SAND2002-0396
Unlimited Release
Initially Printed March 2002, Updated October 2002

LOCA 1.1

Library Of Continuation Algorithms: Theory and Implementation Manual

Andrew G. Salinger, Nawaf M. Bou-Rabee, Roger P. Pawlowski, and
Edward D. Wilkes
Parallel Computational Sciences Department

Elizabeth A. Burroughs, Richard B. Lehoucq, and Louis A. Romero
Computational Mathematics and Algorithms Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1111

Abstract

LOCA, the Library of Continuation Algorithms, is a software library for performing stability analysis of large-scale applications. LOCA enables the tracking of solution branches as a function of a system parameter, the direct tracking of bifurcation points, and, when linked with the ARPACK library, a linear stability analysis capability. It is designed to be easy to implement around codes that already use Newton's method to converge to steady-state solutions. The algorithms are chosen to work for large problems, such as those that arise from discretizations of partial differential equations, and to run on distributed memory parallel machines. This manual presents LOCA's continuation and bifurcation analysis algorithms, and instructions on how to implement LOCA with an application code. The LOCA code is publicly available at `www.cs.sandia.gov/loca`. **Version 1.1 adds** the capability of performing multi-parameter continuation through an interface to the MF library of Mike Henderson of IBM, which is now included in the distribution.

Contents

Chapter 1

Introduction

1.1 LOCA Overview

This document is the theory and implementation manual for version 1.0 of LOCA, the Library Of Continuation Algorithms. When implemented with an application code, LOCA enables the tracking of solution branches as a function of a system parameter and the direct tracking of bifurcation points. LOCA is designed to drive application codes that use Newton's method to locate steady-state solutions to nonlinear problems. The algorithms are chosen to work for large problems, such as those that arise from discretizations of partial differential equations, and to run on distributed memory parallel machines.

The approach in LOCA for locating and tracking bifurcations begins with augmenting the residual equations defining a steady state with additional equations that describe the bifurcation. A Newton method is then formulated for this augmented system; however, instead of loading up the Jacobian matrix for the entire augmented system (a task that involved second derivatives and dense matrix rows), bordering algorithms are used to decompose the linear solve into several solves with smaller matrices. Almost all of the algorithms just require multiple solves of the Jacobian matrix for the steady state problem to calculate the Newton updates for the augmented system. This greatly simplifies the implementation, since this is the same linear system that an application code using Newton's method will already have invested in. Only the Hopf tracking algorithm requires the solution of a larger matrix, which is the complex matrix involving the Jacobian matrix and an imaginary multiple of the mass matrix.

The following algorithms are available in this version of LOCA:

1. ZERO_ORDER_CONTINUATION

2. FIRST_ORDER_CONTINUATION

3. ARC_LENGTH_CONTINUATION

4. TURNING_POINT_CONTINUATION (fold points)

5. PITCHFORK_CONTINUATION

6. HOPF_CONTINUATION

7. PHASE_TRANSITION_CONTINUATION

The LOCA library relies heavily on a robust linear stability analysis capability. This is needed for detecting the first instance of Hopf and Pitchfork bifurcations and generating initial guesses for the null vectors for these routines. We have used the P_ARPACK package [1][2] to indentify a few select eigenvalues and eigenmodes. With this release of LOCA, we are including routines that drive P_ARPACK to perform the Cayley transformation, which we have found to work well for large scale problems [3][4].

(In addition, the LOCA interface has been linked to an rSQP optimization code. The RSQP_OPTIMIZATION option will be discussed in the implementation section, but details on how to use this feature are outside the scope of this document.)

The rest of the document is organized as follows. Some background information on bifurcations follows in Section 1.2. Chapter 2 is the theory manual, and presents the algorithms that are implemented in LOCA. Chapter 3 shows the directory structure of the Loca code and contains instructions on how to implement LOCA around a new application code, including a recipe to follow. Chapter 4 gives details on each of the wrapper routines that give LOCA access to routines in the application code. Chapter 5 goes through each of the elements of the two structures that control the continuation algorithms. Chapter 6 explains the typical strategies employed to track bifurcations. This document does not contain demonstrations of the LOCA library being used with application codes.

Publications and presentations that demonstrate the use of the LOCA library for analysis of applications can be obtained from the LOCA web page:

- **www.cs.sandia.gov/LOCA**

This web page contains links to the download site for the source code and this manual.

The LOCA code is being licensed under the GNU Lesser General Public License, a copy of which is available with the code and can also be found at www.gnu.org. We welcome users to submit suggestions, bug reports, bug fixes, extensions, and results and publications generated using LOCA.

1.2 A Survey of Bifurcation Theory

This section is meant to give the user a brief introduction to bifurcation theory. For more information we refer users to the following references [5, 6, 7, 8, 9]. The Seydel book in particular has an excellent list of references.

1.2.1 Complex Behavior in Nonlinear System

The existence of multiple steady state solutions, or of unstable steady solutions, can lead to interesting behavior in nonlinear systems. This interesting behavior can lead to undesirable effects such as a system operating in one way on certain occasions, and in another way on other occasions. On the other hand, this interesting behavior can also be desirable, as when one is interested in making switches or oscillators.

LOCA, and bifurcation theory in general, is aimed at helping scientists and engineers efficiently map out different regions of parameter space that have qualitatively different behavior. Although one could in principle map out these regions using only transient calculations, this would be inefficient and unreliable[1]. For example, suppose we wanted to know for what values of parameters a system has two stable steady state solutions. For a given value

[1]We should mention that transient calculations could yield information not obtained in a bifurcation analysis. Ideally one would use a transient analysis in conjunction with a bifurcation analysis.

of the parameters we could determine that there are multiple stable steady states by solving many different initial value problems. If some converge towards one solution, and others converge towards another, then we have multiple stable steady states. However, each transient calculation will be time consuming, and it is not clear how many trials we need to do in order to be convinced that only one stable steady solution exists. Furthermore, once we have convinced ourselves how many stable steady states there are for one set of parameters, this information does not greatly reduce the search when we change the parameters slightly.

Using tools like LOCA and P_ARPACK, one only solves for steady state solutions, but takes into account the transient behavior by looking at the eigenvalues of the linearized system. The bifurcation approach has two advantages over the transient approach.

- We can solve steady equations, which is much more efficient computationally then integrating until a steady state is reached.
- We can append equations to our steady state solution that allow us to find the value of a parameter where the system changes its behavior. When we have more than one parameter in our system, we can then map out parameter space very efficiently.

1.2.2 Generic Behavior

Suppose that we have a dynamical system of the form

$$\mathbf{B}\dot{\mathbf{x}} = \mathbf{R}(\mathbf{x}, \lambda) \tag{1.1}$$

where $\mathbf{x}$ is a vector (possibly of infinite dimension) that determines the state of our system, and λ is a parameter. Steady state solutions can be found by solving

$$\mathbf{R}(\mathbf{x}, \lambda) = 0 \tag{1.2}$$

The stability of a steady state solution $\mathbf{x}_0$ is determined by finding the eigenvalues of the system

$$\sigma \mathbf{B}\mathbf{w} = \mathbf{J}(\mathbf{x}_0)\mathbf{w} \tag{1.3}$$

where $\mathbf{J}(\mathbf{x}_0)$ is the Jacobian of the function $\mathbf{R}$ evaluated at the steady solution $\mathbf{x}_0$. If all of the eigenvalues of this system have real parts less than zero, the steady state solution is stable. If any of the eigenvalues have a real part bigger than zero, the system is unstable.

It is possible to find many different ways for a solution to lose its stability. For example, we can mathematically construct systems that have 10 modes all go unstable at the same value of the parameter. It can be exceedingly difficult to analyze such problems, but fortunately problems of this sort are unlikely to occur in practice. One of the basic ideas in bifurcation theory is that of generic behavior. We only concern ourselves with behavior that is likely to occur in a system. Behavior is said to be non-generic if when we perturb our equations, the behavior no longer exists. For example, it is generic for two curves in the plane to intersect with non zero slope. If we slightly change the equations of the two curves, they will still intersect at some other point with non-zero slope. However, it is not generic for two curves in three dimensional space to intersect. Although it is not difficult to find examples of curves that intersect. If we apply almost any arbitrarily small perturbation to the equations, they will not intersect.

We now apply this notion of a generic property to nonlinear stability problems. Suppose we have a steady state solution $\mathbf{x}_0(\lambda)$ that depends on a parameter λ. Suppose that we analyze the stability of this solution, and find that at some value λ_0 this solution goes unstable. This implies that there is a critical eigenvalue that has a zero real part. Elementary bifurcation theory shows that there are only two generic ways that a solution can lose its stability.

- If the critical eigenvalue is zero, the system has a turning point.
- If the critical eigenvalue is imaginary, the system undergoes a Hopf bifurcation.

A turning point is a point in parameter space where two solution branches merge and then disappear. For example the equation $x^2 + \lambda = 0$ has two solutions if $\lambda < 0$, but no solutions if $\lambda > 0$. The point $x = 0$, $\lambda = 0$ is a turning point (Figure 1.1a). Some physical examples of turning points include:

- The buckling of a shallow arch under symmetrical loading.
- The breaking away of a drop from a tube when the volume is too large.

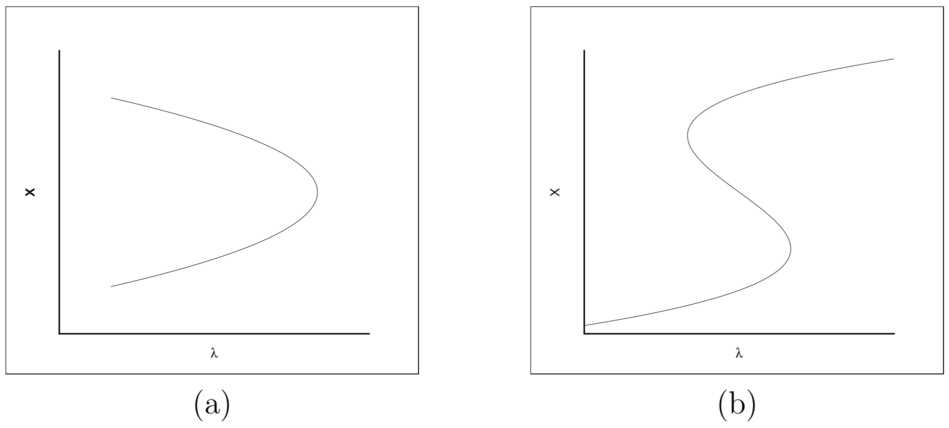


Figure 1.1: Bifurcation diagrams of a turning point: (a) one turning point, (b) multiple turning points (leading to hysteresis).

- The bursting of a balloon when a critical volume is reached.
- Ignition of an explosion.

Elementary bifurcation theory shows that the stability of one mode changes as we go around a turning point. This implies that if one of the solutions is stable, then the other solution will be unstable. Frequently we encounter more than one turning point in a system. In this case we can get an S shaped solution branch (Figure 1.1b). For sufficiently large or small values of the parameters we will have only one solution, but for intermediate values we will have three steady solutions. The upper and lower branches will be stable, the middle branch unstable. This S shaped solution curve can lead to nonlinear hysteresis, where as we increase our parameter value the solution suddenly jumps from the lower to the upper solution. If we then slightly decrease the value of the parameter the solution does not jump back to the lower solution, but will only do this if we greatly reduce the value of the parameter.

The other generic form of loss of stability is the Hopf bifurcation. After a Hopf bifurcation the system no longer settles down to a steady state, but will begin to oscillate periodically. Examples of Hopf bifurcations include the onset of :

- Vertex shedding behind bluff bodies.
- Flutter in airplane wings.

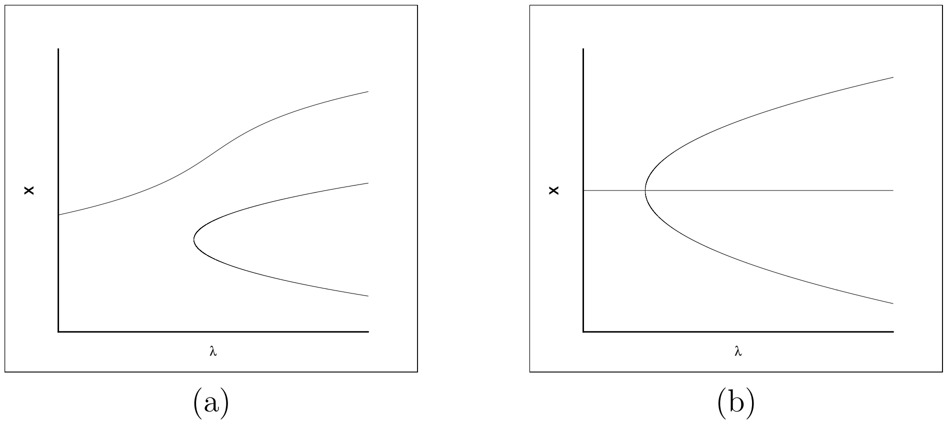


Figure 1.2: Bifurcation diagrams of a symmetry breaking bifurcation: (a) broken pitchfork bifurcation (b) pitchfork bifurcation.

- Oscillations in electrical circuits
- Shimmying of wheels.

Note that when we have located either one of these bifurcations, we are at very interesting points in parameter space. The turning point and Hopf bifurcation are generically the only bifurcations we expect to see in a one parameter system. For this reason they are called co-dimension one bifurcations. If we have more than one parameter in our system we can get more degenerate bifurcations. These more degenerate bifurcations tell us even more about our system than a co-dimension one bifurcation. For example suppose our system has two parameters λ and μ. Suppose that for some value of μ our solution curves as a function of λ look like those in Figure 1.2a. If we know about the turning point T, we know that our solution has at least two solutions for some values of the parameters. However, knowing about the turning point does not tell us that the upper disconnected branch exists. Suppose that as we change the parameter μ we find that for some value of μ our solution curves look like those in Figure 1.2b. The particular value of λ, μ and $\mathbf{x}$ where we have these solution branches intersecting is an example of a co-dimension two bifurcation. (This particular co-dimension two bifurcation is called a pitchfork bifurcation.) If we know about the existence of a co-dimension two bifurcation we know that when we unfold it, we will get behavior like we see in Figure 1.2a (and other behavior as well). This example illustrates the point that there is more information in a co-dimension two bifurcation.

1.2.3 Symmetry Breaking

When analyzing the stability of steady state solutions it is very common to find non-generic behavior. For example, if we take a beam and load it symmetrically we find that the simple solution where it is straight up and down is stable if the loading is not large. When the loading gets to be large enough the beam will buckle by bending either to the left or right. Note that this is not a turning point, or a Hopf bifurcation. It is not a Hopf bifurcation because the beam does not begin oscillating, It is not a turning point because we have two solution branches intersecting each other, rather than a single solution branch turning around on itself.

At first sight this appears to cast some doubt on the usefulness of the concept of generic behavior. However, we can note that if the beam was not built perfectly symmetrical we would find that the pitchfork bifurcation gets split up into two branches that do not intersect. We only get the non-generic behavior in the symmetrical case.

The best way out of this dilemma is to note that scientists and engineers often analyze symmetrical situations. We then ask what type of generic behavior we expect to encounter under the assumption that our system has some symmetry. It should be emphasized that these additional types of bifurcations are co-dimension one bifurcations in the presence of symmetry, but they would be co-dimension two (or higher) bifurcations if no symmetry were present. For this reason we see that the scientist or engineer is actually being wise in choosing to analyze a system with symmetry. This allows them to get more information out of a one parameter system.

Certain sorts of symmetry can ensure the existence of multiple eigenvalues in our system. This can lead to very interesting and complex behavior at bifurcation points. In LOCA we limit ourselves to the simplest sort of bifurcations that can occur when a system has symmetry. These bifurcations assume that the critical mode has a simple eigenvalue.

As in the previous section we assume that we are analyzing the stability of a steady solution $\mathbf{x}_0(\lambda)$. However, we now assume that the governing equations (and our solution $\mathbf{x}_0(\lambda)$) are invariant under some group of symmetry transformations. We now suppose that at some value of λ, our system goes unstable. We assume that the critical mode is a simple eigenvalue (this is not generically requited in all systems with symmetry). In this case if the critical eigenvalue has a nonzero imaginary part we will once again get a Hopf bifurcation. This bifurcation will have some interesting spatio-temporal symmetry. But, it is still a Hopf bifurcation. On the other hand, if the critical eigenvalue is zero, there are two possibilities. In order to understand these possibilities we need to realize that when we analyze a system with symmetry, then the eigenvector associated with a simple real eigenvalue must either be symmetric or anti-symmetric with respect to any of the symmetries of the group. If the eigenvector is symmetric with respect to all of the transformations in the group, it is said to be a fully symmetric eigenvector. If it is anti-symmetric with respect to some transformations, it is said to be a symmetry breaking eigenvector.

