that are physically close to the given node, `node`.

The combination phase of this local method, as described above, may seem trivial. However, because nodes are copied between charts, it is in general impossible to guarantee that nodes lie on regular grids. This causes two problems: First, local discretization becomes more difficult, since many standard methods depend on regular grids (we will see such a method later). Second, it often turns out that in the irregular case, the resulting finite difference equations are not sufficiently structured to be solvable by iterative methods such as relaxation.[8] But the application of direct or semi-direct methods to large matrices can be computationally intensive and numericaly undesirable, and hence the resulting set of linear algebraic equations can become very difficult to solve. The price we paid for simplicity in the combination phase is that the local problem becomes more difficult.

Chesshire and Henshaw avoid these difficulties by using a different approach [7]: Their method uses *locally regular grids* for local discretization, and instead of copying nodes (which destroys the regularity of local grids in the method outlined above), the combination phase is carried out by using *interpolation* functions between nodes. While this works well for some problems, however, it relies on much more complicated procedures for the combination phase and restricts the types of charts one could use. Thus, their method is not explored in this section, although variations on their idea are explored later in the context of finite elements.

The rest of this section focuses on the local problem of obtaining and solving finite difference approximations for PDEs because the problem is already non-trivial at that level, and adding the complication of solving PDEs on manifolds probably would not help.[9] Both

---

[8]Readers unfamiliar with relaxation and other iterative methods for solving large sparse linear systems of equations are referred to Chapter 6 of Vichnevetsky [27]. Appendix A also contains a brief introduction to the subject.

[9]Except that, perhaps, one could choose local coordinate systems to "regularize" the sample point ge-

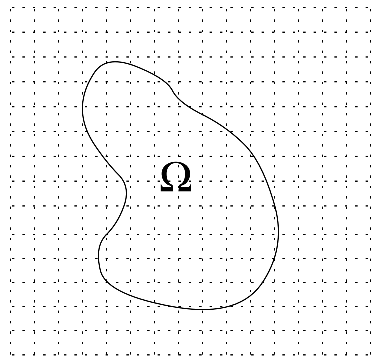


Figure 3-2: The discrete Dirichlet problem.

the simple method described above and that of Chesshire and Henshaw involve more difficulties, and thus in this section we only consider the application of finite differences to irregularly-distributed sample points over subsets of Euclidean space. This is an interesting problem in its own right.

3.3.1 Generating coefficients for irregular sample points

This section discusses the problem of local discretization using finite differences. As such, all domains are open subsets of Euclidean spaces unless otherwise stated.

The discrete Dirichlet's problem

As mentioned in §3.3, one of the primary problems encountered in implementing the algorithm above is the formulation of finite difference techniques using irregularly-distributed sample points. Before tackling this more difficult case, though, let us revisit the canonical example of finite differences: Laplace's equation on a regular rectangular grid and the discrete version of Dirichlet's problem (see Figure 3-2).

The basic idea is this: Let f be a real-valued differentiable function of one real variable. By the definition of the derivative, we have:

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h}. \tag{3.5}$$

This is the *central-difference approximation*, and has nicer numerical properties than the standard forward-difference approximation.

ometry to improve their numerical properties. But this turns out to be a rather difficult problem. For more details, Clark, et. al., present and analyze one possible way of carrying out this procedure, and describe its application to image processing [8].

Applying this approximation twice to f at x, we have an estimate of the second derivative of f:

$$f''(x) \approx \frac{f(x+h)+f(x-h)-2f(x)}{h^2}. \tag{3.6}$$

Now suppose we are interested in solving the boundary value problem for Laplace's equation over some region Ω in R^n. Cover the space R^n by a lattice $L_h = \{(x_1, ..., x_n) : x_i = k_i h, k_i \in Z\}$ with spacing $h > 0$, and choose h sufficiently small so that the domain Ω may be approximated by a subset Ω_h of L_h. Applying the formula above, we obtain:

$$\nabla^2 u(x,y) \approx \frac{u(x+h,y)+u(x-h,y)+u(x,y+h)+u(x,y-h)-4u(x,y)}{h^2}. \tag{3.7}$$

Upon rearrangement and setting the Laplacian of u to 0, this yields the familiar formula:

$$u(x,y) \approx \frac{u(x+h,y)+u(x-h,y)+u(x,y+h)+u(x,y-h)}{4}. \tag{3.8}$$

This formula is sufficient to determine approximate solutions of Laplace's equation over a regular lattice with reasonable accuracy for domains with sufficiently smooth boundaries and boundary data.

Polynomial interpolation

However, we cannot generalize this method to other irregular sample points because we made heavy use of the regularity of the grid in its derivation: The approximation formula (3.6) was valid because the sample points are regularly spaced, and an approximation of the Laplacian operator could be made because the lattice is generated by the orthogonal vectors $h\hat{x}$ and $h\hat{y}$, which lets us take the appropriate derivatives for computing the Laplacian. Thus, this method would not work if the sample points did not lie on a regular grid.

A different approach to finite differences is thus necessary. One natural idea is *polynomial interpolation*[10]: In any finite difference method, the primary goal is to express the partial differential equation as a set of coupled finite difference equations. Since we are only concerned with linear operators here, it is natural to take these finite difference equations to be linear. In particular, let the ith sample point be p_i, and let L be a linear differential operator. Then for each sample point, we would like to find coefficients a_{ij} such that:

$$Lu(p_i) = \sum_j a_{ij} u(p_j), \tag{3.9}$$

[10]Special thanks to Thanos Siapas and Gerald Jay Sussman for telling me about this idea.

where the index j ranges over all other nodes. Furthermore, since many physical problems involve only local interactions, and because of concerns for computational efficiency on parallel machines, the indices a_{ij} are chosen so that a_{ij} is non-zero only if p_i and p_j are physically close. Deciding whether two sample points are close or not is, of course, a parameter that needs to be chosen. Usually, one can call two sample points close if $|p_i - p_j| < R$ for some fixed radius R; in that case, p_i and p_j are called *neighbors*.

One way of computing the coefficients a_{ij} for some fixed i is as follows: Suppose that we would like to choose the coefficients for some point p_i with respect to its neighbors $p_{n_{ik}}, k = 1, ..., n_i$. For concreteness, let the domain be a subset of the plane. Then we can require that the approximation (3.9) is *exact* on some set of test functions, $\phi_1, \phi_2, ..., \phi_{m_i}$. Substituting the basis functions into Equation (3.9), this gives:

$$L\phi_j(p_i) = \sum_{k=1}^{n_i} a_{in_{ik}} \phi_j(p_{n_{ik}}), j = 1, 2, ..., m_i. \tag{3.10}$$

Clearly, this is a set of m_i linear equations in the n_i variables $a_{in_{ik}}$ (recall that i is fixed). If we have enough basis functions ϕ_j so that $m_i = n_i$, and if the basis functions are chosen so that the $L\phi_j$ can be easily computed, then the equations (3.10) provide an efficient means of determining the unknown coefficents $a_{in_{ik}}$. Indeed, when this is applied to the rectangular grid, where each grid point is given its immediate neighbors in the $\hat{x}$ and $\hat{y}$ directions as neighbors, this process yields the approximation (3.8).

3.3.2 Solving linear algebraic equations

While this method gives reasonable approximations of the differential operator L, there is a serious problem: The iterative methods usually used to solve the resulting linear algebraic equations, such as successive overrelaxation, do not converge, while the use of direct or semi-direct methods are often not possible for very large systems of equations.

One idea is to take advantage of the following well-known theorem: If A is a symmetric positive-definite matrix, then successive overrelaxation converges for all overrelaxation factors $0 < \bar{\omega} < 2$. Now, suppose we wish to solve the linear system of equations $Ax = b$ for some non-singular matrix A. Then $A^T Ax = A^T b$ is equivalent to the original system of equations. Furthermore, if A is nonsingular, then $A^T A$ is *positive-definite* and the theorem applies. Additionally, this computation can be carried out *locally*: Since the jth column of A consists of the coefficients $a_{n_{jk}j}, k = 1, 2, ..., n$, which are the coefficients of $p_{n_{jk}}$ with respect to p_j, two columns share a non-zero entry if and only if the corresponding sample points are within *two radii* of each other (see Figure 3-3). Since the entries of $A^T A$ are actually the dot products of the columns of A, the computation of $A^T A$ remains mostly local, with the neighborhood radius of each node increasing from R to $2R$.

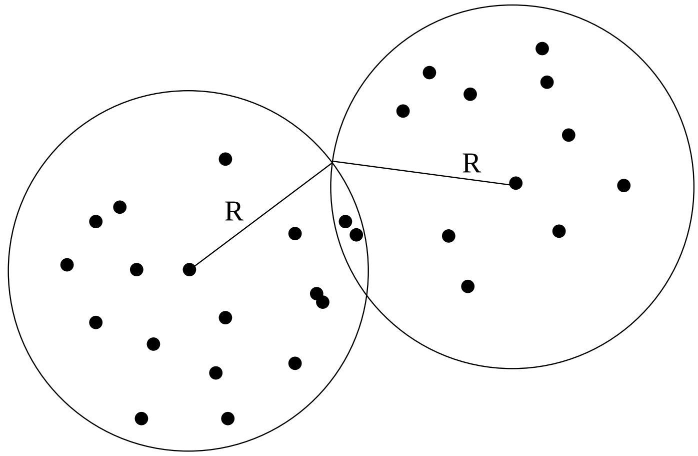


Figure 3-3: The case of irregularly-distributed nodes: In performing the “transpose trick,” two nodes have non-zero coefficients for each other if and only if they are within two hops.

Unfortunately, this clever idea is not as magical as it may seem at first: First, by multiplying a matrix with its own transpose, the *condition number* of the matrix is approximately squared.[11] This tremendously worsens the numerical properties of the matrix. Furthermore, the theorem quoted earlier states that the relevant spectral radius, ρ, is less than 1. However, it does not bound ρ away from 1. Thus, the actual spectral radius is often so near 1 that, in the presence of round-off error, the method converges too slowly to be useful, and we are forced to explore other methods.

3.3.3 Numerical examples

This sections presents the results of some numerical experiments using finite differences. Out of a desire to compute using a large number of nodes rather quickly, the programs have been written in C. Thus, the source code will not be included here because they are not very illuminating.

The problem in which we are interested is the *rectangular slot problem*: Consider the unit square $\Omega = [0,1] \times [0,1]$, depicted in Figure 3-4. Given the electric potential on the boundary of Ω and the condition that there are no charges in the interior of Ω, what is the electric potential everywhere inside Ω? From electrostatics, we know that the solution must satisfy Laplace's equation. Furthermore, analytical solutions of this problem can be easily

[11]The *reciprocal* of the condition number of a matrix measures, in some sense, the distance of a matrix to the set of *singular matrices*. Thus, the larger the condition number is, the closer the matrix is to being singular, and it becomes increasingly difficult to obtain numerically accurate solutions. For a more thorough discussion of condition numbers, as well as a discussion of this particular problem, see [24].

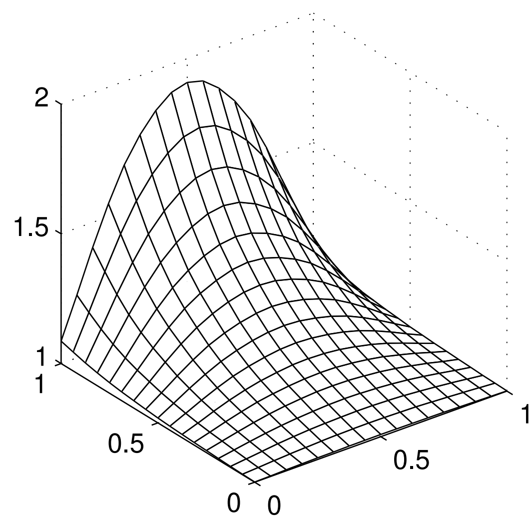


Figure 3-4: Determining the electric potential in a rectangular slot, with boundary conditions specified by Equation (3.11). The plot is generated by dividing the unit square into smaller squares, over which the nodal values are averaged. This reduces the number of points that need to be plotted.

derived using Fourier methods, so that numerical answers can be checked against the true solution.[12]

For our purposes, it is useful to just settle on boundary conditions whose corresponding solution is easy to compute. One such example is:

$$h(x,y) = \begin{cases} 1+\sin(\pi x), & y=1 \\ 1, & otherwise. \end{cases} \tag{3.11}$$

The exact solution for these boundary values is:

$$u(x,y) = 1 + \frac{\sinh(\pi y)\cdot\sin(\pi x)}{\sinh(\pi)}. \tag{3.12}$$

Notice that practically every function involved has the constant 1 added to it. This bounds solution values away from zero so that meaningful relative errors may be computed; it should not add significantly to the numerical error, since 1 is of the same order of magnitude as the solution values.[13]

[12] For more information about this and other related problems, see Haus and Melcher [15].

[13] This is essentially the first term in the Fourier series expansion for the solution of the slot problem with boundary values:

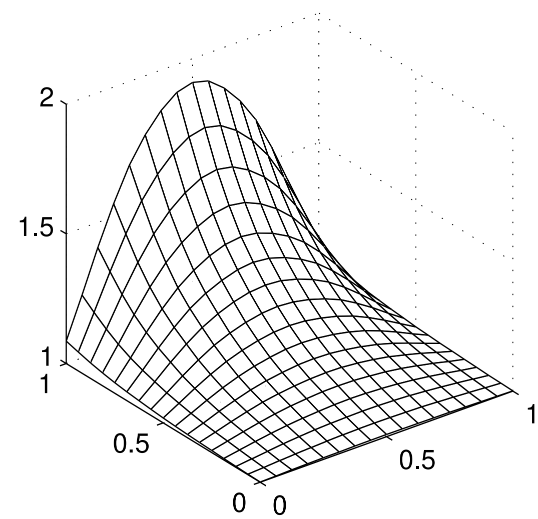


Figure 3-5: The electric potential in a rectangular slot, determined by finite difference computations on a regular rectangular grid. About 100,000 iterations (with $\bar{\omega} = 1.9$) were run, so the solver may not have converged to the "true" approximate solution yet.

A note on graphics. It is vital to note that in this section, all plots of sample values over the unit square are produced by dividing the unit square into rather coarse grids first, and then averaging over the sample values. This simplifies the task of plotting, but at the risk of making the data appear more smooth than it is. So please take care not to be misled by the apparent simplicity of the plots.

Regular grid. First, let us use the approximation (3.8) to approximate the solution on a regular rectangular grid. The actual grid used consists of 10,000 nodes, placed at regular intervals in the unit square Ω on a 100×100 grid. After applying Equation (3.8) to each node for about 10^6 iterations,[14] the resulting values are checked against the actual solution.

Figure 3-4 shows the shape of the electric potential arising from the boundary conditions

---

$$h(x,y) = \begin{cases} 1, & y = 1 \\ 0, & \mathit{otherwise}. \end{cases} \tag{3.13}$$

While this boundary condition is much simpler than the one above, its corresponding solution requires the computation of an infinite series that converges rather slowly; the relevant Fourier series is that of the unit-step funtion, where Gibbs' effect shows up.

Note that this boundary condition is also *discontinuous*, which makes accurate numerical solutions somewhat harder to obtain (especially near the corners). This is one of the many reasons why one may wish to have the ability to use multiple coordinate systems when solving PDEs, thus concentrating computational effort near discontinuities in the boundary data.

[14]Actually, the algorithm used is successive overrelaxation (SOR), with a relaxation factor of 1.9. This helps accelerate the convergence rate; for more information, see Appenix A or Vichnevetsky [27].

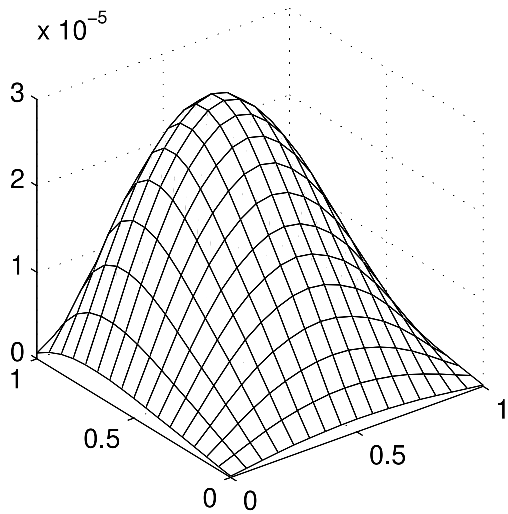


Figure 3-6: The absolute difference between the functions depicted in Figure 3-4 and Figure 3-5. The maximum absolute error is 0.0000291001797184, the minimum absolute error is 0.0000000157547926, and the average absolute error is 0.0000115848344767. The maximum relative error, on the other hand, is 0.0000216438565018, the minimum relative error is 0.0000000157534190, and the average relative error is 0.0000093994479478.

specified in Equation (3.11). Figure 3-5 shows the values obtained from the regular grid approximation. Note that they are qualitatively alike.

In fact, one can plot the error between the two; this is shown in Figure 3-6. Notice that the error reaches its maximum near the non-zero boundary values.

Randomly-distributed sample points and simple averaging. The next idea depends on an alternative derivation of the approximation (3.8): Let u be a function over some region Ω. For every point p and any real $r > 0$, denote the *closed ball* of radius r centered at p, $\{q : |p - q| \leq r\}$, by $B^n_r(p)$, and denote its boundary (the $n-1$-sphere) by $S^{n-1}_r(p)$. Then, u is said to have the *mean-value property* if for every point p and radius r such that $B^n_r(p)$ is contained in Ω, $u(p) = \int_{S^{n-1}_r(p)} u\ dS$ (where dS denotes the appropriate measure for a surface integral). A well-known theorem then states that u satisfies Laplace's equation if and only if it has the mean value property.

The equivalence of Laplace's equation and the mean-value property has many important consequences; Ahlfors [3] contains more details. For our part, it can be used to derive another approach to Laplace's equation: One uses randomly-distributed nodes,[15] but instead

[15]Actually, using uniformly distributed random numbers to place nodes uniformly in a rectangular region

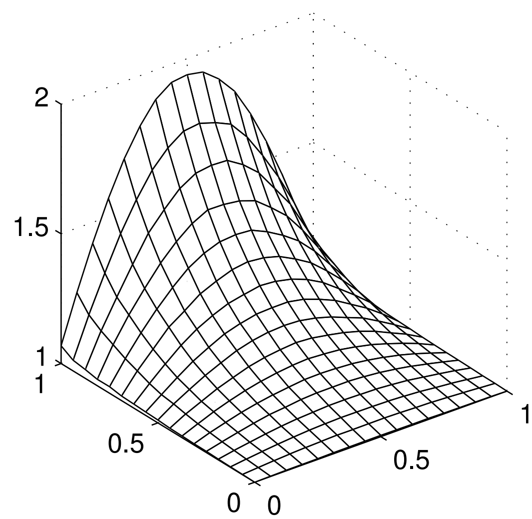


Figure 3-7: The potential computed by simple averaging using randomly-distributed nodes. As in previous figures, this plot is generated by dividing the unit square into smaller squares, over which the nodal values are averaged. So please keep in mind the comments at the beginning of this section: This plot may appear to be more smooth than the actual data because of the averaging procedure.

of trying to perform fancy derivations of finite difference coefficients, each node simply averages the values of its neighbors within a given radius R and sets its own value to this average. The validity of this approach follows from the mean-value property and a simple volume integral over the closed ball of radius R centered at each point p.

Figure 3-7 shows the approximate solution constructed this way; the smooth surface is generated by locally averaging nodal values. In this particular computation, there are 10,000 nodes in the rectangular slot, each having an average of 25 neighbors. Notice that it is qualitatively similar to Figures 3-4 and 3-5. However, as Figure 3-8 shows, the error distribution is much less smooth and is much larger.

Furthermore, we can examine the relationship between the average error and parameters such as the radius R and the number of nodes. Figure 3-9 plots average absolute error against the radius R for a domain having a fixed number of nodes. We see that as R decreases, the error decreases as well. This can be understood in terms of a node's ability to adapt to the approximate solution: Averaging over too large a neighborhood "stiffens" the approximate solution and makes convergence to solutions with large gradients difficult. Also,

tends to create clusters of nodes because the law of large numbers does not give us a very tight bound on the variance of the distribution from the mean, so it is necessary to enforce a minimum distance between nodes to ensure a "uniform" distribution.

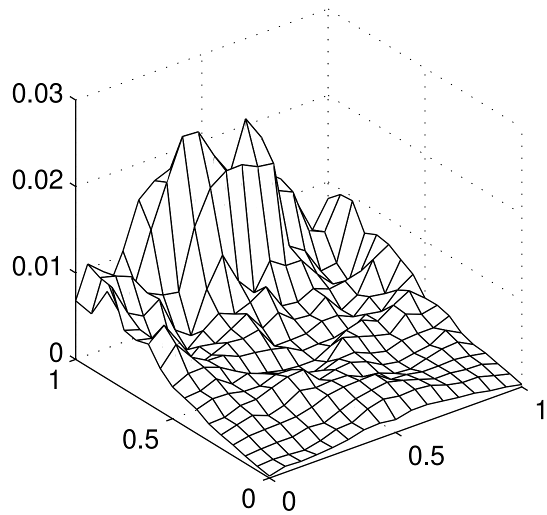


Figure 3-8: The error distribution for the averaging method. The maximum absolute error is 0.0303547633666503, the minimum absolute error is 0.0000001458319725, and the average absolute error is 0.0044948995201661. The maximum relative error, on the other hand, is 0.0246888943589139, the minimum relative error is 0.0000001448811931, and the average relative error is 0.0035004079565252.

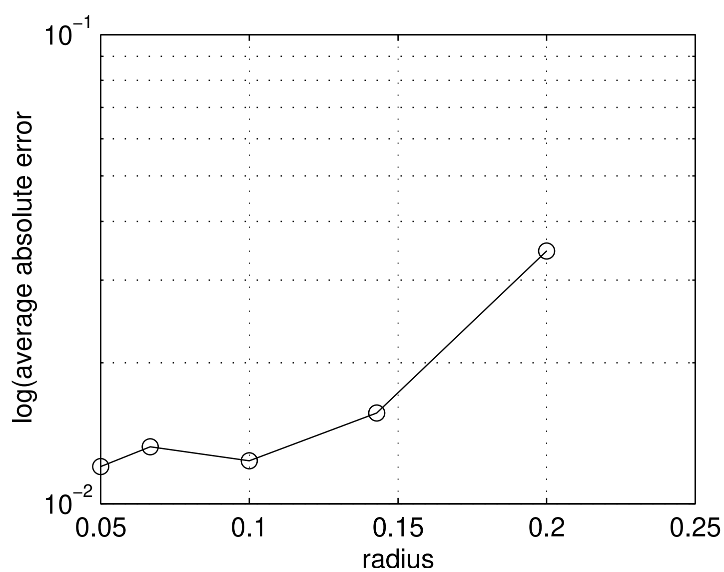


Figure 3-9: The average absolute error versus the radius R. The domain contains 1,000 nodes while the radius ranged from 0.2 to 0.05. Successive overrelaxation is performed on each configuration for 100,000 iterations, with $\bar{\omega} = 1.7$.

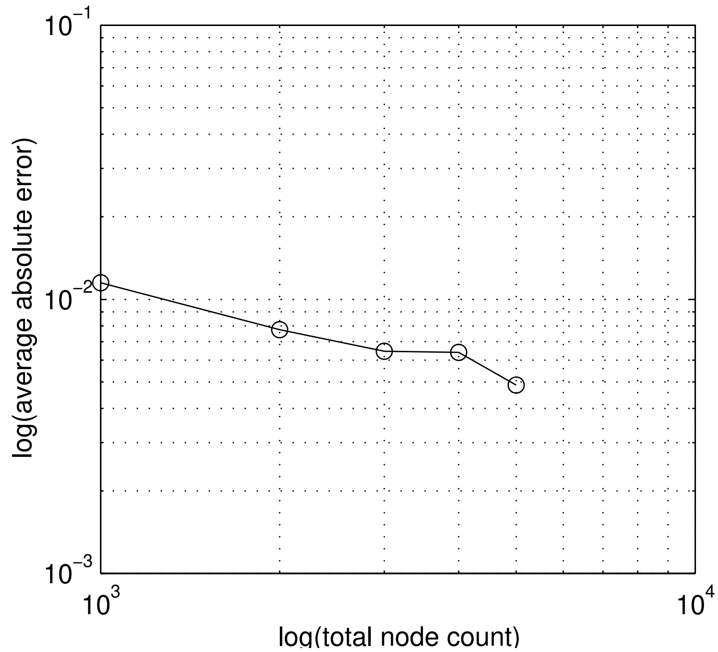


Figure 3-10: The average relative error versus total number of nodes. The number of nodes varies from 1,000 to 5,000, and the radii are changed to keep the average number of nodes per neighborhood at around 27 nodes. Successive overrelaxation is performed on each configuration for 100,000 iterations, with $\bar{\omega} = 1.7$.

the semi-log plot shows that the error decreases approximately exponentially for sufficiently large R, though the curve tapers off as R becomes smaller. However, the error *cannot* be made arbitrarily small by decreasing R along, because nodes can become disconnected from each other for sufficiently small R, and the boundary data would then have no way of "propagating" to interior nodes.

Figure 3-10 shows the analogous plot for the average absolute error versus the total number of nodes, with the density held constant by changing the radius. This log-log plot demonstrates an approximate power law governing the relation between the total number of nodes (given fixed density) and the average absolute error.

However, despite its simplicity and reasonable accuracy, the averaging method is limited by its lack of generality: Because it uses properties specific to Laplace's equation, it is not immediately applicable to other elliptic differential equations. This is one of the advantages of generating finite difference coefficients using polynomial interpolation, as described in §3.3.1.

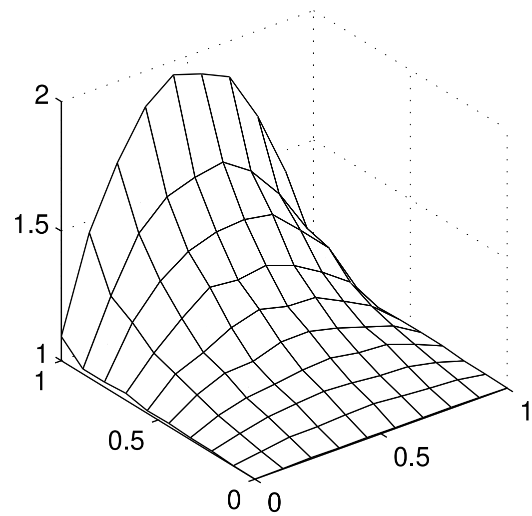


Figure 3-11: The approximate solution generated by applying direct matrix inversion to the system of equations generated by polynomial interpolation. The maximum absolute error is 0.0081833977839731, the minimum absolute error is 0.0000000779550711, and the average absolute error is 0.0004690292039753. The maximum relative error is 0.0076113517976692, the minimum relative error is 0.0000000745384821, and the average relative error is 0.0004207582072576.

