Multifario (MF): Documentation

Michael E. Henderson
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

July 26, 2007

1 Introduction

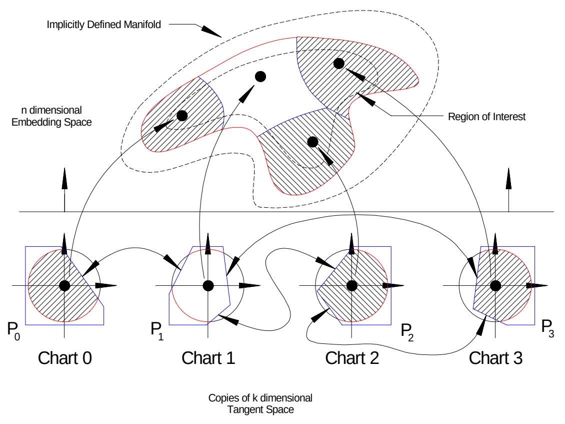
Multifario (MF) is a package for continuing solution manifolds of nonlinear systems of equations. The algorithm is described in some detail in the [?]. This document is a user's guide/reference for the implementation that was used for the examples in that paper. Another example that has been published is [?]. This is a clamped rod, which is a very idealized model for DNA folding.

1.1 Overview

Most of this section can be skipped by the first time user. Several examples are discussed later in this manual, and they are probably a good place to start. This section describes (at a high level) the algorithm and data structures that are used in those examples. Further specific details can be found in the second half of this manual.

The continuation algorithm in multifario takes as input an implicitly defined manifold M (i.e. points which lie in $\mathbb{R}^n$ and satisfy a system of equations $F(u) = 0$). The output is an "atlas" of "charts" which cover the "component" of M which is connected to an input point u_0 by paths entirely in some region of interest Ω.

The continuation is based on the following loop, which is the basic flow of any continuation method:



Implicitly Defined Manifold
n dimensional Embedding Space
Region of Interest
P_0
P_1
P_2
P_3
Chart 0
Chart 1
Chart 2
Chart 3
Copies of k dimensional Tangent Space

Figure 1: The basic obects in the representation of a manifold.

1. Let $m = 0$, $M_0 = \emptyset$.
2. Build a neighborhood of M about u_m.
3. Merge the neighborhood into M_m.
4. Select a new point u_{m+1} in Ω which is near the boundary of M_m.
5. Increment m and repeat until there are no points left near the boundary and in Ω.

The manifold must be represented internally in such a way that a point near the boundary can be found. It is also necessary to add a new chart to the internal representation. The data structure used is called an "atlas of charts", or junt an "Atlas". Each chart in the atlas is represented by a polyhedral "chart domain" in the tangent space. The only point on M which is stored is the center of the chart, but a point in the chart domain $P \subset \mathbb{R}^k$ can be projected onto M. A point in the embedding space $\mathbb{R}^n$ can also be projected onto M, as long as a k dimensional subspace is provided to define the normal space to the projection.

Multifario is organized in a fairly transparent way relavtive to these concepts. Some are related to the formulation of the problem `MFImplicitMF`, `MFNRegion`, `MFNVector`. Others with the algorithm used `MFContinuationMethod`, and with the result of the computation `MFAtlas`. The main data structures are:

`MFErrorHandler e` – is a way for the user to pass a specific method of handling errors down to the routines in multifario. There is a default implementation which prints a message to `stdout` and aborts on an error.

`MFImplicitMF M` – represents the implicitly defined manifold M. A pointer to the function $F(u)$ is stored in M, as well as pointers to routines which find the tangent space of M at a point, and project a point onto M orthogonal to a given k dimensional subspace. In addition the embedding space is stored (an `MFNSpace`).

`MFNVector u` – represents a point in the embedding space. There are several implementations of the `MFNVector`, and the implicitly defined manifold M provides a "factory" to return a zero vetor of the correct type `MFIMFVectorFactory`.

`MFKVector s` – represents a point in the tangent space. There is only one implementations of the `MFKVector`, since the

`MFNKMatrix Phi` – represents a basis for the tangent space. It is a set of k $MFNVectors$, although there are

`MFNRegion Omega` – represents a subset of the embedding space.

`MFNSpace space` – represents the embedding space. It is used to perform inner products, to find distances and the tangent to a geodesic. There are several implementations, but the space is usually created in the constructor for M and the casual user should not have to deal with the `MFNSpace`

`MFAtlas A` – represents a (partially) computed covering of M. It consists mostly of a list of charts (see below).

`MFChart c` – represents a small neighborhood of M. It is represented by a center (an `MFNVector` on M), a tangent space (an `MFNKMatrix`), a k dimensional polyhedron which is the domain of the chart mapping, and a radius R, which is the actual size of the neighborhood (2-norm). The polyhedra are updated so that any point which is a distance R away from the origin (in $\mathbb{R}^k$) and which is interior to the polyhedron is on the boundary of the atlas.

`MFComntinuationMethod` – consists mainly of a single routine `MFComputeAtlas` which takes the definition of a solution component and returns an atlas which covers the component. There are several implementations, with varying levels of sophisticate of the implementations, as well as an interface to `AUTO` which can only be used to compute $k = 1$ dimensional manifolds.

1.2 Create/Free – reference counting

Instances of these data structures are created with routines (constructors or "ctors") like MFCreateNCube, which creates an MFNRegion. The user interacts with the object through "member functions" like MFNRegionInterior(Omega,u), which tests if u is in Omega. MFNRegion and the other data structures are typedef'ed to be pointers to data structures. Reference counting is used, so when no longer needed, the instance is released with a corresponding routine like MFFreeNRegion (a desctructor, or dtor).

Routines may store pointers to objects and increase the reference count so that their deletion will be defered until the pointer is no longer needed anywhere. The ctor returns an object with a reference count of one. The destructor (the Free routine) subtracts one from the reference count, and if the reference count is zero the space allocated to the object is released. When an object is passed to a subroutine, and that subroutine stores a pointer to the object, the reference count is increased by one. The destructor for the object which stored the pointer must then invoke the dtor for the stored object. If everyone follows the rules, there are no memory leaks, and objects remain in memory until they are no longer needed.

1.3 Base classes – choosing the constructor

The MFImplicitMF, MFNRegion, MFNSpace, and MFNVector objects are *base classes*. That is, they don't represent a single data structure for storing the object they represent, but could be one of several. The user choses which data structure by choosing the constructor that is called to create an instance of the object. The constructor calls e.g. MFCreateNVectorBaseClass(), then fills in various routines that "Set" a pointer to the data structure that implements the object (e.g. MFNVectorSetData), and routines to perform the operations on the object. When the user calls, e.g. MFNVAdd, to add two vectors, the routine supplied in the ctor is called ad passed the data pointers for the two vectors to add and the data pointer for the vector where the result is to be placed. Note that binary operations like "add" pretty much have to be passed three vectors of the same type (one is for the result). Otherwise there are up to three different "add" routines that could be used. One would be chosen, and would have to use routines that are common to all vectors to get and set the components of the other two vectors.