When a symmetric system loses its stability at a simple real eigenvalue, there are two possibilities

- If the critical eigenvector is fully symmetric, then the system encounters a turning point. The symmetry of the solution does not change as we go around the turning point.

- If the critical eigenvector is symmetry breaking, then we encounter a pitchfork bifurcation. In this case the symmetrical solution $\mathbf{x}_0(\lambda)$ continues to exist on both sides of the critical value of λ. There is a second solution branch that intersects this symmetrical branch. This second branch only exists locally on one side of the bifurcation point, and the solutions on this branch are not symmetric with respect to all elements of the group. Furthermore the elements on the upper and lower branches are transformed into each other by the elements of the group.

1.2.4 Tranversality Conditions

We used the example of two curves intersecting with non-zero slope in two dimensional space as an example of generic behavior. If we were a bit sloppy we could merely say that it is generic for two curves to intersect in two dimensional space, without mentioning the non-zero slope condition. The condition that they have non-zero slope is known as a transversality condition, and is included to guarantee that we do not have a degenerate situation, and that we can prove that for any small enough perturbation of our equations, we can find a new point of intersection.

In bifurcation theory the conditions stating that we are not at a degenerate point in parameter space are also known as transversality conditions. For example, suppose we have a system that has a turning point at $\mathbf{x} = \mathbf{x}_0$, $\lambda = \lambda_0$. If this system is not degenerate, if we perturb it slightly, the perturbed system should also have a turning point nearby the original turning point. If this is not the case, then we are at a degenerate point in parameter space, and we have a situation similar to two two dimensional curves intersecting with zero slope. There are 3 transversality conditions which guarantee that we are not at a degenerate point. Let $\mathbf{r}$ and $\mathbf{l}$ be the right and left eigenvectors of the Jacobian matrix. The transversality conditions can be stated as follows:

- The zero eigenvalue is a simple eigenvalue.
- The quantity $\mathbf{l}^{\mathsf{T}} \frac{\partial F(\mathbf{x}_0, \lambda_0)}{\partial \lambda}$ must not vanish.
- The quantity $\mathbf{l}^{\mathsf{T}} \mathbf{R}_{\mathbf{xx}}(\mathbf{x}_0, \lambda_0)\mathbf{rr}$ must not vanish.

For a one dimensional problem these conditions can be stated as $\frac{\partial F}{\partial \lambda} \neq 0$, and $\frac{\partial^2 F}{\partial x^2} \neq 0$. For higher dimensional problems the term $\mathbf{l}^{\mathsf{T}} \mathbf{R}_{\mathbf{xx}}\mathbf{rr}$ is the projection of the quadratic terms onto the center manifold.

For a Hopf bifurcation, the transversality conditions guarantee that if we have located a Hopf bifurcation, and then we perturb our system, we can locate a nearby Hopf bifurcation. The transversality conditions are

- The Jacobian matrix does not have a zero eigenvalue, and there is one and only one complex conjugate pair of eigenvalues that has a zero real part.
- The complex conjugate pair of eigenvalues are simple eigenvalues.
- Let $\sigma(\lambda)$ be the critical eigenvalue. The real part of the critical eigenvalue must pass through zero with non zero slope. That is $Re \frac{d\sigma(\lambda_0)}{d\lambda} \neq 0$.

The transversality conditions for a pitchfork bifurcation (in the presence of symmetry) are :

- The critical eigenvalue is simple.
- The critical eigenvalue passes through zero with nonzero slope, $\frac{d\sigma(\lambda_0}{d\lambda} \neq 0$.

Chapter 2

LOCA Bifurcation Algorithms

In this chapter, we present the algorithms that are implemented in the LOCA library. These can be divided into three types: parameter continuation algorithms to track steady state solutions as a function of a single parameter or, new to LOCA version 1.1, as a function of multiple parameters using the MF Library[10]; bifurcation tracking algorithms to calculate a parameter value (referred to as the bifurcation parameter) at which a bifurcation occurs as a function of a second (continuation) parameter; and a linearized stability analysis routine to ascertain the stability of the steady state solution (using the ARPACK library [1, 2]).

For references on computing bifurcations, we recommend Cliffe, Spence, and Tavener (2000) [5] and Govaerts (2000) [11] and references therein. To see examples of the LOCA algorithms being used with large-scale analysis codes, see the publications section of the LOCA web page: www.cs.sandia.gov/LOCA.

It is assumed that the application code uses a fully coupled Newton method to solve for steady states to a set of nonlinear equations. We have n residual equations $\mathbf{R}$ which form the basis of the model. The steady state problem is written as

$$\mathbf{R}(\mathbf{x}, \lambda) = \mathbf{0}, \tag{2.1}$$

which, given an initial guess for $\mathbf{x}$, is solved iteratively with Newton's method,

$$\mathbf{J}\Delta\mathbf{x} = -\mathbf{R}, \qquad \mathbf{x}^{new} = \mathbf{x} + \Delta\mathbf{x} \tag{2.2}$$

where the Jacobian matrix $\mathbf{J} = \frac{\partial \mathbf{R}}{\partial \mathbf{x}}$. The iteration on $\mathbf{x}$ converges when $\|\Delta\mathbf{x}\|$

and/or $\|\mathbf{R}\|$ decrease below some tolerances. For scalability to large applications, the linear solve of the matrix equation 2.2 must be solved iteratively.

2.1 Parameter Continuation Methods

Steady state solution branches are tracked using continuation algorithms. Zero order, first order, and pseudo arc length continuation [12] algorithms have all been implemented in the LOCA library.

2.1.1 Zero Order and First Order Continuation

This section covers zero order and first order continuation in a chosen parameter λ. Both of these algorithms consist of seeking a sequence of steady state solutions to a specified problem,

$$\mathbf{R}\left(\mathbf{x}_i, \lambda_i\right) = \mathbf{0} \tag{2.3}$$

where λ_i is a parameter value in the specified range of continuation and $\mathbf{x}_i$ is the converged solution vector at the steady state corresponding to $\lambda = \lambda_i$. These are the simplest continuation algorithms, as they do not require augmentation or bordering of the problem matrix.

In zero order continuation, the steady state solution $\mathbf{x}_i$ obtained at each step is used as the initial guess for Newton iteration at the following step:

$$\mathbf{x}_{i+1}^{P} = \mathbf{x}_i \tag{2.4}$$

In first order continuation, the tangent vector, or sensitivity of the solution to the continuation parameter, is predicted by doing an additional linear solve of the system

$$\mathbf{J}\frac{\partial \mathbf{x}}{\partial \lambda} = -\frac{\partial \mathbf{R}}{\partial \lambda} \tag{2.5}$$

where $\mathbf{J}$ is the Jacobian matrix previously computed with $\lambda = \lambda_i$ and $\frac{\partial \mathbf{R}}{\partial \lambda}$ is a forward-difference approximation obtained by perturbing λ a small amount δ and reassembling the residual:

$$\frac{\partial \mathbf{R}}{\partial \lambda} = \left[\mathbf{R}\left(\lambda_i + \delta\right) - \mathbf{R}\left(\lambda_i\right)\right]/\delta \tag{2.6}$$

The first order prediction of $\mathbf{x}_{i+1}$ at the next continuation step $\lambda_{i+1} = \lambda_i + \Delta\lambda_i$ is then

$$\mathbf{x}^P_{i+1} = \mathbf{x}_i + \frac{\partial \mathbf{x}}{\partial \lambda}\Delta\lambda_i \tag{2.7}$$

Thus, first order continuation entails one additional linear solve per continuation step, but this should reduce the number of Newton iterations required to attain convergence on each subsequent step.

Step size control refers to the methods used to determine the change in the parameter value λ_i for each continuation step. The relevant LOCA input quantities are (see Section 5.2 for implementation details):

λ_b Beginning parameter value on entering LOCA.

λ_e Ending parameter value used to determine completion of continuation; may be larger or smaller than λ_b.

$\Delta\lambda_0$ Initial step size $\lambda_1 - \lambda_0$.

$\Delta\lambda_{min}$ Smallest allowable absolute step size $|\Delta\lambda|$.

$\Delta\lambda_{max}$ Largest allowable absolute step size $|\Delta\lambda|$.

a Adjustable 'aggressiveness' parameter for increasing the step size, set to zero for constant step size.

N_{max} Maximum Newton iterations per continuation step.

N_c Maximum number of continuation step attempts (including failures).

If the first step fails to converge after N_{max} iterations, the entire continuation run is terminated. Otherwise, the parameter λ is advanced by the specified initial step size $\Delta\lambda_0$ and the second step is attempted. After the second step and each subsequent step, the step size $\Delta\lambda$ is adjusted as described below.

If the solution attempt failed to converge, the previous step size is halved. If the step size falls below $\Delta\lambda_{min}$ or if N_c is exceeded, continuation is terminated. Otherwise, the parameter value (and initial solution guess if doing first order) are updated for the new step size and the step is attempted again.

If the solution attempt did converge, the next step size $\Delta\lambda_i = \lambda_{i+1} - \lambda_i$ is computed as

$$\Delta\lambda_i = \Delta\lambda_{i-1}\left[1 + a\left(\frac{N_{max} - N_i}{N_{max} - 1}\right)^2\right] \tag{2.8}$$

where a is the input step control parameter and N_i is the number of Newton iterations required for convergence on the last step. This value is reset to $\Delta\lambda_{max}$ if it exceeds that value. Also, if the new step size would take the parameter past its final value λ_e, $\Delta\lambda_i$ is reset to $\lambda_e - \lambda_i$, as this will be the last step.

When a constant step size is specified (i.e. $a = 0.0$) and the step size is reduced after one or more failed step attempts, once convergence is again attained the step size is permitted to increase by using $a = 0.5$ in Equation 2.8 until the initial (constant) step size is again reached.

This procedure allows the parameter step size to be controlled adaptively based on the nonlinear solver convergence rate and is used for all LOCA algorithms, but is slightly modified for arc length continuation as discussed in the Section 2.1.3.

2.1.2 Pseudo Arc Length Continuation

When it is desired to perform continuation of a problem at parameter values in the vicinity of a stability limit (i.e. near a turning point), difficulties arise as the Jacobian matrix approaches singularity. Using simple parameter continuation methods would result in more failed step attempts and smaller step sizes as the turning point is approached. The pseudo arc length continuation algorithm [12] is designed to alleviate the singularity by augmenting the linear system with an alternate arc length parameter s and an arc length equation. The augmented system is then described by

$$\mathbf{R}\left(\mathbf{x}(s), \lambda(s)\right) = 0 \tag{2.9}$$

$$n\left(\mathbf{x}(s), \lambda(s), s\right) = 0 \tag{2.10}$$

where $\mathbf{R}$ is the Newton residual and n is an arc length equation. Here, both $\mathbf{x}$ and λ are parameterized as functions of s, which can be defined by

$$||d\mathbf{x}||^2 + (d\lambda)^2 = (ds)^2 \tag{2.11}$$

An arc length equation can then be obtained from Equation 2.11 by differentiating with respect to s:

$$\left\|\frac{\partial \mathbf{x}}{\partial s}\right\|^2 + \left(\frac{\partial \lambda}{\partial s}\right)^2 = 1 \tag{2.12}$$

However, it is more convenient to use a linearized form of Equation 2.12, as in [13]

$$n\left(\mathbf{x}(s), \lambda(s), s\right) = \left(\mathbf{x}_i - \mathbf{x}_{i-1}\right) \cdot \frac{\partial \mathbf{x}_i}{\partial s} + \left(\lambda_i - \lambda_{i-1}\right) \frac{\partial \lambda_i}{\partial s} - \Delta s = 0 \tag{2.13}$$

This algorithm seeks steady state solutions at pre-determined intervals of arc length Δs rather than of the parameter $\Delta\lambda$. In Equation 2.9, the arc length equation establishes the relationship between $\Delta\lambda$ and Δs, and the residual equations establish the steady state solution at the corresponding value of λ.

The augmented system can be expressed in matrix form as

$$\begin{bmatrix} \mathbf{J} & \frac{\partial \mathbf{R}}{\partial \lambda} \\ \left(\frac{\partial \mathbf{x}}{\partial s}\right)^T & \frac{\partial \lambda}{\partial s} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \lambda \end{bmatrix} = - \begin{bmatrix} \mathbf{R} \\ n \end{bmatrix} \tag{2.14}$$

where $\mathbf{J}$, $\mathbf{R}$, and $\Delta\mathbf{x}$ are computed during each Newton iteration using a guessed value of λ. To solve the augmented system (Equation 2.14), it is necessary to update both λ and the solution update $\mathbf{x}$ at each iteration. While it is possible to simply construct the full $(N+1) \times (N+1)$ system and solve it once, the resulting matrix would be more dense than the original Jacobian. LOCA's bordering algorithm (see also [13]) exploits the typically sparse nature of $\mathbf{J}$ by performing one resolve per iteration:

$$\mathbf{J}\mathbf{a} = -\mathbf{R} \tag{2.15}$$

$$\mathbf{J}\mathbf{b} = -\frac{\partial \mathbf{R}}{\partial \lambda} \tag{2.16}$$

where $\mathbf{a}$ and $\mathbf{b}$ are temporary vectors. The new updates are then found by:

$$\Delta\lambda = -\left(n + \frac{\partial \mathbf{x}}{\partial s} \cdot \mathbf{a}\right) / \left(\frac{\partial \lambda}{\partial s} + \frac{\partial \mathbf{x}}{\partial s} \cdot \mathbf{b}\right) \tag{2.17}$$

$$\Delta\mathbf{x} = \mathbf{a} + \Delta\lambda\mathbf{b} \tag{2.18}$$

Like all other LOCA bordering algorithms, this algorithm is called from the application code's nonlinear solver during each Newton iteration after the solution update is solved for but before the solution is updated (see step 2 of Section 3.2). The convergence status of the bordering algorithm updates (true or false) is returned, and becomes an additional criterion for convergence of the nonlinear solver.

2.1.3 Arc Length Scaling and Step Size Control

As discussed by Shadid [13], it is numerically advantageous for the relative magnitudes of the parameter and solution updates to be of similar order. In particular, the advantage of using arc length parameterization can be lost if the solution contribution to the arc length equation becomes very small. In this algorithm, a single scaling factor Θ is used for the solution contribution in order to provide some control over the relative contributions of λ and $\mathbf{x}$. The modified arc length equation is then

$$\left(\frac{\partial \lambda}{\partial s}\right)^2 + \Theta^2 \left\| \frac{\partial \mathbf{x}}{\partial s} \right\|^2 = 1 \tag{2.19}$$

or equivalently

$$\left(\frac{\partial \lambda}{\partial s}\right)^2 \left[1 + \Theta^2 \left\| \frac{\partial \mathbf{x}}{\partial \lambda} \right\|^2\right] = 1 \tag{2.20}$$

Equation 2.20 can be rearranged to find $\frac{\partial \lambda}{\partial s}$:

$$\frac{\partial \lambda}{\partial s} = \pm \left[1 + \Theta^2 \left\| \frac{\partial \mathbf{x}}{\partial \lambda} \right\|^2\right]^{-1/2} \tag{2.21}$$

The sign of $\frac{\partial p}{\partial s}$ is chosen to be the same as that of the quantity

$$\Theta^2 \left(\frac{\partial \mathbf{x}}{\partial \lambda}\right)_i \cdot (\mathbf{x}_i - \mathbf{x}_{i-1}) + (\lambda_i - \lambda_{i-1}) \tag{2.22}$$

such that s increases as the parameter p proceeds in the direction from λ_{i-1} to λ_i [13]. This also enables detection of when a turning point is passed, as $\frac{\partial \lambda}{\partial s}$ will change sign there.

The relevant inputs for arc length scaling are (see Section 5.3 for implementation of these quantities):

$\lambda_g'^2$ Used to set the initial value of scale factor Θ and to periodically rescale the solution (default = 0.5).