Randomly-distributed sample points and polynomial interpolation. Let us now take a look at the finite difference coefficients generated using polynomial interpolation. Unlike the case of regular grids, the iteration diverges rather quickly. For the ease of computation, this section examines systems with smaller numbers of nodes — The tests here use 300 interior nodes distributed uniformly in the unit square and 144 nodes spaced evenly along the boundary, with the same boundary conditions (3.11).

For a system this size, one could explicitly compute the spectral radius for various iteration methods.[16] Indeed, for the example here, the spectral radius for Gauss-Seidel is 73.75932386604968, while that of Jacobi iteration is 6.69818594658326. Thus, both iteration methods diverge for this system. However, as a test of the accuracy of the coefficients themselves, we can directly invert the matrix using LU decomposition.[17] The result, shown in Figure 3-11, demonstrates that polynomial interpolation actually produces fairly accurate answers—If one had the ability to solve the resulting equations.

[16]The computations in this section are done using $MATLAB^{TM}$.

[17]This could be done because the system only has 300 interior nodes, and hence 300 unknowns. With 10,000 unknowns, there is no way to invert the matrix directly! Of course, from the view of error analysis, one should be suspicious of directly inverting even a 300×300 matrix...

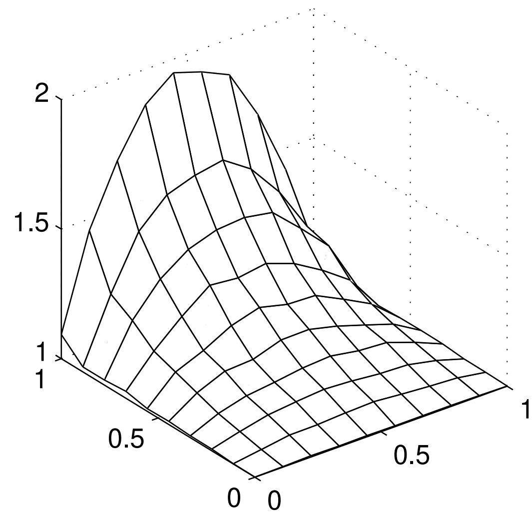


Figure 3-12: The approximate solution generated by applying direct matrix inversion to the system of equations generated by the "transpose trick." The maximum absolute error is 0.0079623718237853, the minimum absolute error is 0.0000001702022259, and the average absolute error is 0.0004585412757934. The maximum relative error is 0.0074057762673312, the minimum relative error is 0.0000001577217479, and the average relative error is 0.0004110492850620.

The transpose trick. So what happens if we actually attempt to apply the "transpose trick" described in §3.3.2? Does this really improve the stability of Gauss-Seidel iterations? The answer is a lackluster affirmative: The spectral radius for Gauss-Seidel iteration is 0.99999999123756, while that of Jacobi iteration is 7.52337630885650. Thus, Gauss-Seidel (in theory) converges for this problem, even though the spectral radius is close enough to 1 that convergence is very slow. Furthermore, the condition number of the matrix before multiplying by the transpose is $2.016135227435024 \times 10^6$, while after multiplying by the transpose it becomes $2.382088963154271 \times 10^{12}$—Roughly squared, as expected.

Thus, instead of applying iteration to these equations, LU decomposition is applied directly as in Figure 3-11. The result is shown in Figure 3-12.

3.4 Finite elements on manifolds

An alternative to finite difference techniques is to employ finite element methods in local discretization, which in general do not require regular grids to perform efficiently (as do finite difference methods). However, the combination of local equations into a global system can be more problematic for finite elements than for finite differences.

The basic idea of finite elements on manifolds is simple: For each chart (U, V, ϕ), one can map the open set U onto the open subset V of R^n. Since V is an open subset of R^n, one can generate a mesh that covers almost all of V in a number of ways: One way is to always ensure that V is of a simple shape by choosing the appropriate mapping ϕ; then it is a easy to generate a regular grid over V. Another way is to generate a set of nodes that fill V "densely," and to triangulate them using a mesh generation algorithm such as quickhull [6], which works for general n-dimensional convex polytopes. Having generated a mesh over each chart, one can then apply standard finite element methods, such as Rayleigh-Ritz or Galerkin's method, to the open subset V of R^n. This yields locally discretized equations for each chart.

The next step is to combine the equations. One straightforward proposal is to choose a set of nodes in the overlap region between charts, and to constrain the unknown value at each of those nodes to the interpolated value from the other chart, thus generating a relation between unknown variables in different charts. The nodes chosen to form these constraints are called *interpolation nodes*, and choosing good ones turns out to be rather tricky: Too few, and not enough information propagates between charts to generate a good solution. Too many, and the resulting equations become overconstrained and cannot come anywhere close to the real solution.

Before discussing these issues in detail, however, it is useful to devlelop a deeper understanding of what it means to integrate functions over manifolds.

3.4.1 Integration on manifolds

Integration is a very powerful tool in the study of partial differential equations, particularly in the formulation of numerical methods. This is because integrals are much easier to compute accurately and have a number of other nice properties, and can often be used to reformulate PDEs in ways that simplify numerical solution methods. For example, finite element methods often rely on variational principles (as in the Rayleigh-Ritz method) or orthogonality conditions (as in Galerkin's method) to discretize PDEs: In the former case, the computation of the action functional to be minimized requires integration over the domain of the PDE, and in the latter case, the evaluation of the inner product on the function space of possible solutions again requires the integration of functions over the domain.

While these ideas are all straightforward to define on subsets of Euclidean space, it is less obvious how one can arrive at a coordinate-independent definition of integration on manifolds. Integration, as opposed to differentiation, is inherently a *global* operation, not a local one, and thus the definition of integrals is more difficult than that of differentials.

There is no unique way to define the integral of a real-valued function on manifolds. However, one could integrate real-valued functions over *Riemannian manifolds* (see §2.1.4), where a "smoothly-varying" inner product is defined on each of tangent spaces. There is another useful approach to integration that relies on "differential forms." Since this material will not be needed for our purposes here, a discussion is postponed until Appendix B.

Partitions of unity

In view of the usefulness of tangent vectors on manifolds, which were defined using the fact that manifolds locally "look like" Euclidean spaces, one natural idea would be to reduce the problem of integrating a function over the whole manifold to the problem of integrating a function over a chart. That is, the problem of integration can be divided into two subproblems: The first is how to reduce the problem of integration to a local problem, and the second is how to define integration locally in a consistent way so that the integral of a function over a small subset of the manifold is independent of the chart chosen to evaluate that integral.

It turns out that the two approaches to integration mentioned above differ only in how they solve the second subproblem. The common solution to the first subproblem, called a "partition of unity," is a simple but powerful idea.

Let $\{\rho_i\}$ be a set of smooth real-valued functions on a manifold M, let U_i denote the interior of the support of ρ_i, and let $\mathcal{A}$ be an atlas that is compatible with the atlas of M. Then $\{\rho_i\}$ is a *partition of unity subordinate to* $\mathcal{A}$ if:

1. $\rho_i(x) \geq 0$ for all $x \in M$.

2. For each i, there exists a chart $(U, V, \phi) \in \mathcal{A}$ such that the support $\bar{U}_i$ of ρ_i is contained in U. Furthermore, $\bar{U}_i$ is *compact*.[18]

3. $\sum_i \rho_i(x) = 1$ for all $x \in M$.

4. Every point $x \in M$ has a neighborhood W such that W is contained in only finitely many of the sets U_i.

For any atlas on any manifold, there exists a partition of unity subordinate to it. For a proof of this fact, see Munkres [21], Guillemin and Pollack [14], or Warner [28]. In this discussion, the atlas to which a partition of unity is subordinate may not be mentioned

[18]For those who have not had exposure to point set topology, compactness in this context is equivalent to saying that the image of $\bar{U}_i$ under ϕ is a closed and bounded subset of V. It is a topological property independent of the chart.

explicitly; in such cases, the atlas of the manifold is assumed.[19] Incidentally, finite element basis functions furnish a nice example of a partition of unity.

Suppose, now, that we have already found a nice way to define an integral operator "$\int$" on real-valued functions over the manifold M. What properties should it have? First of all, integrals should be *linear*; that is, the integral of two functions f and g should satisfy

$$\int_M (af + bg) = a \int_M f + b \int_M g \tag{3.14}$$

for real constants a and b. Now note that for any function f and any partition of unity $\{\rho_i\}$, the following equation holds for all $x \in M$:

$$f(x) = \sum_i \rho_i(x) f(x). \tag{3.15}$$

This expression is well-defined, because even though the collection $\{\rho_i\}$ may be infinite, axiom 4 shows that for each x, only finitely many of the numbers $\rho_i(x)$ is non-zero. Thus, this potentially infinite series is actually a finite sum for each x, and the expression is well-defined. The equation then follows from the fact that the ρ_i sum to 1.

Combining this with the linearity of integrals, we obtain:

$$\begin{aligned} \int_M f &= \int_M \sum_i \rho_i f && (3.16) \\ &= \sum_i \int_M \rho_i f && (3.17) \end{aligned}$$

But each of the functions $\rho_i f$ has compact support. Furthermore, the support of $\rho_i f$ must be a subset of the support of ρ_i, which is contained entirely in some chart.

Conversely, suppose that we have a way of integrating functions whose supports lie entirely within a chart. It is easy to show that the choice of a partition of unity to combine these integrals does not affect the final outcome: Let $\{\rho'_j\}$ be another partition of unity subordinate to the atlas $\mathcal{A}' = \{U'_j\}$. Then:

$$\sum_i \int_{U_i} \rho_i f = \sum_{i,j} \int_{U_i \cap U'_j} \rho_i \rho'_j f = \sum_j \int_{U'_j} \rho'_j f. \tag{3.18}$$

We have thus reduced the problem of finding a reasonable definition of integrals of functions on manifolds to a local problem: How can we integrate functions whose supports lie entirely in a given chart?

[19] In most treatments of partitions of unity, axiom 2 is stated using open covers, not atlases. However, for our purposes, partitions of unity are most useful when the open cover is an atlas.

Integration on Riemannian manifolds

Consider now open subsets V_1 and V_2 of Euclidean n-space. Suppose f is a smooth bounded real-valued function on V_2 (boundedness is generally required to ensure that the integral is finite), and that there exists a smooth bijective map ϕ from V_1 to V_2. Using the change of variables theorem, we know that the integral of f over V_2 can be written in two ways:

$$\int_{y \in V_2} f(y)dy = \int_{x \in V_1} f(\phi(x))|\det D\phi(x)|dx, \tag{3.19}$$

where traditional notation, rather than functional notation, was used for the sake of clarity.

As stated in §3.4.1, one major aspect of defining integration on manifolds is finding a consistent definition of local integrals. In view of Equation (3.19), this amounts to figuring out what geometric information is necessary to construct objects that transform like determinants, so that integrals of functions over "small" subsets of the manifold are the same no matter what chart is used. The approach here is to relate determinants to a local measure of *volume* in tangent spaces of the manifold, so that the function analogous to the determinant can be defined geometrically.[20]

The geometry of determinants. Let us begin with the geometric interpretation of the determinant: Let S be a set of n vectors $B = \{v_1, v_2, ..., v_n\}$ in Euclidean n-space. Every such set S defines a *parallelpiped*:

$$\{v \in R^n : v = \sum_{i=}^{n} a_i v_i, \sum_{i=0}^{n} a_i \leq 1, a_i \geq 0\}, \tag{3.20}$$

where $v_0 = \sum_{i=1}^{n} v_i$. This generates a convex polyhedron with vertices at the origin, each of the points v_i, and the point $v_0 = \sum v_i$; in the case $n = 2$, this is just the definition of a parallelogram. The n-dimensional *volume* of this geometric object is then $|\det A|$, where A is the matrix whose columns are the vectors $v_1, v_2, ..., v_n$.

Now, this definition of volume implicitly used the structure of Euclidean space. The determinant depends on the components of the matrix A, which in turn depend on the particular basis chosen. In the Euclidean case, there is a standard basis, but general vector spaces do not have special bases singled out for them, and hence the determinants of linear transformations are not well-defined. However, for inner product spaces, the determinant *is* well-defined, *up to a sign*:

Let V and W be n-dimensional inner product spaces, and let L be a linear transformation

[20] As discussed in Appendix B, the other approach is to somehow associate determinants to functions, so that instead of integrating real-valued functions, one integrates functions called *differential forms*, whose values are "determinant-like" functions.

from V to W. Choosing bases B_V and B_W for V and W, respectively, we can write L as a matrix with real components. Its determinant is then well-defined with respect to these bases. In particular, let $B_{V,1}$ and $B_{V,2}$ be orthonormal bases for V, and let $B_{W,1}$ and $B_{W,2}$ be orthonormal bases for W. If we let L_i be the matrix representation of L with respect to the bases $B_{V,i}$ and $B_{W,i}$, then elementary linear algebra shows that:

$$L_2 = A_W^{-1} \cdot L_1 \cdot A_V, \tag{3.21}$$

where A_V is the matrix representation of the basis $B_{V,2}$ with respect to the basis $B_{V,1}$, and A_W is the matrix representation of the basis $B_{W,2}$ with respect to the basis $B_{W,1}$. But the bases $B_{V,i}$ and $B_{W,i}$ are chosen to be orthonormal for $i = 1, 2$, so the matrices A_V and A_W are *orthogonal*, and their determinants are ± 1. Thus, $\det L_2 = \pm 1 \det L_1$, and we see that for inner product spaces, one can define the determinant in a consistent way up to a factor of ± 1.

We can therefore make the following definition: Let L be a linear transformation from an inner product space V to another inner product space W, both of dimension n. Then the function $|\det L|$ is *defined* to be the absolute value of the determinant of L with respect to *any* orthonormal bases for V and W. By the argument above, this is well-defined. Furthermore, like ordinary determinants, this has the following properties: $|\det I| = 1$ for the identity operator I, and given inner product spaces V_1, V_2, and V_3, and linear transformations $L_1 : V_1 \to V_2$ and $L_2 : V_2 \to V_3$, where the dimensions of the V_i are all n, $|\det L_2 L_1| = |\det L_2| \cdot |\det L_1|$.

Integrals on compact Riemannian manifolds. Let M be a Riemannian manifold, and for each point $x \in M$, let g_x denote the inner product on the tangent space $T_x M$. Suppose f is a smooth real-valued function on M whose support is a compact subset of U for some chart (U, V, ϕ). Define the integral of f on U by:

$$\int_U f = \int_V f \circ \phi^{-1} |\det d\phi^{-1}|. \tag{3.22}$$

Since the tangent spaces of M are inner product spaces (recall that M is a Riemannian manifold), and V as a subset of R^n has a canonical inner product, the expression $|\det d\phi^{-1}|$ is well-defined.[21] Furthermore, suppose the support of f is contained in both U_1 and U_2 for some charts (U_1, V_1, ϕ_1) and (U_2, V_2, ϕ_2). Let U be the intersection of U_1 and U_2, and let $W_i = \phi_i(U)$. Then f is also supported in U, and:

[21] The differential $d\phi^{-1}$ is well-defined because ϕ^{-1} is a smooth map from the open subset V, which is a manifold itself, into the manifold M.

$$
\begin{aligned}
\int_{U_1} f &= \int_U f & (3.23)\\
&= \int_{W_1} f \circ \phi_1^{-1} |\det d\phi_1^{-1}| & (3.24)\\
&= \int_{W_2} (f \circ \phi_1^{-1} \circ (\phi_1 \circ \phi_2^{-1})) |\det d\phi_1^{-1}| \cdot |\det (d(\phi_1 \circ \phi_2^{-1}))| & (3.25)\\
&= \int_{W_2} f \circ \phi_2^{-1} |\det d\phi_2^{-1}| & (3.26)\\
&= \int_{U_2} f, & (3.27)
\end{aligned}
$$

and the integral $\int_U f$ is well-defined. But by our earlier argument using partitions of unity in §3.4.1, this means the integral is well-defined on manifolds.

One last note: This discussion actually skirts the issue of *convergence*. While each local integral $\int \rho_i f$ is well-defined because $\rho_i f \circ \phi^{-1}$ has compact support in V, there is nothing that guarantees that the sum $\int f = \sum_i \int \rho_i f$ converges. In general, it does not always converge, and one often requires that the partition of unity be finite. A manifold for which there exists a finite partition of unity must be *compact*.[22]

Implementation in Scheme

Having gone to such lengths to discuss integration on manifolds, the reader might suspect that one could build an elaborate computational scheme for computing integrals of real-valued functions over Riemannian manifolds. However, in practice it often happens that the manifold in question is an open subspace of R^n (or, in cases where boundary conditions are necessary, closures of open subspaces of R^n). In such cases, it suffices to use the Euclidean structure directly to define integrals, and the code for manipulating finite element basis functions implement the ideas in the previous section automatically. As a complete implementation of these ideas is not necessary for testing the use of multiple coordinate systems to solve PDEs, such routines have not been implemented at this time. The purpose of this treatment of integration has primarily been for the theoretical insight it provides; like partial differential operators in §3.1, the code used in this chapter can seem ad-hoc and confusing without a proper framework in mind.

[22] A good introduction to general topology and such concepts as compactness, connectedness, and continuity for general topological spaces is Munkres [19].

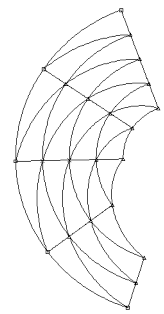


Figure 3-13: A boundary chart for the solid disc in the plane.

3.4.2 More about boundaries

This section picks up where §2.1.8 left off: In order to discuss the computational solution of elliptic boundary-value problems on manifolds, it is necessary to build a computational framework for working with boundary charts and manifolds with boundaries. This section discusses the implementation of manifolds with boundaries in Scheme.

`Add-boundary-to-chart` and `make-boundary-chart` are the primary procedures for computing with boundaries of manifolds. `Add-boundary-to-chart` takes as arguments a `chart` (U, V, ϕ), an index `i`, and an optional argument `level` L, and declares the subset $\{p \in U : x_i = L, x = \phi(p)\}$ of V the *boundary* of the chart. This creates boundary charts for the original manifold. While this is a slight deviation from the definition of boundary charts in §2.1.8, it is clearly equivalent and slightly simplifies programming with these abstractions. `Make-boundary-chart`[23] then constructs a chart for the boundary manifold out of a boundary chart for the original manifold.

The actual construction of a manifold with boundary can be rather messy, so the code is omitted here. Figures 3-13 through 3-15 show three charts that cover the solid disc $\{x \in R^2 : |x| \leq 1\}$, the first two being boundary charts and the third covering the center of the disc. Figure 3-16 shows how these charts overlap.

3.4.3 Computing with finite elements on manifolds

The previous sections, together with Appendix A, contain the material necessary for developing finite elements on manifolds. Since the subject of partial differential equations is sufficiently vast and complicated that many issues of theoretical and computational importance need to be resolved in very different fashions in different cases, the programs have

[23]This procedure is a bit of a misnomer, since boundary charts, as defined, are really charts of the manifold M, not charts of the boundary manifold ∂M.

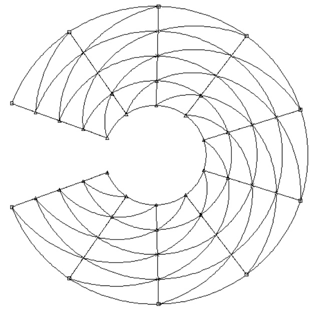


Figure 3-14: Another boundary chart for the solid disc in the plane.

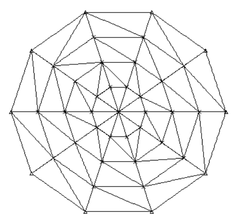


Figure 3-15: A third chart for the solid disc in the plane; this one covers only the interior and does not intersect the boundary.

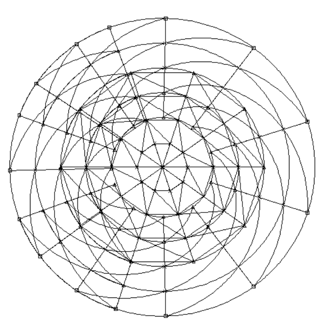


Figure 3-16: All three charts together, covering the unit disc.

been designed to provide only a logical skeleton into which all the components fit, and the individual components, such as the finite element basis functions and their integraton over domains, are very flexible. Consequently, the best way to understand the algorithms and representations used for these computations is to examine how it works for a concrete example; otherwise the program can seem excessively abstract.

The main program is divided into three parts: The first is a finite element program (FEM) that performs the local finite element assembly, etc., and has no knowledge of manifolds. Indeed, this portion stands on its own as a finite element PDE solver over Euclidean spaces. The second part is a set of additions to the manifold code developed in the Chapter 2 that help manage geometric structures such as boundaries for the sake of setting boundary values and solving PDEs. Finally, the third part is a set of tools that oversee the finite element assembly process on manifolds, and has various routines that combine local equations into global ones in different ways.

The primary example in this section, as in Appendix A, is the boundary value problem for Laplace's equation. The domain of solution is the unit disc (see Figure 3-16), which was given the structure of a manifold with three charts (see Figures 3-13 through 3-15). As stated before, this is a natural problem because of its simplicity and importance in physical problems. Furthermore, one can easily derive analytical solutions for simple boundary values, and for more complicated boundary values traditional finite element methods (over subspaces of Euclidean space) are known to perform reasonably well.

3.4.4 Local finite-elements

First, let us discuss the local finite element program. It depends on explicit computational representations of *nodes* and *elements* and uses these abstractions to isolate different stages in the finite element assembly process and to clarify the interdependence of different components. In this discussion, unless explicitly stated, all objects exist in Euclidean spaces.

In this system, nodes are objects that have coordinates, carry values, and have some extra fields (such as various ID numbers that identify them from other nodes in the ensemble), and flags that identify them as boundary nodes. Since each element object also keeps track of the nodes that they contain, each node is also assigned a *local ID* by the element. Conversely, each node must also keep track of the elements to which they belong.

In terms of elements and nodes, then, the finite element assembly process can be expressed rather concisely as follows:

```
(define (assemble-equations source nodes)

  ;; SOURCE is a function from R^2 to R, and NODES is expected to be a vector.

  (let* ((ncount (vector-length nodes))
```

```
     (bcount 0)
     (index-map (make-vector ncount)))

;; First, assign each node an index and count the number of boundary nodes.

(do ((i 0 (+ i 1)))
    ((>= i ncount))
  (node:set-id! (vector-ref nodes i) i)
  (if (node:boundary? (vector-ref nodes i))
      (set! bcount (+ bcount 1))))

;; Next, create a mapping from node indices into matrix row number. (The
;; matrix has one row per interior node.)

(let loop ((i 0) (row 0))
  (if (< i ncount)
      (if (node:boundary? (vector-ref nodes i))
          (begin
            (vector-set! index-map i #f)
            (loop (+ i 1) row))
          (begin
            (vector-set! index-map i row)
            (loop (+ i 1) (+ row 1))))))

;; Loop over the nodes to create row entries:

(let* ((icount (- ncount bcount))
       (big-matrix (make-sparse-matrix icount (1+ icount))))

  (do ((i 0 (+ i 1)))
      ((>= i ncount))

    (if (not (node:boundary? (vector-ref nodes i)))
        (let ((row (vector-ref index-map i)))

          ;; Compute the source term for this row:

          (sparse-matrix-set! big-matrix row icount
                              (node:compute-source (vector-ref nodes i)
                                                   source))

          ;; Combine boundary values:

          (for-each
           (lambda (pair)
             (let ((id (car pair))
                   (val (cadr pair)))
               (if (node:boundary? (vector-ref nodes id))
                   (sparse-matrix-set!
                    big-matrix row icount
                    (- (sparse-matrix-ref big-matrix row icount)
                       (* val (node:get-value (vector-ref nodes id)))))
                   (sparse-matrix-set! big-matrix row
                                       (vector-ref index-map id) val))))
           (node:assemble (vector-ref nodes i))))))

  big-matrix)))
```

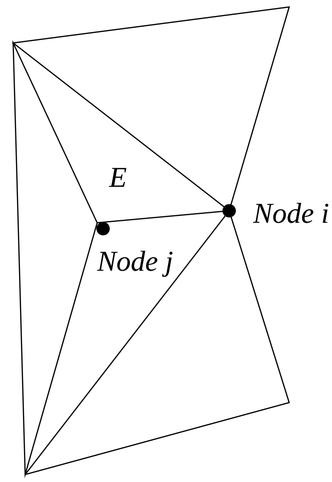


Figure 3-17: As defined in Appendix A, each node corresponds to a vertex in a triangulation, and to each node i there corresponds a finite element basis function ϕ_i. The support of ϕ_i is the union of all those elements adjacent to node i, and hence the intersection of the supports of two basis functions ϕ_i and ϕ_j, $i \neq j$, consists of a union of elements as well. `Element:compute-integrals`, when given an element E and an index i belonging to E, returns the set of all integrals of the form $\int_E \phi_i \cdot L\phi_j$ for all j that are neighbors of i.

Note that this FEM assembly program does not actually compute the integrals, but calls `node:assemble` to recursively construct the appropriate coefficients and combine them.

```
(define (node:assemble node)
  (let ((l (append-map
            (lambda (element index)
              (element:compute-integrals element index))
            (node:get-elements node)
            (node:get-local-ids node))))

    ;; ELEMENT:COMPUTE-INTEGRALS returns a list of pairs, where each pair takes
    ;; the form (node-id . coefficient).  MERGE-TERMS then sorts and adds up
    ;; coefficients that have the same ID.

    (merge-terms l + (lambda (x y) (< (car x) (car y))))))
```

`Node:assemble` calls `element:compute-integrals`, which returns a list of pairs of the form `(node-index .  integral)`, which represent the element's contribution to the finite element integrals involving the basis function centered at the given node. More precisely, let i be the index of the current node, and let j denote the index of one of its neighbors, and let E denote an element shared by these two nodes (see Figure 3-17).[24] Then `element:compute-integrals` and `node:assemble` compute and return a list of pairs of the form:

[24] That is, E is part of the intersection of the supports of the basis functions ϕ_i and ϕ_j.

$$\left(j, \int_E \phi_i \cdot L\phi_j\right). \tag{3.28}$$

`Merge-terms` then adds up contributions corresponding to the same node index j.

This shows that all routines for integrating basis functions and dealing with the differential operator can be isolated in the element abstraction: The FEM assembly program and the nodes exist merely for "book-keeping" purposes, and all the information about the geometry of the domain and the action of the differential operator are encapsulated in the elements. The element abstraction thus isolates all the components that need to be changed in order to modify the type of basis functions used and the method used to integrate them; this simplifies the method's application to manifolds.