1.4 Read/Write

All classes provide a "Write" and a "Read" routine. The idea is that the entire data structure can be written to disk, then read in by another program and completely reconstructed. The difficult part is reading back in user implementations of the base classes. The base class writes the Id of the class to disk, then calls the "Write" routine to allow the class to write it's data. When being read, however, the Id indicates what the class is, but the base class has no idea what ctor to call based on that string. For now the "Read" routine is not stored as a pointer in the class, but is a bare subroutine, and the base class "Read" has a case statement that decides, based on the Id, which ctor to call. In the future I should have some way of registering a ctor for a given Id. But then I need to call something to cal the routine that does the registers ... Dynamic loading might do be a way to do it, but supporting DLL's across a range of platforms doesn't look like fun. Static initializers in C++ would work as well, but then the user would have to give up the "main" program, and I would resort to a C++ main program that calls something Kludgy like "MF_Main". Bleech.

So write works if classes that inmplement the base classes provide a reasonable "WriteData" routine, but the "Read" requires editing the base class code to make an entry to the case statement.

1.5 Print

Most routines have a "Print", e.g. MFPrintNVector(FILE*,MFNVector). This does it's best to give a formatted representation of the object, and can be helpful in debugging. The Print routines are declared in the MFPrint.h header file.

1.6 Error handling

The last argument to all multifario routines is an `MFErrorHandler`. This is a base class which is called when the routine encouters an error. The default action is that an error message is printed, and if the error is not a warning `abort()` is called.

2 Installation

The IMF's provided with the distribution require Lapack, and the makefile assumes that it is available in a library called liblapack.a . The blas will also be required. To install:

1. edit the file share/config.site to give local lib and include dirs where Linpack and Lapack can be found.
2. run the configure script "./configure"
3. create the libraries (installed in lib) "make"
4. create the utilities (installed in bin) "make utilities"
5. create the examples (installed in bin) "make examples"

The documentation is in Doc/MF.tex

3 Examples

Several example programs are provided.

ComputeLine A rather trivial example: computes a line segment ($n = 1$, $k = 1$). Uses the NSpace manifold.

ComputePlane Computes a plane ($n = 2$, $k = 2$). Uses the MFNSpace manifold. (Euclidean n-space).

ComputePlaneClip Computes a plane ($n = 2$, $k = 2$). Uses the MFNSpace manifold. (Euclidean n-space). Shows how to use the "clipping" of the chart polyhedra to make the result look nicer.

Compute3Space Computes the interior of a cube ($n = 3$, $k = 3$). Uses the NSpace manifold.

Compute4Space Computes the interior of a hypercube ($n = 4$, $k = 4$). Uses the NSpace manifold.

ComputeCircle Computes a circle ($n = 2$, $k = 1$). Uses the String interface to the MFAlgebraic manifold.

ComputeSphere Computes a sphere ($n = 3$, $k = 2$). Uses the String interface to the MFAlgebraic manifold.

ComputeSphereSub Computes a sphere ($n = 3$, $k = 2$). Uses the Subroutine interface to the MFAlgebraic manifold. Uses two initial points, and limits the continuation to 100 charts. Also override the projection used for writing the output to the plotfile.

ComputeTorus Computes a torus ($n = 3$, $k = 2$). Uses the String interface to the MFAlgebraic manifold.

ComputeGenusTwo Computes a genus two surface ($n = 3$, $k = 2$). Uses the String interface to the MFAlgebraic manifold. Shows how to use MFIMFProject to projection an initial point near M onto M.

ComputeSpherePacking Computes a set of spheres of dimension n lying on and covering the surface of a unit sphere ($k = n - 1$).

ComputeTranscritical Computes a pair of intersecting surfaces ($n = 3$, $k = 2$). Demonstrates the detection of singular lines and continuing through them.

ComputeCusp Computes the complex cusp $u(u^2 - \lambda) = \mu$ ($n = 4$, $k = 2$). Demonstrates the detection of singular lines and continuing through them.

ComputeTaylor24 Computes a model of mode interactions – a pair of cubic equations ($n = 4$, $k = 2$). Uses the String interface to the MFAlgebraic manifold. Has several sheets bifurcating from a trivial sheet.

ComputeTaylorA Computes a model of (2,4) mode interactions from John Bolstad's Taylor-Couette code – a pair of cubic equations ($n = 4$, $k = 2$). Uses the String interface to the MFAlgebraic manifold. Has several sheets bifurcating from a trivial sheet.

ComputeDomokos Computes the equilibrium configurations of a clamped elastica, from the paper:

> Domokos, G. "Global Description of Elastic Bars". ZAMM – Z. angew. Math. Mech. **74** (1994) 4, T 289–T291.

Uses the MFTPBVP manifold, with $k = 1$. Demonstrates secondary bifurcation from a sequence of pitchfork bifurcations from a trivial branch.

ComputeRod Computes the equilibrium configurations of a clamped elastica. Uses the MFTPBVP manifold.

To build the examples run "make examples". For example, to run ComputeSphere use the command "bin/ComputeSphere ¿ Sphere.out" (the output is currently a bit too verbose for stdout). This produces a text file called Sphere.plotfile, which can be rendered, or converted to a file that can be rendered. (See the next Section.)

4 Utilities

The output of the examples is a *plotfile*. The plotfile is a set of polyhedra in n-space representing the polyhedra in the tangent spaces. The centerfile is a list of the points u_i (the centers of the polyhedra).

The user can also get the output in the form of an *atlas* file, which is a full blown dump of the atlas data structure, or as a *centerfile* (just the points u_i). More on this later.

I usually use Open DataExplorer (www.opendx.org) to interact with the results. I also have used Pov-Ray (a nice, free ray-tracer), and a z-buffer renderer that I wrote a long time ago (called "sh") which is included in this distribution. This last produces a tiff file (if you have libtiff – www.libtiff.org), or a postscript file.

Several postprocessing programs are provided for drawing atlas files, and converting plotfiles to DX, POV-Ray or VBM files:

DrawPlotfile Creates a Tiff or Postscript file with a rendering of a plotfile.

PlotfileToDX Creates a DataExplorer (available at www.opendx.org) file from the plotfile. A sample DX net is included (genericDXNet.net and genericDXNet.cfg) that imports a .dx file generated by PlotfileToDX and renders it.

PlotfileToVBM Creates a VBM (available from Randy Paffenroth) file from the plotfile.

PlotfileToPOV Creates a POV-Ray (available at www.povray.org) file from the plotfile. A sample .pov control file (genericPOVRay.pov) is included which sets up a camera and colors and renders a .pov file generated by PlotfileToPOV.

DrawAtlas Creates a Tiff-file (by choice if libtiff – www.libtiff.org is available) or a Postscript file (if it isn't) with a rendering of an atlas file.

DrawAtlasTS Creates a Tiff or Postscript file with a rendering of an atlas file (with charts drawn in the tangent spaces, faster than DrawAtlas).

DrawDual Creates a Tiff or Postscript file with a rendering of the dual triangulation of the atlas file.

DualToDX Creates a DataExplorer file with the dual triangulation of the atlas file.

To create an image of the sphere run "bin/DrawPlotfile Sphere". This looks in the current directory for a file called "Sphere.view" which contains a viewpoint and extent for the picture to be drawn. If the file can't be found defaults are used. Several options are available to the Draw commands. "bin/DrawPlotfile -help" will print a description of the options.