λ'_{max} Value of $\frac{\partial \lambda}{\partial s}$ at which rescaling is invoked (default = 0.0).

y Used to adjust parameter step size when solution vector changes rapidly (default = 0.0).

τ_m Used only when very small steps must be taken near a turning point (default = 0.0).

After the first step is completed, the initial value of Θ is found such that $\frac{\partial\lambda}{\partial s} = g$, where g is the square root of $\lambda_g'^2$:

$$\frac{\Theta}{\Theta_{old}} = \frac{\left(\frac{\partial\lambda}{\partial s}\right)}{g}\sqrt{\frac{1-g^2}{1-\left(\frac{\partial\lambda}{\partial s}\right)^2}} \tag{2.23}$$

and Θ_{old} is set to one initially. After each subsequent arc length step, the value of $\frac{\partial\lambda}{\partial s}$ is recalculated, and if it exceeds λ'_{max}, Equation 2.23 is used to calculate the new Θ value which will restore $\frac{\partial\lambda}{\partial s}$ to its target value g. However, if the new Θ value exceeds 10^8, it is reset to 10^8 and the new value of $\frac{\partial\lambda}{\partial s}$ (which will then differ from g) is computed. Rescaling of the solution will change the values and step sizes of the arc length parameter s, but will not effect the solution.

After completing the first step, an initial arc length step size is computed as

$$\Delta s_0 = \Delta\lambda_0 / \left(\frac{\partial\lambda}{\partial s}\right)_0 \tag{2.24}$$

Thereafter, step size control for arc length continuation is achieved by the same methods as for parameter continuation (see Section 2.2.1), except that the arc length step Δs is used in Equation 2.8 rather than Δp. However, the limits $\Delta\lambda_{min}$ and $\Delta\lambda_{max}$ still apply to $\Delta\lambda$, which is estimated as $\frac{\partial\lambda}{\partial s}\Delta s$. If this estimate exceeds $\Delta\lambda_{max}$, then Δs is adjusted proportionately. For the final step, Δs is estimated such that λ will reach its end value λ_e, although it may not reach this value exactly.

As in first order continuation, the tangent vector $\frac{\partial\mathbf{x}}{\partial\lambda}$ is computed by one linear solve after each successful step, and the initial solution guess at the next step is computed from it. The tangent vector also provides a means of detecting how fast the solution vector changes as the parameter is advanced. An indicator of this rate of change is the tangent factor:

$$\tau = \frac{\left(\frac{\partial\mathbf{x}}{\partial\lambda}\right)_i \cdot \left(\frac{\partial\mathbf{x}}{\partial\lambda}\right)_{i-1}}{\left\|\frac{\partial\mathbf{x}}{\partial\lambda}\right\|_i \left\|\frac{\partial\mathbf{x}}{\partial\lambda}\right\|_{i-1}} \tag{2.25}$$

τ is the cosine of the angle between two successive tangent vectors, which varies from 1 when $\mathbf{x}$ varies linearly with λ toward zero as $\mathbf{x}$ changes more rapidly with λ, such as when approaching a turning point. An additional means of step control enables the step size to be reduced based on τ:

$$\Delta s = \Delta s \tau^y \tag{2.26}$$

Here, the value of Δs computed by Equation 2.8 is further adjusted by a factor of τ raised to the power y, such that the degree of step control can be adjusted by changing y. (This step control helps generate visually appealing continuation plots. When plotting arclength continuation results, a solution curve is approximated by piecewise linear segments, and basing the step size on τ controls the discontinuity in the slope between those segments, avoiding jagged curves.)

For some problems with complex turning points, difficulties may arise if a converged step takes the solution from a region of minor solution sensitivity to one of large solution sensitivity (to the continuation parameter). This may occur when the parameter value for the step is much closer to the turning point than the previous value, and would be indicated by a small value of τ. Consequently, the next step attempt could be considerably larger than would be required to reach the turning point, and numerous failed steps could occur. For even greater step size control, the tangent factor step limit τ_m can be invoked by setting its input value between zero and one. After each converged step, the tangent vector and τ^y are immediately calculated. If τ^y is less than the specified τ_m, the step is treated as a failure and repeated at smaller step sizes until τ^y increases to at least τ_m. This degree of control should rarely be necessary.

2.1.4 Multi-Parameter Continuation

A new type of parameter continuation made available in LOCA version 1.1 is multi-parameter continuation. This algorithm performs arclength continuation on multiple parameters. When run with one parameter, the result is a curve of solutions just like arclength continuation. On two parameters, the result is a surface of solutions. In general, when looking for steady solutions to n equations as a function of k parameters, the continuation problems is finding a k-dimensional manifold in $(n + k)$-dimensional space.

Multi-parameter continuation is enabled by the publicly-available MF library of Mike Henderson at IBM[10]. This library is now being distributed with LOCA, but documentation on the library and algorithms can be found at the web site `www.research.ibm.com/people/h/henderson`. The MF library creates a tiling of the solution manifold (curve in 1D, surface in 2D, etc.). The algorithm can be visualized in 2D as covering a surface with circles, which generates a tiling of the surface with polygons. The edges of the polygons are the cords connecting the intersection points of two circles.

The MF library needs two main functions from LOCA for the algorithm to proceed. The first is, given a converged solution on the manifold, find an orthonormal basis for the null space. Since there are n equations defining the manifold and $n + k$ unknowns (when including the parameters), there is a k-dimensional null space. This basis can be formed (unless sitting exactly on a bifurcation) using the tangent vectors to the solution with respect to the parameters. This is calculated using the same as a predictor step using Eq. 2.5 for each of the k parameters.

The second function is the projection step, which takes an initial guess for a solution vector and parameter values and solves for a point on the solution manifold. The solution must satisfy the n residual equations $\mathbf{R}$ as well as k arclength constraints. A bordering algorithm is used to solve for this augmented set of equations that is identical to the procedure for arclength continuation in Equations 2.14-2.18, except that now the arclength equation n, the parameter λ, the arclength s and the tangent vector $\mathbf{b}$ are now vectors of length k. The division on Eq. 2.17 is now a simple $k \times k$ linear solve which is performed with LAPACK.

The manifold is tracked in all directions until reaching boundaries set by the user. The default is to give lower and upper bounds for each parameter and solve for the manifold over this rectangular region of parameter space. The user can input different step sizes for different parameters, since they often can have different scalings.

2.2 Bifurcation Tracking Algorithms

In this section we describe the methods implemented in the LOCA library for locating three common instabilities exhibited in nonlinear systems: turning (fold) point, pitchfork, and Hopf bifurcations. Each of the algorithms solves simultaneously for the steady state solution vector $\mathbf{x}$ of length n, the parameter at which the bifurcation occurs, λ, and the null vector $\mathbf{w} = \mathbf{y} + \mathbf{z}i$, which is the eigenvector associated with the eigenvalue that has zero real part. The bifurcations are tracked as a function of a second parameter via simple zero order continuation.

2.2.1 The Turning (Fold) Point Tracking Algorithm

The turning point tracking algorithm uses Newton's method to converge to a turning point (fold point) and simple zero order continuation to track it as a function of a second parameter. At a turning point bifurcation (or fold), there is a single eigenvalue $\gamma = 0$ with associated real-valued null vector $\mathbf{y}$. We use the following formulation of to characterize the turning point [14]:

$$\mathbf{R} = \mathbf{0} \tag{2.27}$$

$$\mathbf{J}\mathbf{y} = \mathbf{0} \tag{2.28}$$

$$\phi \cdot \mathbf{y} = 1 \tag{2.29}$$

Here ϕ is a constant vector. The first vector equation (which is n scalar equations) specifies that the solution be on the steady state solution branch, the second vector equation specifies that a real-valued eigenvector $\mathbf{y}$ exists that corresponds to a zero eigenvalue, and the last scalar equation pins the length of the null vector at length 1 (and removes the trivial solution if $\mathbf{y} = 0$). The vector ϕ is chosen to be the same as the initial guess for $\mathbf{y}$ so that the final equation mimics a L^2 norm yet is linear. This set of $2n + 1$ equations specifies the values of $\mathbf{x}$, $\mathbf{y}$, and λ.

A full Newton method for this system has the form

$$\begin{bmatrix} \mathbf{J} & \mathbf{0} & \frac{\partial \mathbf{R}}{\partial \lambda} \\ \frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}} & \mathbf{J} & \frac{\partial \mathbf{J}\mathbf{y}}{\partial \lambda} \\ \mathbf{0}^T & \phi & 0 \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{y} \\ \Delta \lambda \end{bmatrix} = - \begin{bmatrix} \mathbf{R} \\ \mathbf{J}\mathbf{y} \\ \phi \cdot \mathbf{y} - 1 \end{bmatrix} \tag{2.30}$$

It would be desirable to formulate this system and send it to an efficient linear solver, but this is not practical with many large-scale engineering simulation codes. One hurdle would be the formulation of the matrix $\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{y}$, which requires derivatives not normally calculated in an engineering code and does not lend itself well to efficient numerical differentiation. The second issue is the work involved in determining the sparse matrix storage for iterative linear solvers and partitioning and load balancing for applications sent to parallel computers. The last row and column are not in general sparse and would require frequent global communications. The sparsity of the matrix $\mathbf{J}$ coming from many PDE solution methods (e.g. finite element, finite difference, finite volume) limits communications in the linear solver to only local communications between a processor and a handful (order ten) of its neighbors.

To reduce the effort in implementing the bifurcation algorithms with application codes, bordering algorithms are used to solve the system of equations in (2.27). The linear equations in the Newton iteration for the turning point algorithm (2.27) can be equivalently formulated with four linear solves of the matrix **J** and some simple algebra:

$$\mathbf{J}\mathbf{a} = -\mathbf{R} \tag{2.31}$$

$$\mathbf{J}\mathbf{b} = -\frac{\partial \mathbf{R}}{\partial \lambda} \tag{2.32}$$

$$\mathbf{J}\mathbf{c} = -\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{a} \tag{2.33}$$

$$\mathbf{J}\mathbf{d} = -\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{b} - \frac{\partial \mathbf{J}\mathbf{y}}{\partial \lambda} \tag{2.34}$$

$$\Delta\lambda = \frac{1 - \phi\cdot\mathbf{c}}{\phi\cdot\mathbf{d}} \tag{2.35}$$

$$\Delta\mathbf{x} = \mathbf{a} + \Delta\lambda\mathbf{b} \tag{2.36}$$

$$\Delta\mathbf{y} = \mathbf{c} + \Delta\lambda\mathbf{d} - \mathbf{y} \tag{2.37}$$

The variables **a**, **b**, **c** and **d** are temporary vectors of length n. Each of the 4 linear solves of **J** are performed by the application code, in the same way that this matrix is solved for in Newton iteration (2.2). Work can be saved in the second, third, and fourth solves, by reusing a factorization for a direct solver and the preconditioner for an preconditioned iterative solver. The algorithm requires initial guesses for **x** and λ, which usually come from a steady solution near the turning point as located by an arclength continuation run. LOCA supports the use of an initial null vector from a previous turning point tracking run of the problem if one is provided; otherwise, the initial guess for the null vector and the fixed value of the vector ϕ are both chosen to be scaled versions of the **b** vector from Equation (2.32),

$$\mathbf{y}^{init} = \phi = \frac{\mathbf{b}}{\|\mathbf{b}\|} \tag{2.38}$$

where $\|\mathbf{b}\| = \sqrt{\mathbf{b}\cdot\mathbf{b}}$ This scaling assures that Equation (2.29) is initially satisfied.

The derivatives on the right hand side of Equations (2.32 – 2.34) are all calculated with first order finite differences and directional derivatives. The following formulas are used:

$$\frac{\partial \mathbf{R}}{\partial \lambda} \approx \frac{\mathbf{R}(\mathbf{x}, \lambda + \varepsilon_1) - \mathbf{R}(\mathbf{x}, \lambda)}{\varepsilon_1} \tag{2.39}$$

$$\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{a} \approx \frac{\mathbf{J}(\mathbf{x} + \varepsilon_2 \mathbf{a}, \lambda)\mathbf{y} - \mathbf{J}(\mathbf{x}, \lambda)\mathbf{y}}{\varepsilon_2} \tag{2.40}$$

$$\begin{gathered}\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{b} + \frac{\partial \mathbf{J}\mathbf{y}}{\partial \lambda} \approx \\ \frac{1}{\varepsilon_3}\mathbf{J}(\mathbf{x} + \varepsilon_3 \mathbf{b}, \lambda)\mathbf{y} + \frac{1}{\varepsilon_1}\mathbf{J}(\mathbf{x}, \lambda + \varepsilon_1)\mathbf{y} - \left(\frac{1}{\varepsilon_3} + \frac{1}{\varepsilon_1}\right) \mathbf{J}(\mathbf{x}, \lambda)\mathbf{y}\end{gathered} \tag{2.41}$$

The robustness and accuracy of the algorithm is dependent on the choice of the perturbations ε. The following choices have been found to work well on sample applications for $\delta = 10^{-6}$:

$$\varepsilon_1 = \delta(|\lambda| + \delta) \tag{2.42}$$

$$\varepsilon_2 = \delta(\frac{||\mathbf{x}||}{||\mathbf{a}||} + \delta) \tag{2.43}$$

$$\varepsilon_3 = \delta(\frac{||\mathbf{x}||}{||\mathbf{b}||} + \delta) \tag{2.44}$$

However, some problems have been observed to require an even smaller value, such as $\delta = 10^{-9}$. For this reason, the value to be used is an input to LOCA and must be provided as discussed in Chapter 5.

After convergence to a turning point, a slight modification of simple zero order continuation is used to converge to the next turning point at the next value of a second parameter. The initial guesses for λ and $\mathbf{y}$ are the converged values at the previous turning point, and the constant vector is set to $\phi = \mathbf{y}$. We found more robust convergence when the solution vector $\mathbf{x}$ was perturbed off the singularity by a small random perturbation of relative magnitude 10^{-5}.

2.2.2 The Pitchfork Tracking Algorithm

An algorithm for tracking Pitchfork bifurcations has been developed that requires little modifications to the application code and model. Pitchfork bifurcations occur when a symmetric solution loses stability to a pair of asymmetric solutions. In this algorithm, we require that the user defines the symmetry by supplying a constant vector, ψ, that is antisymmetric with respect to the symmetry being broken. We specify the Pitchfork by the following set of coupled equations:

$$\mathbf{R} + \sigma\psi = \mathbf{0} \tag{2.45}$$

$$\mathbf{J}\mathbf{y} = \mathbf{0} \tag{2.46}$$

$$\langle \mathbf{x}, \psi \rangle = 0 \tag{2.47}$$

$$\phi \cdot \mathbf{y} = 1 \tag{2.48}$$

The variable not defined in the turning point algorithm above is the scalar variable σ that is a slack variable representing the asymmetry in the problem. This additional unknown is associated with the additional equation 2.47, which enforces that the solution vector is orthogonal to the antisymmetric vector. The notation in this equation represents an inner product. For a symmetric model, σ will go to zero at the solution. This system can be shown to be a non-singular formulation of the Pitchfork bifurcation.

We find this to be a convenient alternative to formulating the pitchfork tracking algorithm on half the domain with symmetric boundary conditions for the solution and antisymmetric bounday conditions for the bifurcation problem [5]. For one, the applying of symmetry boundary conditions for some symmetries (e.g. rotational symmetry) can break the sparsity pattern of a Jacobian matrix, and therefore the parallel communication maps. Secondly, it can be inconvenient to transfer solutions from half the domain onto the the whole domain when dealing with large databases for meshes and solutions. With our formulation, the same mesh and boundary conditions are used to locate the pitchfork bifurcation as to follow one of the pitchfork (non-symmetric) branches. However, the gains in efficiency of only solving half the domain make the traditional approach appealing.

There are a few assumptions that were made to ease the implementation of the Pitchfork tracking algorithm, yet can make it trickier to use. First, we require that any odd symmetry in the variables is about zero so that the inner product of the solution vector with the antisymmetric vector is zero. For instance, the cold and hot temperatures in a thermal flow problem should be set at -0.5 and 0.5 instead of 0 and 1. Secondly, our current implementation uses a dot product of the vectors to calculate the inner product $\langle \mathbf{x}, \psi \rangle$; however, this strictly should be an integral over the computational domain. For instance, if the discretization (i.e. finite element mesh) is not symmetric with respect to the symmetry in the PDEs, then the dot product of the solution vector and antisymmetric coefficient vectors would not be zero. We allow the users of the LOCA library to supply the integrated inner product, yet in our applications we have replaced it with the vector dot product. If the mesh is not symmetric with respect to the symmetry in the PDEs that is being broken at the Pitchfork bifurcation, the discretized system will exhibit an imperfect bifurcation. The algorithm presented here will converge to a point that is a reasonable approximation of the Pitchfork bifurcation. However, at this point $\sigma \neq 0$ and therefore we will not have $\mathbf{R} = 0$.