Constructing elements and differential operators. The construction of elements is much more complicated than the mere packaging of data. It takes as arguments three procedures for constructing important data structures. The first of these, `make-operator`, takes a list of nodes and returns a list of structures that represent the differential operator (or an approximation thereof) over the element described by the given nodes. It is organized in such a convoluted way because oftentimes it is useful to have the ability to approximate differential operators with variable coefficients with operators whose coefficients are locally constant. To facilitate this, operators need to "know" the element over which it is operating, and hence we have the `make-operator` constructor.

To complicate matters even more, it is often useful to split a differential operator L into three components: An m-vector-valued differential operator L_{left}, a second m-vector-valued opreator L_{right}, and a bilinear form (on vectors in R^m) $\langle , \rangle$, satisfying the equation

$$\int \langle L_{left} f, L_{right} f\rangle = \int (f \cdot Lf), \tag{3.29}$$

where f is an arbitrary differentiable function of compact support, which, for example, can be a basis function.[25] The reason for this is that finite element basis functions are often piecewise polynomial functions, and hence are only differentiable finitely many times. In general, the more degree of differentiability one requires, the higher the order of the polynomials. Since higher-order polynomials require more nodes, their storage and manipulation require more computational resources. Conversely, one can often reduce the amount of data needed by reducing the order of the polynomials. This is possible if one integrates by

[25] It should be clear what L_{left} and L_{right} mean for functions on Euclidean spaces. In the context of manifolds, think of the operators L_{left} and L_{right} as m-tuples of partial differential operators as defined earlier in §3.1, which would map real-valued functions $f : M \to R$ on M to m-vector-valued functions $Lf : M \to R^m$.

parts and split the differential operator into two parts. For example, the Laplacian is often represented by the gradient operator $L_{left} = L_{right} = \nabla$, which when integrated by parts to yield the (negative) Laplacian operator $-\nabla^2$; this allows the use of basis functions that are continuous with piecewise-continuous first partials, such as piecewise-linear functions.[26]

Thus, `make-operator` returns `left-op`, `right-op`, and `combine`, which correspond to L_{left}, L_{right}, and $\langle , \rangle$, respectively. This structure also allows the use of the usual representation of differential operators: Just let `right-op` compute the differential operator, let `left-op` be the identity operator, and replace `combine` with a function product operation.

The other two arguments of `element-maker` are simpler: `Make-integrator` takes as argument a list of nodes and returns a procedure capable of integrating basis functions over the element defined by those nodes, and `make-basis-function` creates a basis function data structure. Note that basis functions are generally abstract data structures that represent mathematical functions, not computational procedures, and their representations are completely flexible: The entire program works so long as `make-integrator` and `make-basis-function` agreed *a priori* upon a consistent representation of basis functions. In practice, as stated above, piecewise polynomial basis functions are often used because their images under differential operators are easy to compute, as are their integrals.

```
;;; Note that this implicitly assumes that elements are the convex hull of
;;; their vertices.

;;; The (meta-)constructor for element-constructors:

(define (element-maker make-operator
                       make-integrator
                       make-basis-function)

  ;; MAKE-INTEGRATOR should take as argument a list of nodes, and returns a
  ;; procedure that takes a variable number of functions (at least 1) and
  ;; integrates their product over the domain specified implicitly as the
  ;; convex hull of the vertex nodes.

  ;; MAKE-BASIS-FUNCTION should take as argument a list of nodes and the index
  ;; of the node that is to be the center of the basis function, and return
  ;; some structure representing basis functions.

  ;; We place no restrictions on the representation of functions over elements,
  ;; so long as the particular instances of MAKE-BASIS-FUNCTION and
  ;; MAKE-INTEGRATOR agree a-priori on the representation.

  ;; MAKE-OPERATOR should take a list of nodes and return LEFT-OP, RIGHT-OP,
  ;; and COMBINE procedure, satisfying (INTEGRATE (COMBINE (LEFT-OP F)
  ;; (RIGHT-OP G))) = (INTEGRATE F (OP G)), i.e. implement integration by parts
  ;; so that basis functions can be less smooth.
```

[26] In the case of Laplace's equation, the symmetric positive semi-definite form $-\int \langle , \rangle$ on the space of differentiable functions is called the *Dirichlet form*.

```
;; The list of nodes facilitates the interpolation of variable coefficients
;; in the operator.  This may not be a good interface, as it makes artificial
;; assumptions on the contract between basis functions and operators (as is
;; the explicit use of LEFT-OP and RIGHT-OP).

(define (make-element vertex-nodes other-nodes)

  ;; The first part stores the coefficients, the second part the source
  ;; terms.  What about coefficients?  Maybe we should incorporate the
  ;; source term into the differential operator.

  (let* ((nodes (append vertex-nodes other-nodes))
         (number-of-nodes (length nodes))
         (n-choose-2 (choose (+ number-of-nodes 2) 2))
         (element
          (vector (make-vector n-choose-2 0)
                  (make-vector n-choose-2 0)
                  vertex-nodes
                  other-nodes
                  (make-vector number-of-nodes #f)))
         (op (make-operator nodes)))

    ;; Add the element to the nodes:

    (let loop ((nodes nodes) (i 0))
      (if (not (null? nodes))
          (begin
            (node:add-element (car nodes) element i)
            (loop (cdr nodes) (+ i 1)))))

    ;; Initiailize elements (and hiding the hair)...

    (let ((integrate (make-integrator vertex-nodes))
          (local-form (operator:get-local-form op)))

      (do ((i 0 (+ i 1)))
          ((>= i number-of-nodes))
        (element:set-basis-function!
         element i (make-basis-function nodes i)))

      (do ((i 0 (+ i 1)))
          ((>= i number-of-nodes))

        (let ((f (element:get-basis-function element i)))

          (do ((j i (+ j 1)))
              ((>= j number-of-nodes))
            (let ((g (element:get-basis-function element j)))
              (element:set-coeff! element i j
                                  (integrate (local-form f g)))
              (element:set-source! element i j (integrate f g)))))))

    element))
make-element)
```

This also shows that, as a matter of efficiency, elements can be called on to evaluate the integrals first when one constructs the domain. One can then work with different boundary values (or source functions, in the case of Poisson's equation) without recomputing the finite element integrals.

3.4.5 Basic FEM algorithm on manifolds

There are two top-level programs that manage the computation of finite element equations on manifolds. The first program manages mesh generation and element construction, while the second program uses these elements and the local finite-element assembly program to generate a sparse matrix that represents the discretized system of linear equations.

What follows is the main portion of the code for the first program:[27]

```
(define (pde:domain-maker generate-node-lists process-complex)
  (lambda (M
           make-vertices
           make-extra-nodes
           tesselate
           . argl)

    ;; First, make the bounding nodes of the convex domain, and then
    ;; triangulate and make the extra nodes:

    (let ((atlas (manifold:get-finite-atlas M)))

      (if (not atlas)
          (error "Error: Can only do FEM with finite atlases."))

      (write-line '(tesselating domain...))

      ;; Do something more complicated here to reduce the overlap:

      (let loop ((charts atlas)
                 (node-lists (generate-node-lists make-vertices atlas argl)))
        (if (not (null? charts))

            ;; TESSELATE should return a list of lists, where each list
            ;; contains the elemental faces of a given dimension (in some given
            ;; polytope).  In the planar case, this reverses the convention in
            ;; fem.scm: The list should be sorted by dimension in *descending*
            ;; order.

            (let* ((chart (car charts))
                   (nodes (car node-lists))
                   (complex (process-complex (tesselate nodes) (cdr charts)))
                   (extra-nodes (make-extra-nodes complex)))

              ;; By default, use FEM-DISCRETIZE.  Can replace with others.
```

[27] A little matter of terminology: Many procedures in this code manipulate data structures called "complexes" (as in `chart:get-complex`). The term refers to *simplicial complexes*, which are spaces that can be formed as the union of points, lines, triangles, tetrahedra, and their higher-dimensional generalizations called *simplices*. Not only are simplicial complexes useful for finite element computation, they are also very important for studying the structure of topological spaces and form one of the starting points for *algebraic topology*. For more details, see Munkres [20]. For our purposes, however, it is just a convenient way to package data structures that describe triangulations on charts.

```
          (make-pde-chart chart extra-nodes fem-discretize complex)
          (loop (cdr charts) (cdr node-lists)))))

  ;; Construct elements.  We don't need to explicitly mark boundaries
  ;; because manifolds should already have such structures defined.

  (lambda (operator make-integrator make-basis-function)
    (let ((element-maker (pde:element-maker operator
                                            make-integrator
                                            make-basis-function)))

      (write-line '(constructing elements...))

      (for-each

       (lambda (chart)

         ;; Construct the elements:

         (write-line
          `(making ,(length (complex->faces (chart:get-complex chart)))
                   elements...))

         (let* ((make-element (element-maker chart))
                (new-elements (map make-element
                                   (complex->faces
                                    (chart:get-complex chart))
                                   (chart:get-extra-nodes chart))))
           (chart:set-elements! chart new-elements)))

       atlas))))))
```

This program is a "meta-constructor" for domain constructors, and returns a procedure that adds sufficient structure to a given manifold (such as nodes and local triangulations, etc.) that finite element analysis can be performed. It provides only a logical skeleton into which other procedures fit; the real work is done by procedures like `generate-node-lists`, `process-complex`, `make-vertices`, `make-extra-nodes`, and `tesselate`.

Given the appropriate procedures for constructing nodes and meshes on charts, the program generates nodes and constructs meshes for each chart. Then, some of the nodes are "pruned" away to control the size of the number of nodes shared between charts.[28] The expression `(make-pde-chart chart extra-nodes fem-discretize complex)` attaches extra data structures to chart, so that in a later stage the information obtained here can be used to construct the elements.[29] Finally, yet another procedure is returned that takes the information obtained above, as well as representations of the differential operator, constructors for basis functions, and integrators of basis functions, and actually constructs the

[28]This will be discussed in more detail in the next section.

[29]The procedure `fem-discretize` is stored away and called later for the local finite element assembly procedure. It provides a simple interface to the program of the previous section. It can always be replaced by a different FEM routine, of course.

elements.

Having constructed elements and prepared the domain of solution for finite element analysis, the second top-level program generates the discretized equations given boundary data and a source function:

```
;;; Given a domain with constructed elements, a source function, and a boundary
;;; value function, produce the appropriate discretized equation.  The nodes
;;; are left with indices that specify their corresponding row in the matrix.

(define (pde:equation-maker merge-equations)
  (lambda (domain source boundary-value . extra-args)

    ;; EXTRA-ARGS gives us finer control over the discretization.

    ;; DOMAIN should be a manifold that already has PDE structures constructed.
    ;; Hence, it contains information about the operator (through the elements
    ;; in its discretized charts).

    ;; BOUNDARY-VALUE is irrelevant for domains without boundary.  Just specify
    ;; anything (but do put in something).

    (let* ((M domain)
           (charts (manifold:get-finite-atlas M))
           (nodes (list->vector (append-map chart:get-nodes charts)))
           (ncount (vector-length nodes)))

      ;; CHART:DISCRETIZE-PDE should return a list of linear equations.  First,
      ;; set the boundary values:

      (write-line `(,ncount nodes generated...))
      (write-line '(setting boundary values...))

      (do ((i 0 (+ i 1)))
          ((>= i ncount))
        (let ((node (vector-ref nodes i)))
          (if (node:boundary? node)
              (node:set-value! node (boundary-value node)))))

      ;; Next, compute the local equation systems:

      (write-line `(computing ,(length charts) local systems of equations...))

      (let ((equations (append-map
                        (lambda (chart)
                          (chart:discretize-pde chart source extra-args))
                        charts)))

        ;; Compute constraints:

        (write-line '(merging local equations...))
        (merge-equations domain equations)))))
```

Once again, this program only serves as a logical skeleton. All the major components of the programs, such as the procedure `merge-equations`, are easily modifiable. This facilitates the testing of different methods for performing these tasks. Indeed, the fol-

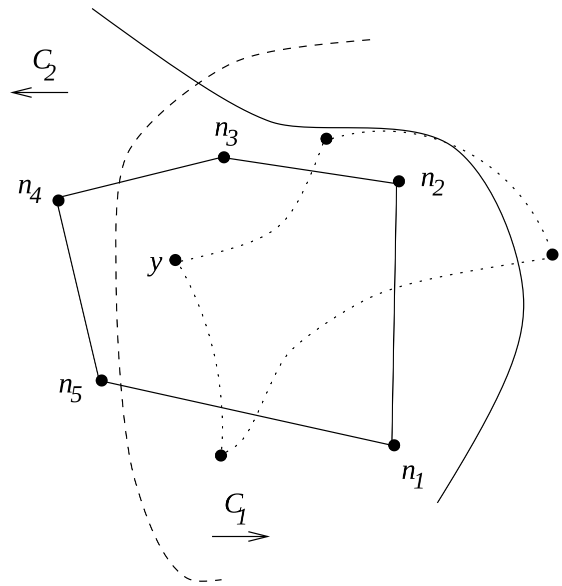


Figure 3-18: A point y in some chart, with its "neighbors" n_1 through n_5, which are the nodes belonging to the element that contains y. (Since the elements of a triangulation partition whatever chart they cover, each point lies in only one element *except* for points lying in the boundaries of elements.) This figure is drawn using the coordinate system of C_2, and it illustrates two charts, with the dotted lines outlining the element (of C_1) to which the point $x = \phi_1(p)$ belongs while the dashed line delineates the boundary of the image $\phi_2(U_1 \cap U_2)$ of C_1 in this coordinate system. The dotted element has a curved boundary because the entire image is seen in the coordinate system of C_2.

lowing sections will explore a couple different implementations of `generate-node-lists`, `process-complex`, and `merge-equations` that control how much charts overlap and how local equations are merged into a global set of equations.

3.4.6 Interpolation between charts

Finally, we come to the most delicate part of the problem: How does one actually combine local equations into a global set of equations? This process is determined by the procedures `generate-node-lists`, `process-complex`, and `merge-equations`, which are passed into `pde:domain-maker` and `pde:equation-maker` as arguments.

As mentioned at the beginning of §3.4, one natural idea is the following: Let $C_1 = (U_1, V_1, \phi_1)$ and $C_2 = (U_2, V_2, \phi_2)$ be charts on the manifold M. Suppose the ith node in the discretized domain is at the point $p \in M$, and that p lies in the intersection $U_1 \cap U_2$. Let $x = \phi_1(p)$ be the coordinate vector corresponding to p in V_1, and let $y = \phi_2(p)$ be the coordinate vector corresponding to p in V_2. Then one could simply constrain the unknown value at x, a_i, to the value at the corresponding point $y = \phi_2(p)$, interpolated from basis functions in C_2. More precisely, let n_i be the indices of the nodes in the element E containing y in C_2 (see Figure 3-18). Then the constraint we want is:

$$a_i = \sum_k \phi_{n_k}(y) a_{n_k}, \tag{3.30}$$

where a_j denotes the sample value $u(p_j)$ of the approximate solution at the jth node, with position p_j. Since the expressions $\phi_{n_i}(y)$ can be computed without reference to any unknowns, we see that this is a linear equation relating unknown nodal values. Thus, the constraints generated this way may simply be "appended" onto the system of locally-discretized equations for each chart, each of which is also linear. Doing this for a sufficiently large number of nodes that lie in the overlap of two charts should generate enough extra equations to relate the local equations derived for each chart. It should be noted that this process of appending constraints produces overdetermined systems, for which exact solutions generally do not exist. Thus, a least-squares approximation is the best one could do. This can be done by computing the *normal equations*, which finds an approximate solution to the overdetermined system $Ax = b$ by minimizing the magnitude of the error $Ax - b$ with respect to the natural inner product of Euclidean space. As will be explained later, however, the formation of the normal equations again runs the risk of producing an *ill-conditioned system*.[30]

A program that implement a general procedure for combining equations and constraints into a large matrix is shown below. It relies on `make-constraints` to construct the constraint equations, and the main body of the program performs the tedious task of constructing the matrix row by row:

```
;;; This complicated-looking procedure performs the simple task of forming a
;;; sparse matrix out of locally-discretized equations and constraint
;;; equations.  The constraints are generated with the help of
;;; MAKE-CONSTRAINTS.

(define (append-constraint-equations make-constraints)
  (lambda (domain equations)

    ;; First, set IDs and clear hidden states:

    (write-line '(setting node ids...))

    (let loop ((id 0) (nodes (manifold:get-nodes domain)))
      (if (not (null? nodes))
          (let ((node (car nodes)))
            (node:set-constraint! node #f)
            (if (node:boundary? node)
                (begin
                  (node:set-id! node 'boundary-node!)
```

[30] That is, a system of equations with a very large condition number.

```
            (loop id (cdr nodes)))
          (begin
            (node:set-id! node id)
            (loop (+ id 1) (cdr nodes))))))

;; Next, generate constraints:

(write-line '(generating constraints...))

(with-values
    (lambda () (make-constraints domain))

  (lambda (c-count clists)
    (let* ((eq-count (length equations))
           (m (+ eq-count c-count))
           (n (+ eq-count 1)))

      (write-line `(constructing a matrix of dimension (,m ,n)...))

      (let ((mat (make-sparse-matrix m n)))

        ;; First, copy the equations:

        (write-line `(copying ,eq-count equations...))

        (for-each
         (lambda (eq)
           (let ((i (equation:get-id eq)))
             (sparse-matrix-set!
              mat i eq-count (equation:get-constant eq))
             (for-each
              (lambda (term)
                (sparse-matrix-set! mat i (term:get-id term)
                                    (term:get-coeff term)))
              (equation:get-terms eq))))
         equations)

        ;; Next, copy the constraints:

        (write-line `(copying ,c-count constraints...))

        (let next-clist ((i eq-count) (clists clists))
          (if (null? clists)
              mat
              (let next-constraint ((clist (car clists)) (i i))
                (if (null? clist)
                    (next-clist i (cdr clists))
                    (let ((constraint (car clist)))
                      (sparse-matrix-set!
                       mat i eq-count (equation:get-constant constraint))
                      (for-each
                       (lambda (term)
                         (sparse-matrix-set! mat i (term:get-id term)
                                             (term:get-coeff term)))
                       (equation:get-terms constraint))
                      (next-constraint (cdr clist) (+ i 1))))))))))))
```

While the basic idea of interpolating unknown values from other charts is simple enough, there are some unresolved details here: For one thing, what does it mean to create constraints for "a sufficiently large number of nodes"? Is it necessary to create constraints for *all* nodes in the overlap, or just some specially-chosen interpolation nodes? Which ones should we use? Furthermore, let C_1, C_2, and C_3 be charts, let $p \in M$ be a point contained in all three charts, and let $x_i = \phi_i(p)$ be the image of p in the chart C_i. Since there are three charts, there are three different constraints we can generate using the recipe above by considering different pairs of charts. Is it better to generate all three constraints, or to generate only one or two of them? Since the basis functions and triangulations in different charts are by no means related to each other, one would expect that the constraints are independent of each other, and hence this is a non-trivial question. Clearly, this problem extends in general to any node that lies in more than two charts, and if not all possible constraints are to be generated, then which ones should we use?

Since there are many possible choices here and no obvious candidate, it seems reasonable to try a couple of different ideas and see how well they perform:

1. Generate all constraints for all nodes in the overlaps between all pairs of charts.

2. Put the set of all charts in some linear ordering, and generate all constraints for all nodes in the overlaps of adjacent charts (in the given ordering).

The following program, `make-all-constraints`, implements the first of the ideas enumerated above by generating all constraints between all pairs of charts:

```
(define (make-all-constraints domain)
  (let ((constraints
         (append-map
          (lambda (pair)
            (let ((chart-1 (car pair))
                  (chart-2 (cadr pair)))
              (append (constrain-all-nodes chart-1 chart-2)
                      (constrain-all-nodes chart-2 chart-1))))
          (pairs (manifold:get-finite-atlas domain)))))
    (values (length constraints) (list constraints))))

(define (constrain-all-nodes chart-1 chart-2)
  (append-map
   (lambda (node)
     (if (node:boundary? node)
         '()
         (let ((eq (chart:pointwise-constraint node chart-2)))
           (if eq
               (list eq)
               '()))))
   (chart:get-nodes chart-1)))

(define (pairs l)
  (let loop ((l l) (result '()))
    (if (null? l)
```

```
        result
        (loop (cdr l)
              (let ((a (car l)))
                (let loop ((l (cdr l)) (result result))
                  (if (null? l)
                      result
                      (loop (cdr l) (cons (list a (car l)) result))))))))
```

It can be passed into `append-constraint-equations` to construct the constraints. This program is rather straightforward: For all pairs of distinct charts, generate all possible constraints from nodes in the overlap between these two charts.

The next program implements the second idea, which involves ordering the charts. Since atlases are represented by Scheme lists, the implicit ordering of lists is used to linearly order the charts.

```
(define make-all-ordered-constraints
  (let ((exists? (lambda (node) #t)))
    (lambda (domain)
      (let* ((charts (manifold:get-finite-atlas domain))
             (result-1 (charts->constraints charts exists?))
             (result-2 (charts->constraints (reverse charts) exists?)))
        (values (+ (car result-1) (car result-2))
                (append (cadr result-1) (cadr result-2)))))))

;;; The charts come in a ordered list, so that implicit ordering is used as the
;;; linear ordering we need.

(define (charts->constraints charts good-node?)

  ;; The predicate GOOD-NODE? lets the calling procedure control which nodes to
  ;; use.  In this case, it simply uses all non-boundary nodes

  (let next-chart ((charts charts)
                   (count 0)
                   (clists '()))
    (if (null? charts)

        (list count clists)

        ;; Go through each node in the chart and check for constraints:

        (let ((chart (car charts)))
          (let next-node ((nodes (chart:get-nodes chart))
                          (count count)
                          (clist '()))
            (if (null? nodes)
                (next-chart (cdr charts) count (cons clist clists))
                (let ((node (car nodes)))

                  ;; We only want to create constraints for nodes that do not
                  ;; already have a constraint:

                  (if (and (good-node? node)
                           (not (node:get-constraint node))
                           (not (node:boundary? node)))
```

```
                    (let ((eq (make-constraint node (cdr charts))))
                      (if eq
                          (next-node (cdr nodes) (+ count 1) (cons eq clist))
                          (next-node (cdr nodes) count clist)))
                    (next-node (cdr nodes) count clist)))))))))

(define (make-constraint node charts)
  (let loop ((charts charts))
    (if (null? charts)
        #f
        (let ((eq (chart:pointwise-constraint node (car charts))))
          (if eq
              eq
              (loop (cdr charts)))))))
```

This program is a bit more complicated: `Charts->constraints` takes a list of charts and produces a list of constraints, such that a node n in a chart C_i is constrained to a chart C_j if and only if j is the least integer greater than i such that C_j contains n. The same procedure is then called again to construct constraints in the reverse direction, so that constraints exist for charts adjacent in this linear ordering (or as close to being adjacent as possible).

Both of the programs above call `chart:pointwise-constraint`, which can be implemented thusly:

```
(define (chart:pointwise-constraint node chart)

  ;; The coefficients of a linear constraint for some node x should simply be
  ;; the value at p of the basis function centered at x.  This linearity
  ;; depends only on the fact that the solution is approximated by a linear
  ;; combination of basis functions.

  (if (chart:member? (node:get-point node) chart)
      (let* ((x (chart:point->coords (node:get-point node) chart))
             (element (chart:coords->any-element x chart)))
        (if element
            (let loop ((nodes (element:get-nodes element))
                       (i 0)
                       (const 0)
                       (terms (list (make-term node -1))))
              (if (null? nodes)
                  (begin
                    (node:set-constraint! node chart)
                    (make-equation node const terms))
                  (let ((neighbor (car nodes))
                        (coeff (evaluate-basis-function
                                (element:get-basis-function element i) x)))
                    (if (node:boundary? neighbor)
                        (loop (cdr nodes)
                              (+ i 1)
                              (- const (* (node:get-value neighbor) coeff))
                              terms)
                        (loop (cdr nodes)
                              (+ i 1)
                              const
```

```
                         (cons (make-term neighbor coeff) terms))))))
      #f))
   #f))
```

It simply finds an element of `chart` to which `node` belongs, and loops through the nodes of the given element to evaluate the basis functions and compute the coefficients.

3.4.7 Some numerical results.

To test the ideas above, we should perform some numerical experiments. The canonical problem on which every FEM program should cut its teeth is the boundary value problem for Laplace's equation. For us, the domain will be the unit disc $\{(x, y) \in R^2 : x^2 + y^2 \leq 1\}$ in the plane (see Figure 3-16), with the boundary value

$$f(\theta) = \cos(2\theta). \tag{3.31}$$

Using the angle addition formula for cosines, one finds that $f(\theta) = \cos^2\theta - \sin^2\theta$. But the function $g(x, y) = x^2 - y^2$ satisfies Laplace's equation everywhere, and $g(\cos\theta, \sin\theta) = f(\theta)$ for all θ, so g must be the true solution corresponding to the boundary data f. This gives us a convenient problem on which to test the ideas above and an exact solution against which to compare answers.

So far we have only seen how to implement the auxiliary procedure `merge-equations`: The constructor `append-constraint-equations`, given either `make-all-constraints` or `make-all-ordered-constraints`, should return a procedure that constructs constraint equations for `pde:equation-maker`. But we also need to implement the auxiliary procedures for `pde:domain-maker`. To do this, we need the procedures `make-nodes-for-each-chart` and `do-nothing-to-complex`, which, as their names suggest, are very simple procedures. We will need more complicated auxiliary procedures later on, but these simple programs suffice for now.