5 Example – computing a sphere, the string interface to MFAlgebraic manifolds

You probably want to begin by modifying one of the provided examples. Below we dissect the ComputeSphere example, which should be enough to get you started. This example illustrates most of the routines needed to find

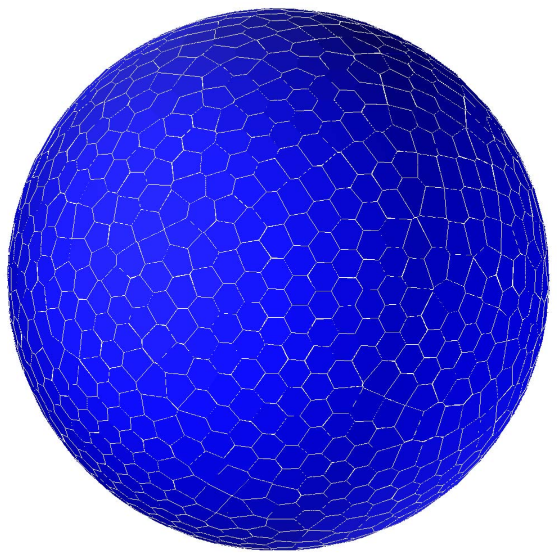


Figure 2: The output from the ComputeSphere example.

a representation of an implicitly defined manifold as an atlas of charts. It can be found in src/ComputeSphere.c. A slightly different version, which uses subroutines to define the sphere can be found in in src/ComputeSphereSub.c.

First we include the header file which defines the interfaces to the Atlas and other objects, and declare various variables.

```
01   #include <MFAtlas.h>
02
03   int main(int argc, char *argv[])
04    {
05     MFImplicitMF M;
06     MFNRegion Omega;
07     MFAtlas A;
08     MFNVector u0;
09     MFContinuationMethod H;
10
```

Next an MFImplicitMF object, which describes the problem, is created in lines 11–12 (below). The corresponding "free" is below at line 34. The "Algebraic" manifold is created using the ctor which takes an expression giving $n-k$ functions defining the function F. The first argument is a list of variables, and the second is a list of functions. The ctor counts the number of variables to determine n, and the number of functions to get $n-k$ and thus k.

```
11      M=MFIMFCreateAlgebraicExpression("[x,y,z]",
12                                          "[x**2+y**2+z**2-1.]",e);
```

The region of interest Ω, is created at line 13. The corresponding "free" is below at line 35. In this case I've chosen a cube centered at the origin if radius 1.1 (i.e. the coordinates of the corners are $(\pm 1.1, \pm 1.1, \pm 1.1)$.

```
13      Omega=MFNRegionCreateHyperCube(3,1.1,e);
14
```

The initial point u_0 is created at line 15 and the coordinates are set in lines 16–18 to those of the initial point $(0, 1, 0)$. The vector is "free'd" below at line 36.

```
15      u0=MFCreateNVector(3,e);
16      MFNVSetC(u0,0, 0.,e);
17      MFNVSetC(u0,1, 1.,e);
18      MFNVSetC(u0,2, 0.,e);
```

With these objects we have provided enough information to create the manifold. Before that we choose which algorithm to use by creating H at line 20. It is "free'd" at line 37. In lines 21–27 we set some parameters that control how the algorithm works. Line 21 sets "epsilon", which is the maximum allowed distance between the tangent space and the manifold, which controls the accuracy and the size of the chart domains. At line 22 we say that the algorithm should continue until the entire manifold is computed (the first time it might be wise to set a moderate number of charts, so that you don't wait forever to get a result). At line 23 we ask for some output (to stdout) indicating the progress of the computation. Line 24 says to "page out" charts once they are no longer needed. For the sphere this is probably silly, but for large problems the memory required to store all charts is large. At line 25 we ask that the output be in the form of a "plotfile" on disk, and at line 26 we say that we don't want the chart centers in a disk file. And in line 27 we give a prefix for files (e.g. the plotfile will be Sphere.plotfile).

```
19
20      H=MFCreateMultifariosMathod(e);
21      MFMultifariosMathodSetEpsilon(H,.1,e);
22      MFMultifariosMathodSetMaxCharts(H,-1,e);
23      MFMultifariosMathodSetVerbose(H,1,e);
24      MFMultifariosMathodSetPage(H,1,e);
25      MFMultifariosMathodSetDumpToPlotFile(H,1,e);
26      MFMultifariosMathodSetDumpToCenterFile(H,0,e);
27      MFMultifariosMathodSetFilename(H,"Sphere",e);
```

Finally at line 29 the manifold is passed to the routine, MFComputeAtlas, which creates and returns an Atlas, which is "free'd" below at line 33.

```
28
29      A=MFComputeAtlas(H,M,Omega,u0,e);
```

Since the output is being written to a plotfile as we go along we don't do anything with the Atlas, just close it (line 31), which makes sure that all of the polyhedra have been written to the plotfile.

```
30
31      MFCloseAtlas(H,A,e);
```

And finally, we free all the objects we Create'd. Note that MFComputeAtlas creates an atlas and returns it, so that must be free'd as well.

```
32
33      MFFreeAtlas(A,e);
34      MFFreeImplicitMF(M,e);
35      MFFreeNRegion(Omega,e);
36      MFFreeNVector(u0,e);
37      MFFreeContinuationMethod(H,e);
38
39      return 0;
40     }
```

Of course, when main ends all the storage is free'd anyway, but this is a good habit.

6 Example – solving a two point boundary value problem, the MFTPBVP manifold

Below we dissect the ComputeDomokos example, which should be enough to get you started with the TPBVP solver. The ComputeRod is more realistic, but the solution has sheets with symmetries, and the example separates these by controlling crossings of the planes of symmetry. The problem is from the paper

> Domokos, G. "Global Description of Elastic Bars". ZAMM – Z. angew. Math. Mech. **74** (1994) 4, T 289–T291.

and is a noninear eigenvalue problem – a two-point boundary value problem for three functions $(\alpha(x), M(x), y(x))$ and two parameters (H, V). There are four boundary conditions, so we expect the solutions to be curves (and they are).

$$\begin{aligned}
&\alpha' = M \\
&M' = -H\sin\alpha + V\cos\alpha \\
&y' = \sin\alpha \\
\\
&\alpha(0) = \alpha(1) = 0 \\
&y(0) = y(1) = 0
\end{aligned}$$

When $V = 0$ there is a trivial solution $(\alpha(x) = 0, M(x) = 0, y(x) = 0)$, and a linear analysis indicates bifurcations from the trivial solution.