To start the algorithm, we expect the user to supply the vector ψ. The null vector $\mathbf{y}$ has the antisymmetry that we are requiring of ψ. We calculate ψ and the initial guess for $\mathbf{y}$ by first detecting the Pitchfork bifurcation with an eigensolver. The eigenvector associated with the eigenvalue that is passing through zero at the Pitchfork is used for ψ and the initial guess for $\mathbf{y}$. For problems that have multiple pitchfork bifurcations in the same region of parameter space, which is often the case when the system can go unstable to different modes, the pitchfork algorithm can be started multiple times with different ψ vectors to track each pitchfork separately. We choose $\sigma = 0$ as an initial guess and we rarely see it increase past 10^{-10} throughout the iterations.

We have tried a few different vectors for the constant vector ϕ, which is used to get an approximate norm of the null vector $\mathbf{y}$.

As with the turning point algorithm, we use a fully coupled Newton method to converge to the Pitchfork bifurcation and a bordering algorithm to simplify the solution of the Newton iteration. The Newton iteration for this system is

$$\begin{bmatrix} \mathbf{J} & \mathbf{0} & \psi & \frac{\partial \mathbf{R}}{\partial \lambda} \\ \frac{\partial \mathbf{Jy}}{\partial \mathbf{x}} & \mathbf{J} & \mathbf{0} & \frac{\partial \mathbf{Jy}}{\partial \lambda} \\ \frac{\partial \langle \mathbf{x}, \psi \rangle}{\partial x} & \mathbf{0} & \mathbf{x}^T & 0 \\ \mathbf{0} & \phi & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{y} \\ \Delta \sigma \\ \Delta \lambda \end{bmatrix} = - \begin{bmatrix} \mathbf{R} \\ \mathbf{Jy} \\ \langle \mathbf{x}, \psi \rangle \\ \phi \cdot \mathbf{y} - 1 \end{bmatrix} \tag{2.49}$$

This system can be solved using a mathematically (but not numerically) equivalent bordering algorithm:

$$\mathbf{J}\mathbf{a} = -\mathbf{R} \tag{2.50}$$

$$\mathbf{J}\mathbf{b} = -\frac{\partial \mathbf{R}}{\partial \lambda} \tag{2.51}$$

$$\mathbf{J}\mathbf{c} = -\psi \tag{2.52}$$

$$\mathbf{J}\mathbf{d} = -\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{a} \tag{2.53}$$

$$\mathbf{J}\mathbf{e} = -\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{b} - \frac{\partial \mathbf{J}\mathbf{y}}{\partial \lambda} \tag{2.54}$$

$$\mathbf{J}\mathbf{f} = -\frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}}\mathbf{c} \tag{2.55}$$

$$\Delta\sigma = -\sigma + \frac{(\langle \mathbf{x}, \psi\rangle + \langle \mathbf{b}, \psi\rangle)\phi \cdot \mathbf{e} + \langle \mathbf{b}, \psi\rangle(1 - \phi \cdot \mathbf{d})}{\langle \mathbf{b}, \psi\rangle\phi \cdot \mathbf{b} - \langle \mathbf{c}, \psi\rangle\phi \cdot \mathbf{e}} \tag{2.56}$$

$$\Delta\lambda = \frac{1 - \phi \cdot \mathbf{d} - \phi \cdot \mathbf{f}(\Delta\sigma + \sigma)}{\phi \cdot \mathbf{e}} \tag{2.57}$$

$$\Delta\mathbf{x} = \mathbf{a} + \Delta\lambda\mathbf{b} + (\Delta\sigma + \sigma)\mathbf{c} \tag{2.58}$$

$$\Delta\mathbf{y} = \mathbf{d} + \Delta\lambda\mathbf{e} + (\Delta\sigma + \sigma)\mathbf{f} - \mathbf{y} \tag{2.59}$$

This algorithm has 6 temporary vectors $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathrm{and}\mathbf{f})$, each of which is the result of a linear solve with the same matrix $\mathbf{J}$. The vector $\mathbf{c}$ does not vary throughout the Newton iteration so this solve is only performed on the first Newton iteration of each solve to a pitchfork bifurcation. The right hand sides of these 6 linear systems are mostly the same as for the turning point algorithm, and so we reuse the same routines (and therefore the same differencing schemes and perturbations) that were presented above (equations 2.39 and 2.42).

2.2.3 The Hopf Point Tracking Algorithm

The algorithm for tracking Hopf bifurcations is characterized by a complex pair of eigenvalues that have a zero real part. The purely imaginary eigenvalues can be written $\gamma = \pm\omega i$ with complex eigenvectors $\mathbf{w} = \mathbf{y} + \mathbf{z}i$. By separating the real and complex parts, the following set of equations can be used to describe a Hopf bifurcation in real arithmetic [15, 16],

$$\mathbf{R} = \mathbf{0} \tag{2.60}$$
$$\mathbf{J}\mathbf{y} = -\omega\mathbf{B}\mathbf{z} \tag{2.61}$$
$$\mathbf{J}\mathbf{z} = \omega\mathbf{B}\mathbf{y} \tag{2.62}$$
$$\phi \cdot \mathbf{y} = 1 \tag{2.63}$$
$$\phi \cdot \mathbf{z} = 0 \tag{2.64}$$

where $\mathbf{B}$ is the matrix of coefficients of time dependent terms. This system of $3N + 2$ equations and unknowns solves for the the solution vector [$\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ω, and λ]. The first vector equation specifies that we are on the solution branch, the next two equations specify that we are at place where there is a purely imaginary eigenvalue, and the last two scalar equations set the phase and amplitude of the eigenvectors (which are otherwise free). The Hopf bifurcation tracking algorithm is the complex valued equivalent to the real valued turning point tracking algorithm. Setting ω to zero yields two redundant turning point tracking algorithms. Also note that the same Hopf bifurcation can admit a second solution to this system of equations at ($\mathbf{x}$, $\mathbf{y}$, -bfz, $-\omega$, λ).

One Newton iteration for the fully coupled solution of this system is the linear system,

$$\begin{bmatrix} \mathbf{J} & \mathbf{0} & \mathbf{0} & 0 & \frac{\partial \mathbf{R}}{\partial \lambda} \\ \frac{\partial \mathbf{J}\mathbf{y}}{\partial \mathbf{x}} + \omega \frac{\partial \mathbf{B}\mathbf{z}}{\partial \mathbf{x}} & \mathbf{J} & \omega\mathbf{B} & \mathbf{B}\mathbf{z} & \frac{\partial \mathbf{J}\mathbf{y}}{\partial \lambda} + \omega \frac{\partial \mathbf{B}\mathbf{z}}{\partial \lambda} \\ \frac{\partial \mathbf{J}\mathbf{z}}{\partial \mathbf{x}} - \omega \frac{\partial \mathbf{B}\mathbf{y}}{\partial \mathbf{x}} & -\omega\mathbf{B} & \mathbf{J} & -\mathbf{B}\mathbf{y} & \frac{\partial \mathbf{J}\mathbf{z}}{\partial \lambda} - \omega \frac{\partial \mathbf{B}\mathbf{y}}{\partial \lambda} \\ \mathbf{0} & \phi & \mathbf{0} & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \phi & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\mathbf{y} \\ \Delta\mathbf{z} \\ \Delta\omega \\ \Delta\lambda \end{bmatrix} = - \begin{bmatrix} \mathbf{R} \\ \mathbf{J}\mathbf{y} + \omega\mathbf{B}\mathbf{z} \\ \mathbf{J}\mathbf{z} - \omega\mathbf{B}\mathbf{y} \\ \phi \cdot \mathbf{y} - 1 \\ \phi \cdot \mathbf{z} \end{bmatrix} \tag{2.65}$$

In this derivation we have allowed for $\frac{\partial \mathbf{B}}{\partial \mathbf{x}} \neq 0$ and $\frac{\partial \mathbf{B}}{\partial \lambda} \neq 0$. While in many situations these terms can be neglected, the matrix $\mathbf{B}$ can depend on the solution vector through dependence of the inertial coefficients (e.g. density and heat capacity) on the local state vector. The matrix $\mathbf{B}$ will depend on the parameter very strongly when λ is a geometric parameter that moves the mesh locations.

Again we solve this linear system by a bordering algorithm that breaks it into simpler linear solves. It is not possible to solve this system by solves of just the matrix $\mathbf{J}$, but also requires solves of the complex matrix $\mathbf{J} + \omega\mathbf{B}i$. The bordering algorithm for the Newton iteration of the Hopf tracking algorithm, written in terms of real-valued variables, is,

$$\mathbf{Ja} = -\mathbf{R} \tag{2.66}$$

$$\mathbf{Jb} = -\frac{\partial \mathbf{R}}{\partial \lambda} \tag{2.67}$$

$$\begin{bmatrix} \mathbf{J} & \omega\mathbf{B} \\ -\omega\mathbf{B} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix} = \begin{bmatrix} \mathbf{Bz} \\ -\mathbf{By} \end{bmatrix} \tag{2.68}$$

$$\begin{bmatrix} \mathbf{J} & \omega\mathbf{B} \\ -\omega\mathbf{B} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{e} \\ \mathbf{f} \end{bmatrix} = \begin{bmatrix} -(\frac{\partial \mathbf{Jy}}{\partial \mathbf{x}} + \omega \frac{\partial \mathbf{Bz}}{\partial \mathbf{x}})\mathbf{a} \\ -(\frac{\partial \mathbf{Jz}}{\partial \mathbf{x}} - \omega \frac{\partial \mathbf{By}}{\partial \mathbf{x}})\mathbf{a} \end{bmatrix} \tag{2.69}$$

$$\begin{bmatrix} \mathbf{J} & \omega\mathbf{B} \\ -\omega\mathbf{B} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{g} \\ \mathbf{h} \end{bmatrix} = \begin{bmatrix} -(\frac{\partial \mathbf{Jy}}{\partial \mathbf{x}} + \omega \frac{\partial \mathbf{Bz}}{\partial \mathbf{x}})\mathbf{b} - (\frac{\partial \mathbf{Jy}}{\partial \lambda} + \omega \frac{\partial \mathbf{Bz}}{\partial \lambda}) \\ -(\frac{\partial \mathbf{Jz}}{\partial \mathbf{x}} - \omega \frac{\partial \mathbf{By}}{\partial \mathbf{x}})\mathbf{b} - (\frac{\partial \mathbf{Jz}}{\partial \lambda} - \omega \frac{\partial \mathbf{By}}{\partial \lambda}) \end{bmatrix} \tag{2.70}$$

$$\Delta\lambda = \frac{(\phi \cdot \mathbf{c})(\phi \cdot \mathbf{f}) - (\phi \cdot \mathbf{e})(\phi \cdot \mathbf{d}) + (\phi \cdot \mathbf{d})}{(\phi \cdot \mathbf{d})(\phi \cdot \mathbf{g}) - (\phi \cdot \mathbf{c})(\phi \cdot \mathbf{h})} \tag{2.71}$$

$$\Delta\omega = \frac{(\phi \cdot \mathbf{h})\Delta\lambda + (\phi \cdot \mathbf{f})}{\phi \cdot \mathbf{d}} \tag{2.72}$$

$$\Delta\mathbf{x} = \mathbf{a} + \Delta\lambda\mathbf{b} \tag{2.73}$$

$$\Delta\mathbf{y} = \mathbf{e} + \Delta\lambda\mathbf{g} - \Delta\omega\mathbf{c} - \mathbf{y} \tag{2.74}$$

$$\Delta\mathbf{z} = \mathbf{f} + \Delta\lambda\mathbf{h} - \Delta\omega\mathbf{d} - \mathbf{z} \tag{2.75}$$

This algorithm has 8 temporary vectors **a** through **h**, which are solved with two solves of the **J** matrix and three solves of the $2n \times 2n$ matrix $\begin{bmatrix} \mathbf{J} & \omega\mathbf{B} \\ -\omega\mathbf{B} & \mathbf{J} \end{bmatrix}$. This algorithm differs from the turning point and pitchfork tracking algorithms which only require solution of the steady state Jacobian **J**, a routine which codes using Newton's method already have. Since the location of the nonzeros in the sparse matrix **B** are a subset of those for the matrix **J**, a parallel iterative solver for the $2n \times 2n$ matrix can use the same local communication maps as used for solces of **J**. An algorithm for solving complex matrix equations with a real-valued sparse iterative solver has recently been published [17] and implemented in the Komplex extension to the Aztec library of preonditioned iterative Krylov solvers. This algorithm also requires the formulation of the **B** matrix, a routine which a code performing linear stability analysis of equation 2.89 will already have.

To initialize the routine, we assume that an initial Hopf bifurcation has been detected with an eigensolver, by having the real part of a complex pair of eigenvalues pass through zero with successive steps in the parameter. This gives good starting values for all the unknowns in the Hopf tracking algorithm.

2.2.4 The Phase Transition Tracking Algorithm

A phase transition occurs when two different phases (i.e. liquid and vapor) can coexist under the exact same thermodynamic conditions and have the same free energy. The phase transition tracking algorithm uses Newton's method to converge to a parameter value and two solution vectors which have equal free energies. Simple zero order continuation tracks the phase transition as a function of a second parameter. Use of this algorithm in the Tramonto code usually has a partial pressure as the first 'bifurcation' parameter, and temperature as the second continuation parameter. We characterize the phase transition by the following set of $2n + 1$ equations:

$$\mathbf{R}(\mathbf{x}_1, \lambda) = \mathbf{0} \tag{2.76}$$

$$\mathbf{R}(\mathbf{x}_2, \lambda) = \mathbf{0} \tag{2.77}$$

$$G = \Omega(\mathbf{x}_1, \lambda) - \Omega(\mathbf{x}_2, \lambda) = 0 \tag{2.78}$$

Here $\mathbf{x}_1$ and $\mathbf{x}_2$ are two solution vectors and Ω is the expression for the free energy.

A full Newton method for this system has the form

$$\begin{bmatrix} \mathbf{J}_1 & \mathbf{0} & \frac{\partial \mathbf{R}_1}{\partial \lambda} \\ \mathbf{0} & \mathbf{J}_2 & \frac{\partial \mathbf{R}_2}{\partial \lambda} \\ \frac{\partial \Omega_1}{\partial \mathbf{x}_1} & -\frac{\partial \Omega_2}{\partial \mathbf{x}_2} & \frac{\partial G}{\partial \lambda} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_1 \\ \Delta \mathbf{x}_2 \\ \Delta \lambda \end{bmatrix} = - \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \\ G \end{bmatrix} \tag{2.79}$$

Here the subscript i on the variable $\mathbf{R}$, $\mathbf{J}$, and Ω represent evaluation with solution vector $\mathbf{x}_i$.