The definitions of key data structures are shown below:

```
;;; The procedure that prepares the domain for the PDE solver:

(define pde:make-simple-domain
  (pde:domain-maker make-nodes-for-each-chart do-nothing-to-complex))


;;; Two different ways for generating constraints:

(define combine-equations-with-overlap1
  (pde:equation-maker
   (append-constraint-equations make-all-constraints)))

(define combine-equations-with-overlap2
  (pde:equation-maker
```

```
   (append-constraint-equations make-all-ordered-constraints)))


;;; Construct the domain of the PDE:

(define disc
  (make-ball 2 make-spherical-sphere))


;;; Construct the Laplacian.  Note that OPERATOR:IMBEDDED-POLY-OP simply
;;; packages the operators left-op, right-op, and combine.  This splits the
;;; Laplacian into two parts through integration by parts.

(define imbedded-poly-laplacian
  (make-operator
   disc
   (operator:imbedded-poly-op
    poly-gradient
    poly-gradient
    (lambda (v w) (basis:scalar* -1 (basis:dot v w))))))


;;; The true solution of Laplace's equation that we're trying to approximate:

(define (test-function node)
  (let ((x (x-coord-map node))
        (y (y-coord-map node)))
    (- (square x) (square y))))
```

Having defined the necessary auxiliary procedures, we can now try to compute the solution of Laplace's equation:

```
;;; Prepare the domain for FEM:

(define make-test-domain
  (pde:make-simple-domain disc                 ;; The domain.
                          make-mesh            ;; A generic vertex generator.
                          make-no-extra-nodes  ;; No edge nodes, just vertices.
                          planar-triangulate   ;; A generic mesh generator.

                          ;; Some extra parameters:
                          '(rectangular 10 5)
                          '(spherical 5 10)))
(tesselating domain...)
;Value: make-test-domain


;;; Construct the elements and initialize finite element integrals:

(make-test-domain
 ;; The Laplacian we just constructed.
 imbedded-poly-laplacian

 ;; Integrates directly in Euclidean space -- It cheats!
 make-triangular-imbedded-integrator

 ;; Make some generic piecewise-polynomial basis functions.
 pde:make-imbedded-poly-basis-function)
```

```
(constructing elements...)
(making 72 elements...)
process time: 4880 (4470 RUN + 410 GC); real time: 5744
(making 72 elements...)
process time: 4960 (4540 RUN + 420 GC); real time: 5761
(making 70 elements...)
process time: 4810 (4370 RUN + 440 GC); real time: 5616
;No value


;;; Assemble the equations, generate constraints, and build the matrix
;;; equation:

(define mat1
  (combine-equations-with-overlap1 disc             ;; The domain again.
                                   0-function       ;; No source term.
                                   test-function))  ;; The true solution.
(141 nodes generated...)
(setting boundary values...)
(computing 3 local systems of equations...)
(40 equations generated for 50 nodes.)
(40 equations generated for 50 nodes.)
(41 equations generated for 41 nodes.)
(merging local equations...)
(setting node ids...)
(generating constraints...)
(constructing a matrix of dimension (267 122) ...)
(copying 121 equations...)
(copying 146 constraints...)
process time: 13560 (12180 RUN + 1380 GC); real time: 20325
;Value: mat1


;;; Try the other method:

(define mat2
  (combine-equations-with-overlap2 disc 0-function test-function))
(141 nodes generated...)
(setting boundary values...)
(computing 3 local systems of equations...)
(40 equations generated for 50 nodes.)
(40 equations generated for 50 nodes.)
(41 equations generated for 41 nodes.)
(merging local equations...)
(setting node ids...)
(generating constraints...)
(constructing a matrix of dimension (235 122) ...)
(copying 121 equations...)
(copying 114 constraints...)
process time: 9800 (8860 RUN + 940 GC); real time: 14915
;Value: mat2


;;; Neither matrices are square, of course, because of the constraint
;;; equations:

(sparse-matrix-size mat1)
;Value 62: (267 122)
```

Total number of nodes	Absolute error			Relative error	
	Maximum	Minimum	Average	Maximum	Minimum
121	0.186186	0.000346268	0.0321264	3.04321	-1.98107
253	0.174211	5.49186e-05	0.0296698	18.854	-79.1678
433	0.170829	3.3609e-05	0.0299808	11.44	-14.8512
661	0.167295	0.00010959	0.0310626	30.1526	-16.9254
937	0.163327	1.76278e-05	0.031479	38.7539	-44.89
1261	0.160884	3.38679e-06	0.0323654	53.2708	-52.5735
1633	0.160982	4.98298e-06	0.0327103	76.6556	-64.7096
2053	0.162743	7.24373e-06	0.0334327	102.64	-83.955
2521	0.163858	1.6586e-05	0.0342905	109.606	-131.524
3037	0.163727	8.36536e-06	0.0352394	153.382	-154.745
3601	0.165879	1.22342e-05	0.0365282	200.408	-188.901

Table 3.1: Statics of the results generated by `make-all-constraints`.

```
(sparse-matrix-size mat2)
;Value 63: (235 122)


;;; Use least-squares to solve these guys:

(define mat1 (sparse-normal-equations mat1))
;Value: mat1

(define v1 (sor mat1 1000 1.9 ))
(residual: 5.731092683758376e-16)
;Value: v1

(define mat2 (sparse-normal-equations mat2))
;Value: mat2

(define v2 (sor mat2 1000 1.9))
(residual: 7.216449660063518e-16)
;Value: v2
```

Note that we tested both constraint-generation systems without having to recompute the finite element integrals. This is one of the principal advantages of structuring the program to exploit the modularity of the finite element method.

The numerical experiments consist of a series of 11 tests, with the number of nodes ranging from 63 to 3,601; note that because some methods discard unnecessary nodes, the actual number used for computation may change between methods. The code used to run the numerical experiments themselves are very similar to what is shown above, and hence will not be listed separately. Table 3.4.7 shows the statistics based on results generated using `make-all-constraints`, while Table 3.4.7 shows the statistics for the results generated using the other method.

Total number of nodes	Absolute error			Relative error	
	Maximum	Minimum	Average	Maximum	Minimum
121	0.192343	0.000764133	0.0337441	2.17113	-1.65951
253	0.180941	5.30935e-06	0.029554	27.2596	-68.7197
433	0.166176	9.66714e-06	0.0281985	13.4546	-8.88716
661	0.16295	5.27949e-05	0.0291729	18.9087	-19.8467
937	0.15868	4.36369e-07	0.0294747	34.479	-23.4737
1261	0.158527	2.13021e-05	0.0300879	39.5776	-43.8798
1633	0.159253	4.22971e-06	0.0305563	58.4892	-56.2954
2053	0.157801	1.44311e-05	0.030769	62.5005	-86.9364
2521	0.159835	1.02857e-06	0.031765	108.839	-90.1434
3037	0.161139	1.10304e-05	0.032667	123.56	-132.926
3601	0.163639	1.53283e-06	0.033938	165.795	-163.194

Table 3.2: Statistics of the results generated by `make-all-orderd-constraints`.

Note that in both tables, the maximum absolute error remains fairly constant. This may hint at a deeper reason for the method's failure. Such issues are discussed in the next section, where this situation is analyzed a little more closely.

Figure 3-19 plots the average absolute error against the number of nodes using the data from Table 3.4.7, while Figure 3-20 does the same for Table 3.4.7.

Figure 3-21 plots the true solution, while Figure 3-22 plots one solution obtained by `make-all-constraints`. As one can see, they are qualitatively similar, even though numerically the solution is fairly far off.

3.4.8 The problem with interpolation

As can be seen from the data in the previous section, neither of the methods work very well, even though they employed relatively straightforward algorithms and obtained qualitatively reasonable results.

The main problem appears to be that the interpolation approach produces more equations than unknowns, which in general yields *overdetermined* systems of equations. There are two consequences of this overdetermination: First, geometrically speaking, the basis functions become too *rigid*. Becuase these methods enforced too many constraints on nodal values in overlaps, the basis functions in different charts become very tightly dependent on each other, and the approximate solution itself (which consists of linear combinations of basis functions) becomes too "stiff" to conform to the real solution (see Figure 3-23). As a result, much of the numerical accuracy is lost.

A second problem may be that in order to solve a large system of overdetermined system

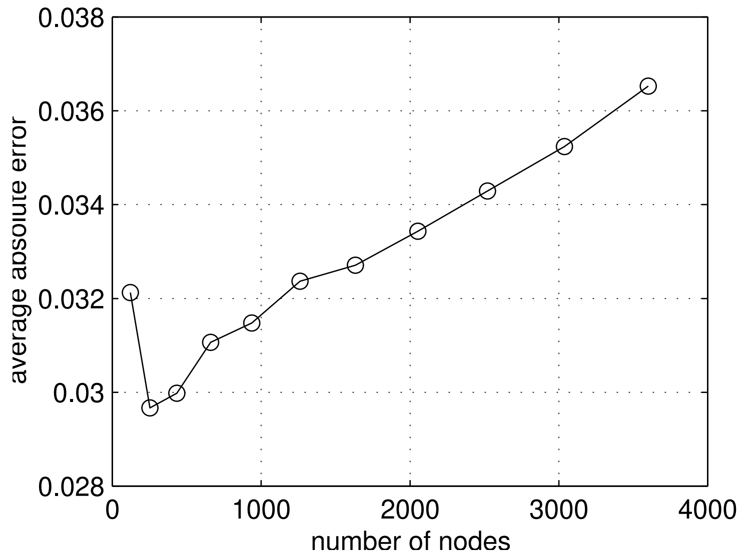


Figure 3-19: Average absolute error versus number of nodes. The results were generated using `make-all-constraints`.

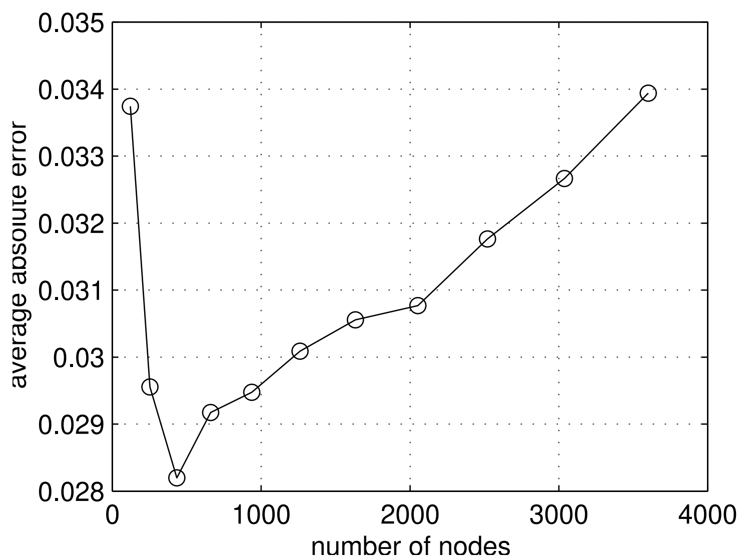


Figure 3-20: Average absolute error versus number of nodes. The results were generated using `make-all-ordered-constraints`.

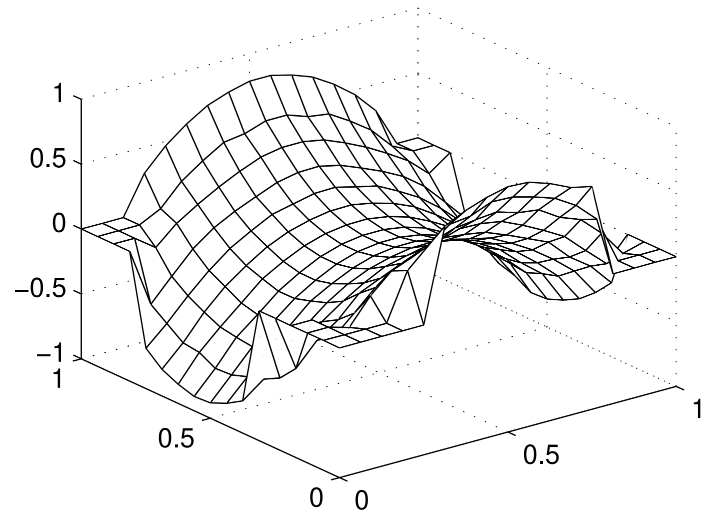


Figure 3-21: The true solution to the disc problem. Note that this plot is generated in a fashion similar to Figures 3-5 through 3-12: The domain is divided into a simple square grid, over which the sample values are averaged. This reduces the number of points to be plotted. The surface generated is a hyperbolic paraboloid of one sheet, as expected.

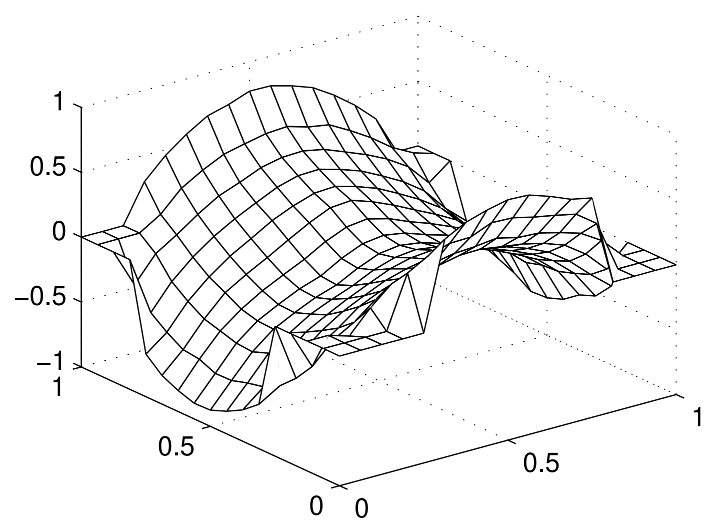


Figure 3-22: The sample solution generated by using all possible constraints.

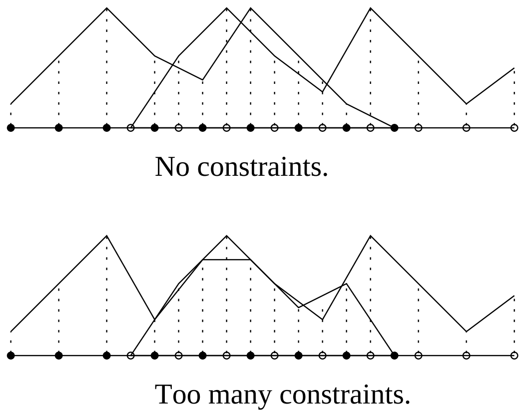


Figure 3-23: Enforcing too many constraints causes basis functions to become *too* dependent on each other.

of equations,

$$Ax = b, \tag{3.32}$$

where the number of rows of A far exceeds its number of columns, one would normally have to compute the normal equations:[31]

$$A^T A = A^T b. \tag{3.33}$$

Now, this should look somewhat familiar. It is, in fact, our friend from §3.3.2, where the "transpose trick" was used in an attempt to make relaxation converge for a class of sparse matrices. In this case, however, more than convergence is at stake: If A is not square, it simply does not make sense to apply relaxation! But in multiplying A by A^T, we have once again made the system of equation even more ill-conditioned. Furthermore, the resulting Gauss-Seidel iteration matrix again has a spectral radius close to 1, making convergence extremely slow.

[31] This is what the procedure `sparse-normal-equations` does. While there exist much better methods for producing least-squares solutions to overdetermined systems, such as singular value decomposition (also known as SVD; see [24]), they do not apply easily to large systems of equations. In order to use iterative solution methods, the normal equations are the easiest way to facilitate the use of iterative solution methods like relaxation on overdetermined systems.

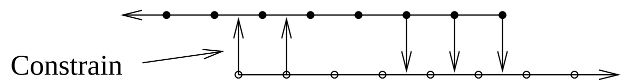


Figure 3-24: Only nodes near the edge in their own charts are allowed to become interpolation nodes. This reduces the amount of "rigidity" in the approximate solution.

3.4.9 Other approaches to FEM on manifolds

How can we avoid the problems associated with overdetermined systems of equations? There are a few alternatives. First, we can use more sophisticated methods of generating constraint equations and choosing interpolation nodes, such as the methods proposed in Chesshire and Henshaw [7] or Petersson [23]. While this will not avoid the necessity of computing the normal equations, it does hold the hope of minimizing the effects of the rigidity problem.

Improving interpolation methods

For the sake of completeness, let us take a brief look at how well these variations on interpolation methods work. The basic algorithms tested here are:

1. The idea of Chesshire and Henshaw, CMPGRD.

2. Same as `make-all-ordered-constraints`, except nodes in overlap regions are allowed to become interpolation nodes if and only if they are near the chart's edge.

The second idea above attempts to create an interpolation geometry depicted in Figure 3-24. Contrast this with Figure 3-23, and one sees that this should help make the system of equations less overdetermined while still propagating enough information to arrive at a reasonable solution.

Figure 3-25 shows the result of the Chesshire-Henshaw algorithm, while 3-25 shows the results of using the second idea. The accuracy should have improved slightly. However, relaxation converges sufficiently slowly that the improvement in accuracy, if any at all, is probably lost in the noise.

A method that works

This section describes a method that actually works fairly well compared to the interpolation methods of earlier sections. It avoids the problem of generating overdetermined systems of equations, and the global matrix of equations it generates is guaranteed to be symmetric

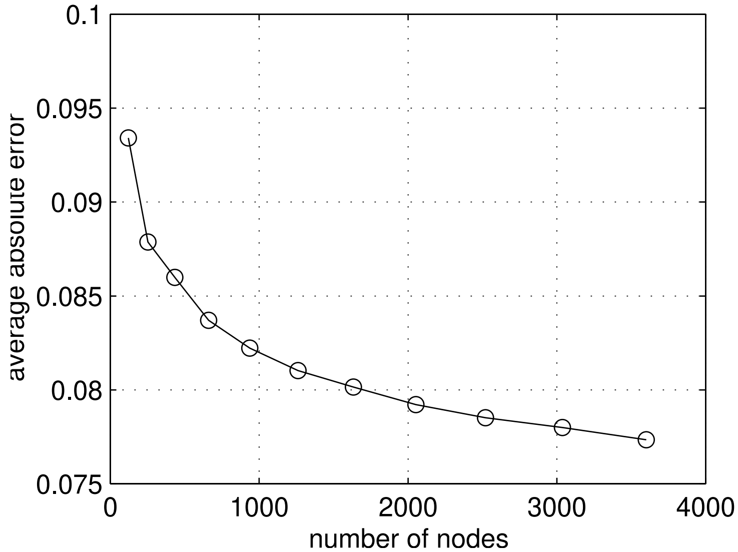


Figure 3-25: The results generated using Chesshire and Henshaw's CMPGRD algorithm.

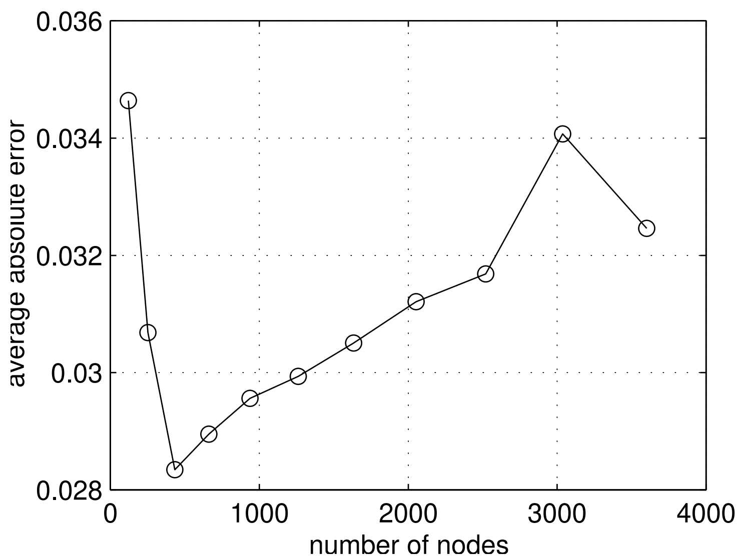


Figure 3-26: The results generated using the idea depicted in Figure 3-24.

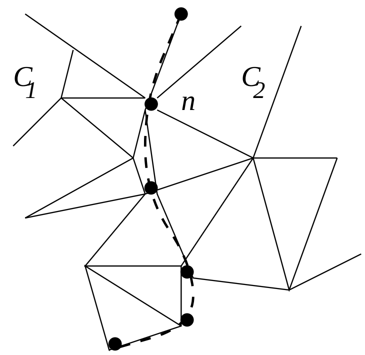


Figure 3-27: The idealized case, where charts do not overlap but intersect nicely along a common edge.

positive-definite, and thus solvable by relaxation without having to worry about normal equations and condition numbers. This method involves "pretending" as if the mesh were global, even if it were not, and for this reason it is referred to as the "semi-local method" here, even though by our earlier definition this is a strictly local discretization method.

The basic idea is simple: Suppose that charts, instead of *overlapping*, fit together like jigsaw puzzle on the manifold along well-defined boundaries (see Figure 3-27).[32]

Suppose now that the ith node lies on the boundary between these two "charts." From C_1 the node obtains an equation of the form

$$u_i = \sum_j a_{ij} u_j + b, \tag{3.34}$$

where the u_j are the unknowns sample values, and the a_{ij} are the finite element coefficients. Similarly, from C_2 the node obtains:

$$u'_i = \sum_j a'_{ij} u'_j + b'. \tag{3.35}$$

Now, consider what the constraint approach actually does: In this idealized case, the node in question does not lie inside an element, but rather is also a node of the other chart. Thus, the constraint approach must append the equation

$$u'_i - u_i = 0. \tag{3.36}$$

[32]Actually, images of charts on manifolds are generally open sets, so they *cannot* intersect along a boundary in the way described here. However, their *closures* can behave this way.

This is equivalent to the system of two equations:[33]

$$\begin{aligned} u_i &= \tfrac{1}{2}\left(\sum_j a_{ij}u_j + \sum j a'_{ij}u'_j + b + b'\right), \\ u_i &= \tfrac{1}{2}\left(\sum_j a_{ij}u_j - \sum j a'_{ij}u'_j + b - b'\right). \end{aligned} \tag{3.37}$$

But consider the finite element integrals in Equation (A.28) of §A.2.3: In order to obtain the correct finite element equation over the *whole* mesh, the correct equation is the top equation, which is the *sum* of the two contributions from the charts. In the context of finite elements, the bottom equation makes no sense at all.[34] Thus, the constraint approach overdetermines the discretized system of equations, and the addition of this extra equation destroys the accuracy of the approximation method in this idealized case.

This is a fairly clear indication that we should add the equations corresponding to the same node in different charts. Furthermore, this generates *one* equation for each interior node, instead of two as in the interpolation case. And, because of the form of the finite element integrals in Equation (A.28), the matrix is guaranteed to be symmetric positive semi-definite; invertibility then guarantees positive-definiteness.

Now, in general, charts willl not cover the manifold this nicely. However, we can always try to make the overlap as small as possible (in terms of nodes shared by charts), and then *pretend* as if we are in the idealized case and apply the equations above.

More formally, the following is the *semi-local algorithm.* Note that $\{C_i\}$ is a given list of charts.

1. Construct a set of nodes N_i for each chart C_i.

2. For each node n in N_i and for each chart C_j with $j > i$, check if n belongs to C_j. If so, remove n from N_i. This *completely* removes the overlap (in terms of sample points) between charts.

3. For each remaining node n in N_i and each chart C_j with $j < i$, check if n belongs to C_j. If so, make a copy of n and add it to N_j. This restores some overlap. Furthermore, while this cannot guarantee that local meshes agree in intersections of charts, it does guarantee that all charts share all nodes in overlap regions.

4. Triangulate and initialize elements; perform local FEM computation. The previous step may have restored too much overlap, so the meshes may have to be "trimmed."

[33] These equations are obtained by identifying the variables u_i and u'_i, and then taking the sum and the difference of the two resulting equations contributed by the two charts.

[34] This can come about if the elements had opposite orientations, so that the integrals pick up an extra minus sign.

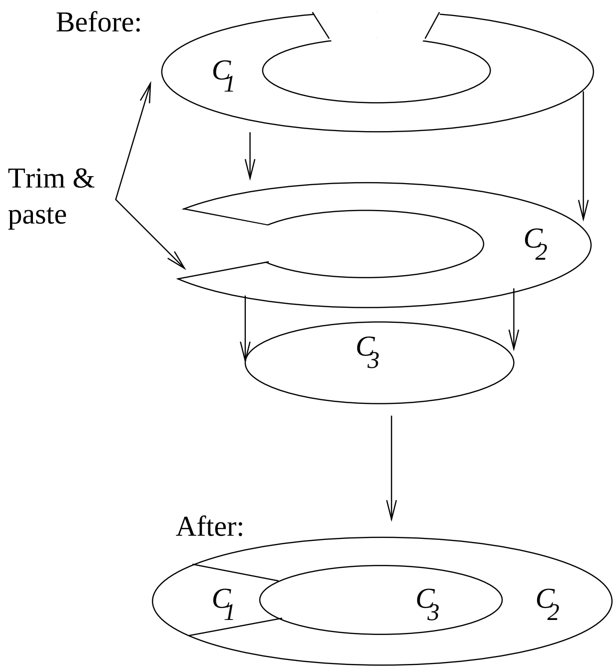


Figure 3-28: The closer a chart is to the bottom of the "stack," the more likely it will keep its nodes. The lower nodes are then copied to the top charts. Intuitively, think of this as cutting holes from the top charts, and then "pasting" them downwards onto lower charts.

5. For each node n' of chart C', if it is a copy of some node n in another chart C, then add the equation of n' in C' to the equation of n in C, and remove the variable corresponding to n'.