First we include the header file which defines the interfaces to the Atlas and other objects, and declare various variables.

```
#include <MFAtlas.h>
#include <math.h>
int MFDomokosProjectToDraw(MFNVector,double*,void*,MFErrorHandler);
void MFTPBVPSetStability(MFImplicitMF,MFNVector,MFNKMatrix,
                                                    void*,MFErrorHandler);
int MFStopTPBVP(MFImplicitMF,MFNVector,MFNKMatrix,MFNVector,
                                           MFNKMatrix,void*,MFErrorHandler);

#define PI 3.14159265358979323846264338327950288

#define NX 100
```

```
09
10   int main(int argc, char *argv[])
11    {
12     int i,j,n;
13     MFImplicitMF M;
14     MFNRegion Omega;
15     MFAtlas S;
16     MFNVector ug;
17     MFNVector u0;
18     MFNKMatrix Tan;
19     MFContinuationMethod H;
20     double p0[2];
21     double p1[2];
22     double *r0;
23     double dr;
24     double xy0[3];

25     int nx=NX;
26     int np=2;
27     int nu=3;
28     int nbc=4;
29     int nic=0;
30     int k;
```

Next an MFImplicitMF object, which describes the problem, is created. The corresponding "free" is below at line 66. The "TPBVP" manifold is created using the ctor which takes subroutines giving 4 functions defining the boundary value problem. This is of the form

$$\begin{aligned}
&u' = f(t,u,p,u0,p0)\\
&a(u(0),u(1),p,u0(0),u0(1),p0) = 0\\
&\int_0^1 l(t,u(t),p,u0(t),p0)\mathrm{d}\ t + m(p,p0) = 0
\end{aligned}$$

The routines are passed as triples (except for m), of the routine to evaluate the function, and it's derivatives w.r.t. u and the parameters p. The pair $u0$ and $p0$ are a nearby function and parameter (for imposing phase constraints).

```
31  k=nu+np-nbc-nic;
```

```
32  M=MFIMFCreateTPBVP(k,nx,nu,np,f,fu,fl,nbc,a,au,al,nic,
                                             l,lu,ll,m,ml,e);
33  MFIMFSetR(M,6./(4*PI*PI),e);
34  MFIMFSetProjectForDraw(M,MFDomokosProjectToDraw,e);
35  MFIMFSetSetStability(M,MFTPBVPSetStability,e);
36  MFIMFSetStop(M,MFStopTPBVP,e);
```

The call to `MFIMFSetR` defines a maximum radius, which is needed on the trivial branch (where the curvature is zero, which would give an infinite radius). The call to `MFIMFSetProjectForDraw` provides a routine (`MFDomokos-ProjectToDraw`), which projects the full solution into a smaller space for the Plotfile. The routines `MFIMFSetSetStability` and `MFIMFSetStop` are a clumsy way of stating that bifurcations are to be located. By default the `MFTPBVP` does not do this.

The region of interest Ω, is created at line 39. The corresponding "free" is below at line 67. The `MFNRegionCreateTPBVP` is just an interval on each of the parameters $(p0, p1)$, and on the norm of u, $(-200, 200)$.

```
37  p0[0]= -600./(4*PI*PI); p1[0]=600./(4*PI*PI);   /* H */
38  p0[1]= -20.; p1[1]=20.;   /* V */
39  Omega=MFNRegionCreateTPBVP(nx,nu,np,p0,p1,-200.,200.);
```

Next an initial point u_0 is created. This is done by defining a mesh in the array $r0$, (lines 40–42) and solving the initial value problem to get a guess at a solution `ug` (line 48). The routine `MFTPBVPIntegrateForInitialSolution` solves the initial value problem. Its arguments are `xy0`, the initial conditions, `p0` the parameters, and the mesh `r0`.

Once we have a guess we find the tangent space at the guess, and project the guess to get the initial point `u0`. The vector is "free'd" below at line 68. The guess is "free'd" at line 52, and the tangent at line 5 at line.

```
40  r0=(double*)malloc((nx+1)*sizeof(double));
41  dr=1./(nx-2);
42  for(i=0;i<nx+1;i++)r0[i]=(i-.5)*dr;

43  xy0[0]=0.;
44  xy0[1]=0.;
45  xy0[2]=0.;
46  p0[0]=1.15;
```

```
47  p0[1]=0.;

48  ug=MFTPBVPIntegrateForInitialSolution(M,xy0,p0,r0,e);

49  Tan=MFIMFTangentSpace(M,ug,e);

50  u0=MFCreateNVector(n,e);

51  MFFreeNKMatrix(Tan,e);
52  MFFreeNVector(ug,e);
```

With these objects we have provided enough information to create the manifold. Before that we choose which algorithm to use by creating H at line 53. It is "free'd" at line 69. In lines 45–62 we set some parameters that control how the algorithm works. Line 54 sets "epsilon", which is the maximum allowed distance between the tangent space and the manifold, which controls the accuracy and the size of the chart domains. The `DotMin` set at line 55 says that branches whose dot product of tangents is bigger than this number are the same (bifurcating branchs make an angle bigger than the arccos of the dotmin). At line 56 we say that the algorithm should continue until the entire manifold is computed (the first time it might be wise to set a moderate number of charts, so that you don't wait forever to get a result). At line 57 we ask for some output (to stdout) indicating the progress of the computation. Line 58 says to "page out" charts once they are no longer needed. For the sphere this is probably silly, but for large problems the memory required to store all charts is large. At line 59 we ask that the output be in the form of a "plotfile" on disk, and at line 60 we say that we don't want the chart centers in a disk file. At line 61 we ask that for 50 times when we rnu out of manifold we invoke the branch switcher to move on to another branch. (50 is just a large enough number – we want them all.) Fnally in line 62 we give a prefix for files (e.g. the plotfile will be Domokos.plotfile).

```
53  H=MFCreateMultifariosMathod(e);
54  MFMultifariosMathodSetEpsilon(H,.03,e);
55  MFMultifariosMathodSetDotMin(H,.9,e);
56  MFMultifariosMathodSetMaxCharts(H,-1,e);
57  MFMultifariosMathodSetVerbose(H,1,e);
58  MFMultifariosMathodSetPage(H,1,e);
```

```
59  MFMultifariosMathodSetDumpToPlotFile(H,1,e);
60  MFMultifariosMathodSetDumpToCenterFile(H,0,e);
61  MFMultifariosMathodSetBranchSwitch(H,50,e);
62  MFMultifariosMathodSetFilename(H,"Domokos",e);
```

Finally at line 63 the manifold is passed to the routine, MFComputeAtlas, which creates and returns an Atlas, which is "free'd" below at line 33.

```
63     S=MFComputeAtlas(H,M,Omega,u0,e);
```

Since the output is being written to a plotfile as we go along we don't do anything with the Atlas, just close it (line 64), which makes sure that all of the polyhedra have been written to the plotfile.

```
64     MFCloseAtlas(H,S,e);
```

And finally, we free all the objects we Create'd. Note that MFComputeAtlas creates an atlas and returns it, so that must be free'd as well.

```
65     MFFreeAtlas(S,e);
66     MFFreeImplicitMF(M,e);
67     MFFreeNRegion(Omega,e);
68     MFFreeNVector(u0,e);
69     MFFreeMultifariosMathod(H,e);

70     return 0;
71    }
```

Of course, when main ends all the storage is free'd anyway, but this is a good habit.