As in the bifurcation tracking algorithms above, a bordering algorithm is used to solve the system of equations in (2.76). The linear equations in the Newton iteration for the turning point algorithm (2.76) can be equivalently formulated with two linear solves of the matrix $\mathbf{J}_1$, two of the matrix $\mathbf{J}_2$, some simple algebra:

$$\mathbf{J}_1\mathbf{a} = -\mathbf{R}_1 \tag{2.80}$$

$$\mathbf{J}_1\mathbf{b} = -\frac{\partial \mathbf{R}_1}{\partial \lambda} \tag{2.81}$$

$$\mathbf{J}_2\mathbf{c} = -\mathbf{R}_2 \tag{2.82}$$

$$\mathbf{J}_2\mathbf{d} = -\frac{\partial \mathbf{R}_2}{\partial \lambda} \tag{2.83}$$

$$\Delta\lambda = -\frac{G + (\frac{\partial \Omega_1}{\partial \mathbf{x}_1}\mathbf{a} - \frac{\partial \Omega_2}{\partial \mathbf{x}_2}\mathbf{c})}{\frac{\partial G}{\partial \lambda} + (\frac{\partial \Omega_1}{\partial \mathbf{x}_1}\mathbf{b} - \frac{\partial \Omega_2}{\partial \mathbf{x}_2}\mathbf{d})} \tag{2.84}$$

$$\Delta\mathbf{x}_1 = \mathbf{a} + \Delta\lambda\mathbf{b} \tag{2.85}$$

$$\Delta\mathbf{x}_2 = \mathbf{c} + \Delta\lambda\mathbf{d} \tag{2.86}$$

The variables $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{d}$ are temporary vectors of length n. The quantities in equation 2.84 with the $\frac{\partial \Omega}{\partial \mathbf{x}}$ terms can be quickly approximated with directional derivatives, such as,

$$\frac{\partial \Omega_1}{\partial \mathbf{x}_1}\mathbf{a} - \frac{\partial \Omega_2}{\partial \mathbf{x}_2}\mathbf{c} \approx \frac{G(\mathbf{x}_1 + \varepsilon\mathbf{a}, \mathbf{x}_2 + \varepsilon\mathbf{c}) - G(\mathbf{x}_1, \mathbf{x}_2)}{\varepsilon} \tag{2.87}$$

where the perturbation is chosen as

$$\varepsilon = \delta\sqrt{\frac{\mathbf{x}_1 \cdot \mathbf{x}_1}{\mathbf{a} \cdot \mathbf{a} + \delta} + \frac{\mathbf{x}_2 \cdot \mathbf{x}_2}{\mathbf{c} \cdot \mathbf{c} + \delta}} \tag{2.88}$$

with $\delta = 10^{-6}$.

The algorithm requires initial guesses for $\mathbf{x}_1$, $\mathbf{x}_2$ and λ, which usually come from picking two solutions from near a phase transition from an arclength continuation run. As with the bifurcation tracking algorithms, the strength of the phase transition algorithm is not locating the first occurrence, but the automatic tracking of the phase transition with respect to a second parameter.

2.3 Linearized Stability Analysis

The stability of the steady solutions to small perturbations can be ascertained through linearized stability analysis. Linearization of the transient equations around the steady state lead to a generalized eigenvalue problem of the form

$$\mathbf{J}\mathbf{w} = \gamma \mathbf{B}\mathbf{w}, \tag{2.89}$$

where $\mathbf{B}$ is the matrix of coefficients of time dependent terms, γ is an eigenvalue of the system (generally complex), and $\mathbf{w}$ is the associated eigenvector, which can be written in terms of real values vectors $\mathbf{w} = \mathbf{y} + \mathbf{z}i$. The linear theory shows that a perturbation in the solution vector in the direction of $\mathbf{w}$ will evolve in time (t) with amplitude $e^{\gamma t}$. It is clear from this that a solution will be linearly stable if all eigenvalues satisfy $\text{Real}(\gamma) < 0$, and therefore decay in time. If any eigenvalue has positive real part, then perturbations with any component in the direction of the associated eigenvector will grow exponentially, and the steady state solution is deemed unstable. A system loses stability, and experiences a bifurcation, when a stable steady state solution branch, as parameterized by a system parameter λ, passes through a point where $\text{Real}(\gamma) = 0$.

We have developed a robust linearized stability analysis capability for large scale problems that accurately approximates leading eigenvalues of the system in equation 2.89. This method is based on the Cayley transformation to make the eigenvalues of interest have largest magnitude, and then uses the implicitly restarted Arnoldi method of the ARPACK and P_ARPACK libraries [1, 2]. Details of the methods are found in [3] and benchmarking and application of the methods to incompressible flows are found in [4], [18], [19].

Chapter 3

LOCA Software and Implementation

The LOCA library is "C" code that performs parameter continuation and bifurcation tracking for an application code. LOCA is designed for use around codes that use Newton's method to locate steady-state solutions. The algorithms in LOCA (presented in the previous Chapter) are designed to be as non-invasive as possible, and make use of the application code's own routines. The algorithms in LOCA are programmed to wrapper routines which must be filled in by the user. For the most part, these routines will be already available in a code performing Newton's methods: residual fills, Jacobian matrix fills, and linear solves of the Jacobian matrix. (More involved development is usually needed to enable the Hopf bifurcation tracking routine, where a Mass matrix and complex matrix solve capabilities are required.)

In this chapter, we will first detail the directory structure of the LOCA code, and give details on how to compile the library, the test_driver, and how to run the test suite. The second section gives the recipe for implementing LOCA in a new application code. Items **3** and **4** of this recipe include numerous sub-tasks, with details in the following two Chapters of this manual.

3.1 LOCA Directory Structure

The LOCA software has the directory structure shown in Figure 3.1, and consists of 6 directories under the `Loca` top directory. The most important of these is the `src` directory of source code with `loca` and `test_driver` subdirectories. In `src/loca` is the source code for the LOCA library, which are the files that are independent of the application code. These files can be compiled once per platform to create a `libloca.a` library. The Fortran files are only needed when the user wants to enable the linear stability analysis (eigenvalue) capability, and require linking to the ARPACK library. In `src/test_driver` is the file `loca_interface.c`, which is the file that must be extensively modified to interface LOCA with a new application. Here it is filled in to interface the test_driver problem with LOCA. The test_driver is the remaining files in this directory, and solves for the stability of a beam conveying fluid (the garden hose problem).

The `build` directory is for compiling. The makefiles are set up to require three environment variables to be set:

1. setenv LOCA_HOME *path-to-top-LOCA-directory*

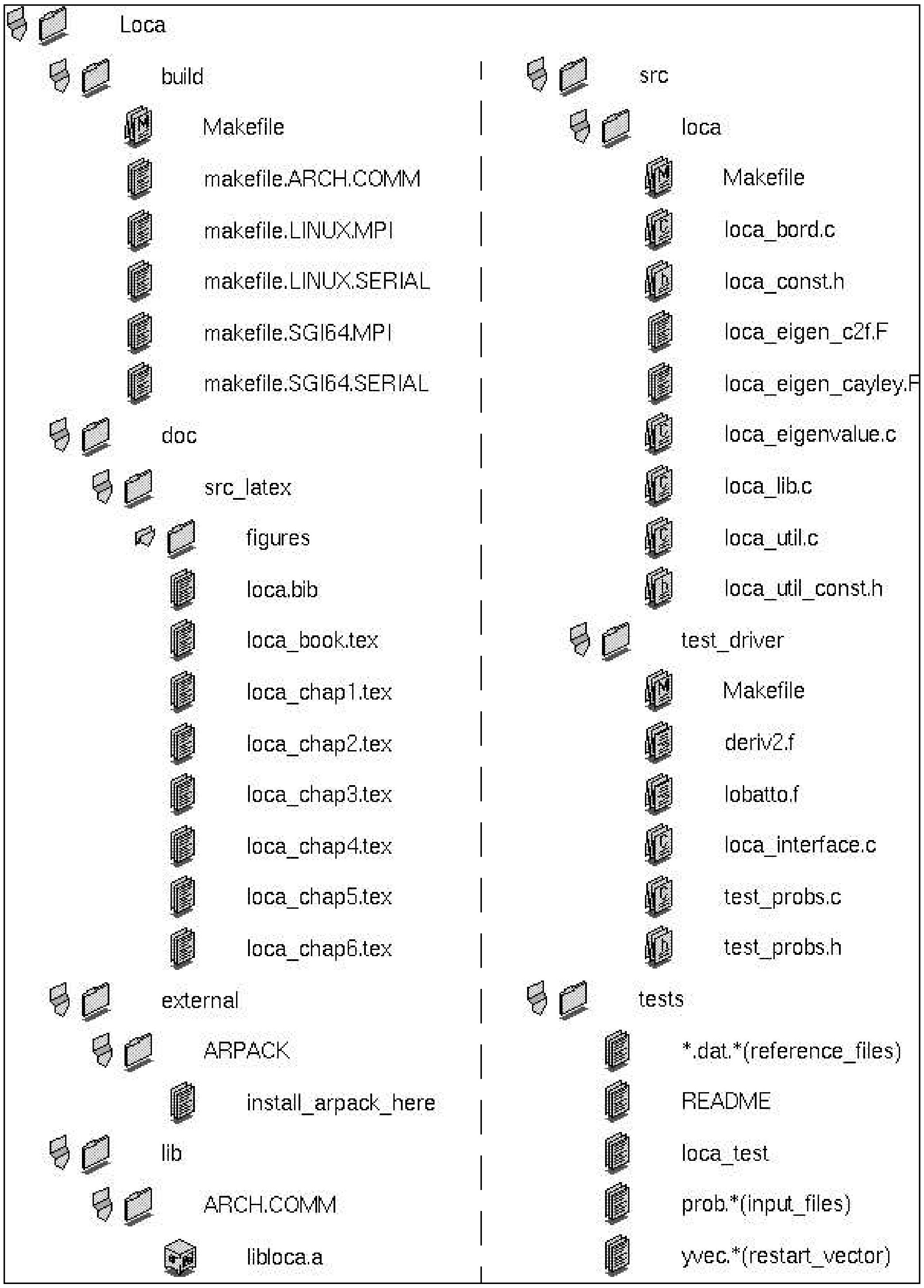
2. setenv LOCA_ARCH *architecture-name*

3. setenv LOCA_COMM *communication-type*

LOCA_ARCH is used to organize the compilation of LOCA for different platforms, (e.g. SGI64, LINUX, SOLARIS) and LOCA_COMM is usually set to SERIAL or MPI to distinguish the communication type compiled for. When compiling, the makefile automatically looks for compiling information from a file named makefile.LOCA_ARCH.LOCA_COMM, and the object code and the library are placed in subdirectories (withing the `src` and `lib` directories) with the name LOCA_ARCH.LOCA_COMM.

Within the machine dependent makefiles, the paths to compilers, lapack, blas, and arpack libraries must be set. A define flag, `EIGENVALUE_DEFINES`, should be set equal to `EIGEN_SERIAL` or `EIGEN_PARALLEL` if the linear stability analysis capability is desired. These flags control the compilation of the three files with names `src/loca_eigen*`.

Similarly, the environment variable `LOCA_MF` needs to be set if the multi-parameter continuation capability is desired. This flag will cause the com-



```
Loca
    build
        Makefile
        makefile.ARCH.COMM
        makefile.LINUX.MPI
        makefile.LINUX.SERIAL
        makefile.SGI64.MPI
        makefile.SGI64.SERIAL
    doc
        src_latex
            figures
            loca.bib
            loca_book.tex
            loca_chap1.tex
            loca_chap2.tex
            loca_chap3.tex
            loca_chap4.tex
            loca_chap5.tex
            loca_chap6.tex
    external
        ARPACK
            install_arpack_here
    lib
        ARCH.COMM
            libloca.a
    src
        loca
            Makefile
            loca_bord.c
            loca_const.h
            loca_eigen_c2f.F
            loca_eigen_cayley.F
            loca_eigenvalue.c
            loca_lib.c
            loca_util.c
            loca_util_const.h
        test_driver
            Makefile
            deriv2.f
            lobatto.f
            loca_interface.c
            test_probs.c
            test_probs.h
    tests
        *.dat.*(reference_files)
        README
        loca_test
        prob.*(input_files)
        yvec.*(restart_vector)
```

Figure 3.1: Directory structure of the LOCA code:, with directories `build`, `doc`, `external`, `lib`, `src` and `tests`. The `src/loca` directory holds the source code for the LOCA library, while the `src/test_driver` contains the interface file `loca_interface.c` (here filled out to work with a test code) which must be extensively modified to link LOCA to a new application code. In version 1.1, the MF library has been added in `external/mf` and the inpilation of the source file `src/loca/loca_mf.c` and linking to the compiled MF library in directory `external/mf`.

After setting the three environment variables, the code can be compiled from within the `build` directory by typing one of the following:

1. `gmake loca` (compiles just src/loca to form libloca.a)

2. `gmake test_driver` (compiles just src/test_driver)

3. `gmake` (compiles both loca and test_driver)

The makefile also recognizes `clean` and `clobber` targets.

`external` is the directory to put the ARPACK library, and now contains the MF library as well. It is expected that the compiled arpack libraries will reside in this directory, and the paths to the arpack libraries in the makefiles will point to here. `doc` is the documentation directory, which includes latex code for this manual. `lib` is the directory where compiled `libloca.a` libraries are placed, within subdirectories that are created from the name of the environment variables: LOCA_ARCH.LOCA_COMM.

The `tests` directory contains an automatic test script for running the test_driver code and checking against reference output files. The test script currently runs ten different input files (named `prob`) and the reference output files are also in this director, and contain the string `dat` in the name. The test script is executed by running:

```
perl loca_test
```

For more information on the physical problem being solved by the test_driver application, please see the paper entitled "A Multi-parameter, Numerical Stability Analysis of a Standing Cantelever Conveying Fluid" by Bou-Rabee, Romero, and Salinger, which can be accessed from the LOCA web page: `www.cs.sandia.gov/LOCA`.

Almost all of the code development needed to implement LOCA around a new application code occurs in the file `loca_interface.c`. All the other LOCA files should in theory remain unmodified. The other files include the header file `loca_const.h`, which contains definitions of flags (e.g. `#define ARC_LENGTH_CONTINUATION 2`) used in the library, and definitions of the `con` structure of structures which holds all the problem specific information.

The file `loca_lib.c` contains the parameter continuation loop, including step control logic and predictor calculations. The file `loca_bord.c` contains the bordering algorithms that get called within each Newton iteration. The file `loca_mf.c`, new in version 1.1, contains the interface between LOCA and the MF multi-parameter continuation library. The file `loca_util.c` contains utility routines needed by the rest of the codes, such as vector copies and dot products.

3.2 Implementation Recipe

0. Modularize your code. LOCA needs to be able to call the nonlinear solver (Newton) iteration loop. If your nonlinear solver includes code before the Newton loop begins, such as reading in a mesh or creating an initial guess for a solution vector, these need to be separated into another routine. LOCA also needs to call a residual (right hand side) fill routine and a Jacobian matrix fill routine. Make sure your code still works after these changes.

1. Put in a call to `do_loca()`. This will cede control to the LOCA library. This call needs to come *a*fter all initialization is done, including generating an initial guess for the solution vector. This is the same place where the Newton iteration loop is called for steady state calculations and a time stepper is called for transient problems. You can choose the arguments you want to send to do_loca(), which are usually the same as you send to the nonlinear solver. A piece of code might look like this:

```
if (method==STEADY)
    newton_solver(arglist);
else if (method==TRANSIENT)
    time_integration(arglist);
else if (method==CONTINUATION)
    do_loca(arglist);
```

In the file `loca_interface.c`, the first routine is `do_loca()`. Fill in the argument list as you have chosen it. Also, any header files that are included at the top of the nonlinear solver function should also be included at the top of `loca_interface.c`. Add a prototype for the call to `do_loca()`.

2. A hook is needed in the middle of your Newton iteration to invoke the bordering algorithms. (This is not needed if all you want to use is ZERO_ORDER_CONTINUATION and FIRST_ORDER_CONTINUATION.) First, add an argument to your nonlinear solver routine called `void *con_ptr`, and pass it a value of `NULL` wherever you usually call your nonlinear solver. Define an integer variable called `continuation_converged` in your nonlinear solver. This will be a flag indicating whether or not the part of the Newton iteration performed in the bordering algorithms is converged.

In the Newton loop, after the linear system has been solved for the update vector, but before the update is added to the current value of the solution vector, put in the following code:

```
if (con_ptr==NULL)
    contination_converged=TRUE;
else {
    continuation_converged=FALSE;
    continuation_converged =
       continuation_hook(x, delta_x, con_ptr, rtol, atol);
}
```

Here `double *x` is the current solution vector, `double *delta_x` is the Newton update to the solution vector, `double rtol` and `double atol` are tolerance to determine convergence of the update (with `reltol`=10^{-2} and `abstol`=10^{-6} as typical values).

Note that the vector `delta_x` gets modified in the bordering algorithms. The next lines after this hook are usually the update loop (`x[i] = x[i] + delta_x[i]`).

Add the condition that `continuation_converged==TRUE` to the convergence criteria for the Newton iteration.

Finally, make sure your nonlinear solver returns the number of Newton iterations taken before convergence, whether in the argument list or as a return value.

3. The next part of the implementation, and the most time consuming, is the filling in of all the needed wrapper routines in the file `loca_interface.c`. Wrapper routines (which include `linear_solver_conwrap`, `assign_parameter_conwrap`, and `solution_output_conwrap`) provide LOCA access to routines in the application code. A complete list of the wrapper routines is given in Chapter 4, each with a description of the required functionality, argument lists, return values, and a list of which continuation strategies use it. (For example, wrapper routines dealing with the mass matrix are only required for Hopf tracking and eigenvalue calculations.)