Figure 3-28 depicts what the semi-local algorithm does to the overlap between charts. Steps 2 and 3 above are carried out by the following implementation of the auxiliary procedures `generate-node-lists`:

```
;;; Generate lists of nodes for each chart, and then reduce the overlap:

(define (generate-node-lists make-nodes charts argl)

  ;; Generate a list of nodes for each chart, then loop over the charts.  Note
  ;; that the earlier a chart is in the list, the less likely its nodes are to
  ;; survive.

  (let next-chart ((charts charts)
                   (lists (make-nodes-for-each-chart make-nodes charts argl))
                   (result '())
                   (reversed '())
                   (count 0))
    (if (null? charts)
```

```
        (copy-overlap-nodes count result reversed)
        (next-chart (cdr charts)
                    (cdr lists)
                    (cons (remove-overlap-nodes (car lists) (cdr charts))
                          result)
                    (cons (car charts) reversed)
                    (+ count 1)))))

(define (make-nodes-for-each-chart make-nodes charts extra-args)
  (map (lambda (chart) (apply make-nodes (cons chart extra-args))) charts))


;;; Take out all nodes in NODES that belong to any of the charts in CHARTS.

(define (remove-overlap-nodes nodes charts)
  (let next-node ((nodes nodes) (result '()))
    (if (null? nodes)
        result
        (let* ((node (car nodes))
               (p (node:get-point node)))
          (let next-chart ((charts charts))
            (if (null? charts)
                (next-node (cdr nodes) (cons node result))
                (if (chart:member? p (car charts))
                    (next-node (cdr nodes) result)
                    (next-chart (cdr charts))))))))))


;;; For each node list in LISTS, take each node and see if it's in one of the
;;; charts that come after the node's own chart in list-order.  If so, make a
;;; copy of that node and put it in the corresponding chart.  Note that the
;;; order of node lists is reversed.

(define (copy-overlap-nodes count lists charts)
  (let ((v (make-vector count '())))
    (let next-list ((lists lists) (charts charts) (i 0) (result '()))
      (if (null? lists)
          result
          (let next-node ((nodes (car lists)))
            (if (null? nodes)
                (next-list (cdr lists) (cdr charts) (+ i 1)
                           (cons (append (vector-ref v i) (car lists)) result))
                (let ((node (car nodes)))
                  (if (or (node:local-boundary? node)
                          (node:boundary? node))
                      (let ((p (node:get-point node)))
                        (let next-chart ((charts (cdr charts))
                                         (j (+ i 1))
                                         (l (cdr lists)))
                          (if (null? charts)
                              (next-node (cdr nodes))
                              (let ((chart (car charts)))
                                (if (chart:member? p chart)
                                    (let ((other (close-node p (car l))))
                                      (if other
                                          (node:set-constraint! other node)
                                          (vector-set! v j
                                                       (cons
                                                        (node:copy node chart)
```

```
                                                (vector-ref v j))))))
                                (next-chart (cdr charts) (+ j 1) (cdr l))))))
                      (next-node (cdr nodes))))))))))


;;; A kluge to make sure nodes do not become too close to each other:

(define close-node
  (let* ((close-enuf? (make-comparator .01))
         (too-close? (lambda (p q)
                       (close-enuf? (vector:distance p q) 0))))
    (lambda (p l)
      (let loop ((l l))
        (if (null? l)
            #f
            (if (too-close? p (node:get-point (car l)))
                (car l)
                (loop (cdr l))))))))
```

After this stage, the amount of overlap between charts (in terms of how many nodes are shared) should have been reduced. But more importantly, the fact that nodes are shared will help us construct the equations later.[35] However, the amount of overlapping after this stage may still be too much, so after triangulation it is necessary to "trim" the mesh a bit. This is accomplished through the following implementation of the auxiliary procedure `process-complex`:

```
;;; After filtering out nodes, local boundary information becomes useless...

(define (exact-overlap complex charts)
  (kill-extra-nodes complex charts)
  (resurrect-only-connected-nodes complex charts)
  (keep-only-live-nodes complex charts))

(define (kill-extra-nodes complex charts)

  ;; Figure out which nodes to keep by looking at the overlaps:

  (write-line `(processing ,(length (complex->vertices complex)) nodes...))

  (let next-node ((nodes (complex->vertices complex)))
    (if (not (null? nodes))
        (let ((node (car nodes)))
          (let ((p (node:get-point node)))
            (let next-chart ((charts charts))
              (if (null? charts)
                  (next-node (cdr nodes))
                  (let ((chart (car charts)))
                    (if (chart:member? p chart)
                        (let ((node (car nodes)))
                          (node:kill! node)
                          (node:set-local-boundary! node #f)
```

[35] Nodes are shared in the sense that if n belongs to a chart C_1, and its location on the manifold also places it in the chart C_2, then a node at exactly the same location exists in C_2, and hence the two nodes can be identified later on.

```
                        (next-node (cdr nodes)))
                      (next-chart (cdr charts))))))))))

(define (resurrect-only-connected-nodes complex charts)

  ;; Only keep nodes that are connected to live ones:

  (write-line '(figuring out overlaps...))

  (let loop ((faces (complex->faces complex)) (keep '()))
    (if (null? faces)
        (for-each
         (lambda (face)
           (for-each
            (lambda (node)
              (if (not (node:active? node))
                  (begin
                    (node:set-local-boundary! node #t)
                    (node:resurrect! node))))
            face))
         keep)
        (if (at-least-one-live-node? (car faces) charts)
            (loop (cdr faces) (cons (car faces) keep))
            (loop (cdr faces) keep)))))

(define (keep-only-live-nodes complex charts)

  ;; Figure out which faces/edges/etc. to keep:

  (write-line '(processing complex...))

  (let loop ((complex complex) (result '()))
    (if (null? complex)
        (reverse result)
        (let inner-loop ((faces (car complex)) (okay-faces '()))
          (if (null? faces)
              (loop (cdr complex) (cons okay-faces result))
              (let* ((face (car faces))
                     (list? (list? face)))
                (if (or (and list? (not (memq #f (map node:active? face))))
                        (and (not list?) (node:active? face)))
                    (inner-loop (cdr faces) (cons face okay-faces))
                    (inner-loop (cdr faces) okay-faces))))))))

(define (at-least-one-live-node? face charts)
  (memq #t (map node:active? face)))
```

This works much like the earlier routines: It removes all possible overlap, then "grows" the mesh back a little bit. But because this stage occurs *after* the triangulation, the structure of the mesh can be used to control how much overlap there is. And because the earlier stage ensured that intersecting charts *share* nodes in overlap regions, this guarantees that this geometric configuration is as close to the ideal situation in Figure 3-27 as possible.

The following definitions then combine the local equations into a global system of equations, and construct the top-level programs:

```
;;; Generate the sparse matrix by adding appropriate equations together:

(define (merge-equations domain equations)
  (let ((nodes (manifold:get-nodes domain))
        (count 0)
        (mat #f))

    ;; First, assign IDs to nodes, and create the matrix:

    (write-line '(creating matrix...))

    (let loop ((nodes nodes) (i 0))
      (if (null? nodes)
          (begin
            (set! count i)
            (set! mat (make-sparse-matrix count (+ count 1))))
          (let ((node (car nodes)))
            (cond ((node:boundary? node)
                   (node:set-id! node 'boundary-node!)
                   (loop (cdr nodes) i))
                  ((node:get-constraint node)
                   (node:set-id! node 'constrained-node!)
                   (loop (cdr nodes) i))
                  (else
                   (node:set-id! node i)
                   (loop (cdr nodes) (+ i 1)))))))

    ;; Next, start filling in equations while keeping track of constraints:

    (write-line '(copying equations...))

    (let next-eq ((equations equations))
      (if (null? equations)
          (begin
            (write-line '(done!))
            mat)
          (let* ((eq (car equations))
                 (i (node:get-real-id (equation:get-node eq))))

            (sparse-matrix-set! mat i count
                                (+ (equation:get-constant eq)
                                   (sparse-matrix-ref mat i count)))

            (let next-term ((terms (equation:get-terms eq)))
              (if (null? terms)
                  (next-eq (cdr equations))
                  (let* ((term (car terms))
                         (j (node:get-real-id (term:get-node term)))
                         (val (term:get-coeff term)))
                    (sparse-matrix-set! mat i j (+ (sparse-matrix-ref mat i j)
                                                    val))
                    (next-term (cdr terms))))))))))


;;; Construct the top-level programs:

(define combine-equations-without-overlap
  (pde:equation-maker merge-equations))
```

Total number of nodes	Absolute error			Relative error	
	Maximum	Minimum	Average	Maximum	Minimum
63	0.105298	0.000442996	0.0131643	0.547058	-1.19462
130	0.0745854	0.000150227	0.00777591	2.99789	-10.7455
225	0.049322	0.00010038	0.00369715	0.991634	-2.70366
337	0.0532307	1.88416e-06	0.0067024	5.5545	-1.92022
485	0.0762939	1.29948e-06	0.00677602	22.5097	-7.5364
655	0.0420157	4.13828e-06	0.00207558	2.01222	-0.65778
843	0.0232905	1.88216e-07	0.00137413	5.82382	-1.90627
1062	0.0270354	7.00353e-07	0.0012742	2.89441	-0.983369
1297	0.0233997	4.69356e-06	0.00224749	3.20791	-9.62969
1562	0.0187541	1.03235e-07	0.00139054	0.542516	-1.62893
1862	0.0172077	7.19112e-07	0.000983776	6.20396	-2.05861

Table 3.3: Statistics of the results generated by the "semi-local method."

```
(define pde:make-domain-without-overlaps
  (pde:domain-maker generate-node-lists exact-overlap))
```

Like `append-constraint-equations`, this program mostly performs the tedious task of matrix construction. Overlaps between elements from different charts are a source of error for this method. However, the algorithm very carefully reduces the amount of overlap between charts to the minimum required for the merging process.

Table 3.4.9 shows the results generated by this method, while Figure 3-29 shows an approximate solution generated this way.

What is more interesting is a plot of the relative error in Figure 3-30: By a comparison with Figure 3-16, one sees that the the areas with the highest relative error are very much correlated with chart boundaries, which is where we would expect the errors to be maximized.

Figure 3-31 shows a plot of the average absolute error versus the number of nodes, which should be convincing evidence that this method, while not extremely accurate, does converge to the true solution at a reasonable rate as the number of nodes is increased.

Why it works

The success of the method described in the previous section depends very much on the fact that Laplace's equation comes from a *variational principle*. That is, solutions of Laplace's equation minimize an action integral with respect to a simple Lagrangian density function we can construct. In general, finite element methods owe their success to the existence of variational principles and the relative smoothness of solutions, and in problems where

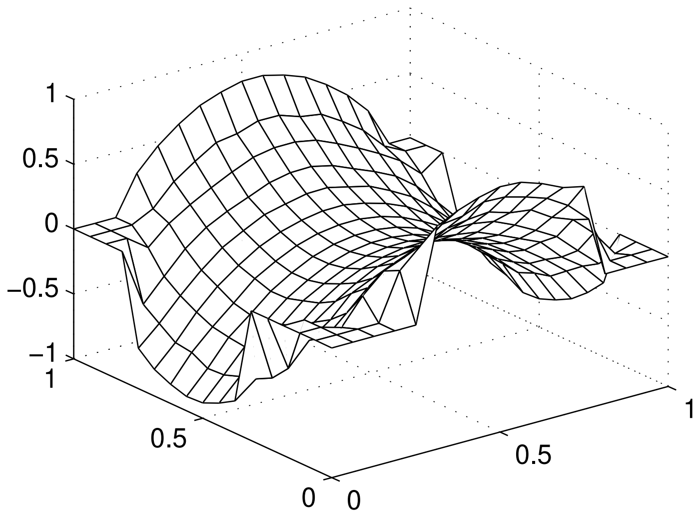


Figure 3-29: An approximate solution of the boundary value problem generated by the semi-local method.

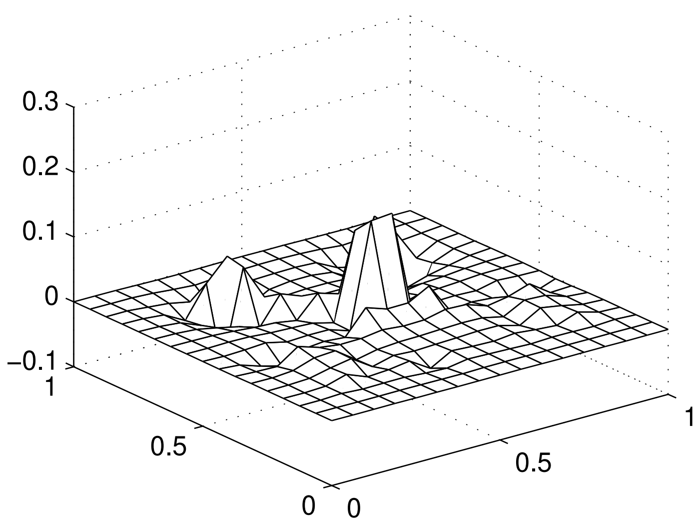


Figure 3-30: The relative error for the solution plotted in Figure 3-29.

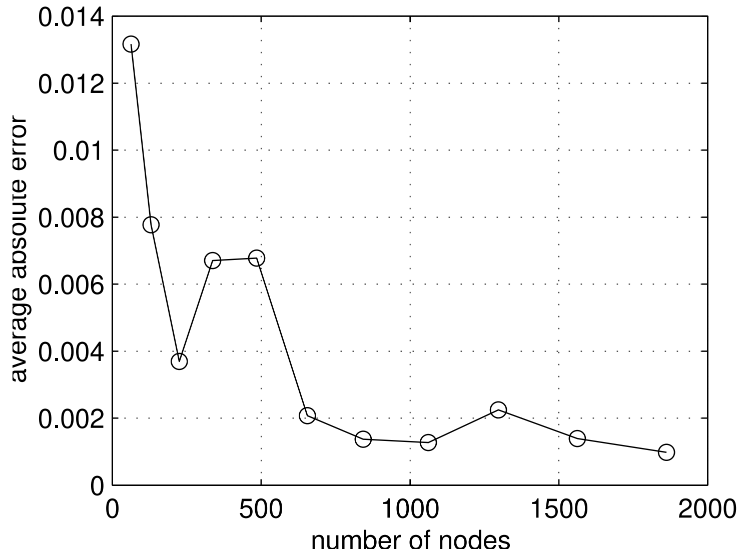


Figure 3-31: The average absolute error versus the number of nodes. The data is generated using the semi-local methods, with the same parameters as earlier experiments.

such principles are unavailable (for example, in many problems involving the dissipation of energy) or where the solutions contain singularities and shocks, finite element methods are not nearly as effective.

For such problems, then, finite difference methods are much more general and are sometimes the only available tools. In those cases, the algorithm outlined in the previous section cannot be expected to work, and we must resort to more sophisticated methods, like that of Chesshire and Henshaw.

3.5 Some comments on mesh generation

This section contains a few brief comments regarding the difficulty of triangulating manifolds, and hence using global discretization methods, for integrating PDEs on manifolds. In particular, a standard theorem of differential topology states that every manifold can be covered by a mesh of "triangular" elements.[36] More precisely, Munkres [18] presents a proof that every manifold has the structure of a simplicial complex. The proof is constructive and works by first triangulating each chart locally (in Euclidean space), and then refining the triangulations on overlapped regions between charts so that they can be "pasted together." While this construction is very suggestive from a computational viewpoint, there is a catch:

[36]In three dimensions, triangles become tetrahedrons, and in even higher dimensions they are called *simplices*. A space that is formed by "pasting" together simplices is known as a *simplicial complex*.

The proof requires the computation of the interesections between a large number of simplicies. While this mostly involves only linear equations, and is in principle computable, in practice this can be extremely expensive in terms of computational resources. Thus, the mathematical proof does not actually supply a solution to the computational problem of triangulating a manifold.

In fact, the merging of local meshes into global ones is the main bottleneck of the entire process. As shown by the quickhull algorithm [6], one can always efficiently triangulate convex subsets of Euclidean spaces. Thus, the only major problem is the merging of local meshes into global ones.

One possible solution is to use abstractions other than manifolds to describe spaces with complex geometries. For example, instead of building local coordinate systems that overlap arbitrarily, one could imagine building complex spaces by deforming and "pasting" lines and squares and cubes and other such topological objects. One can indeed build a large class of spaces this way (in theory), and such spaces are called *CW complexes*. Differentiable manifolds are all examples of CW complexes, so in principle one could use this abstraction to do local triangulation and, because the pieces fit along the boundary *exactly* (instead of in some hard-to-determine overlap), one could merge the meshes more easily.

One important thing to note is that, in the end, a decision on how spaces are constructed should be driven by actual applications because it is almost impossible to arrive at a general computational framework for any class of numerical problems without a context. For example, even though many computational geometry algorithms are restricted to low-dimensions (2 or 3), for most structural engineering problems this is sufficient to generate reasonable models. Furthermore, in fluid problems, the spatial dimension is often low, and while the geometry of the domain is a significant part of the difficulty of simulating fluid flows, it is not the only difficulty. The abstract manifold approach developed in this report are probably most suited to solving problems from mathematical physics, where abstract mathematical spaces are perhaps more commonly encountered.

3.6 Directions for future work

There are a number of alternatives that may help surmount the difficulties described in earlier sections.

3.6.1 Improvements to finite differences

There are a few directions in which finite difference methods may be improved. One is to develop better algorithms for solving large sparse systems of linear equations, so that the unstable coefficients generated by finite difference techniques using irregular sample points would become solvable.

A distinctly different approach would be to simply do finite differences on regular grids, and to basically follow the Chesshire-Henshaw idea. While their idea works well for some special problems, however, there are cases when their idea produces less reasonable answers. For a discussion of this, see [23].

3.6.2 Improvements to finite elements

To improve the performance of finite element methods on manifolds, on the other hand, probably requires more work. While FEMs work admirably well with irregular sampling geometry, the complexity of the geometric problem of combining local equations into a global system can be rather daunting, as was shown in this section. Clearly, much more work needs to be done in this domain, and there are many variations on these ideas. Part of the difficulty of this problem is that, in view of the variational formulation of Laplace's equation, the problem of combining local equations is that of a *constrained minimization problem*, which are often non-trivial. On the other hand, perhaps a standard technique like Lagrange multipliers would work nicely for this case. There are many other things to try.

On the other hand, one of the difficulties that arises with the semi-local method is that it gives charts little control over the geometry of their local meshes because nodes are copied between charts. Thus, while the method produces reasonably good results and has nice convergence properties, it does accumulate quite a bit of truncation error due to geometric defects. It would be very useful to generalize the idea in a way that still allows regular local grids, so as to minimize the effects of geometry on accuracy.

3.6.3 Other methods

Finally, there could be breakthroughs in mesh generation on arbitrary n-manifolds. Although most current work have focused on low-dimensional problems because of their potential applications in engineering and computer graphics, this is a rather active research area and much is being discovered. A global finite element method should work rather nicely on a manifold.

Or one could exploit the *meshless methods* developed by Duarte and Oden [11], which explicitly build partitions of unity using discrete sample points without first generating a mesh. This has the advantage that one does not need to think about combining meshes to use these methods on manifolds. Furthermore, their method can utilize essentially Rayleigh-Ritz or Galerkin approximations, so that the resulting linear equations are solvable by iterative methods.

Chapter 4

Hyperbolic equations

This topic of this chapter is the numerical solution of partial differential equations that describe how certain physical systems evolve in time. Again, as in the solution of elliptic boundary value problems on manifolds, it is possible to break this problem into two components: First, we must have a way of *locally* integrating the PDE; and second, the local solutions must be combined to form a global solution. It is also possible, of course, to discretize the entire manifold first before solving the equations, but it will turn out that the difficulties one must overcome in global methods are not all that different from those of local methods. Because of the nontrivial nature of solving such equations even in the case where the domain has trivial geometry, this chapter focuses on the local problem.

Standard PDE solvers generally perform finite element or finite difference approximations *in space* first, so as to compute the time derivative, and then step forward *uniformly in time* at regular intervals—As one would with ordinary differential equations.[1] While this approach works well enough for many problems, it is rather unsatisfactory philosophically: We have good reason to believe that physical reality does not distinguish among time-like directions, and that any time axis is just as fundamental and just as arbitrary as any other. Thus, a coordinate-independent description of fundamental physical processes and the equations that govern them should not depend on the existence of a unique time axis. More pragmatically, there exist physical problems for which it is helpful to use different frames of reference, and a properly coordinate-independent formulation of PDEs should not be restricted to advancing along an arbitrarily chosen time axis. The use of regular time steps implicitly gives the time coordinate a special status, which complicates any attempt at coordinate-independent representations and solutions.

[1] A notable exception occurs in numerical general relativity, where the use of Regge calculus suggests some interesting ideas for the work at hand. Einstein's field equations are very much beyond the scope of this report, though, and will not be discussed here. For more information on Regge calculus, see Sorkin [26]. For a good introduction to general relativity, see Schutz [25].

One natural solution to this dilemma is the following: Instead of discretizing the spatial dimensions and stepping forward in time, one simply discretizes the equation over *spacetime*[2] and solve for the unknown solution over the entire spacetime region of interest in one step. One might expect, for example, that standard finite element techniques may be applied directly to the entire spacetime domain, and that the unknown solution can be solved over all spacetime events by solving *one* very large system of algebraic equations.

Perhaps not too surprisingly, this simple idea does *not* work, even though there are no obvious problems in the derivation. One reason for this failure is proposed in the next section, and, in view of this proposal, various ways for improving the accuracy are suggested in §4.3. §4.4 discusses some of the difficulties that arise in these improved methods, and also presents some problems that spacetime methods must, in general, overcome. Finally, possible directions for future research in this area are suggested in §4.5.

This chapter is more about open questions than solutions to well-posed problems, and as such may be seem less coherent than earlier chapters. However, it is hoped that the questions asked here will lead to other questions whose answers will some day shed light on the mathematical, physical, and computational structures involved in understanding partial differential equations. Also, because everything here is performed in subsets of Euclidean space, explicit programs probably do not aid in understanding, and are thus omitted in this chapter.

As in earlier chapters, the focus here will be on the simplest possible example that exhibits interesting behavior, which in this case is the linear wave equation.

4.1 The linear wave equation

While Laplace's equation is arguably one of the most important PDEs, there are other important equations that have fundamentally different behavior. One of these is the *linear wave equation.* This equation describes, for example, the propagation of electromagnetic waves in free space. It is therefore useful to identify one of the variables as time in some frame of reference, and to define $D_t = D_{n+1}$ so that time and space derivatives can be more easily distinguished. The wave equation in $(n+1)$ dimensions (n space dimensions plus time) is then:

$$(D_t^2 - c^2\Delta)u = 0, \tag{4.1}$$

where $c > 0$ is a real constant and $\Delta = \nabla^2 = \sum_{i=1}^{n} D_i^2$ is the Laplacian operator over the space variables. For concreteness, this discussion will be restricted to the case $n = 1$. In

[2] *Spacetime* is simply the set of all spatial positions of our space along with time indices. Points in spacetime are often called *events*, and Figure 4-1 would be an example of a *spacetime diagram.*

this case, the wave equation also describes the behavior of a vibrating string with small oscillations. For convenience, let us define $D_x = D_1$ so that $\Delta = D_x^2$.

In constrast to Laplace's equation, the boundary value problem for the wave equation is *ill-posed.* That is, it does not always have solutions for arbitrary boundary conditions, and even when such solutions exist, they are often not unique. However, in the case when $n = 1$ and Ω is the unit square $\{(x,t) \in R^2 | 0 \leq x \leq 1, 0 \leq t \leq 1\}$, one can specify *initial conditions*

$$u(x,0) = f(x), \quad D_t u(x,0) = g(x), \tag{4.2}$$

$$u(0,t) = h(t), \qquad u(1,t) = k(t), \tag{4.3}$$

for some prescribed functions f, g, h, and k. Then the wave equation does have a unique solution. This is called the *initial value problem.*[3]

It is tempting to apply the finite element method directly to the initial value problem for the wave equation. In particular, Galerkin's method may seem generally applicable. However, there is good evidence that Galerkin's method, as presented in Appendix A, will almost always do poorly for the linear wave equation. This does *not*, of course, imply that finite element methods cannot be somehow adapted for the wave equation. First, though, let us take a closer look at why boundary value problems are ill-posed for the linear wave equation.

4.2 Initial value problems and characteristics

As stated in in the previous section, boundary value problems are ill-posed for the wave equation. The root of this problem is the existence of "characteristic manifolds," which describe the "propagation" of initial data. In this section, these notions will be examined a little more closely. However, a close analysis of the ill-posedness of the boundary value problem for the wave equation in terms of these concepts can be fairly complicated and involves many technical details.[4] Thus, this discussion will instead focus on a simpler example, from which we can derive some informal observations on the wave equation.

[3] Technically, this is known as a *mixed initial-boundary value problem* because it contains both initial data in time (the top two equations) and boundary data in space (the bottom two).

[4] Specifically, this problem is ill-posed in that there is no generally applicable existence *and* uniqueness theorem for such problems. On the other hand, for *special cases* of the wave equation over rectangular regions, there are existence and uniqueness results for the boundary value problem. See Fox and Pucci [13] and Payne [22].

4.2.1 Characteristic curves for a first-order equation

There are many equations for which the boundary value problem is ill-posed. Among these are *hyperbolic equations*, for which initial value problems are well-posed.[5] This is because of the existence of *characteristics*, along which one cannot specify arbitrary values of the solution and its derivatives of order less than m (where m is the order of the equation). Equivalently, characteristics *propagate* data about values of the solution and its lower-order partials, because the interdependence of the solution and its lower-order derivatives leads to equations that determine the evolution of the solution along characteristics.

To illustrate, consider the first-order linear equation

$$(D_t + cD_x)u = 0, \tag{4.4}$$

with constant $c > 0$. Defining the coordinate transformation f with inverse g by

$$f_\xi(x,t) = x - ct, \quad f_\tau(x,t) = t, \tag{4.5}$$

$$g_x(\xi,\tau) = \xi + c\tau, \quad g_t(\xi,\tau) = \tau, \tag{4.6}$$

we obtain the new equation

$$\begin{aligned}(D_t + cD_x)u &= (D_\tau v \circ f)D_t f_\tau + (D_\xi v \circ f)D_t f_\xi) + \\ &\quad c[(D_\tau v \circ f)D_x f_\tau + (D_\xi v \circ f)D_x f_\xi] &(4.7)\\ &= D_\tau v \circ f - cD_\xi v \circ f + cD_\xi v \circ f &(4.8)\\ &= D_\tau v \circ f &(4.9)\\ &= 0, &(4.10)\end{aligned}$$

where $v(\xi,\tau) = u(g_x(\xi,\tau), g_t(\xi,\tau))$. Thus, under this coordinate transformation, the equation becomes $D_\tau v = 0$, so that v is constant in τ and depends only on ξ. Thus, $v(\xi,\tau) = F(\xi)$ for some function F. Changing back to the old coordinates, this implies that a solution $u(x,t)$ of Equation (4.4) must take the form

$$u(x,t) = F(x - ct). \tag{4.11}$$

Equivalently:

$$u(x,t) = u(x - ct, 0). \tag{4.12}$$

[5] There exist equations, such as the diffusion equation $(D_t - k\Delta)u = 0$, where neither initial value nor boundary value problems are well-posed.

Conversely, any differentiable function in the form (4.11) satisfies the original equation. So the solution is completely determined by its values along the line $t = 0$. The initial values $u(x, 0)$ are thus "propagated" along the lines $x = ct$, which are called *characteristic curves* (or simply *characteristics*). As shown above, one cannot specify arbitrary values at two distinct points along the same characteristic. Thus, the boundary value problem for the first-order linear equation (4.4) is, in general, ill-posed: Every characteristic intersects the boundary of any bounded spacetime region at least twice,[6] and admissible boundary data are thus severely constrained.

4.2.2 Characteristics for general equations

Now consider an mth-order partial differential equation over an $(n+1)$-dimensional domain Ω. Let S be an n-dimensional subspace of Ω. In general, one can prescribe values for the derivatives of order less than m on S, subject to some *compatibility conditions*—Partial derivatives of orders less than m in directions tangent to S must satisfy the chain rule.[7] These compatibility conditions, together with the differential equation, usually produce enough equations to determine all derivatives $D^\alpha u$ of u with $|\alpha| \leq m$, including the normal derivatives with respect to S up to order m. If this is true everywhere on S, then S is said to be *non-characteristic*. If the equations are singular everywhere on S, then S is *characteristic*.

Intuitively, information on a characteristic subspace S does not determine how the solution evolves outside of S. Since the coefficients of *linear equations*[8] formed by the compatibility conditions and the differential equation consist of combinations of the unknown solution and their lower derivatives, the singularity of such a system of equations on a characteristic manifold implies that the quantities are not independent of each other. These constraints in turn determine derivatives *tangential* to the characteristic in terms of lower-order normal derivatives and solution values, so that such data can be propagated along

[6] A "bounded spacetime region" is a subset of the spacetime domain that is bounded *in spacetime*, not just bounded in space.