Finally, there are the routines defining the problem and the projection. These could be in a separate library, or in the same file. The differential equation –

```
void f(double r,int nu,double *u,int np,double *p,
                       double *u0,double *l0,double *f)
 {
  f[0]=2*PI*u[1];
  f[1]=-p[0]*sin(2*PI*u[0])+p[1]*cos(2*PI*u[0]);
  f[2]=sin(2*PI*u[0]);
  return;
 }

void fu(double r,int nu,double *u,int np,double *p,
                       double *u0,double *l0,double *fu)
 {
  int i;

  for(i=0;i<nu*nu;i++)fu[i]=0.;
  fu[0+nu*1]=2.*PI;
  fu[1+nu*0]=-2*PI*p[0]*cos(2*PI*u[0])-2*PI*p[1]*sin(2*PI*u[0]);
  fu[2+nu*0]=2*PI*cos(2*PI*u[0]);
  return;
 }

void fl(double r,int nu,double *u,int np,double *p,
                       double *u0,double *l0,double *fl)
 {
  int i;

  for(i=0;i<nu*np;i++)fl[i]=0.;
  fl[1+nu*0]=-sin(2*PI*u[0]);
  fl[1+nu*1]= cos(2*PI*u[0]);

  return;
 }

void a(int nbc,int nu,double *uL,double *uR,int np,double *p,
                    double *u0L,double *u0R,double *l0,double *a)
 {
  a[0]=uL[0];
  a[1]=uR[0];
  a[2]=uL[2];
  a[3]=uR[2];
  return;
 }
```

```
void au(int nbc,int nu,double *uL,double *uR,int np,double *p,
                    double *u0L,double *u0R,double *l0,double *au)
 {
  int i;

  for(i=0;i<nbc*2*nu;i++)au[i]=0.;
  au[0+nbc*(0+0*nu)]=1.;
  au[1+nbc*(0+1*nu)]=1.;
  au[2+nbc*(2+0*nu)]=1.;
  au[3+nbc*(2+1*nu)]=1.;
  return;
 }

void al(int nbc,int nu,double *uL,double *uR,int np,double *p,
                    double *u0L,double *u0R,double *l0,double *al)
 {
  int i;
  for(i=0;i<nbc*np;i++)al[i]=0.;
  return;
 }
```

The routines for the integral constraints aren't really needed (we could pass NULL's), since there are no integral equations.

Finally there's the projection. The protocol here is that it is called first with NULL arguments, which is a signal to return the length required in `x` for the projected point.

```
int MFDomokosProjectToDraw(MFNVector u, double *x, void *d)
 {
  int nx=NX;
  int np=2;
  int nu=3;

  if(x==(double*)NULL)return 3;

  x[0]=MFNV_C(u,nx*nu,e);
  x[1]=MFNV_C(u,nx*nu+1,e);
```

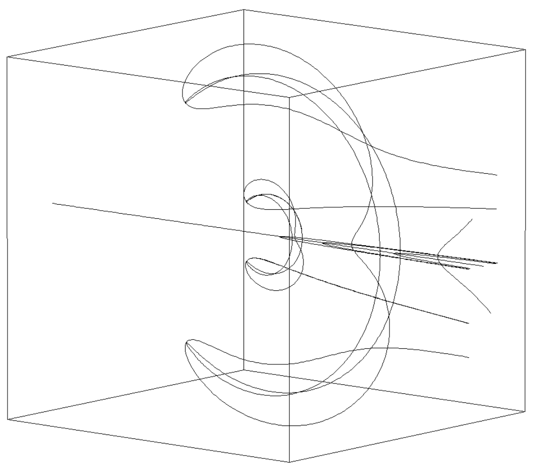


Figure 3: The output from the ComputeSDomokos example.

```
  x[2]=MFNV_C(u,1,e);

  return 3;
 }
```

To run this example, use something like

```
bin/ComputeDomokos > Domokos.out
```

To create an image of the bifurcation diagram run "bin/DrawPlotfile Domokos". This looks in the current directory for a file called "Domokos.view" which contains a viewpoint and extent for the picture to be drawn. If the file can't be found defaults are used. Several options are available to the Draw commands. "bin/DrawPlotfile -help" will print a description of the options.

7 Implicit Manifolds Provided

I've written a couple of continuation codes for my own use over the years. I intentionally decided not to do that this time, since some very good ones are already out there. The idea was to provide only the core of a continuation code, and use those other codes to do the projections, tangent calculations and singular point detection and branch switching. Like all good dreams this one was a little optimistic. So I've implemented a couple of solvers so that I have something to play with, and have written interfaces to AUTO and LOCA, available at

`http://www.sourceforge.net/projects/auto2000/`
`and http://www.cs.sandia.gov/projects/loca/`

What the implementations below are missing (and AUTO and LOCA provide) is singular point detection and branch switching, and a way of writing those related systems so that bifurcating branches of different type may be followed (e.g. periodic motions from steady state, homoclinic from periodic, etc.). I'm working on it though.

MFAlgebraicMF – an algebraic system defined by strings or subroutines The ComputeSphere example in the previous section shows how to use the ctor with strings. There's a version of the example using the ctor that passes subroutines in the ComputSubroutine example.

MFTPBVP – a Two Point Boundary Value Problem defined by subroutines. For now see the ComputeRod example. I'm trying to find a cleaner example.

8 MFImplicitMF – an Implicitly Defined Manifold

The Implicit manifold has a fairly complicated interface, and is where most of the work in a continuation method is performed. Below M is of type MFImplicitMF, u and v are of type MFNVector, and Phi is of type MFNKMatrix,

int MFIMF_N(M,e); – Returns the dimension of the space in which the manifold is embedded.

int MFIMF_K(M,e); – Returns the dimension of the manifold.

int MFIMFProject(M,u0,Phi,u,e); – Returns a point u on the manifold which is the projection of u0 orthogonal to the columns of the matrix

Phi. If the projection fails a "0" is returned, otherwise the result is "1".

MFNKMatrix MFIMFTangentSpace(M,u,e); – Returns a matrix whose columns form an orthonormal basis for the tangent space of M at the point u.

MFNKMatrix MFIMFTangentSpaceWithGuess(M,u,Phi0,e); – Returns a matrix whose columns form an orthonormal basis for the tangent space of M at the point u. The user provides a guess at the tangent space in the matrix Phi0. Some approaches to finding the tangent space can take advantage of this guess.

double MFIMFScale(M,u,Phi,e); – Returns a radius for the ball at a point u, at which the columns of the matrix Phi give an o.n. basis for the tangent space. The idea is that for points in the tangent space that are closer to the origin then R the project will not fail and the distance between the point in the tangent space $(u + \Phi s)$ and the projection is within ϵ.

MFNSpace MFIMFNSpace(M,e); – Returns the space in which the manifold is embedded.

void MFIMFEvaluate(M,u,MFNVector F,e); – This routine is not needed for the continuation, but if it is present it evaluates $F(u)$ and returns it in the first $n - k$ coordinates of the vector F.

void MFIMFApplyJacobian(M,u,Phi,Psi,e); – This routine is also not needed for the continuation, but if it is present it evaluates $F_u(u)$, applies it to the columns of the matrix Phi and returns each result in the corresponding column of the matrix Psi. Note that Phi is not necessarily a basis for the null space of the Jacobian.

void MFIMFApplySecDer(M,u,MFNVector phi0,MFNVector phi1,MFNVector psi,M
This routine is also not needed for the continuation, but if it is present it evaluates $F_{uu}(u)$, applies it to the vectors phi0 and phi1 and returns result in psi, that is

$$\psi = F_{uu}(u)\phi_0\phi_1, \qquad \text{or in tensor notation} \qquad \psi^i = F^i_{,j,k}\phi_0^j\phi_0^k.$$

int MFIMFStop(M,u0,Phi0,u1,e); – Determines if the continuation should move from u0 to u1. This is the way singular points are detected. It is really a second way to limit the extent of the manifold (the first way being Ω).

int MFIMFProjectToSave(M,u,double *y,e); – Provides a projection tha tis used to save a point to disk. The idea is that the entire MFNVector may be more than is needed.