The algorithms in LOCA access the solution vector and residual vector, and assumes that these quantities are continuous in memory on each processor. They can therefore be identified by a pointer to the beginning of the array and a length. (Future version of LOCA will relax these assumptions and allow the user to supply routines for all vector operations.) LOCA is however blind to the matrix and all other attributes of the problem, such as the mesh. Because of this, the wrapper routines have minimal argument lists. For instance, when LOCA needs the application code to multiply the Jacobian matrix times a vector `x` to render a vector `y`, the call is simply

```
matvec_mult_conwrap(x,y);
```

The function `void matvec_mult_conwrap(double *x, double *y);` needs to know about the matrix, but doesn't receive information about it through the argument lists. This is accomplished by making all information that is needed in the wrapper routines – but which is not passed through LOCA – *g*lobal to the file `loca_interface.c`.

Since the steady-state nonlinear solver of the application code must have information about the matrix (address, sparsity, etc.), this same information will be available in the routine `do_loca` at the top of the file `loca_interface.c`. This is because step **1** above gave `do_loca` the same argument list and include files as the nonlinear solver. To access the matrix information in the wrapper routines below in the same file, this information needs to be made global to the file. If the information is already in a header file included at the top of the file, then it is already global to the file. If it is passed in through the argument list to `do_loca`, then it needs to be made global to the file by the `passdown` structure. The `passdown` structure is not used explicitly by the LOCA library but is made available for the user to pass parameters to the wrapper routines.

For example, if the variable `double *J` is passed to `do_loca`, and is needed by the matrix-vector multiply wrapper routine (`matrix_residual_fill_conwrap`), the structure element `passdown.J` of the same variable type can be defined. Then the following line of code in the routine `do_loca` can be added:

```
passdown.J = J;
```

This variable can now be accessed anywhere in the file as `passdown.J`.

Following this example further, let's assume a users code uses the following prototype to fill a matrix:

```
fill_residual(*x, *rhs, *J, *options)
```

Where `x` is the solution vector used to evaluate the residual and matrix fill, `*rhs` is the residual vector, `*J` is the matrix storage structure, and `*options` contains a list of options needed by the users code during the matrix fill. The user would add the required objects to the `passdown` structure in the file `loca_interface.c`:

```
struct passdown_struct {
    MATRIX *J;
    int *options;
} passdown;
```

Then the `void matrix_residual_fill_conwrap` would use the `passdown` structure information in the call to the users fill function:

```
void matrix_residual_fill_conwrap(double *x, double *rhs,
                                              int matflag) {
    switch (matflag) {
        case RHS_ONLY:
            passdown.options->fill_flag = RESIDUAL;
            break;
        case MATRIX_ONLY:
            passdown.options->fill_flag = MATRIX;
            break;
        case RHS_MATRIX:
            passdown.options->fill_flag = MATRIXRESIDUAL;
            break;
    }
    matrix_fill(x, rhs, passdown.J,
            passdown.options);
}
```

4. Set the elements of the `con` structures in the routine `do_loca`. The `con` structure contains 8 structures which the user must load with parameters and flags that control the continuation runs. For instance, the `con.general_info` structure contains storage space for the solution vector, the parameter, and the problem size. These structures are defined in the file `loca_const.h`. A description of each of the elements of these structure is given in Chapter 5.

Most of the values in these structures need to be set through an input file and not by hardwiring them in source code. Therefore, each application code needs to read in the relevant parameters to control the continuation algorithms, pass that information to the `do_loca` routine, and then assign them to the appropriate `con` structures. This task includes determining which quantities needed by LOCA are already provided in the existing application code and which new ones need to be added. For those which already exist (such as a solution vector), it is only necessary to pass the input values in to LOCA and assign them to the relevant LOCA structures. For inputs which must be added (such as a step control variable), the following procedure is recommended to provide LOCA with its required inputs.

a Create and define a C structure to hold all LOCA inputs to be added to the code. Ensure that there is an input to indicate that LOCA will be used.

b Provide a subroutine to allocate this structure and call it prior to parsing the code's input file.

c Add the necessary lines to the input file and add routines to read them in to the code's input parser. Some of these values should be assigned defaults if not included in the input file, following the examples of existing inputs.

d If the application code runs in parallel, use its established procedure to communicate these inputs to each processor.

e Include this structure in the argument list for `do_loca` and assign each value to the relevant LOCA structure entries.

The template `loca_interface.c` file for running the example problems shows an example of how this is done. Also included with the example problems is an input file reader that can be adapted for use with a new application code.

Chapter 4

LOCA's Wrapper Routines

The wrapper routines provide the interface between LOCA and the user's code. All wrapper routines are contained in the file `loca_interface.c`. This file (and only this file) must be extensively modified for every new code that uses the library. It consists of `do_loca`, the top level routine called by the application code, a `passdown` structure for user supplied arguments and $\sim$15 wrapper routines, which must be filled for your specific code. Some wrapper routines are specific to certain tracking algorithms. Therefore, the user need only complete the wrapper routines that are required for the particular tracking algorithms they wish to utilize. In this chapter we describe the requirements of each wrapper routine, including the input, output, and return value, as well as a list of which continuation methods require that wrapper. For instance, implementing LOCA without enabling the Hopf tracking capability saves the need to fill several of the wrapper routines.

The wrapper routines make extensive use of the `passdown` structure. This structure is used to make information available to wrapper routines without passing that information through LOCA. The `passdown` structure is not used explicitly by the LOCA library but is made available for the user to pass parameters to the wrapper routines. The structure provides a means to pass objects to the wrapper routines without forcing the user to declare the objects globally. The `passdown` structure and an example of its use are described in section 3.2.

4.1 nonlinear_solver_conwrap

`int nonlinear_solver_conwrap(double *x, void *con_ptr, int step_num);`

Description: Put a call to your Newton method nonlinear solver here, using `double *x` as the initial solution vector. The argument `void *con_ptr` needs to be added to your nonlinear solver argument list (see item 2 of the recipe in section 3.2). You can also use the nonlinear solver in the example version of `loca_interface.c`, which does a simple Newton, instead of your own.

On Input:

`x` Current guess of the solution vector.

`con_ptr` Pointer to the `con` structure, cast to a pointer to void. This pointer equals `NULL` for zero order and first order continuation, and not otherwise. When not equal to `NULL`, the bordering algorithms are invoked.

`step_num` Continuation step number, which may be used by some nonlinear solvers.

On Output:

`x` is the updated solution vector for the finished nonlinear iteration.

Return Value: `[int]` The number of Newton iterations taken for convergence needs to be returned, with any negative number signaling a failed Newton step. The number of Newton iterations is used to decide the size of the next continuation step (see Eq. 2.8).

This wrapper required for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION

This wrapper not needed for: RSQP_OPTIMIZATION

4.2 linear_solver_conwrap

`int linear_solver_conwrap(double *x, int jac_flag, double *tmp);`

Description: Put the call to your linear solver here. This is always proceeded by a matrix fill call.

On Input:

`x` is the current guess of the update vector to the linear system.

`jac_flag` is the flag indicating the status of the Jacobian so that preconditioners can be used:
`NEW_JACOBIAN`: recalculate preconditioner.
`OLD_JACOBIAN`: reuse preconditioner.
`SAME_BUT_UNSCALED_JACOBIAN`: Must rescale the matrix and can reuse preconditioner. This happens when the matrix has been recalculated at the same conditions as before.
`CHECK_JACOBIAN`: Jacobian may be scaled or unscaled depending on whether it was reassembled or previously saved and recopied; rescale only if necessary.

`tmp` is the work space array with the same length as x. This only comes in allocated for the `SAME_BUT_UNSCALED_JACOBIAN` option; otherwise it is set to `NULL` and should not be accessed.

On Output:

`x` is the solution vector for the linear solve.

Return Value: `[int]` flag that indicates the success of the linear solve. Any negative number signals a failed Newton step.

This wrapper required for: FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION, RSQP_OPTIMIZATION, EIGENVALUE_CALCULATIONS

This wrapper not needed for: ZERO_ORDER_CONTINUATION

4.3 komplex_linear_solver_conwrap

```
int komplex_linear_solver_conwrap(double *c, double *d, int
jac_flag, double *omega, double *tmp);
```

Description: Put the call to your complex linear solver here. complex solves an NxN linear system of complex vectors by separating the real and imaginary components and performing a coupled 2Nx2N linear solve using all real values. The complex linear solve is composed of:

$$\begin{bmatrix} \mathbf{J} & \omega\mathbf{B} \\ -\omega\mathbf{B} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{4.1}$$

See section 2.2.3 for a more detailed explanation.

On Input:

`c` is the right hand side vector (real part) in equation 4.1.

`d` is the right hand side vector (complex part) in equation 4.1.

`jac_flag` is the flag indicating the status of the Jacobian so that preconditioners can be used:
`NEW_JACOBIAN`: recalculate preconditioner.
`OLD_JACOBIAN`: reuse preconditioner.
`OLD_JACOBIAN_DESTROY`: reuse preconditioner, then destroy the preconditioner.

`omega` is the imaginary value of the eigenvalue that is tracked with the Hopf bifurcation tracking algorithm.

`tmp` is an allocated temporary storage vector with the same size as `x`.

On Output:

`c` is the real part of update vector to the komplex linear solve.

`d` is the complex part of update vector to the komplex linear solve.

Return Value: `[int]` flag that indicates the success of the linear solve. Any negative number signals a failed Newton step.

This wrapper required for: HOPF_CONTINUATION

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, PHASE_TRANSITION_CONTINUATION RSQP_OPTIMIZATION

4.4 matrix_residual_fill_conwrap

```
void matrix_residual_fill_conwrap(double *x, double *rhs, int
matflag);
```

Description: Put the call to your matrix/residual fill routine here.

On Input:

`x` is the solution vector used to evaluate the matrix/residual fills.

`rhs` is storage space for the residual vector, and comes in allocated even when the residual vector are not requested.

`matflag` is a flag indicating the type of fill requested:
`RHS_ONLY`: residual fill only.
`MATRIX_ONLY`: matrix fill only.
`RHS_MATRIX`: fill both the residual and matrix.
`RHS_MATRIX_SAVE`: fill the residual, and optionally fill and save the unperturbed and unscaled Jacobian matrix.
`RECOVER_MATRIX`: matrix fill only, optionally recover from a saved copy rather than reassembling.

On Output:

`rhs` contains the residual vector, when requested by `matflag`.

`passdown.matrix` The Jacobian matrix **J** should be calculated when requested by `matflag`. The user can choose how to store this matrix so that it is available in the `linear_solver_conwrap` and `matvec_mult_conwrap` routines. This would typically be done via the `passdown` structure.

Return Value: `[void]`

This wrapper required for: FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION, RSQP_OPTIMIZATION, EIGENVALUE_CALCULATIONS

This wrapper not needed for: ZERO_ORDER_CONTINUATION

4.5 mass_matrix_fill_conwrap

```
void mass_matrix_fill_conwrap(double *x, double *rhs);
```

Description: Put the call to your mass matrix fill routine here.

On Input:

`x` is the solution vector used to perform the mass matrix fill.

`rhs` is the residual vector, allocated but not used.

On Output:

`passdown.mass_matrix` The Mass matrix (the coefficient of time dependent terms, **B**) should be calculated. The user can choose how to store this matrix so that it is available in the `komplex_linear_solver_conwrap` and `mass_matvec_mult_conwrap` routines. This would typically be done via the `passdown` structure.

Return Value: `[void]`

This wrapper required for: HOPF_CONTINUATION, EIGENVALUE_CALCULATIONS

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, PHASE_TRANSITION_CONTINUATION, RSQP_OPTIMIZATION

4.6 matvec_mult_conwrap

```
void matvec_mult_conwrap(double *x, double *y);
```

Description: Put the call to your matrix-vector multiply routine, using the Jacobian matrix $\mathbf{J}$ here.

On Input:

`x` is the vector of length number of unknowns.

On Output:

`y` contains the vector $\mathbf{J}\ \mathbf{x}$.

Return Value: `[void]`

This wrapper required for: TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, RSQP_OPTIMIZATION, EIGENVALUE_CALCULATIONS

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, PHASE_TRANSITION_CONTINUATION

4.7 mass_matvec_mult_conwrap

`void mass_matvec_mult_conwrap(double *x, double *y);`

Description: Put the call to your matrix-vector multiply routine, using the mass matrix **B**, here.

On Input:

`x` is the vector of length number of unknowns.

On Output:

`y` contains the vector **B x**.

Return Value: `[void]`

This wrapper required for: HOPF_CONTINUATION EIGENVALUE_CALCULATIONS

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, _TRANSITION_CONTINUATION, RSQP_OPTIMIZATION

4.8 assign_parameter_conwrap

`void assign_parameter_conwrap(double param);`

Description: Put the call to a routine here to assign the continuation parameter value `param` in the users code. For instance, if you want to continue in value of a global parameter `alpha`, then this routine consists of the single line:

```
alpha = param;
```

On Input:

`param` is the new value of the continuation parameter.

On Output:

Return Value: `[void]`

This wrapper required for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION

This wrapper not needed for: RSQP_OPTIMIZATION

4.9 assign_bif_parameter_conwrap

```
void assign_bif_parameter_conwrap(double bif_param);
```

Description: Put the call here to a routine that assigns the bifurcation parameter value `bif_param` in the application code. This is the parameter that is part of the solution in the tracking routines.

On Input:

`bif_param` is the new value of the continuation parameter.

On Output:

Return Value: `[void]`

This wrapper required for: TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, RSQP_OPTIMIZATION

4.10 assign_multi_parameter_conwrap

`void assign_multi_parameter_conwrap(double* param_vec);`

Description: Put the call here to a routine that assigns the vector of continuation parameters `param_vec` in the application code. This is the only new conwrap routine for the new multi-parameter continuation capability in LOCA version 1.1.

On Input:

`param_vec` is the vector containing new values of the continuation parameters.

On Output:

Return Value: `[void]`

This wrapper required for: MANIFOLD_CONTINUATION

4.11 calc_scale_vec_conwrap

`void calc_scale_vec_conwrap(double *x, double *scale_vec, int numUnks);`

Description: Put the call to a routine to calculate a scaling vector here.

On Input:

`x` is the solution vector of length number of unknowns.

`scale_vec` is a dummy vector of length number of unknowns.

`numUnks` is the number of unknowns on this proc (the length of x and scale_vec).

On Output:

`scale_vec` Vector of length number of unknowns used to scale variables so that one type of unknown (e.g. pressure) doesn't dominate over others. Used to balance the variables and the arc-length variable in arc-length continuation, and for scaling the null vector in turning point tracking. Using reciprocal of the average value of that variable type is a good choice. Vector of all ones should suffice for most problems.

Return Value: `[void]`

This wrapper required for: ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION, RSQP_OPTIMIZATION

4.12 gsum_double_conwrap

```
double gsum_double_conwrap(double sum);
```

Description: Put the call to a routine to calculate a global sum. Just return sum for single processor jobs. This is used by the global dot product routines.

On Input:

`sum` is the value of double on this processor to be summed on all procs.

On Output:

Return Value: `[double]` The global sum is returned on all processors.

This wrapper required for: ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION, RSQP_OPTIMIZATION

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION

4.13 perturb_solution_conwrap

```
void perturb_solution_conwrap(double *x, double *x_old, double
*scale_vec, int numOwnedUnks);
```

Description: Put a routine to perturb the solution vector a little bit here. This is to move a converged solution at a singularity off of the singularity before doing continuation. We have used a random vector with elements of order 10^{-5} times the solution vector as the perturbation. This isn't pretty but has helped convergence on some turning point tracking problems. Leaving this routine empty works fine.

On Input:

`x` is the current solution vector.

`x_old` is the current solution vector, and should not be modified in this routine.

`scale_vec` is the work space for a vector to scale x.

`numOwnedUnks` is the length of owned nodes part of x, x_old, scale_vec

On Output:

`x` Solution vector perturbed a bit.