[7] *Normal derivatives* of order less than m can be specified arbitrarily. For a more coherent and less vague exposition of this material, see John [16].

[8] The general nonlinear partial differential equation can be transformed into a *quasilinear equation* by differentiating with respect to its highest-order derivative. A quasilinear equation is one that is linear in the highest-order derivatives, but the coefficients may depend on the unknown solution and its lower derivatives. Since the order of the equation is increased by this transformation, additional constraints can and must be derived from the original data and appended to the new data. However, this allows us to define characteristic surfaces for all equations.

This also shows why nonlinear equations are complicated: The characteristics of linear equations depend only on the coefficients themselves, and thus are almost always well-defined. However, for nonlinear (quasi-linear) equations, since the coefficients themselves can depend on the unknown solution and its derivatives, the characteristic manifolds (and hence the directions of information propagation) depend on the particular solution, thus complicating the problem tremendously.

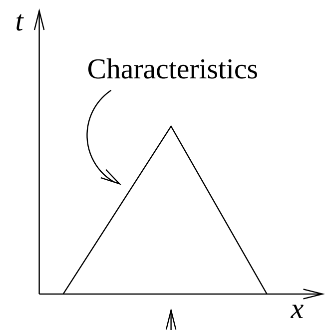


Figure 4-1: Characteristic lines of the wave equation. The interval on the initial line $\{t = 0\}$ bounded by characteristics is called the *domain of dependence* of the solution u at the given point: The value of the solution at the "tip" of the triangular region bounded by the characteristics (called an *inverted light cone*) can only depend on data in the domain of dependence; nothing outside the interval can affect the solution at that point.

the characteristic via another differential equation.

For mth-order quasilinear partial differential equations $L[u] = b$, one can derive an algebraic criterion for characteristics (only the result is stated here): Let $L = \sum_{|\alpha| \leq m} A_\alpha D^\alpha$, where the A_α are functions of spacetime events, values of the unknown solution, and its derivatives of order strictly less than m. Then S is *characteristic* if and only if for every point p on S and non-zero vector v normal to S at p, the equation $\sum_{|\alpha|=m} A_\alpha v_p^\alpha = 0$ holds. For example, in the case of the linear wave equation, $v_t^2 - c^2 v_x^2 = 0$ must hold, so if a vector $v = (v_x, v_t)$ is normal to characteristics, then it satisfies $v_t = cv_x$ or $v_t = -cv_x$. Thus, the characteristics for the linear wave equation are the lines $x = ct$ and $x = -ct$ (see Figure 4-1). Because solution values along characteristics cannot be completely independent, we see that the boundary value problem for the wave equation cannot be well-posed in the strictest sense.

4.2.3 Variational principles revisited

We will now examine variational principles more closely, and to develop some tools useful for analyzing the application of finite elements to the linear wave equation. It may be helpful for the reader to review the material in Appendix A first, particularly the derivation of the Rayleigh-Ritz method and its relation to Galerkin's method.

First, we need to derive a necessary condition for a function to minimize an action. Let $L : R^5 \to R$ be a differentiable function, which is called the *Lagrangian density*, and for any real-valued function u on Ω let γ_u be the function defined by

$$\gamma_u(x, y) = (u(x, y), D_1 u(x, y), D_2 u(x, y), x, y). \tag{4.13}$$

Once again, define the *action* by

$$S(u) = \int_\Omega L \circ \gamma_u, \tag{4.14}$$

and note that if L is defined by

$$L(u, v, w, x, y) = \frac{1}{2}(v^2 + w^2), \tag{4.15}$$

then the action S above becomes the action defined in Equation (A.21).

At this point, it is important to note that in what follows, it will be necessary to differentiate both the function L, which has a 5-dimensional domain, and u, which has a 2-dimensional domain. To avoid confusion, in the rest of this section, differential operators on functions over R^5 will be written as ∂_i instead of D_i; operators on functions over the 2-dimensional domain Ω will continue to be denoted by D_i.

To determine a necessary condition for action-minimizing functions, it is helpful to generalize the idea of *directional derivatives.* Let h be any real-valued function that vanishes on the boundary $\partial\Omega$ of Ω. Consider the real-valued function of a real variable,

$$V_h(s) = S(u + sh) = \int_\Omega L \circ \gamma_{u+sh}. \tag{4.16}$$

As in the case when u and h belong to a finite-dimensional vector space, $V_h(s)$ computes the function S along the one-dimensional subspace spanned by h. Hence, $DV_h(0)$ is the directional derivative of S in the direction of h at u. If u is indeed a minimum of S, then it follows that $DV_h(0) = 0$ for all "directions" h. Differentiating V_h under the integral sign yields

$$DV_h(0) = \int_\Omega (\partial_1 L \circ \gamma_u) \cdot h + (\partial_2 L \circ \gamma_u) \cdot (D_1 h) + (\partial_3 L \circ \gamma_u) \cdot (D_2 h) = 0. \tag{4.17}$$

Integrating by parts and noting that h vanishes on the boundary $\partial\Omega$ gives us:

$$\int_\Omega (\partial_1 L \circ \gamma_u - D_1(\partial_2 L \circ \gamma_u) - D_2(\partial_3 L \circ \gamma_u)) \cdot h = 0. \tag{4.18}$$

This equation holds for all functions h that vanish on the boundary of $\partial\Omega$, so the following equation must hold:

$$\partial_1 L \circ \gamma_u = D_1(\partial_2 L \circ \gamma_u) + D_2(\partial_3 L \circ \gamma_u). \tag{4.19}$$

With L define as in Equation (4.15), this gives us Laplace's equation.[9]

Note that even though the equivalence of the variational principle with Equation (4.19) has historically been called the *principle of* least *action*, the derivation above really finds the *stationary points* of the action functional. Thus, it is more appropriate to call it the *principle of* stationary *action*, though in the case of Laplace's equation it really is a minimum action principle.

4.2.4 Galerkin's method and the initial value problem

Let us now return to the question of applying Galerkin's method to the linear wave equation. The main problem is that the wave equation arises from a variational principle, and that Galerkin's method is equivalent to the Rayleigh-Ritz method. This would not be a problem if one is interested in solving boundary value problems, for then the stationary points of the action functional are solutions of the wave equation. But the boundary value problem for the wave equation is ill-posed, as indicated in §4.2, and in most applications initial value problems are more important. The difference between initial and boundary value problems is that data are specified at different parts of the domain, and in the initial value problem not all of the boundary of the domain has specified values. This geometric difference is where finite element methods break down.

Specifically, let L be defined by

$$L(u, v, w, x, t) = \frac{1}{2}(w^2 - c^2 v^2). \tag{4.20}$$

Using Equation (4.19), this generates the wave equation (4.1). But recall now that in the derivation of Equation (4.19), one of the crucial steps is integrating by parts and using the fact that the perturbation h vanishes on the boundary to get rid of boundary terms. But such perturbations were natural because we were solving boundary value problems. However, if one is interested in the initial value problem, then the appropriate class of perturbations h should vanish on the set $\{x = 0\} \cup \{x = 1\} \cup \{t = 0\}$, and furthermore $D_t h$ should vanish on the initial line $\{t = 0\}$. The function h can now be nonzero along the subset $\{t = 1\}$ of the boundary, and hence integrating by parts would not yield Equation

[9] Equation (4.19) may seem a bit unwieldy in our notation, but consider how one would write this in traditional notation: One is tempted to simply write

$$\frac{\partial L}{\partial u} = \frac{\partial}{\partial x}\left(\frac{\partial L}{\partial u_x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial L}{\partial u_y}\right).$$

But both L and u have x and y as arguments, and the notation $\frac{\partial}{\partial x}$ does not distinguish between them. So this equation is *wrong!* The correct way to write this in traditional notation requires writing out all the arguments, which is an even bigger mess than Equation (4.19).

(4.18). Instead, it gives

$$\int_\Omega (\partial_1 L \circ \gamma_u - D_x(\partial_2 L \circ \gamma_u) - D_t(\partial_3 L \circ \gamma_u))h + \int_{t=1} (\partial_3 L \circ \gamma_u)h = 0. \tag{4.21}$$

Supposing u is continuous and has continuous first derivatives, the boundary term in Equation (4.21) vanishes for all h only if $D_3 L \circ \gamma_u = 0$ for $t = 1$, which in the case of the wave equation means $D_t u(x, 1) = 0$ for $0 \le x \le 1$. This cannot in general be true. Therefore, the boundary term is almost always nonzero, which implies that the integrand in the first term is also nonzero, and thus u cannot satisfy the wave equation.[10]

One can easily show that Galerkin's method for the wave equation is equivalent to finding the stationary points of the approximate action, using an argument almost identical to that of §A.2.4. Thus, in the limit as the finite element approximation becomes more exact, the approximation constructed by Galerkin's method would converge to some stationary point satisfying the initial conditions (if it converges at all). As shown above, this function cannot satisfy the wave equation. In fact, one can derive lower bounds on the error using the variational principle.

We can also strengthen the argument to show that if such a action-minimizing function exists in the case of the initial value problem and has continuous first derivatives, then for every point $p = (x, 1)$ of the line $\{t = 1\}$ such that $D_t u(p) \neq 0$, the residual $D_t^2 u - c^2 D_x^2 u$ is *unbounded* in every neighborhood of p. Thus, u cannot even have continuous second derivatives, and any solution that minimizes the action must contain singularities.

4.3 Variations on a theme of Lagrange

In view of the analysis above, there are a few natural variations on the Rayleigh-Ritz idea that *may* help produce reasonable solutions to the wave equation. In particular, it is possible to eliminate the boundary term from Equation (4.21), so that stationary points of the action functional are indeed solutions of the wave equation. There are a few ways of accomplishing this, and this section proposes two of them.[11]

4.3.1 Modifying the action principle

The first idea is to simply modify the Lagrangian density to change the form of Equation (4.21), so that the boundary integral

$$\int_{t=1} (\partial_3 L \circ \gamma_u)h = 0 \tag{4.22}$$

[10] That is, if such a stationary point u exists at all.

[11] Apologies are due to Professors Guillemin and Sternberg for borrowing the title of their book.

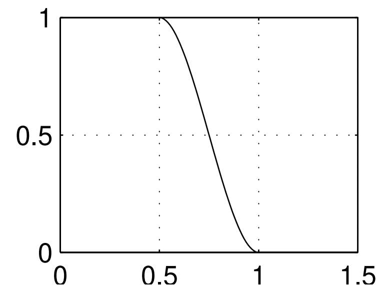


Figure 4-2: A typical cut-off function.

vanishes. This allows the rest of the derivation of Equation (4.19) to be carried through, so that the stationary points of the action do exist and correspond to solutions of the wave equation (or so one would hope).

More specifically, consider the following Lagrangian:

$$L(u, v, w, x, t) = \frac{1}{2}(c^2 v^2 - \rho(t) w^2), \tag{4.23}$$

where ρ is a *cut-off function*, as depicated in Figure 4-2. Cut-off functions provide a nice way to change the behavior of the differential equation in different regions of spacetime. In this particular case, we wish to choose constants t_1 and t_2 such that $\rho(t) = 1$ for all $t \leq t_1$ and $\rho(t) = 0$ for all $t \geq t_2$. For our purposes, set $t_1 = \frac{1}{2}$ and $t_2 = 1$. ρ then vanishes on the final line $\{t = 1\}$.

We can apply Equation (4.19) to the Lagrangian density above, obtaining:

$$\rho(t) D_t^2 u(x, t) + D\rho(t) \cdot D_t u(x, t) - c^2 D_x^2 u(x, t) = 0. \tag{4.24}$$

Thus, for $t < \frac{1}{2}$, the equation is just the linear wave equation. For $\frac{1}{2} < t < 1$, the equation slowly changes until at $t = 1$, it becomes:

$$D_x^2 u(x, t) = 0, \tag{4.25}$$

which is obviously no longer well-posed because it says nothing about the behavior of u over time. Time ceases to have any meaning in this modified system after $t = t_2 = 1$.

The boundary integral term that we wanted to eliminate becomes:

$$\int_{t=1} (\partial_3 L \circ \gamma_u) h = \int_{t=1} \rho(t) D_t u(x, t) dx = 0, \tag{4.26}$$

because ρ was chosen to vanish on the line $\{t = 1\}$.

Note the characteristics are no longer straight lines, and hence the speed of the wave is

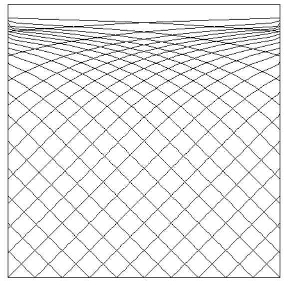


Figure 4-3: The characteristics of this modified wave equation. The top boundary is where "time ends."

also no longer constant (see Figure 4-3). The top boundary, $\{t = 1\}$, is where the meaning of time breaks down. The speed of propagation at time t:

$$\frac{c}{\sqrt{\rho(t)}}, \tag{4.27}$$

Thus, the speed of the wave approaches infinity as time approaches t_2.

Causality

Note that there is something suspicious about this method. After all, we are hoping to obtain, via this trick, accurate solutions of the wave equation in the spacetime region $\{t < 1/2\}$ by modifying the equation in the region $\{t > 1/2\}$. How can changes in the *future* affect the accuracy of solution in the *past*? Has some notion of causality been violated? Indeed, even though this trick does not provide accurate numerical solutions, it does generate symmetric systems of linear algebraic equations, which implies that unknown data from the future does somehow affect the past.

The "solution" to this apparent paradox is that the finite element method really has no built-in directionality. Thus, the Rayleigh-Ritz equations do not enforce any *causal* structure in spacetime, but instead only gives *correlations* between sample points. In a very informal sense, this can actually be advantageous: By correlating predications made from past data with constraints imposed in the future, one might even hope to improve the solution over the entire spacetime region of interest.

The Lorentz metric

The usual Lagrangian for the wave operator can be expressed in terms of the *Lorentz metric*:[12]

$$L(u(x,t), D_x u(x,t), D_t u(x,t), x, t) = \frac{1}{2}[c^2(D_x u(x,t))^2 - (D_t u(x,t))^2] \quad (4.28)$$

$$= \frac{1}{2} g^*(du_{(x,t)}, du_{(x,t)}), \quad (4.29)$$

where g is the metric tensor, g^* its dual metric on the dual space, and $du_{(x,t)}$ denotes, as in Chapter 2, the *differential* of u at (x,t).

This has several consequences. First, it gives us a coordinate-independent way of describing the wave equation on arbitrary manifolds equipped with a Lorentz metric: Because metric tensors and differentials are already coordinate-free objects on manifolds, Equation (4.28) gives a coordinate-free way of describing the Lagrangian. Now, the variational principle itself can also be stated in a coordinate-free way, since integration of scalar functions can also be defined with respect to a Lorentz metric, as was done for Riemannian metrics in §3.4.1.[13] So using this Lagrangian and Equation (4.19) gives us a consistent way of generalizing the wave equation to Lorentz manifolds.[14] In the usual case of Euclidean spacetime with the flat metric, this gives us the usual wave equation.

Furthermore, this description also tells us what we are *really* doing when we put the time-dependent factor ρ into the Lagrangian density: The metric itself is being made time-dependent! Thus, spacetime is no longer flat, and Equation (4.27) shows that the "speed of light" becomes infinite in a finite amount of time in this coordinate system (see Figure 4-3. This may seem problematic from a physical point of view, and it is. It introduces *curvature* into spacetime and may even violate some conservation laws due to the coarseness of the discretization. The numerical results of the next section show that this method does not work very well.

---

[12] This section supposes some familiarity with relativistic concepts.

[13] Symmetric nondegenerate tensor fields, such as Lorentz metrics, are known as *pseudo-Riemannian metrics.* Because they have orthogonal eigenvectors, the basic argument that defined integration on Riemannian manifolds also works on any *pseudo-Riemannian manifold:* The key result is the fact that with respect to a Lorentz metric, we can define *orthonormal bases*, which are orthogonal basis vectors with magnitude ± 1. Then the matrix representatio of bases are also orthogonal matrices, and their determinants are ± 1. Taking absolute values defines local integrals consistently.

[14] Compactness is required for computing the action, but not for computing Equation (4.19) in local coordinates.

Number of nodes	Absolute error			Relative error	
	Maximum	Minimum	Average	Maximum	Minimum
14	3.46339	0.00629682	1.36863	5.65633e+16	-4.27969e+15
27	17.2273	0.577482	7.74683	55.7488	-45.7939
44	7.10821	0.365194	3.23993	1.98809e+15	-9.46197e+16
65	4.34967	0.211686	2.05495	11.2617	-14.2968
90	2.5196	0.0978345	1.06008	1.5842e+15	-3.17402e+16
119	2.05826	0.034336	0.820245	9.95072	-10.4461
152	3.05573	0.0313615	1.13394	1.45998e+16	-2.40796e+16
189	8.72154	0.157251	3.54988	31.5282	-44.0292
230	3.31555	0.0326151	1.20425	4.68941e+15	-2.98963e+16
275	2.48151	0.0037325	0.881522	11.2937	-11.9207
324	2.22377	0.000174037	0.753538	5.73477e+14	-2.11547e+16
377	2.01804	0.00592493	0.675755	8.69369	-12.2134
434	1.99763	0.000252334	0.634148	3.13461e+14	-1.5434e+16

Table 4.1: Statics of the results generated by modifying the wave equation.

Numerical reults

In order to perform actual numerical experiments, it is necessary to choose a specific cut-off function. The actual ρ used is:

$$\rho(t) = \rho_0 \left(\frac{t - t_1}{t_2 - t_1} \right), \tag{4.30}$$

where ρ_0 is defined by:

$$\rho_0(t) = \begin{cases} 1, & t < t_1, \\ 2t^3 - 3t^2 + 1, & t_1 \leq t \leq t_2, \\ 0, & t_2 < t. \end{cases} \tag{4.31}$$

The function ρ has the following properties (see Figure 4-2):

$$\begin{aligned} \rho(t_1) &= 1, & \rho(t_2) &= 0, \\ D\rho(t_1) &= 0, & D\rho(t_2) &= 0, \end{aligned} \tag{4.32}$$

so that it provides a fairly smooth transition between the linear wave equation (in the range $t < t_1$) to the degenerate equation (4.25) (in the range $t \geq t_2$).

Table 4.3.1 shows the data from numerical experiments performed using this method. It is unclear why the relative error jumps between entries, but it may have to do with accidental geometric configurations (i.e. the placement of nodes in the charts and how they overlap), since these jumps also exist in Table 4.3.2. The statistics are only collected over

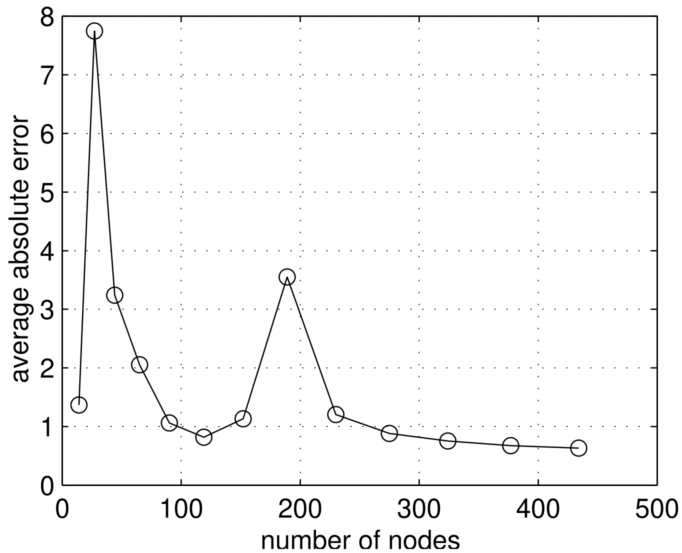


Figure 4-4: Average absolute error versus number of nodes.

those nodes for which $t < 1/2$, i.e. in the region where the modified equation agrees with the wave equation. The true solution, in this case, is:

$$u(x,t) = \cos(2\pi(x - t)), \tag{4.33}$$

where c was set to 1 for convenience. The discretized equations are solved directly using LU decomposition with partial pivoting.

Figure 4-4 plots some of the results of Table 4.3.1. Clearly, this method does not work very well, although it does appear to slowly converge to the true solution.

Figure 4-5 plots the true solution of the wave equation over this square domain, while Figure 4-6 plots the solution generated by this method. As one can see, this method produces solutions that are only vaguely similar to the true solution in a qualitative sense. Figure 4-7 shows the absolute error distribution, which is sufficiently structured to lead one to suspect the existence of deeper causes of error and possible ways of improving the performance of this method. However, what those causes should be is not entirely clear.[15]

A discussion of possible reasons for the poor performance of this method is postponed until §4.4.1. First, let us take another look at a different approach to eliminating the troublesome boundary term in Equation (4.21).

[15]Noting the jumps in relative error in alternating entries of Table 4.3.1 and its similarity to Table 4.3.2, the problem does seem to be related to the *parity* of the mesh used.

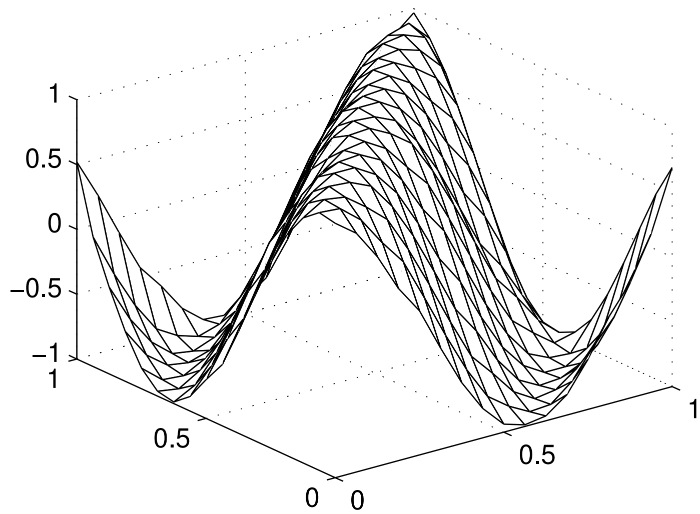


Figure 4-5: The “true” solution to the wave equation given in Equation (4.33).

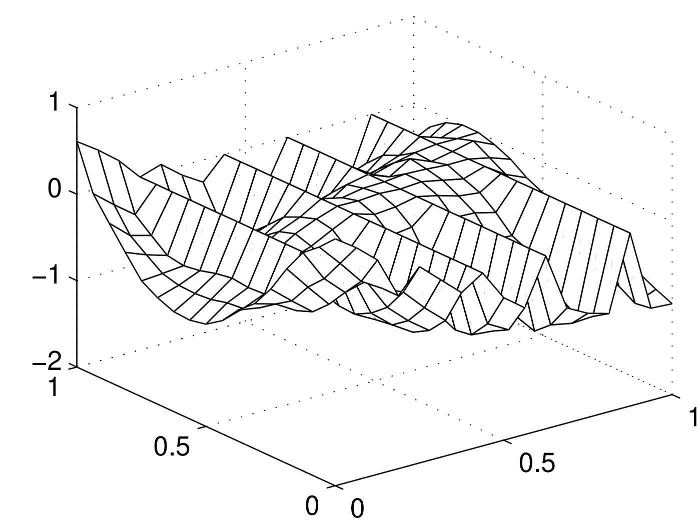


Figure 4-6: The approximate solution generated by this method.

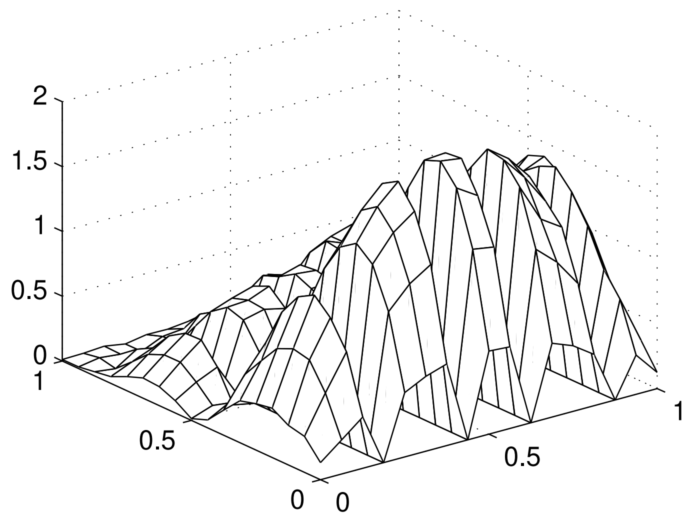


Figure 4-7: The absolute error. Note that this error is very structured, and hence hints at a deeper cause.

4.3.2 Modifying the domain

The second idea depends on modifying the geometry of the domain so that the "final line" $\{t = 1\}$ does not exist at all (see Figure 4-8). More specifically, we extend and modify the geometry of the domain by "attaching" a triangle to the original spacetime domain. It is important to ensure that the triangular part of the domain has sides whose slopes are greate than $1/c$; this makes sure that the boundaries remain *timelike*, so that boundary conditions can be imposed without making the problem ill-posed. With respect to Equation (4.21), this means the boundary term would no longer exist because boundary data would

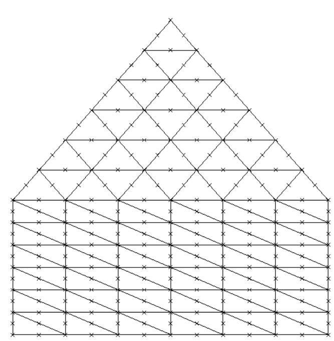


Figure 4-8: Attempting to eliminate the boundary term in equation (4.21) by changing the shape of the domain.

be imposed over the entire boundary.

Geometry and metrics

At first glance, this method and that of the previous section may seem very different: One modifies the wave equation but does not modify spacetime itself, while the other changes the *shape* of the domain without modifying the equation. However, the two are really more similar than they seem.

Let X_1 be the rectangular spacetime of Figure 4-3, and let X_2 denote the "house"-shaped spacetime of Figure 4-8. Consider the comments of §4.3.1: The wave equation really arises from the metric of spacetime, and the method of the previous section works by introducing curvature into spacetime. On the other hand, the "geometric method" of this section seems to have deformed the space without modifying the metric. But X_1 and X_2 are *topologically equivalent*—i.e. One can be continuously mapped onto the other bijectively. Thus, we can always map the oddly-shaped X_2 onto X_1 via a continuous transformation ϕ.

Now, this mapping has an inverse $\phi^{-1} : X_1 \to X_2$ that is also smooth. By using its differential[16], we can "pull back" the flat metric from X_2 onto the space X_1:

$$g_p^1(v,w) = g_q^2(d\phi_q^{-1}(v), d\phi_q^{-1}(w)), \tag{4.34}$$

where g^i is the metric of X_i, $p \in X_1$, and $q = \phi^{-1}(p) \in X_2$. The "pulled-back" metric g^1 then induces a dual metric $(g^1)^*$, which can be used to produce the modified wave equation on X_1 that is equivalent to the "flat" wave equation on X_2, in the sense that:

$$u_1(\phi(x,t)) = u_2(x,t), \tag{4.35}$$

where u_i is the solution of the wave equation associated with the Lorentz metric g^i on the space X_i.