This and the following projections use the protocol that if they are called with either u or y is NULL (zero), they return the number of coordinates in the projection. If u and y are non-NULL, y will be of that length.

int MFIMFProjectToDraw(M,u,double *y,e); – Provides a projection that is used to save a point to plotfile.

int MFIMFProjectToBB(M,u,double *y,e); – Provides a projection that is used to store a point in a hierarchical bounding box during the computation. Points that are well separated should project to well separated points, and the smaller the dimension the less work.

void MFIMFSetStability(M,u,Phi,e); – Sets the "index" of a vector with tangent space Phi. This is sufficient information to detect bifurcations.

int MFIMFSingular(M,u,Phi,v,e); – Finds a singular vector at a bifurcation point. The null space of the Jacobian will typically be of dimension $k+1$, and the returned vector is expected to be that one which is orthogonal to the k-dimensional space spanned by the columns of Phi.

void MFFreeImplicitMF(M,e); – Releases the storage associated with the MFImplicitMF object.

9 MFNRegion – a subset of n-space

These are subsets of the embedding space which restrict the part of the manifold that is to be computed. The Region has only a test routine, which indicates if an NVector is in the region or not.

MFNRegion MFNRegionCreateRectangle(x0,y0,x1,y1,e); – Creates a 2-dimensional rectangular region. The arguments are all double's.

MFNRegion MFNRegionCreateCube(x0,y0,z0,x1,y1,z1,e); – Creates a 3-dimensional rectangular region. The arguments are all double's.

MFNRegion MFNRegionCreateHyperCube(int n,double R,e); – Creates a region which is the interior of a n-dimensional hypercube centered at the origin and sides of length $2R$.

MFNRegion MFNRegionCreateHyperCubeByCorners(int n,ll,ur,e); – Creates a region which is the interior of a n-dimensional hypercube with corners ll (lower left) and ur (upper right). Both corners are MFNVector's.

int MFNRegionInterior(MFNRegion,MFNVector,e); – Returns 1 if the point is in the region, otherwise returns 0.

void MFFreeNRegion(MFNRegion,e); – Release a reference to the region. When the reference count goes to zero the storage associated with the object is free'd.

10 MFNSpace – an n-dimensional embedding space

The manifold is embedded in an n-dimensional space. Rather than assume that this is $\mathbb{R}^n$ we allow the user to supply a way to measure distance and to specify the vector from one point to another. This allows the embedding space to be, for example, periodic, or to have a norm which weights some directions more heavily than others. below space is of type MFNSpace.

Normally a user does not need to create an n-space – they are created when an implicitly defined manifold is created.

double MFNSpaceInner(space,u0,u1,e); – Computes the inner product of the two MFNVector's u0 and u1.

double MFNSpaceDistance(space,u0,u1,e); – Computes the distance between the two MFNVector's u0 and u1.

void MFNSpaceDirection(space,u0,u1,du,e); – Computes the unit vector du pointing from u0 to u1. The MFNVector du must be created by the user (use MFCloneNVector(u0)).

void MFNSpaceAdd(space,u0,u1,sum,e); – Computes the sum of from u0 and u1. The MFNVector sum must be created by the user (use MFCloneNVector(u0)).

void MFNSpaceScale(space,double s,u,v,e); – Multiplies u by the scalar s and puts the result in v. The MFNVector v must be created by the user (use MFCloneNVector(u)).

void MFFreeNSpace(space,e); – Release a reference to the NSpace. When the reference count goes to zero the storage associated with the object is free'd. An NSpace created with an MFImplicitMF is Free'd when the manifold is Free'd.

11 MFNVector – a point in the embedding space

These are points lying in the embedding space. Again, the interface is quite a bit simpler than the IMF.

MFNVector MFCreateNVector(int,MFErrorHanfdler); – Creates and returns an N vector of the given length. This ctor creates a vector stored as an array of doubles. It should be Free'd with the MFFreeNVector routine when it is no longer needed.

MFNVector MFCreateNVectorWithData(int,double*,MFErrorHanfdler); – Creates and returns an N vector of the given length, with coordinates copied from the array. This ctor creates a vector stored as an array of doubles. It should be Free'd with the MFFreeNVector routine when it is no longer needed.

MFNVector MFCloneNVector(MFNVector u,MFErrorHandler e); – Creates and returns an N vector of the same length and coordinates as u. Note: this is a "deep" copy, so changing a coordinate of the cloned vector does not change the corresponding coordinate of the original.

This is a ctor, and the new vector should be Free'd with the MFFreeNVector routine when it is no longer needed.

int MFNV_NC(MFNVector,MFErrorHandler); – Returns the number of coordinates of an NVector (i.e. n).

double MFNV_C(MFNVector,int,MFErrorHandler); – Returns the specified coordinate of an NVector.

void MFNVSetC(MFNVector,int,double,MFErrorHandler); – Changes the specified coordinate of an NVector.

void MFNVAdd(a,b,c,e); – Adds two NVectors c=a+b. c must have been created by the user.

void MFNVDiff(MFNVector,MFNVector,MFNVector,MFErrorHandler); – takes the difference of two NVectors c=a-b. c must have been created by the user.

char *MFNVType(MFNVector,MFErrorHandler); – Returns a string indicating the type of vector. A Dense vector has the type "DENSE".

void MFFreeNVector(MFNVector,MFErrorHandler); – Release a reference to the NVector. When the reference count goes to zero the storage associated with the object is free'd.

12 MFKVector – a point in the tangent space

These are points lying in the tangent space of the manifold. They are stored as a vector of doubles. The user normally would not need to use these objects.

MFKVector MFCreateKVector(int,MFErrorHandler); – Creates and returns an K vector of the given length. This ctor creates a vector stored as an array of doubles. It should be Free'd with the MFFreeKVector routine when it is no longer needed.

MFKVector MFCreateKVectorWithData(int,double*,MFErrorHandler); – Creates and returns an K vector of the given length, with coordinates copied from the array. This ctor creates a vector stored as an array of doubles. It should be Free'd with the MFFreeKVector routine when it is no longer needed.

int MFKV_NC(MFKVector,MFErrorHandler); – Returns the number of coordinates of an KVector (i.e. k).

double MFKV_C(MFKVector,int,MFErrorHandler); – Returns the specified coordinate of an KVector.

void MFKVSetC(MFKVector,int,double,MFErrorHandler); – Changes the specified coordinate of an KVector.

void MFKVAdd(a,b,c,e); – Adds two KVectors c=a+b. c must have been created by the user.

void MFKVDiff(MFKVector,MFKVector,MFKVector,MFErrorHandler); – Takes the difference of two KVectors c=a-b. c must have been created by the user.

void MFKVScale(double,MFKVector,MFErrorHandler); – Multiplies a KVector by a scalar (in place).

void MFKVScaleMul(double,MFKVector,MFKVector,MFErrorHandler); – Multiplies a KVector by a scalar and returns the result in a vector supplied by the user. The result vector must have been allocated by the user.

double MFKVDot(MFKVector,MFKVector,MFErrorHandler); – Returns the inner product of two KVectors (Euclidean).

double MFKVNorm(MFKVector,MFErrorHandler); – Returns the norm of a KVector (Euclidean).

void MFFreeKVector(MFKVector,MFErrorHandler); – Release a reference to the KVector. When the reference count goes to zero the storage associated with the object is free'd.