Return Value: `[void]`

This wrapper required for: TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION, RSQP_OPTIMIZATION

4.14 solution_output_conwrap

```
void solution_output_conwrap(int num_soln_flag, double *x,
double param, double *x2, double param2, double *x3, double
param3, int step_num, int num_its, struct con_struct *con);
```

Description: Put the call to your solution output (both file and screen) routines here. This routine gets called with a flag value for how many vector-parameter pairs are being sent. The solution vector and continuation parameter value are always passed. The real null vector and bifurcation parameter are also passed if bifurcation tracking is being performed. For Hopf tracking, the imaginary part of the null vector and the frequency ω are passed as well. The call to the eigensolver should be put inside this routine.

On Input:

`num_soln_flag` Number of solution vector - parameter pairs to print out. This is 1 for parameter continuation, 2 for turning point, pitchfork, and phase transition tracking, and 3 for Hopf tracking.

`x` is the solution vector to be printed.

`param` is the continuation parameter value.

`x2` is the real part of the null vector for turning point, pitchfork, and Hopf tracking, and the second solution vector for phase transition tracking (set to NULL when `num_soln_flag` $= 1$).

`param2` is the bifurcation parameter value (set $= 0$ when `num_soln_flag` $= 1$).

`x3` is the imaginary part of the null vector for Hopf tracking (set to NULL when `num_soln_flag` < 3).

`param3` is the frequency ω for Hopf tracking (set $= 0$ when `num_soln_flag` < 3).

`step_num` is the index to output to (step_num is 0 based).

`num_its` is the number of Newton iterations taken to converge this step.

`con_struct` is the `con` structure used in the problem, for passing information to the eigensolver.

On Output:

Return Value: `[void]`

This wrapper required for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION RSQP_OPTIMIZATION

This wrapper not needed for:

4.15 eigenvector_output_conwrap

`void eigenvector_output_conwrap(int j, int num_soln_flag, double *xr, double evr, double *xi, double evi, int step_num);`

Description: Put in code to print out an eigenvalue and eigenvector here. This routine gets called only from `loca_eigenvalue.c`. When the eigenvalue is a complex conjugate pair, this routine is called once with the real and imaginary parts of the eigenvector and eigenvalue.

On Input:

`j` is the eigenvector index, which may be needed for writing multiple eigenvector modes.

`num_soln_flag` is 1 when a real eigenvalue/eigenvector is to be printed, and 2 when a complex pair is to be printed.

`xr` is the real part of the eigenvector.

`evr` is the real part of the eigenvalue.

`xi` is the imaginary part of the eigenvector (set to NULL when `num_soln_flag` = 1).

`evi` is the imaginary part of the eigenvalue.

`step_num` is the index to output to (step_num is 0 based).

On Output:

Return Value: `[void]`

This wrapper required for: EIGENVALUE_CALCULATIONS

4.16 free_energy_diff_conwrap

`double free_energy_diff_conwrap(double *x, double *x2);`

Description: Call to return the free energy difference between two solutions. This is the function G in Eq. 2.76. This can be generalized to any constraint on the problem that must be satisfied for two different solutions.

On Input:

`x` first solution vector, $\mathbf{x}_1$ in Section 2.2.4.

`x2` second solution vector, $\mathbf{x}_2$ in Section 2.2.4.

On Output:

Return Value: `[double]` The difference in the free energy between the two solutions.

This wrapper required for: PHASE_TRANSITION_CONTINUATION

This wrapper not needed for: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION, TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, RSQP_OPTIMIZATION

4.17 create_shifted_matrix_conwrap

```
void create_shifted_matrix_conwrap()
```

Description: Call to allocate a new matrix (the same size and sparsity as the **J** matrix) for use by the Cayley-enabled ARPACK Eigensolver. The matrix is never seen by LOCA, but is accessed by other conwrap routines, so should be part of the passdown structure.

On Input:

On Output:

Return Value: `[void]`

This wrapper required for: EIGENVALUE_CALCULATIONS

4.18 shifted_matrix_fill_conwrap

`void shifted_matrix_fill_conwrap(double sigma)`

Description: Routine to fill the shifted matrix for the eigensolver, which is just $(\mathbf{J} - \sigma\mathbf{B})$. This call is proceeded by calls to the Jacobian and Mass matrices, so it can be assumed that these are up to date.

On Input:

`sigma` Value of the σ parameter of the Cayley transformation. This routine is called once with $\sigma = 0$ as part of a projection step, and sometimes σ is changed after a restarting of Arnoldi's method in the eigensolver, so make sure to use the `sigma` variable that is passed in and not just the input value of σ.

On Output:

Return Value: `[void]`

This wrapper required for: EIGENVALUE_CALCULATIONS

4.19 shifted_linear_solver_conwrap

`void shifted_linear_solver_conwrap(double *x, double *y, int jac_flag, double tol)`

Description: Put a call to the linear solver here, using the shifted matrix (as filled in the previous wrapper routine). This will be very similar to the `linear_solver_conwrap`, but using a different matrix. Also, the argument list is currently different. For iterative solvers, scaling should be turned off for this solve.

On Input:

`x` Right hand side vector.

`jac_flag` is the flag indicating the status of the Jacobian so that preconditioners can be reused:

`NEW_JACOBIAN`: The Jacobian is a new one, so recalculate preconditioner (or recalculate LU factorization for a direct solver).
`OLD_JACOBIAN`: The Jacobian is the same as the previous call, so reuse preconditioner (or just back solve a previous LU factorization when using a direct solver).

`tol` Acceptance tolerance for an iterative linear solver, computed dynamically to adjust to the problem scaling.

On Output:

`y` Solution vector of the linear solve: $(\mathbf{J} - \sigma\mathbf{B})\mathbf{y} = \mathbf{x}$.

Return Value: `[void]`

This wrapper required for: EIGENVALUE_CALCULATIONS

4.20 destroy_shifted_matrix_conwrap

`void destroy_shifted_matrix_conwrap()`

Description: Call to free memory for the shifted matrix, as allocated by the `create_shifted_matrix_conwrap` routine above.

On Input:

On Output:

Return Value: `[void]`

This wrapper required for: EIGENVALUE_CALCULATIONS

4.21 gmax_int_conwrap

`int gmax_int_conwrap(int max)`

Description: Put a global max operation here for parallel runs. Just return `max` for serial runs.

On Input:

`max` Integer value on this processor.

On Output:

Return Value: `[int]` Maximum of all the `max` values over all processors.

This wrapper required for: EIGENVALUE_CALCULATIONS

4.22 random_vector_conwrap

`void random_vector_conwrap(double *x, int numOwnedUnks)`

Description: Put a call to a random vector generating routine here. This is used as an initial guess for the iterative eigensolver.

On Input:

`numOwnedUnks` Length of the vector `x` that needs to be filled with random real values.

On Output:

`x` Double precision vector filled with random components.

Return Value: `[void]`

This wrapper required for: EIGENVALUE_CALCULATIONS

Chapter 5

LOCA Inputs

All the problem specific information that must be supplied to LOCA from the application code gets set in the `con` structure. This is a structure made up of several structures (referred to hereafter as sub-structures), each dealing with a separate part of the problem. The elements of the sub-structures that must be loaded in the `do_loca` interface routine can be seen in the file `loca_const.h`, and a sample working version of the `do_loca` routine can be seen in `loca_interface.c`.

Table 5.1 shows the names of the sub-structures, a description of what type of information this structure holds, and for what problem types this structure is accessed. As can be seen in this table, the `con.general_info` and `con.stepping_info` structures must always be set, at most one of the next six structures must be set, and the `con.eigen_info` structure needs only be set when eigenvalues calculations are required.

5.1 The `con.general_info` structure

The following structure elements must be supplied for all problems.

`int con.general_info.method` Flag which sets the continuation strategy that LOCA should perform. The choices are: ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, ARC_LENGTH_CONTINUATION,

Structure Name	**Requested Info**	**When Needed**
`con. general_info`	Solution method, problem size, solution vector, parameter, etc.	**Always**
`con. stepping_info`	Initial step size, number of steps, step size control, etc.	**Always**
`con. arclength_info`	Arclength equation scaling control, step size control	**Arclngth Cont. Only**
`con. turning_point_info`	Initial guess of Bifurcation parameter	**Turning Point Only**
`con. pitchfork_info`	Antisymmetric ψ vector and bifurcation parameter guess	**Pitchfork Only**
`con. hopf_info`	Guesses for frequency, eigen-vectors and bifurcation param	**Hopf Only**
`con.phase_ transition_info`	Guesses for second solution vector, bifurcation parameter	**Phase Trans Only**
`con. manifold_info`	Number of parameters, parameter bounds, step sizes	**Multi-Param Only**
`con. eigen_info`	Cayley parameters, Arnoldi space size, tolerances, etc.	**Eigenvalues Requested**

Table 5.1: Table describing the 8 'sub-'structures to the `con` structure. This is the problem-specific information that the user must supply to LOCA.

TURNING_POINT_CONTINUATION, PITCHFORK_CONTINUATION, HOPF_CONTINUATION, PHASE_TRANSITION_CONTINUATION, LOCA_LSA_ONLY. Each of these strings is assigned an integer in `loca_const.h` which can be used in place of the name.

`double con.general_info.param` Initial value of continuation parameter. This parameter gets assigned to a parameter in the application code in the `assign_parameter_conwrap` wrapper routine (see Section 4.8. This is the parameter that gets stepped in all algorithms except the pseudo arclength continuation, where it is solved for. This parameter is λ in Section 2.1, but not the parameter λ in Section 2.2 (which is the `bif_param` in the following sections.

`double *con.general_info.x` Pointer to current solution vector, allocated and filled.

`double con.general_info.perturb` Parameter used to apply perturbations to several quantities.

`int con.general_info.numUnks` Number of owned plus ghost (external) unknowns on current processor. This is used for allocating new vectors of the same length as the solution and residual vectors, and when vectors are copied.

`int con.general_info.numOwnedUnks` Number of owned unknowns on current processor. This is the length of the vector that is acted upon, e.g. by dot product routines.

`int con.general_info.printproc` Logical flag indicating if this processor executes print statements. This flag is usually set to TRUE for serial runs and for the first processor on parallel runs. It is usually set false for all other processors in parallel runs, but can be set to TRUE for debugging.

`int con.general_info.nv_restart` Logical flag indicating if the turning point algorithm is to be restarted using a previously saved null vector.

`int con.general_info.nv_save` Logical flag indicating that the final null vector from a turning point tracking run is to be written to a separate file for later restart.

5.2 The `con.stepping_info` structure

The following structure elements must be supplied for all problems. These parameters control the continuation steps. This is used even for bifurcation tracking routines, by controlling the zero order continuation of the bifurcation point.

`double con.stepping_info.first_step` Initial parameter step size, which is variable λ_b in Section 2.1.1. This is only approximate for pseudo arclength continuation.

`int con.stepping_info.base_step` Starting number of continuation steps for code-specific output, should be 0 or 1.

`int con.stepping_info.max_steps` This is the maximum number of continuation steps allowed for the run, variable N_c in Section 2.1.1. Failed steps count as steps.

`int con.stepping_info.last_step` Logical flag which is TRUE when the final continuation step is in progress.

`int con.stepping_info.max_param` Ending parameter value for the run. This can be a max or min, and is variable λ_e in Section 2.1.1.

`double con.stepping_info.max_delta_p` Largest allowable parameter step size (entered as an absolute value). This is the variable $\Delta\lambda_{max}$ in Section 2.1.1. Step size is reset to this value if it is exceeded.

`double con.stepping_info.min_delta_p` Not currently implemented. Smallest allowable parameter step size (entered as an absolute value). This is variable $\Delta\lambda_{min}$ in Section 2.1.1. If step size decreases below this value due to failed steps, continuation run is aborted.

`double con.stepping_info.step_ctrl` This is an adjustable parameter used to control the rate of step size increase. This is the variable a in Eq. 2.8 in Section 2.1.1. When set to zero, step size will remain constant unless there is a convergence failure.

`int con.stepping_info.max_newton_its` The maximum number of Newton iterations allowed by the nonlinear solver. This is used to adjust the continuation step size, since it is variable N_{max} in Eq. 2.8..

5.3 The `con.arclength_info` structure

The following structure elements must be supplied for arc length continuation problems, when `con.general_info.method` =ARC_LENGTH_CONTINUATION. This structure is not accessed for any other method.

`double con.arclength_info.dp_ds2_goal` Desired fraction of parameter contribution to arc length equation; used to set and reset solution scale factor for arc length continuation. This is variable $\lambda_g'^2$ in Section 2.1.3.

`double con.arclength_info.dp_ds_max` Used in arc length continuation for periodic solution rescaling. The solution scale factor is recalculated when the parameter sensitivity to arc length exceeds this value. This is variable λ'_{max} Section 2.1.3.

`double con.arclength_info.tang_exp` Adjustable parameter used to decrease step size in arc length continuation near turning points. A value of zero has no effect, and large values (e.g. 5.0) greatly decrease the step size in regions of high curvature. This is the exponent y in Eq. 2.26 in Section 2.1.3).

`double con.arclength_info.tang_step_limit` Used in arc length continuation to provide greater step size control near complex turning points. If the cosine of the tangent to the solution branch at two consecutive steps (τ in Eq. 2.25) drops below this value, the step is failed. See discussion below Eq. 2.26 in Section 2.1.3).

5.4 The `con.turning_point_info` structure

The following structure elements must be supplied for turning point continuation problems, when `con.general_info.method` =TURNING_POINT_CONTINUATION. This structure is not accessed for any other method.

`double con.turning_point_info.bif_param` Initial guess for the bifurcation parameter, which is λ in Eq. 2.30. This is the parameter that is solved for as part of the solution procedure for the bifurcation. This parameter must be assigned in `assign_bif_parameter_conwrap`.

`double *con.turning_point_info.nv` Pointer to a restarted null vector, used only for restarting a turning point tracking run when a previous null vector is provided and con.general_info.nv_restart has been set to TRUE. This pointer must be set to the location of the previous null vector within do_loca, and is then used to initialize the null vector array in the turning point bordering algorithm.

5.5 The `con.pitchfork_info` structure

The following structure elements must be supplied for pitchfork continuation problems, when `con.general_info.method` =PITCHFORK_CONTINUATION. This structure is not accessed for any other method.

`double con.pitchfork_info.bif_param` Initial guess for the bifurcation parameter, which is λ in Eq. 2.49. This is the parameter that is solved for as part of the solution procedure for the bifurcation. This parameter must be assigned in `assign_bif_parameter_conwrap`.

`double *con.pitchfork_info.psi` The ψ vector that is antisymmetric with respect to the symmetry being broken by the pitchfork bifurcation (see Eq. 2.45). This is also used as the initial guess for the null vector $\mathbf{y}$. This is usually computed first by the eigensolver.

5.6 The `con.hopf_info` structure

The following structure elements must be supplied for Hopf continuation problems, when `con.general_info.method` =HOPF_CONTINUATION. This structure is not accessed for any other method.

`double con.hopf_info.bif_param` Initial guess for the bifurcation parameter, which is λ in Eq. 2.65. This is the parameter that is solved for as part of the solution procedure for the bifurcation. This parameter must be assigned in `assign_bif_parameter_conwrap`.

`double con.hopf_info.omega` Initial guess for the frequency ω of the Hopf bifurcation (see Eq. 2.60). This is the imaginary part of the eigenvalue whose real part is zero. This is usually computed first by the eigensolver.

`double *con.hopf_info.y_vec` This is the initial guess for the real part of the eigenvector at the Hopf point, $\mathbf{y}$ in Eq. 2.60. This is usually computed first by the eigensolver.

`double *con.hopf_info.z_vec` This is the initial guess for the imaginary part of the eigenvector at the Hopf point, $\mathbf{z}$ in Eq. 2.60. This is usually computed first by the eigensolver along with $\mathbf{y}$.

`int con.hopf_info.mass_flag` Flag specifying whether or not to compute (a) the mass matrix derivative with respect to the bifurcation parameter ($\frac{\partial \mathbf{B}}{\partial \lambda}$) andor (b) the mass matrix derivative with respect to the solution vector ($\frac{\partial \mathbf{B}}{\partial \mathbf{x}}$), during Hopf bifurcation tracking. These derivatives are often zero and need not be calculated, but for some formulations and parameters (e.g. geometry parameters) these can not be ignored. The possible values are: 1 calculate both (a) and (b); 2 calculate (b) only; 3 calculate (a) only; 4 calculate neither (a) nor (b); 5 calculate (a) and (b) once, and if the norms are non-negligible then include these terms in the Newton iteration.