While X_2 has a flat metric g^2, the metric g^1 induced by ϕ on X_1 is in general not flat, because the transformation ϕ is generally nonlinear. Thus, we see that this new method really can be thought of as just another way to modify the metric of spacetime. The modification, of course, differs from that of the previous section, and generates much more complicated characteristic curves.

One important thing to note is that one can only go so far in modifying the geometry of a space by changing its metric—The *topology* of the manifold will always stay invariant if the metric is smooth everywhere, even though the *geometry* changes. In order to generalize

[16]It should be clear the we can choose ϕ so that it is continuous *almost everywhere*, except at the corners on the boundary of X_2. Similar comments apply to ϕ^{-1}.

Number of nodes	Absolute error			Relative error	
	Maximum	Minimum	Average	Maximum	Minimum
6	14.2485	1.74795	9.29633	1.35509e+17	-14.2485
20	3.89164	0.138369	1.95244	7.78328	-4.68721
42	0.627265	0.00518237	0.247108	1.02444e+16	-6.52581e+15
72	3.24599	0.00177442	1.1747	10.5042	-5.74983
110	0.289144	0.00082902	0.0979768	2.97096e+15	-7.02175e+14
156	0.318759	6.53735e-05	0.12427	1.18395	-1.43249
210	0.194245	0.000291551	0.084536	3.06143e+15	-4.19103e+14
272	0.0736915	0.000189996	0.0240319	0.286878	-0.300047
342	0.0496121	3.03278e-05	0.0153055	2.52581e+14	-1.80042e+14
420	0.149277	1.30728e-05	0.0369261	1.04892	-0.677157
506	0.0927114	0.000108788	0.0293263	5.23132e+14	-1.48843e+15
600	0.812075	0.000495892	0.320429	3.90818	-6.36069
702	0.0492277	5.3695e-07	0.0108849	1.4358e+14	-2.39808e+13

Table 4.2: Statics of the results generated by modifying the spacetime domain.

this particular idea of deforming the spacetime domain to equations on more complicated manifolds, it may be necessary to apply topological transformations as well, so that this method would no longer be simply a variant of the algorithm presented in the previous section.

Numerical results

Table 4.3.2 shows the data collected using this method. The first few entries were obtained using LU decomposition, but such direct methods fail for larger systems of equations, so relaxation had to be used. Since the wave operator does not produce symmetric positive-definite matrices (as does the Laplacian), it is necessary to generate the normal equations by multiplying the matrix with its own transpose. Thus, the accuracy of the solution obtained by relaxation is rather limited (see §3.3.2). However, despite these difficulties, this method clearly outperforms our previous attempt.

Figure 4-9 plots the average absolute error against the number of nodes. As one can see, this method works much better, although it still leaves much room for improvement.

Figure 4-10 shows the approximate solution generated this way, and Figure 4-11 shows the absolute error between this solution and the solution shown in Figure 4-5. Note that the solution in Figure 4-10 is at least qualitatively reminiscent of Figure 4-5.

For this particular method, there is one more parameter we can control: The only constraint on the slope of the triangular "extension" to our domain is that its sides have slope greater than $1/c$. Thus, the slope of the sides can be varied, which affects the accuracy

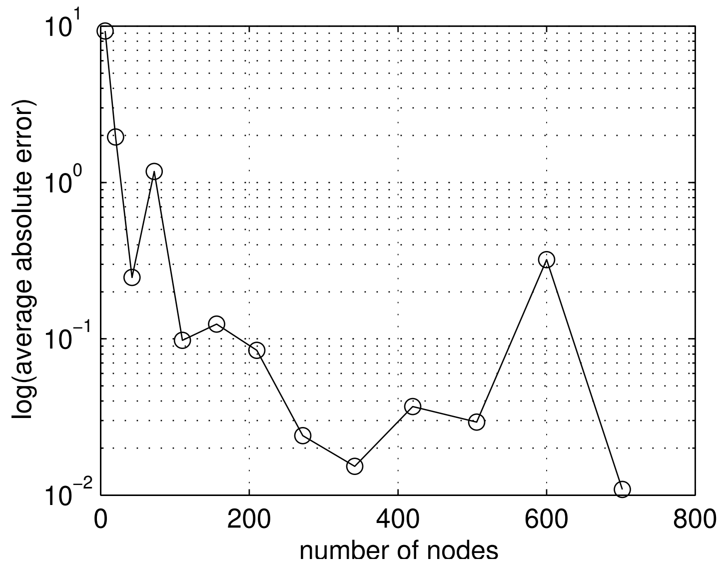


Figure 4-9: Average absolute error versus number of nodes. Results are generated by modifying the domain of solution.

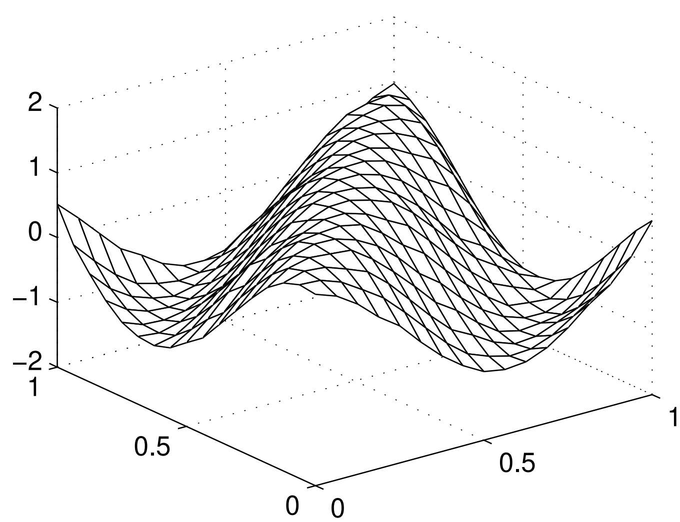


Figure 4-10: Approximate solution generated by extending and changing the shape of the domain.

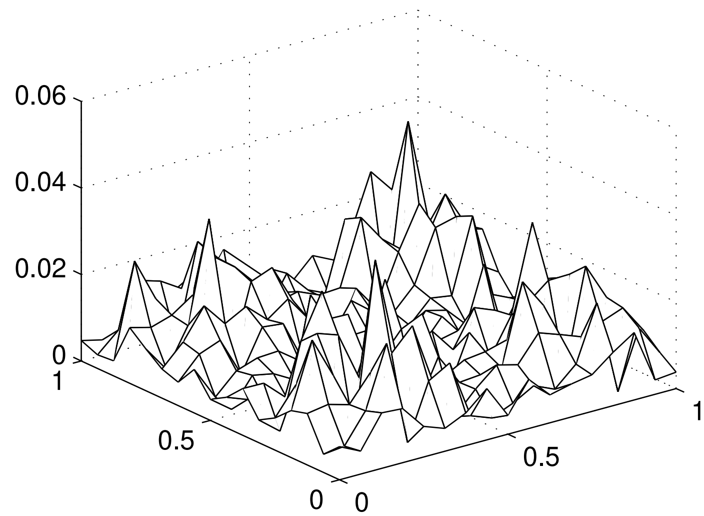


Figure 4-11: The absolute error between Figure 4-5 and 4-10.

Boundary slope	Absolute error			Relative error	
	Maximum	Minimum	Average	Maximum	Minimum
2.0	0.0492277	5.3695e-07	0.0108849	1.4358e+14	-2.39808e+13
2.1	0.021287	8.38556e-06	0.00718122	1.62293e+14	-1.168e+14
2.2	0.104445	0.000264129	0.0333464	8.29833e+14	-7.40989e+14
2.3	0.204355	0.000355765	0.0454508	1.09504e+15	-8.71415e+14
2.4	0.0987477	6.25076e-05	0.0366781	1.18056e+15	-7.67629e+13
2.5	0.0390811	3.05883e-06	0.00954907	2.82624e+14	-2.05056e+14
2.6	0.0415744	1.61621e-05	0.0103743	4.5445e+13	-2.62967e+14
2.7	0.563769	0.000158676	0.22424	5.51249e+15	-3.11361e+14
2.8	0.738567	1.63661e-05	0.252891	7.57622e+15	-1.83677e+15
2.9	1.12121	0.000821135	0.309993	7.87134e+15	-6.65048e+15
3.0	0.858581	6.42513e-05	0.270326	6.98004e+15	-3.78587e+15

Table 4.3: Statistics obtained by varying the size of the triangular region added.

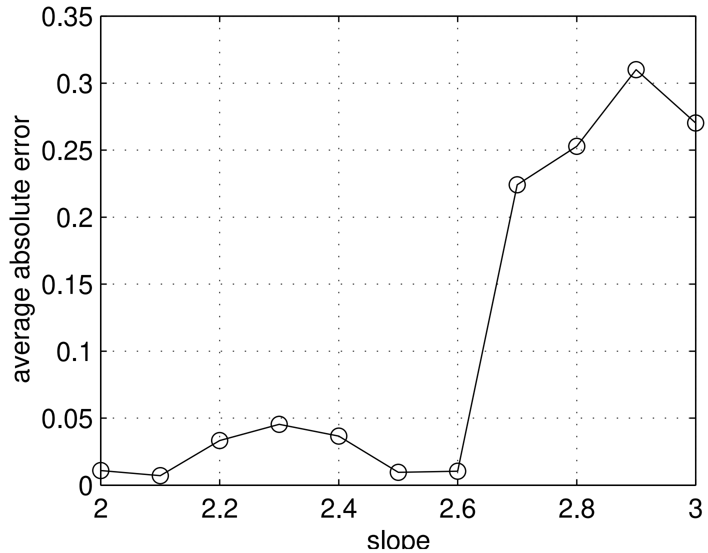


Figure 4-12: The relative error between Figure 4-5 and 4-10.

of solution. Table 4.3.2 shows this data.

Figure 4-12 plots this data. As one can see, there is no clear indication of how one choose the slope to minimize the error.

4.4 Difficulties with the spacetime approach

This section offers some tentative explanations for the failure of the ideas from the previous section. Furthermore, we will discuss some issues faced by spacetime methods in general.

4.4.1 Why the variations failed

It turns out that both of the methods described above probably fail for the same reason: Numerical solvers for the wave equation (and all hyperbolic equations) seem to depend rather sensitively on the geometry of characteristics. In particular, it is often necessary to ensure that information is "propagated" in characteristic direction. Very informally, in terms of finite elements, this means that every node is connected to at least one neighbor in a characteristic direction, so that at least some information is propagated along characteristic lines.

Now, while this condition holds true for both methods over some regions of spacetime, it fails for both methods after some time t_{crit}: For the first method, $t_{crit} = t_1 = 1/2$ because the slope of characteristic curves change after that time but the mesh stays the same. For the second method, $t_{crit} = 1$ because after that, the mesh changes to match the shape of the triangular region. Note that this indicates that we should try to choose the slope of the extended triangular region in the second method to be as close to $1/c$ as possible, and also offers a hint of why the second method performs better than the first.

One might wonder how changes in characteristics or mesh geometry after t_{crit} affects the accuracy of the solution *before* t_{crit}. The answer is that the comments on causality in §4.3.1 apply to both methods: Because the finite element method has no built-in notion of time and provides only *correlations* between past and future events, errors arising from inconsistencies between the mesh and characteristics after t_{crit} naturally affect the accuracy of solution before t_{crit}.

4.4.2 Other problems

Aside from that of accuracy, there are other problems associated with applying spacetime methods to hyperbolic PDEs. One of the most serious is the computational resources required: While standard finite difference methods (or finite element methods with regular time steps) need only keep in memory the data associated with the current time step, plus or minus a few neighboring steps, spacetime methods—by their very nature—require *all* of the data over the spacetime domain. This can be costly in terms of storage requirements if the domain is large. For example, if one needs to understand both the short-term and long-term behavior of solutions, the spacetime region is likely to require a large number of sample points to represent.

Yet another issue is the solution of the discretized equations. Unlike Laplace's equation (or elliptic equations in general), hyperbolic equations almost never generate systems of linear equations that are solvable by relaxation directly. It is for this reason that we were forced to compute the normal equations before applying relaxation to produce Table 4.3.2. While direct methods work fairly well, they are limited by the size of the system one can solve, and in view of the comments above, one can see that spacetime methods can easily generate very large systems of equations.

One last issue is the solution of "true" initial value problems: As stated before, the particular version of the wave equation considered here is a mixed initial-boundary value problem because space, in our case, has finite extent, and both initial values in time and boundary values in space are given. In simulating the propagation of electromagnetic waves in free space, it would be necessary to understand how to simulate large space domains, since finite elements can only work for compact domains.[17]

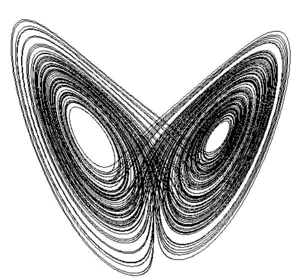
[17] For such problems, it is necessary to consider *absorbing boundary conditions*, which help make space "look" infinite using a finite number of spatial sample points. For more information, see Engquist and Majda [12].

4.5 Directions for future work

Aside from the difficulties mentioned in the previous section, there are other issues of interest here. For one thing, the derivation of Equation (4.19) from the variational principle makes no reference to the initial data $D_t u(x,0) = g(x)$, only the boundary data. Furthermore, the existence and importance of characteristics never arises, even though the variational principle is an *equivalent* way of formulating the wave equation. One natural question, then, is this: Is there a way to analyze the Lagrangian density itself, perhaps as a by-product of the tools used to derive Equation (4.19), that clarifies the importance of characteristics? And why does the initial time derivative not matter in the derivation? What is different between variational principles for PDEs and ODEs, such that the Euler-Lagrange equations hold for ODEs, even though it is initial value problems that are of interest in classical mechanics?

Yet another interesting direction, though only tangentially related to this topic, is that of *information propagation*. This idea has been mentioned informally throughout this chapter; it would be very interesting to formalize it. In particular, can we compute how much information is "propagated along characteristics"? Is there a way to understand the well-posedness of initial value problems for the wave equation, as well as the ill-posedness of boundary value problems, in terms of information propagation? What connections, if any, exist between information propagation and the Lagrangian density? How is information propagation in the PDE itself related to information propagation in the PDE solver, and can we use such ideas to estimate numerical accuracy? Finally, how can we extend these ideas to PDEs that do *not* arise from variational principles?

It is the author's hope to follow up on some of these questions, and that they may lead to a deeper understanding of hyperbolic equations in general (both linear and nonlinear), and the wave equation in particular.



Appendix A

Background Material on Partial Differential Equations

Two general classes of numerical methods for solving partial differential equations are *finite difference methods* and *finite element methods.* While other methods, such as spectral decomposition methods, are very effective in special situations, they do not have the general applicability of finite differences and finite elements.

Finite difference methods are very simple. They depend upon the approximation of derivatives by difference quotients. For example, we know from differential calculus that the *forward difference*

$$\frac{f(x+h)-f(x)}{h} \tag{A.1}$$

approximates the derivative of f at x for sufficiently small h. Finite difference methods are very popular because they are easy to understand and program, and generally run very efficiently on most computers. However, they often depend sensitively upon the particular way in which the domain is discretized, and can easily become numerically unstable. As a result, the literature is full of long, excruciating analyses of convergence criteria and error estimates. The reader will not be subjected to such tortures here.

Instead, this appendix treats finite elements in more depth. This will bring out several important ideas in the theory of partial differential equations along the way.

A.1 Matrix inversion

Before all else, one should know that the numerical solution of partial differential equations generally involves the solution of large systems of linear algebraic equations. Thus, it is useful to first examine some of the more popular methods for solving such systems of equations, and to keep these methods in mind throughout the rest of this appendix and the report itself. The reader is assumed to have some knowledge of elementary linear algebra, including familiarity with *direct methods* such as Gauss-Jordan elimination and LU decomposition (which terminate after a finite number of operations). This section describes some basic *iterative methods.*

A.1.1 Iterative methods and relaxation

The basic problem is this: We wish to solve a system of linear equations:

$$Ax = b, \tag{A.2}$$

where A is an $n \times n$ matrix and x, b are n-vectors, and where n is a large positive integer. For such problems, direct methods such as Gaussian elimination or LU-decomposition require too much space and time to be useful.

One way of computing the solution x is by noting that x is the *fixed point* of the system of finite-difference equations:

$$x_{k+1} = (I - A)x_k + b, k = 0, 1, 2, ... \tag{A.3}$$

Iterating the equation above generates a sequence of vectors $\{x_k, k = 0, 1, 2, ...\}$. If the sequence converges, then one would obtain a solution to the original linear system of equations (A.2). Letting $B = I - A$ in the above equation, it follows by induction that:

$$x_k = B^k x_0 + \sum_{i=0}^{k-1} B^i b, \tag{A.4}$$

where by convention $\sum_{i=0}^{-1} B^i = 0$. B is called the *iteration matrix*, and the sequence $\{x_k\}$ converges to the solution x for all initial conditions x_0 if and only if $\lim_{k \to \infty} B^k = 0$ and the infinite series $\sum_{i=0}^{\infty} B^i$ converges. One could then show that this holds if and only if the *spectral radius* $\rho(B)$ is less than 1.[1]

A.1.2 Jacobi iteration

For general A, the iteration matrix $B = I - A$ often has large eigenvalues, so the iteration would not converge. However, there are some modifications that *do* produce convergent iterations in many instances. These iterative methods, where a difference equation B is obtained from the matrix A and then iterated, are called *relaxation methods*. In the following, let L denote the off-diagonal lower-triangular entries of A, let D denote the diagonal entries of A, and let U denote the off-diagonal upper-triangular entries of A, so that $A = L + D + U$.

The simplest among these methods, called *Jacobi iteration*, simply normalizes each row of the matrix by the diagonal entries, so that instead of $B = I - A$, one has:

$$B = I - D^{-1}A = -D^{-1}(L + U). \tag{A.5}$$

In components, this is equivalent to:

$$x_{k+1}(i) = \frac{-\sum_{j=1, j \neq i}^{n} a_{ij} x_k(j) + \sum_{j=1}^{n} a_{ij} b_j}{a_{ii}}. \tag{A.6}$$

Thus, one could perform the iterations rather efficiently if the matrix is *sparse*; i.e. has a large number of zeros. This method, of course, does not always converge, and Vichnevetsky contains a discussion of such issues [27].

[1] The spectral radius of a matrix A is the maximum among the absolute vaules of the eigenvalues of A. Those familiar with some point set topology should notice that this criterion is equivalent to saying that the function defined by $f(x) = Bx + b$ is a *contraction mapping*.

A.1.3 Gauss-Seidel iteration

A slight variation, called *Gauss-Seidel iteration*, uses:

$$x_{k+1}(i) = \frac{-\sum_{j=1}^{i-1} a_{ij} x_{k+1}(j) - \sum_{j=i+1}^{n} a_{ij} x_k(j) + \sum_{j=1}^{n} a_{ij} b_j}{a_{ii}}. \tag{A.7}$$

That is, instead of updating all components $x_k(i)$ synchronously, the new components are used as soon as they become available. In matrix form, this means:

$$(L + D)x_{k+1} + Ux_k = b, \tag{A.8}$$

or

$$x_{k+1} = -(L + D)^{-1} U x_k + (L + D)^{-1} b. \tag{A.9}$$

This method is somewhat better than the Jacobi method in that it updates the components successively instead of synchronously, so the storage requirements are less stringent and programs are generally more compact and efficient. However, one should be careful in using these methods because their convergence properties are different, although for a large class of problems they both converge.

A.1.4 Overrelaxation

These iterative methods are, in general, relatively slow. In order to speed up the convergence, one often uses *overrelaxation* techniques by taking larger "steps" in each iteration. For Jacobi iteration, this means using:

$$x_{k+1} - x_k = \bar{\omega}((I - D^{-1}(L + U))x_k + D^{-1}b), \tag{A.10}$$

or

$$x_{k+1} = ((1 - \bar{\omega})I - \bar{\omega} D^{-1}(L + U))x_k + \bar{\omega} D^{-1} b. \tag{A.11}$$

The number $\bar{\omega}$ is called the *overrelaxation factor* when $1 < \bar{\omega} < 2$, and called the *underrelaxation factor* when $0 < \bar{\omega} < 1$. One could show that the iteration must necessarily diverge (that is, the spectral radius of the resulting iteration matrix B must be greater than 1) unless $0 < \bar{\omega} < 2$. However, the converse does not hold: $0 < \bar{\omega} < 2$ does not guarantee convergence.

For Gauss-Seidel, a similar derivation yields:

$$x_{k+1} = ((1 - \bar{\omega})I - \bar{\omega}(L + D)^{-1}U)x_k + \bar{\omega}(L + D)^{-1} b. \tag{A.12}$$

This is known as *successive overrelaxation.*

A.2 A brief introduction to finite elements

A.2.1 Introduction

This section briefly summarizes how numerical solutions of partial differential equations can be computed using the finite element method. In particular, it contains a derivation of the standard discretization of Laplace's equation in two dimensions. Most of this material comes from Vichnevetsky [27]; it is an excellent introduction to numerical methods for partial differential equations. Johnson [17] also contains a clear and more detailed account of finite element methods. For an analytical approach, the opening chapters of Fritz John's text [16] offer a good introduction. The classic treatise on partial differential equations is Courant and Hilbert [10], which may be too encyclopedic to serve as an introduction but contains a lot of good stuff. A very brief but clear survey article appears in the McGraw-Hill Encyclopedia of Science & Technology [5].

Notational and mathematical conventions

This section will not rigorously define such terms as *open set*, *closed set*, and *boundary*, since these topological concepts should be fairly intuitive in this setting. It will only define some notations and terms not commonly covered in introductory calculus courses.

The boundary of a region Ω is denoted by $\partial\Omega$, and its *closure* $\overline{\Omega}$ is defined as the union of Ω and its boundary. Given a real-valued function f over Ω, its *support* is defined as the closure of the subset of points over which f is nonzero, i.e. the set $\overline{\{x \in \Omega | f(x) \neq 0\}}$.

For the sake of precision (which is important for turning ideas into programs), functional notation will be used wherever appropriate. Thus, the integral of a real-valued function f over open set Ω is

$$\int_{\Omega} f, \tag{A.13}$$

instead of

$$\int_{\Omega} f(x,y)dxdy. \tag{A.14}$$

That the above is an area integral should be clear from the context, since Ω is an open subset of the plane. Similarly, differential operators will operate on *functions*, not *expressions*. More precisely:

$$\frac{d}{dt}f(t) = (Df)(t), \frac{\partial}{\partial x}f(x,y) = (D_1 f)(x,y), \frac{\partial^n}{\partial x^n}f(x,y) = (D_1^n)f(x,y), \tag{A.15}$$

and so on. And, unless otherwise specified, all functions considered here will be continuously differentiable up to whatever order is required in its context. Note that, for emphasis, the derivative of a function evaluated at t was written as $(Df)(t)$ above, but in general the differential operator D takes precedence over functional evaluation, and $(Df)(t) = Df(t)$.

One last bit of notational convenience is the *multi-index notation*. A *multi-index* α is an n-tuple of non-negative integers $(\alpha_1, \alpha_2, ..., \alpha_n)$. Given an n-vector x, define x^α to be $x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot ... \cdot x_n^{\alpha_n}$. Also, define the *gradient operator* $\nabla = D = (D_1, D_2, ..., D_n)$. Then D^α gives us a useful way to denote the differential operator $D_1^{\alpha_1} ... D_n^{\alpha_n}$. For convenience, define $|\alpha|$ to be $\alpha_1 + ... + \alpha_n$.

A.2.2 Partial differential equations

Before a discussion of algorithms for solving partial differential equations, some terminology and examples are needed. The focus here is on scalar differential equations, though some of these methods generalize to systems of equations.

Basic definitions

A *partial differential equation* for a real-valued function u of n real variables is a relation of the form

$$F(x, u(x), D_1u(x), ..., D_nu(x), D_1^2u(x), ...) = 0, \tag{A.16}$$

where F is real-valued function of finitely many real variables and x denotes a real vector with n components. A function u is a *solution* of the PDE over the domain Ω if it satisfies Equation (A.16) for all x in Ω. F constrains the value of the solution u and a *finite* number of its partial derivatives, and may depend on the coordinates. The *order* of a partial differential equation is the order of the highest-order partial derivative that appears in Equation (A.16). Depending on the specific function F, Equation (A.16) may have no solution, a unique solution, or more than one solution; the existence theory for solutions of partial differential equations is a large and complicated subject, and this report makes no attempt at presenting it. An mth-order PDE is *linear* if it can be written as

$$\sum_{|\alpha| \le m} A_\alpha D^\alpha u = Lu = b, \tag{A.17}$$

where the coefficients A_α of L, as well as b, are functions of the coordinates. This class of equations will be the most important to us.

Many equations arising from applications have *infinitely* many solutions, so one must prescribe additional constraints to obtain unique solutions. For an equation of order m on a domain of dimension n, these constraints usually involve specifying the values of the solution and its derivatives of order less than m on some $(n-1)$-dimensional subspace of the domain of solution. If a partial differential equation along with a constraint has a unique solution, the problem is said to be *well-posed*.[2] As we shall see, different types of equations require different constraints to have existence and uniqueness of solutions. For example, some constraints make the equation *overdetermined*; that is, there may not be a solution of the differential equation that satisfies the given constraint. On the other hand, some constraints may make the equation *underdetermined*, and there may be more than one solution. In these cases, the problem is said to be *ill-posed*.

This section deals with equations over open subsets Ω of the plane $(n = 2)$. Moreover, it concentrates on equations that are *linear* and *homogeneous* with *constant coefficients*:

$$aD_1^2u + bD_1D_2u + cD_2^2u + dD_1u + eD_2u + fu = 0, \tag{A.18}$$

where a, b, c, d, e, and f are arbitrary real constants. Slightly more general equations are treated later.

Laplace's equation

Here are, without proof, a number of facts regarding Laplace's equation. Given an open subset Ω of the plane, *Laplace's equation* for two variables is

$$D_1^2u + D_2^2u = 0, \tag{A.19}$$

[2]The idea of well-posedness is due to Jacques Hadamard, the great French mathematician, who also discovered some of the earliest examples of *ill-posed* problems. As a result, well-posed problems are sometimes called *well-posed in the sense of Hadamard* in mathematical literature.

where u is a real-valued function on $\overline{\Omega}$. A function satisfying Laplace's equation is said to be *harmonic*. It is clear that Laplace's equation is a special case of Equation (A.18). In general, it has infinitely many solutions on a given domain Ω. However, given a real-valued function f on the boundary $\partial\Omega$, the requirement that the solution u agrees with f on $\partial\Omega$, i.e.,

$$u(x, y) = f(x, y), (x, y) \in \partial\Omega, \tag{A.20}$$

uniquely determines the solution u; this is one of the clasical results in the theory of PDEs. Note that in this case, our constraint only specifies the values of the solution on the boundary, not the values of its first partials.