13 MFNKMatrix – a basis for the tangent space

These are used to store an orthonormal basis for the tangent space of the manifold, though they can be used as general matrices. They are stored as a list of the columns of the matrix as NVectors.

MFKKMatrix MFCreateNKMatrix(int k,MFNVector *cols,MFErrorHandler);
– Creates and returns an $n \times k$ matrix of k columns, and *copies* (by cloning), the given columns into the matrix. The NKMatrix should be Free'd with the MFFreeNKMatrix routine when it is no longer needed.

MFNKMatrix MFCreateNKMatrixWithData(int,int,double*,MFErrorHandler);
– Creates and returns an $n \times k$ matrix of k columns, and *copies*, the given elements into the matrix (by creating a dense NVecto for each column). The NKMatrix should be Free'd with the MFFreeNKMatrix routine when it is no longer needed.

MFNKMatrix MFCloneNKMatrix(MFNKMatrix,MFErrorHandler);
– Creates and returns an $n \times k$ matrix which is a copy of the one passed. Note: this is a "deep" copy, so changing an element of the cloned matrix does not change the corresponding coordinate of the original. This is a ctor, and the new matrix should be Free'd with the MFFreeNKMatrix routine when it is no longer needed.

int MFNKMatrixK(MFNKMatrix,MFErrorHandler); – Returns the number of columns in the matrix.

int MFNKMatrixN(MFNKMatrix,MFErrorHandler); – Returns the length of the columns in the matrix.

MFNVector MFMColumn(MFNKMatrix,int,MFErrorHandler); – Returns the requested column.

void MFNKMSetC(MFNKMatrix,int i,int j,double,MFErrorHandler);
– Changes the ith element of jth column to the given value.

void MFMSetColumn(MFNKMatrix,int j,MFNVector c,MFErrorHandler);
– Replaces the jth column of the matrix with the vector c. You are on your honor to make sure the new column is the same type as the others in the matrix.

void MFGramSchmidt(MFNSpace,MFNKMatrix,MFErrorHandler);
– Performs a Gram-Schmidt orthogonalization on the matrix (the space is needed for the inner products!).

void MFFreeNKMatrix(MFNKMatrix,MFErrorHandler); – Release a reference to the NKMatrix. When the reference count goes to zero the storage associated with the object is free'd.

14 MFChart – a small piece of a manifold

MFChart MFCreateChart(MFImplicitMF M,MFNVector u,MFNKMatrix TS, doubl
– Creates a new chart.

MFImplicitMF M The manifold on which the chart lies.

MFNVector u The center of the chart.

MFNKMatrix TS An orthonormal basis for the tangent space of the matrix.

double R The radius of the domain of the chart.

The polyhedron of the chart is initially a hypercube centered at the origin with halfside R.

void MFSubtractHalfSpaceFromChart(MFChart,int,MFKVector n,double d0,MFErro
– This is the core operation of the continuation method. It updates the chart's polyhedron, subtracting the half space $\mathbf{s}.\mathbf{n} < d0$.

MFPolytope MFChartPolytope(MFChart,MFErrorHandler); – Returns the Polyhedron associated with a chart.

MFNVector MFChartCenter(MFChart,MFErrorHandler); – Returns the center of a chart.

MFNKMatrix MFChartTangentSpace(MFChart,MFErrorHandler);
– Returns an o.n. basis for the tangent space of the manifold at the center of a chart.

double MFChartRadius(MFChart,MFErrorHandler); – Returns the radius of a chart.

int MFChartEvaluate(MFChart,MFKVector s,MFNVector u,MFErrorHandler);
– Projects a point in the domain of the chart onto the manifold. The NVector u must have been allocated by the user and should be the same type as the chart center.

int MFChartInterior(MFChart,MFKVector,MFErrorHandler); – Tests to see if a point is interior to the polyhedron of a chart.

int MFChartHasBoundary(MFChart,MFErrorHandler); – Tests to see if all vertices of the polyhedron of a chart have radius less than the radius of the chart.

int MFChartK(MFChart,MFErrorHandler); – Returns the dimension of the manifold of a chart.

int MFChartN(MFChart,MFErrorHandler); – Returns the dimension of the embedding space of a chart.

void MFChartProjectIntoTangentSpace(MFChart,MFNVector u,MFKVector s,MFEr
– Projects a point in the embedding space orthogonally onto the tangent space of the manifold at the center of a chart. The KVector s must have been allocated by the user.

void MFChartPointInTangentSpace(MFChart,MFKVector s,MFNVector u,MFErrorH
– Returns the point in the embedding space corresponding to the first order Taylor series approximation to the manifold. The NVector u must have been allocated by the user and should be the same type as the chart center.

void MFFreeChart(MFChart,MFErrorHandler); – Release a reference to the Chart. When the reference count goes to zero the storage associated with the object is free'd.

15 MFContinuationMethod – an algorithm for computing an atlas of charts for a manifold

A continuation method is an "algorithm" - that is is has a number of internal parameters and provides a method for "doing" something. In the present case that something is computing an Atlas of charts which cover some part of an implicitly defined manifold. Currently there is only one ContinuationMethod (mine!), but when I get some spare time, a grad student, or a vounteer, I'd like to cast the other algorithms in this frame. In the following description –

MFContinuationMethod H;

MFAtlas A;

MFImplicitMF M;

MFRegion Omega;

MFAtlas MFComputeAtlas(H,M,Omega,u0,MFErrorHandler); – Returns an atlas computed using the given algorithm, with starting point u0 (an MFNVector).

MFAtlas MFComputeAtlasWithTangent(H,M,Omega, MFNVector u0, MFNKMatrix
– Returns an atlas computed using the given algorithm, with starting point u0 (an MFNVector), and tangent space Phi0 (an MFNKMatrix). (This is useful for starting at bifurcation points where the tangent space is not unique).

MFAtlas MFComputeAtlasMultiple(H,M,Omega, int m,u0,MFErrorHandler);
– Returns an atlas computed using the given algorithm, with starting points u0[] (an array of m MFNVectors).