5.7 The `con.phase_transition_info` structure

The following structure elements must be supplied for phase transition continuation problems, when `con.general_info.method` =PHASE_TRANSITION_CONTINUATION. This structure is not accessed for any other method.

`double con.phase_transition_info.bif_param` Initial guess for the bifurcation parameter, which is λ in Eq. 2.79. This is the parameter that is solved for as part of the solution procedure for the phase transition. This parameter must be assigned in `assign_bif_parameter_conwrap`.

`double *con.phase_transition_info.x2` This is the initial guess for the second solution vector at the phase transition point, $\mathbf{x}_2$ in Section 2.2.4. The first, $\mathbf{x}_1$, is supplied in `con.general_info.x`.

5.8 The `con.manifold_info` structure

The following structure elements must be supplied for multi-parameter (i.e. manifold) continuation problems, when `con.general_info.method` =MANIFOLD_CONTINUATION. This structure is not accessed for any other method. The first entry is the integer k, the number of parameters on which to perform multi-parameter continuation. The remaining four are each a vector of doubles of length k.

`int con.manifold_info.k` The number of parameters on which to perform multi-parameter continuation. This will most often be 2.

`int con.manifold_info.param_vec` A vector of length k of the starting values of the parameters.

`int con.manifold_info.param_lower_bounds` A vector of length k of the minimum values of the parameters, defining the lower bounds on parameter space over which to compute the manifold of steady solutions.

`int con.manifold_info.param_upper_bounds` A vector of length k of the maximum values of the parameters, defining the upper bounds on parameter space over which to compute the manifold of steady solutions.

`int con.manifold_info.param_steps` A vector of length k of desired step sizes of each of the parameter values.

5.9 The `con.eigen_info` structure

The following structure elements must be supplied when eigenvalue calculations are requested. These parameters are passed to the Cayley-enabled version of P_ARPACK that has been linked with LOCA. This can be turned on for any method choice in `con.general_info.method`.

`int con.eigen_info.Num_Eigenvalues` This is the number of eigenvalues that the eigensolver should try to converge, `nev` in ARPACK.

`int con.eigen_info.Num_Eigenvectors` This is the desired number of eigenvalues (modes) to be output at each continuation step.

`int con.eigen_info.sort` Logical flag indicating that the converged eigenvalues, and their associated eigenvectors, are to be sorted in descending order of real eigenvalue part.

`double con.eigen_info.Shift_Point[3]` This length 3 array contains the Cayley parameters σ and μ, and the experimental parameter δ which in general should be set to 1.0.

`int con.eigen_info.Arnoldi` This is the requested size of the Arnoldi space, known as `ncv` in ARPACK.

`double con.eigen_info.Residual_Tol[2]` This array of length 2 contains the convergence tolerance for the eigensolver, `tol` in ARPACK, and the convergence tolerance for the iterative linear solver, η.

`int con.eigen_info.Max_Iter` This is the maximum number of outer iterations of the restarted Arnoldi method (1 means no restarts). This is `iparam[MAX_ITRS]` in ARPACK.

`int Every_n_Steps` This is the desired number of ontinuation steps between eigensolver calls.

Chapter 6

Nonlinear Analysis Strategies

This chapter describes procedures for performing nonlinear analysis on a system of equations using the algorithms provided by LOCA. These procedures are strategies for dealing with the fact that Newton's method is only locally convergent. For example, a Hopf bifurcation tracking requires highly accurate guesses for the eigenvectors of the complex pair of eigenvalues associated with the Hopf bifurcation. This section will discuss what the user is required to supply prior to running the tracking algorithms. A typical nonlinear analysis would follow these steps:

1. Calculate a steady state solution for the set of nonlinear equations from a trivial solution. This may require changing a parameter value so that the problem is more linear (e.g. decreasing the Reynolds number).

2. Use parameter continuation (zero order, first order, and arc-length) and eigenvalue analysis (linear stability analysis) to continue to parameter regions of interest and locate bifurcation points on the the steady state branch.

3. Using solution and eigenvalue/eigenvector information calculated near a bifurcation point, the tracking algorithms are used to "lock-on" to the solution and value of the bifurcation parameter at the bifurcation point, and track it as a function of a second parameter.

6.1 Steady State Solutions

We assume that the code was designed for steady state calculations and that this can be achieved trivially. If the initial steady state can not be found using pure Newton's method, a globalized Newton method can be applied [20, 21, 22, 23] (including integration in in time) or continuation in real or contrived parameters (Homotopy) can be used to get to the initial steady state of interest. For parameter continuation the idea is to converge to a solution at simplified conditions (where Newton's method converges with a trivial initial guess). Then repeatedly adjust the parameters, while using previous solution information to generate a good initial guess for the solution vector, until you reach the steady state conditions of interest. For example, this can be critical in reacting flow applications comprised of coupled sets of descretized PDEs. Here one would converge to a solution at isothermal conditions and slowly increase (take continuation steps) in temperature until the desired conditions are achieved.

6.2 Parameter Continuation

Once a steady state has been calculated, one can search along the steady state branch by stepping in the continuation parameter, λ. LOCA provides algorithms for performing zero order, first order and arc-length parameter continuation, set by the flags ZERO_ORDER_CONTINUATION, FIRST_ORDER_CONTINUATION, AND ARC_LENGTH_CONTINUATION, respectively.

When a zero order or first order continuation run repeatedly has failed steps that results in cuts in the step size, this often indicates that the algorithm is approaching a turning point. This is where the solution branch doubles back and no solution exists locally for the next parameter step. Near this point one should restart the algorithm using the initial guess at the last converged solution but switch to arc-length parameter continuation. If the parameter doubles back on a new branch, a turning point has been located.

6.2.1 Locating a Bifurcation Point

A turning point can be identified directly by using the arc-length continuation algorithm as discussed in the previous section. To identify any other bifurcation we must use the eigenvalue capabilities in LOCA to locate bifurcation points by monitoring the rightmost eigenvalues. Section 1.2 gives a detailed description of bifurcation phenomena which we summarize here. A bifurcation point occurs when the real part of an eigenvalue of the system under study is zero. If the eigenvalue is purely real the bifurcation may be a turning point or a higher codimension bifurcation such as a pitchfork bifurcation. If the eigenvalue has an imaginary component (i.e. is a complex conjugate pair), the bifurcation is a Hopf bifurcation. Eigenvalue calculations can also ascertain branch stability. If all the eigenvalues are on the left side of the complex plane (i.e. all the eigenvalues have negative real parts) then the solution is a stable steady state.

To locate a bifurcation point, step in the continuation parameter while monitoring the eigenvalues. When the real part of an eigenvalue passes zero, you have evidence of a bifurcation point. Once a bifurcation point is identified, the solution and corresponding eigenvalues and eigenvectors should be calculated and saved for initial guesses for the bifurcation tracking algorithms discussed in the next section. Often the bifurcation tracking algorithms will converge using a solution and the appropriate eigenvectors from somewhere near the bifurcation point, and it is not necessary to zero in on the bifurcation using parameter continuation and eigenvalue calculations before starting the tracking algorithm . However, we have sometimes found that this extra work is needed to launch the tracking algorithms.

6.3 Bifurcation Tracking

The bifurcation tracking algorithms calculate the parameter value (referred to as the bifurcation parameter) that corresponds to a bifurcation point while doing zero order continuation in another parameter (referred to as the continuation parameter). In order to perform bifurcation tracking, certain steps must be taken beforehand. First, a bifurcation point must be located as discussed in section 6.2. The solution nearest the bifurcation should be used as the initial guess for the solution vector of the bifurcation tracking run. The tracking algorithms calculate the steady state solution, eigenvector(s) (also called the NULL vector(s)), and a bifurcation parameter value. Some tracking routines require accurate initial guesses for the NULL vectors while others can use a trivial guess. We will discuss the typical ways to converge the initial step of a tracking algorithm individually.

Since LOCA was written for massively parallel distributed memory systems, we assume an iterative linear solver is used. The bordering algorithms force us to solve nearly-singular linear systems so we typically tighten the linear solver tolerances and loosen the nonlinear solver tolerances during bifurcation tracking. Convergence is determined by driving the scaled update norm for the solution vector, NULL vector, bifurcation parameter and any other algorithm specific unknowns to **less than one**:

$$\text{Scaled Vector Update Norm} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left[\frac{|x_i|}{|x_i| * RTol + ATol}\right]^2} \tag{6.1}$$

where $RTol$ and $ATol$ are the relative and absolute tolerances for the nonlinear solver, N is the number of unknowns, and x_i is the ith unknown value in the vector. We often run with tolerances around 1.0E-2 and 1.0E-5 for the relative and absolute tolerances in the nonlinear solver. Linear solver tolerances are usually kept in the range 1.0e-4 to 1.0E-8 and can greatly effect the convergence of the Newton method.

6.3.1 Turning Point Tracking

The turning point tracking only requires initial guesses for the solution vector and bifurcation parameter. Section 6.2.1 describes the process for identifying the turning point with arclength continuation. Starting with the extreme parameter value and associated solution vector from the arclength continuation run will usually be adequate starting points for the turning point algorithm. No auxiliary variables are required by the code.

6.3.2 Pitchfork Tracking

For pitchfork tracking, the code requires an initial guess for the solution vector, the antisymmetric vector (ψ), and the bifurcation parameter. We use the ψ vector as an initial guess for $\mathbf{y}$. This vector is calculated by first detecting the Pitchfork bifurcation with an eigensolver. By choosing as initial guesses the solution and parameter value at the point where the eigenvalue was closest to zero, and choosing the associated eigenvector as ψ, the pitchfork tracking algorithm will usually converge. For problems that have multiple pitchfork bifurcations in the same region of parameter space, which is often the case when the system can go unstable to different modes, the pitchfork algorithm can be started multiple times with different ψ vectors to track each pitchfork separately.

6.3.3 Hopf Tracking

The Hopf tracking algorithm requires an initial guess for the solution vector, the bifurcation parameter, the value of the imaginary part of the eigenvalue, ω, and the real and complex parts of the corresponding eigenvector, $\mathbf{y}$ and $\mathbf{z}$. The procedure is the same as that for the pitchfork bifurcation, except that a complex pair of eigenvalues crosses the imaginary axis. By finding the point on a continuation run where the real part is closest to zero, all the information is available to create initial guesses to start the Hopf tracking algorithm. Since the eigenvectors given by the eigensolver will not generally satisfy the normalization conditions (Eq.s 2.63 and 2.64), the initial guesses for the eigenvectors are initially rotated and scaled so that these conditions are met. We have found that excess Newton iterations are taken after the solution vector and bifurcation parameter are converged if we keep the same tolerances for both the solution vector and the eigenvectors. Therefore, while we require the scaled update norm to be less than one for the solution, imaginary eigenvalue, and bifurcation parameter, we only require the eigenvectors to have a scaled update norm of less than 100.

6.3.4 Phase Transition Tracking

Phase transition tracking is initiated after first identifying one instance of a phase transition. When arclength continuation locates a region of hysteresis, the solution branch can be plotted using the free energy as a measure and a phase transition can be visually picked off. By using other solution on each branch nearest the visible crossing as initial guesses for $\mathbf{x}_1$ and $\mathbf{x}_2$, and the parameter value where they appear to cross, we have found the phase transition algorithm to be very robust. The system of governing equations does become singular at a critical point, so there is a limit to how close the algorithm can come to this point.

6.3.5 Multi-Parameter Continuation

The multi-parameter continuation capability is new and has not yet been heavily exercised. Please contact the authors with experiences, questions, or improvements. The density of solutions along the solution manifold is in part governed by the parameter step sizes that is part of the user input. In certain cases the overall step size can also be manipulated in the routine `MFScaleLOCA` in file `loca_mf.c`. To compute the manifold with bounds other then a rectangle over the parameter space, including bounds on the solution, the user can program the function `LOCATest` in the same file. The output from MF includes a metric of the solution and the parameter values. The metric is set to be the norm of the solution vector but can be reprogrammed in `MFPrintMetricLOCA`. A post-processing code external/mf/DrawLOCA.c was written to translate the MF output to gnuplot or other formats.

Acknowledgements

We would like to thank John Shadid, David Womble, and Sudip Dosanjh for their efforts in defining, guiding, and supporting this project. We would also like to thank John Shadid, David Day, Karen Devine, Mike Heroux, Mark Adams, Laura Frink, Randy Schunk and Eric Phipps for many useful discussions that have helped us make decisions on algorithms, software design, and licensing.

We would also like to thank Mike Henderson of IBM for working to interface MF with LOCA to generate the multi-parameter continuation capability, and for allowing the distribution of MF with LOCA. We would also like to thank Professor Danny Sorenson of Rice University for making the ARPACK software freely available.

We are grateful for the financial support of DOE, in particular the MICS division for algorithmic research, the ASCI program for software development and documentation, and the CSRF program for scalability studies with initial applications.

Bibliography

[1] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK USERS GUIDE: Solutions of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA, 1998.

[2] K. J. Maschhoff and D. C. Sorensen. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In J. Wasniewski, J. Dongarra, K. Madsen, and D. Oleson, editors, *Applied Parallel Computing in Industrial Problems and Optimization*, volume 1184 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.

[3] R. B. Lehoucq and A. G. Salinger. Large-scale eigenvalue calculations for stability analysis of steady flows on massively parallel computers. *International Journal for Numerical Methods in Fluids*, 36:309–327, 2001.

[4] E. A. Burroughs, L. A. Romero, R. B. Lehoucq, and A. G. Salinger. Large scale eigenvalue calculations for computing the stability of buoyancy driven flows. *Sandia Technical Report*, SAND2001-0113, 2001.

[5] K.A. Cliff, A. Spence, and Tavener S.J. Numerical bifurcation analysis with applications to incompressible flows. *Acta Numerica*, pages 39–131, 2000.

[6] J. Guckenheimer and P. Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*. Applied Mathematical Science. Springer-Verlag, 2 edition, 1983.

[7] R. Seydel. *From Equilibrium to Chaos: Practical Bifurcation and Stability Analysis*. Elsevier, 1988.

[8] G. Iooss and D. D. Joseph. *Elementary Stability and Bifurcation Theory*. Springer Verlag, 1981.

[9] M. Golubitsky and D. Schaeffer. *Singularities and Groups in Bifurcation Theory*. Springer-Verlag, 1985.

[10] M.E. Henderson. Multiple parameter continuation: Computing implicitly defined k-manifolds. *Int'l J. Bifurcation and Chaos*, 12(3):451–476, 2002. see www.research.ibm.com/people/h/henderson.

[11] W. Govaerts. *Numerical Methods for Bifurcations of Dynamic Equilibria*. SIAM. Philadelphi, PA, 2000.

[12] H. B. Keller. Numerical solution of bifurcation and nonlinear eigenvalue problems. In P. H. Rabinowitz, editor, *Applications of Bifurcation Theory*, pages 159–384. Academic Press, New York, 1977.

[13] J.N. Shadid. *Experimental and Computational Study of the Stability of Natural Convection Flow in an Inclined Enclosure*. Ph.D. Thesis, University of Minnesota, Minneapolis, Minnesota, 1987.

[14] G. Moore and A. Spence. A numerical algorithm for the tracking of turning points. *SIAM J. Numer. Anal.*, 17(4):567–576, 1980.

[15] A.D. Jepson. *Numerical opf Bifurcation*. Ph.D. Thesis, California Institute of Technology, Paseda, California, 1987.

[16] A. Griewank and G. Reddien. The calculation of hopf points by a direct method. *IMA J. Numerical Analysis*, 3:295–303, 1983.

[17] D. Day and M. Heroux. Solving complex-valued linear systems via equivalent real formulations. *SIAM J. Sci. Comp.*, 23(2):480–498, 2001.

[18] R. B. Lehoucq and A. G. Salinger. Massively parallel linear stability analysis with P_ARPACK for 3D fluid flow modeled with MPSalsa. *Lecture Notes in Computer Science*, 1541:286–295, 1998.

[19] A. G. Salinger, R. B. Lehoucq, and L. Romero. Stability analysis of large-scale incompressible flow calculations on massively parallel computers. *CFD Journal*, 9(1):529–533, 2001.

[20] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Computational Mathematics. Prentice-Hall, 1983.

[21] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 1999.

[22] J.J. Moré and D. Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software*, 20(3):286–307, 1994.

[23] J.J. Moré and D. C. Sorenson. Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing*, 4(3):553–572, 1983.