Equation (A.20) is called the *boundary condition*, and Equations (A.19) and (A.20) together form the *boundary value problem*. Solving this equation allows us to determine, for example, the electric potential in a bounded, charge-free region given the potential on the boundary.

There is a beautiful way to reformulate the boundary value problem for Laplace's equation as a minimization problem. Let f be a real-valued function on $\partial\Omega$, and let X_f be the set of all real-valued functions u on $\overline{\Omega}$ that agree with f on $\partial\Omega$. Define the real-valued mapping

$$S(u) = \frac{1}{2}\int_{\Omega}\left((D_1 u)^2 + (D_2 u)^2\right) \tag{A.21}$$

on the *function space* X_f; S is called the *action*. One can show that, among all functions u that satisfy the boundary conditions, the solution of Laplace's equation *minimizes* S. This is an example of a *variational principle*, and is discussed in more detail in §refsec:variational.

A.2.3 The Rayleigh-Ritz Method

Typically, the numerical solution of a partial differential equation involve two distinct steps. First, a way of representing the approximate solution is chosen, and the differential equation is reduced to some set of simpler equations that determine the approximate solution; this is known as *discretization*. Next, the discretized equations are solved, yielding the approximate solution. This section discusses only discretization methods, whereas the solution of large systems of linear algebraic equations was briefly described in §A.1. For a more thorough discussion of both aspects of this problem, see Vichnevetsky [27].

The specific discretization method developed here is known as the *Rayleigh-Ritz method*. The basic idea behind this method is simple: Given the domain Ω and a prescription of the boundary value f, choose a set of N functions $\{\phi_i\}$ on $\overline{\Omega}$ and express the solution u as a linear combination

$$u = \sum_{i=1}^{N} a_i \phi_i. \tag{A.22}$$

The functions ϕ_i are the *basis functions*, and the Rayleigh-Ritz method requires them to have some specific properties (these are discussed later). These properties allow us to interpret the coefficients a_i as values of the approximate solution u at pre-specified *sample points* (or *nodes*) p_i. Having specified a representation of approximate solutions, an approximation of the action can be computed as a function of the unknown coefficients a_i and the given boundary values. Minimizing this approximate action turns out to produce a system of linear equations, which can be solved to yield the coefficients a_i.[3]

[3]In the case of Laplace's equation, the coefficients of the discretized equations form a positive-definite

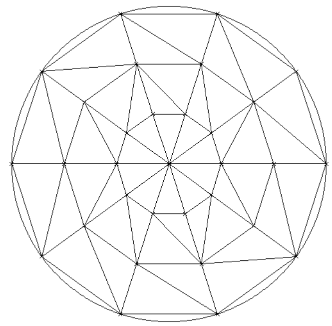


Figure A-1: Finite elements on the unit disc.

Constructing basis functions

The Rayleigh-Ritz method and a large class of other methods are collectively referred to as *finite element methods* because they all represent approximate solutions as linear combinations of a special type of basis functions. They rely on dividing the domain into a finite number of simple shapes, called *elements*, and expressing the approximate solution over each element as a sum of simple shapes. In what follows, the shapes are assumed to be triangles for simplicity, though in general they can be more complicated.

Here is a more detailed description of triangular elements: Choose a finite set of sample points $\{p_i\}$ in the domain $\overline{\Omega}$, such that the subset of sample points lying on the boundary $\partial\Omega$ is non-empty. Choose a finite collection of triangular subsets T_i of $\overline{\Omega}$, such that the T_i intersect each other only along their boundaries, and the sample points are precisely the vertices of the triangles. Furthermore, the union of the triangles T_i should closely approximates Ω.[4] As an example, Figure A-1 shows a crude division of the unit circle into triangular elements. In general, the more finely the elements tesselate the domain Ω, the more accurate the approximate solution will be.

To each *sample point* p_i we now associate a basis function ϕ_i. Intuitively, the basis function ϕ_i is produced by "pasting together" simple shapes over each element adjacent to p_i; this arrangement, as will be shown later, makes the computation of finite element coefficients more efficient. More precisely, these are the requirements on the basis functions:

1. $\phi_i(p_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$
2. $\sum_{i=1}^{N} \phi_i(x, y) = 1$ for all $(x, y) \in \overline{\Omega}$.
3. The functions ϕ_i should be piecewise-differentiable, if not smooth everywhere.
4. The function ϕ_i should be nonzero only in the elements immediately adjacent to p_i.

matrix, and may be inverted using many methods, such as relaxation, LU factorization, or conjugate gradient methods.

[4]This report does not attempt to precisely define this notion, but the union of the elements should at least be topologically equivalent to the original domain Ω.

The first requirement guarantees that if a function u is expressed as linear combination of the basis $\{\phi_i\}$, as in Equation (A.22), then its ith coefficient is simply

$$a_i = u(p_i). \tag{A.23}$$

This is a particularly nice property, for if p_i is a sample point on the boundary $\partial\Omega$, then the value of the approximate solution, $u(p_i)$, is just the given boundary value $f(p_i)$. But then $a_i = f(p_i)$, so that in the linear combination (A.22), the coefficients which correspond to boundary nodes do not need to be computed at all, thus reducing the number of unknowns. This is the way through which boundary values help determine the unknowns.

The second requirement ensures that if $a_1 = a_2 = ... = a_N = a$, then $u(x, y) = a$ for all $(x, y) \in \overline{\Omega}$; that is, constant functions are interpolated exactly by these basis functions. This ensures, for example, that if two approximate solutions constructed from these basis functions have the same values at all sample points, then they are equal everywhere.[5]

The third requirement is necessary because in the process of discretizing the PDE, it is necessary to take partial derivatives of the approximate solution. Finally, the last requirement makes precise the idea of pasting together simple shapes over elements adjacent to p_i. Note that the support of a basis function corresponding to a node p is simply the union of the elements with p as a vertex, and that the intersection of the supports of two basis functions must also be a union of elements. This is an important property for finite element computation.

One way of constructing basis functions that satisfy the requirements above are the so-called "tent functions," which are piecewise-linear functions constructed by linearly interpolating between neighboring nodes, and to let ϕ_i vanish uniformly outside of the elements adjacent to p_i. Note that by continuity, the last condition implies that $\phi_i = 0$ at all nodes except for p_i, so requirements 1 and 4 are redundant.

Discretization

With a specific representation of approximate solutions, one can now compute the unknown coefficients. To do this, use the approximate solution u to compute an approximation of the action (A.21). This is then a real-valued mapping that depends on the (finitely many) unknown coefficients of u. One can then minimize this approximate action by equating its derivative to zero, which yields a set of linear equations.[6]

At this point, it is helpful to keep track of nodes that lie on the boundary $\partial\Omega$. Thus, let us relabel the sample points so that the basis functions $\beta_i, i = 1, 2, ..., N$ correspond to sample points on the *boundary* of Ω, and let $\phi_i, i = 1, 2, ..., M$ continue to denote those that correspond to interior nodes. Let a_i denote the coefficients of ϕ_i, and let b_i denote those of β_i. As noted in the previous section, the N variables $b_1, b_2, ..., b_N$ are precisely the boundary values at the sample points on the boundary, so the only unknown values are $a_1, a_2, ..., a_M$.[7]

---

[5] Unlike finite difference methods, which only work with values of solutions at sample poitns, finite elements explicitly interpolates between sample points in discretizing PDEs.

[6] These are actually *stationary points* of the approximate action. For Laplace's equation, this is indeed the minimum. For other equations where variational principles apply, stationary points need not minimize the action.

[7] This step brings up a subtle point: There are two conditions that the approximate solution must satisfy, and together they produce a unique solution. One is that the approximate solution minimizes the action, and the other is that the solution has the required boundary values. This can be thought of as a constrained minimization problem. There are two approaches to these sorts of problems: The first (the one used here) is

Let T be a real-valued function of the M unknown variables a_i, defined by

$$
\begin{aligned}
T(a_1, ..., a_M) &= S(u[a_1, a_2, ..., a_M]) && \text{(A.24)} \\
&= \frac{1}{2} \int_\Omega (D_1 u[a_1, a_2, ..., a_M])^2 + (D_2 u[a_1, a_2, ..., a_M])^2, && \text{(A.25)}
\end{aligned}
$$

where $u[a_1, a_2, ..., a_M]$ is defined by

$$
u[a_1, a_2, ..., a_M] = \sum_{i=1}^{M} a_i \phi_i + \sum_{i=1}^{N} b_i \beta_i \tag{A.26}
$$

and the action S was defined in Equation (A.21).

To minimize T, simply differentiate under the integral sign. Via the chain rule, the partial derivatives of T, $D_j T(a_1, ..., a_M)$, are:

$$
\int_\Omega \left(\left(\sum_{i=1}^{M} a_i D_1 \phi_i + \sum_{i=1}^{N} b_i D_1 \beta_i \right) \cdot D_1 \phi_j + \left(\sum_{i=1}^{M} a_i D_2 \phi_i + \sum_{i=1}^{N} b_i \beta_i \right) \cdot D_2 \phi_j \right), \tag{A.27}
$$

for $j = 1, 2, ..., M$. Equating the derivatives of T to 0 produces a system of equations:

$$
\sum_{i=1}^{M} a_i \int_\Omega (D_1 \phi_i \cdot D_1 \phi_j + D_2 \phi_i \cdot D_2 \phi_j) = - \sum_{i=1}^{N} b_i \int_\Omega (D_1 \beta_i \cdot D_1 \phi_j + D_2 \beta_i \cdot D_2 \phi_j), \tag{A.28}
$$

with $j = 1, 2, ..., M$. This is a system of M linear equations in M unknowns. Indeed, let

$$
a_{ij} = \int_\Omega (D_1 \phi_i \cdot D_1 \phi_j + D_2 \phi_i \cdot D_2 \phi_j), \tag{A.29}
$$

and let A be the matrix (a_{ij}). Define the M-vector

$$
b = \left(- \sum_{i=1}^{N} b_i \int_\Omega (D_1 \beta_i \cdot D_1 \phi_j + D_2 \beta_i \cdot D_2 \phi_j) \right). \tag{A.30}
$$

Then Equation (A.28) becomes simply

$$
Au = b, \tag{A.31}
$$

where u is the vector of the unknown coefficients a_i.

Some comments on finite elements

The derivation of the discretized equations (A.28) involves many integrals. But recall now that the basis functions were chosen so that a basis function associated with the node p_i is nonzero only over those elements adjacent to p_i. Thus, the integrals in Equation (A.28) need only be evaluated over a finite number of elements. One can generally choose element shapes

---

to enforce the constraint first, and then minimize the action. The second involves minimizing the action first, and then enforcing the constraint. A careful analysis will show that the second approach actually produces a *overdetermined system* of equation; in order to arrive at the same equations one must justify the elimination of the "extra" equations involving the inner product of the residual and basis functions corresponding to boundary nodes.

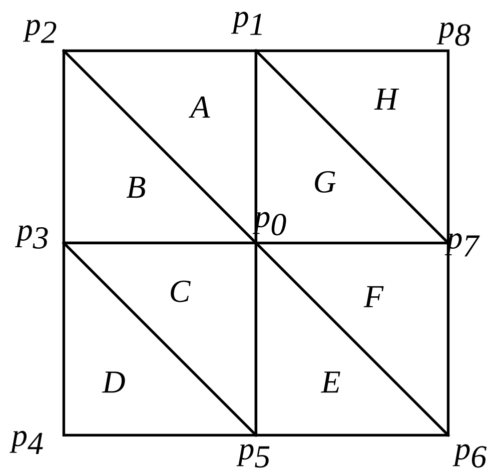


Figure A-2: Rectangular finite elements.

and basis functions to simplify the computation of these integrals, and the primary reason for the popularity of finite element methods is the efficiency with which these coefficients can be computed.

Additionally, this locality mirrors the fact that in many physical systems, most interactions are local and effects propagate with finite speed through the system. And because a coefficient is nonzero only if two nodes are neighbors (in the sense that they are vertices of the same element), the matrix A defined by Equation (A.29) is usually *sparse*; that is, it contains many zeros. This lessens the storage requirements when working with systems with large numbers of sample points, as well as making iterative solution methods like relaxation more efficient.[8]

Example

As an example, let us derive the standard finite difference equations for the boundary value problem using the Rayleigh-Ritz method. Consider a rectangular grid of points in the plane, a *subset* of which is shown in Figure A-2.

Let's use piecewise-linear tent functions on the elements, and suppose that the elements are isoceles triangles with base and height h. Let ϕ_i denote the basis function corresponding to the node p_i; it is then a tent function with its tip at the point p_i. To compute the coefficients corresponding to a typical node p_0 in the matrix $\mathbf{A} = (a_{ij})$ of Equation (A.29), let $c_i = a_{0,i}$.[9] Since the interpolants are linear, their gradients are constant. Hence, the coefficients are simply the dot products of the interpolants multiplied by the area of the intersection of their supports; denote the intersection $Support(\phi_i) \cap Support(\phi_j)$ by $\Omega_{(i,j)}$. The row of $\mathbf{A}$ corresponding to p_0 can have at most six non-zero entries, since p_0 has only six neighbors – namely p_1, p_2, p_3, p_5, p_6, and p_7. p_4 and p_8 (as well as any nodes in the system

[8] For solving large systems of linear equations, iterative methods are generally preferred over direct methods (such as LU decomposition) because of speed and the accumulation of round-off errors.

[9] The matrix is denoted by boldface in this section because the symbol A also refers to one of the regions in Figure (A-2).

that are not pictured in Figure A-2) are not neighbors of p_0 and hence those coefficients must vanish.

To compute c_1, note that $\Omega_{(1,0)} = A \cup G$. Over the region A, the gradients are

$$\nabla\phi_1 = \frac{1}{h}(1,1), \nabla\phi_0 = \frac{1}{h}(0,-1), \tag{A.32}$$

and over the region G, they are

$$\nabla\phi_1 = \frac{1}{h}(0,1), \nabla\phi_0 = \frac{1}{h}(-1,-1). \tag{A.33}$$

The area of each element is $\frac{1}{2}h^2$, so the coefficient c_1 is simply -2. Similarly, $c_3 = c_5 = c_7 = -2$ and $c_2 = c_6 = 0$. Finally, $c_0 = 1+1+1+1+2+2 = 8$, so the p_0 equation of the system $\mathbf{A}x = b$ is $8c_0 - 2c_1 - 2c_3 - 2c_5 - 2c_7 = 0$, which upon rearrangement yields

$$c_0 = \frac{c_1 + c_3 + c_5 + c_7}{4}. \tag{A.34}$$

Equation (A.34) is simply the standard finite difference approximation for Laplace's equation, and similar computations yield the same equations for the case when p_0 is on the boundary $\partial\Omega$.

A.2.4 Galerkin's method

Another commonly-used finite element method is *Galerkin's method.* In many cases, it produces equations equivalent to the Rayleigh-Ritz equations. However, this method differs in that it is slightly more difficult to justify mathematically, even though it is more generally applicable, especially in situations where a variational principle is not available. We derive Galerkin's method by a close analogy with a slightly more general function-expansion method, which also uses expansion in terms of basis functions to solve differential equations.

As before, basis functions are denoted by $\{\phi_i, i = 1,2,3,...\}$; however, these functions are not, for the moment, necessarily of the type considered in Rayleigh-Ritz. Furthermore, representations of functions as (possibly infinite) linear combinations of these basis functions is assumed to be *exact*, so the set of basis functions (called the *basis*) will no longer be finite. Given two real-valued functions f and g on Ω, define the inner product $\langle f, g\rangle$ by

$$\langle f, g\rangle = \int_\Omega f \cdot g \tag{A.35}$$

The basis is required to be *complete*, in the sense that if a function u satisfies $\langle u, \phi_i\rangle = 0$ for all i, then $u = 0$ uniformly on Ω. For example, if Ω is a bounded interval of the real line, one can choose the ϕ_i to be Legendre polynomials or sinusoidal functions; both form complete bases.

Back to Laplace's equation now: Recall that this involves finding a real-valued function u on $\overline{\Omega}$ such that

$$D_1^2 u + D_2^2 u = 0 \tag{A.36}$$

on Ω and $u = f$ on the boundary $\partial\Omega$ for some prescribed function f. Expanding the solution as an infinite series

$$u = \sum_{i=1}^{\infty} a_i \phi_i(x,y) \tag{A.37}$$

over a complete basis, the problem reduces to the determination of the unknown coefficients. Using completeness, this is equivalent to

$$\int_\Omega (D_1^2 u + D_2^2 u)\phi_i = 0, i = 1, 2, 3, ... \tag{A.38}$$

Expanding u in its infinite series, the above equation becomes

$$\sum_{i=1}^{\infty} a_i \int_\Omega (D_1^2 \phi_i + D_2^2 \phi_i)\phi_j = 0, j = 1, 2, 3, ... \tag{A.39}$$

Galerkin's method generalizes this procedure to the case when the basis is finite, and therefore *not complete.*

More specifically, let $\{\phi_i, i = 1, 2, ..., M\}$ and $\{\beta_i, i = 1, 2, ..., N\}$ now denote the finite element basis functions considered in §A.2.3, where, as before, M nodes lie in the interior of Ω and N nodes lie on the boundary. Since the finite element basis is finite, it cannot be a complete basis for the solution space (which is generally infinite-dimensional). However, by analogy with Equation (A.39), one can still require that the residual be orthogonal to the basis functions, producing

$$\sum_{i=1}^{M} a_i \int_\Omega (D_1^2 \phi_i + D_2^2 \phi_i)\phi_j = -\sum_{i=1}^{N} b_i \int_\Omega (D_1^2 \beta_i + D_2^2 \beta_i)\phi_j, \tag{A.40}$$

with $j = 1, 2, ..., M$. Integrating by parts and noting that each basis functions vanishes outside a bounded region, the equations become

$$-\sum_{i=1}^{M} a_i \int_\Omega (D_1\phi_i \cdot D_1\phi_j + D_2\phi_i \cdot D_2\phi_j) = \sum_{i=1}^{N} b_i \int_\Omega (D_1\beta_i \cdot D_1\phi_j + D_2\beta_i \cdot D_2\phi_j), \tag{A.41}$$

again for $j = 1, 2, ..., M$. These equations are identical, up to a sign, to (A.28).

Let u denote the approximate solution given by Galerkin's method. Galerkin's method only requires that the residual $D_1^2 u + D_2^2 u$ lies in the orthogonal complement of the span of the basis. Thus, without some other criterion to justify the equations, Galerkin's method does not actually guarantee that the approximate solution satisfies the differential equation in any sense. Notice the resemblance between the orthogonality condition and least-squares approximations: Recall that if ϕ is a function to be approximated, and u is linear combination of basis functions ϕ_i, then the orthogonality condition $\langle \phi - u, \phi_i \rangle = 0$ indeed produces the least-squares approximation. But in this case, the exact solution ϕ is not available to us. Thus, Galerkin's method does not actually produce the least squares approximation. Orthogonalizing the residual does not minimize it. Indeed, since the basis is not complete, the error residual can be arbitrarily large while still being orthogonal to all the basis functions.

Appendix B

Integration of Differential Forms on Manifolds

This appendix briefly sketches the construction of *differential forms*, which are mathematical objects that can be integrated on *oriented manifolds*. While they are of less importance in the theory of differential equations on manifolds, they are very much essential in the study of *differential topology*. However, those applications would take us too far afield and will not be discussed here. For more information, please see either Guillemin and Pollack [14] or Warner [28].

Recall the change of variables theorem (3.19):

$$\int_{y \in V_2} f(y)dy = \int_{x \in V_1} f(\phi(x))|\det D\phi(x)|dx \tag{B.1}$$

where f is a function on B and $\phi : A \to B$ is a smooth bijection. In §3.4.1, this theorem is used to define integrals of scalar-valued functions on compact Riemannian manifolds. Another possible approach, which we will briefly sketch here, involves assigning "determinant-like" functions to each tangent space of the manifold. Such an assignment is called a "differential form."

Let V be a finite-dimensional vector space. A scalar-valued function T on $V \times ... \times V$ is *multilinear* if it is linear in each of its components, and is *alternating* if exchanging any two arguments changes T to $-T$. The *degree* of T is the number of arguments T has. It is a theorem of linear algebra that such functions, called *alternating tensors*, are always proportional to the determinant function on V with respect to some basis.

Now, let ω be a function that assigns to each point $p \in M$ an alternating tensor ω_p on T_pM. One can show (although it is not done here) that the usual notions of smoothness also apply to these *alternating tensor fields*. A *differential form* on a manifold M is then a smooth alternating tensor field on M.

Because alternating tensors are proportional to the determinant function, it is not very difficult to show that one could obtain a consistent definition of integration for differential forms. To do this, first choose a partition of unity so that the problem is reduced to a local one. Then, note that tensors transform naturally in the following way:

$$T'(v_1, ..., v_k) = T(Lv_1, ..., Lv_k), \tag{B.2}$$

where T' is a tensor on some vector space W, T is a tensor on V, $L : V \to W$ is a linear transformation, and k is the degree of T'. Generalizing this to differential forms on manifolds, we can simply replace L by the differential $d\phi$ of the transition map ϕ. A little bit of linear algebra shows that this *almost* gives us the change of variables theorem:

$$\int_{y\in V_2} f(y)dy = \int_{x\in V_1} f(\phi(x)) \det D\phi(x)dx, \tag{B.3}$$

where $f \det$ is a (local) differential form on V_2 and $(f \circ \phi) \det$ is a differential form on V_1. (Note that both their degrees have to agree with the dimension of the space, n, because of the dimensions of $D\phi$ as a matrix.) As one can see, this is the change of variables theorem *except* for the absolute value. Thus, if one could choose charts so that all the transition maps have positive determinants:

$$\det D\phi(x) > 0, \tag{B.4}$$

then we can define integrals consistently. Manifolds for which such atlases exist are called *orientable manifolds*, and we can thus define integration of differential forms of degree n on compact orientable n-manifolds.

B.1 Stokes's theorem

One of the most important things one can do with differential forms is to generalize Stokes's theorem to compact orientable manifolds. This is done through a map called the *exterior derivative*, which takes a differential form ω of degree k to another differential form $d\omega$ of degree $k+1$. Defining d takes a little bit of work and will not be done here. But to show how much Stokes's theorem is simplified, here is the statement of the theorem using differential forms:

$$\int_M d\omega = \int_{\partial M} \omega. \tag{B.5}$$

This is actually so abstract that it does not say much, unless one has studied differential forms in some depth. However, note that the boundary operator on manifolds satisfies:

$$\partial(M \times N) = (\partial M \times N) \cup (M \times \partial N), \tag{B.6}$$

just like the product rule. As there is a corresponding product rule for exterior derivatives, this shows that there is a rather deep *duality* between geometric objects on the one hand and algebraic structures (such as differential forms) on the other.

Bibliography

[1] Harold Abelson, Tom Knight, and Gerald Jay Sussman. *Amorphous Computing (draft).* MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1995.

[2] Harold Abelson and Gerlad Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 2 edition, 1996.

[3] Lars Ahlfors. *Complex Analysis.* McGraw-Hill, New York, 3rd edition, 1979.

[4] V. I. Arnold. *Mathematical Methods of Classical Mechanics.* Springer-Verlag, New York, 2 edition, 1989.

[5] T. Balderes. Finite element method. *McGraw-Hill Encyclopedia of Science & Technology*, 1992.

[6] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *Submitted to ACM Transactions on Mathematical Software*, January 9, 1995.

[7] G. Chesshire and W. D. Henshaw. Composite overlapping meshes for the solution of partial differential equations. *Journal of Computational Physics*, 90:1–64, 1990.

[8] James J. Clark, Matthew R. Palmer, and Peter D. Lawrence. A transformation method for the reconstruction of functions from nonuniformly spaced samples. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 33(4):1151–1165, October 1985.

[9] William Clinger and Jonathan Rees, editors. *Revised^4 Report on the Algorithmic Language Scheme.* MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1991.

[10] Richard Courant and David Hilbert. *Methods of Mathematical Physics, Volume II: Partial Differential Equations.* John Wiley & Sons, New York, 1989.

[11] C. Armando Duarte and J. Tinsley Oden. *H-p* clouds—an *h-p* meshless method. *Numerical Methods for Partial Differential Equations*, 12(6):673–705, November 1996.

[12] Bjorn Engquist and Andrew Majda. Absorbing boundary conditions for the numerical simulation of waves. *Mathematics of Computation*, 31(139):629–651, July 1977.

[13] D. Fox and C. Pucci. The Dirichlet problem for the wave equation. *Ann. Mat. Pura Appl.*, 46:155–182, 1958.

[14] Victor W. Guillemin and Alan Pollack. *Differential Topology.* Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[15] Hermann A. Haus and James R. Melcher. *Electromagnetic Fields and Energy.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[16] Fritz John. *Partial Differential Equations.* Springer-Verlag, New York, 4th edition, 1981.

[17] Claes Johnson. *Numerical Solutions of Partial Differential Equations by the Finite Element Method.* Cambridge University Press, New York, 1987.

[18] James R. Munkres. *Elementary Differential Topology.* Princeton University Press, Princeton, New Jersey, 1966.

[19] James R. Munkres. *Topology: A first course.* Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[20] James R. Munkres. *Elements of Algebraic Topology.* Addison-Wesley, Reading, Massachusetts, 1984.

[21] James R. Munkres. *Analysis on Manifolds.* Addison-Wesley, Reading, Massachusetts, 1991.

[22] L. E. Payne. *Improperly posed problems in partial differential equations.* Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, Pennsylvania, 1975.

[23] N. Anders Petersson. An algorithm for constructing overlapping grids. *Submitted to SIAM J. Sci. Comput.*, March 16, 1997.

[24] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[25] Bernard F. Schutz. *A First Course in General Relativity.* Cambridge University Press, New York, 1990.

[26] Rafael Sorkin. Time-evolution problem in Regge calculus. *Physical Review D*, 12(2):385–397, 15 July 1975.

[27] Robert Vichnevetsky. *Computer Methods for Partial Differential Equations, Volume 1: Elliptic Equations and the Finite-Element Method.* Prentice-Hall, New Jersey, 1981.

[28] Frank W. Warner. *Foundations of Differentiable Manifolds and Lie Groups.* Springer-Verlag, New York, 1983.

[29] Jack Wisdom and Matthew Holman. Symplectic maps for the n-body problem. *The Astronomical Journal*, 102(4):1528–1538, October 1991.