MFAtlas MFComputeAtlasMultipleWithTangents(H,M,Omega, int m, MFNVector *
– Returns an atlas computed using the given algorithm, with starting points u0[] (an array of m MFNVectors) and corresponding tangent spaces Phi0[] (an array of m MFNKMatrix's).

void MFExtendAtlas(A,H,M,Omega,u0,MFErrorHandler); – Same as MFComputeAtlas, but begins with a previously computed atlas. The manifold must be the same, but the Continuation Method might be different, as well as the region Omega.

void MFExtendAtlasMultiple(A,H,M,Omega, int m,u0,MFErrorHandler);
– Same as MFComputeAtlasMultiple, but begins with a previously computed atlas. The manifold must be the same, but the Continuation Method might be different, as well as the region Omega.

void MFExtendAtlasWithTangent(A,H,M,Omega, MFNVector u0,MFNKMatrix Phi
– Same as MFComputeAtlasWithTangent, but begins with a previously computed atlas. The manifold must be the same, but the Continuation Method might be different, as well as the region Omega.

void MFExtendAtlasMultipleWithTangents(A,H,M,Omega, int m, MFNVector *u0,
– Same as MFComputeAtlasWithTangents, but begins with a previously computed atlas. The manifold must be the same, but the Continuation Method might be different, as well as the region Omega. All of the routines described above call this one.

void MFFlushAtlas(H,A,MFErrorHandler); – Allows the continuation method to perform any pending tasks.

void MFCloseAtlas(H,A,MFErrorHandler); – Allows the continuation method to perform any pending tasks and end the continuation.

Turning now to the parameters that are specific to my algorithm –

MFContinuationMethod MFCreateMultifariosMathod(MFErrorHandler);
– Creates a continuation method with default values for the parameters.

void MFMultifariosMathodSetVerbose(H,int,MFErrorHandler); – Sets a flag indicating how much output the user wants to see on stdout. Zero is minimal, a higher number is more. The current setting can be retreived using the routine MFMultifariosMathodGetVerbose.

void MFMultifariosMathodSetMaxCharts(H,int,MFErrorHandler);
– Sets the maximum number of charts that willbe computed. A -1 indicates no limit. The current setting can be retreived using the routine MFMultifariosMathodGetMaxCharts.

void MFMultifariosMathodSetMinR(H,int,MFErrorHandler); – Sets the minimum chart radius. Must be positive. A chart with radius below this limit is treated as if it is interior. The current setting can be retreived using the routine MFMultifariosMathodGetMinR.

void MFMultifariosMathodSetMaxR(H,int,MFErrorHandler); – Sets the maximum chart radius. Must be positive. No chart is larger than this. The current setting can be retreived using the routine MFMultifariosMathodGetMaxR.

void MFMultifariosMathodSetEpsilon(H,double,MFErrorHandler);
– Sets the maxumum allowed distance over a chart between the first order approximation and the manifold. Must be positive. The current setting can be retreived using the routine MFMultifariosMathodGetEpsilon.

void MFMultifariosMathodSetPage(H,int,MFErrorHandler); – Sets a flag indicating whether to page out charts to a plotfile or centerfile as the continuation progresses. Page in doesn't work yet, so the MFAtlas that is returned is missing the centers and tangentspaces of the charts that have been paged out. MFCloseAtlas causes the remaining charts to be paged out. The current setting can be retreived using the routine MFMultifariosMathodGetPage.

void MFMultifariosMathodSetPageEvery(H,int,MFErrorHandler); – Sets a flag indicating how often charts are checked and paged out. The current setting can be retreived using the routine MFMultifariosMathodGetPageEvery.

void MFMultifariosMathodSetDumpToPlotFile(H,int,MFErrorHandler); – Sets a flag indicating whether charts being paged out are written to a plotfile. The current setting can be retreived using the routine MFMultifariosMathodGetDumpToPlotFile.

void MFMultifariosMathodSetDumpToCenterFile(H,int,MFErrorHandler); – Sets a flag indicating whether charts being paged out are written to a centerfile. The current setting can be retreived using the routine MFMultifariosMathodGetDumpToCenterFile.

void MFMultifariosMathodSetCheckPoint(H,int,MFErrorHandler); – Sets a flag indicating the atlas is to be written to disk (as an atlasfile), as the continuation progresses. The current setting can be retreived using the routine MFMultifariosMathodGetCheckPoint.

void MFMultifariosMathodSetCheckPointEvery(H,int,MFErrorHandler); – Set the interval (in number of charts) between checkpoints. The current setting can be retreived using the routine MFMultifariosMathodGetCheckPointEvery.

void MFMultifariosMathodSetBranchSwitch(H,int,MFErrorHandler); – Set a flag indicating whether the continuation is to attempt to cross singular curves. The current setting can be retreived using the routine MFMultifariosMathodGetBranchSwitch.

NOTE: I'm still working on this.

void MFMultifariosMathodSetFilename(H,char*,MFErrorHandler);
– Set the basename used for writing plotfile, centerfiles, and atlasfiles. The current setting can be retreived using the routine MFMultifariosMathodGetFilename.

void MFMultifariosMathodAddClipF(H,double (*)(MFNVector),MFErrorHandler);
– This is a "brute force" way of tidying up an atlas for rendering. The clipping functions assign a scalar to each vertex in a chart polyhedron, and before the chart is written to a plotfile a linear interpolant is used to clip off the part of the polyhedron that has any positive clipping function value. The functions may be nonlinear, but this procedure won't do what is intended unless there is only one change in sign on any polyhedral edge.

Clipping functions are useful because I compute a covering, and so the computed manifold extends slightly outside the region Omega. When drawing the manifold it looks better to have smooth edges.

void MFMultifariosMathodClearClipF(H,MFErrorHandler); – Resets the number of Clipping Functions to zero.

void MFMultifariosMathodSetDumpToRestartFile(H,int,MFErrorHandler);
– Sets a flag indicating whether a restart file is written as the continuation progresses. Plotfiles do not contain points that lie on the manifold, and centerfiles don't have tangents (needed for branch switching) and can be too large to be useful in extracting representative points on the manifold. A restart file corresponds to AUTO's labeled points, and contains a representative sample of regular and singular points on the manifold. The current setting can be retreived using the routine

int MFMultifariosMathodGetDumpToRestartFile .

NOTE: I'm still working on this.

void MFMultifariosMathodSetDumpToRestartFileEvery(H,int,MFErrorHandler);
– Indicates how dense the points in the restart file are. As charts are added each is assigned a number, which is the minimum of the numbers assigned to its neighbors, plus one. This is a rough indication of how many charts separate it from a chart numbered 0. When a chat has a number greater than the number provided through this routine it is written to the restart file and assigned the number 0. Singular points

have their own set of independant numbers. The current setting can be retreived using the routine MFMultifariosMathodGetDumpToRestartFileEvery.

NOTE: I'm still working on this.

void MFFreeMultifariosMathod(H,MFErrorHandler); – Release a reference to the ContinuationMethod. When the reference count goes to zero the storage associated with the object is free'd.

16 Using an Atlas of Charts

An atlas is a set of charts, and is the data structure holding the results of a continuation.

The following routines access the data structure (for more details, consult the subroutine reference). The variable A is of type MFAtlas.

MFAtlas MFCreateAtlas(MFImplicitMF,MFErrorHandler); – Creates an empty atlas representing the manifold. The user would normally not use this ctor, instead creating the atlas with MFComputeAtlas or its ilk.

int MFAtlasK(A,MFErrorHandler); – Returns the dimension of the manifold corresponding to the atlas.

int MFAtlasN(A,MFErrorHandler); – Returns the dimension of the embedding space of the manifold corresponding to the atlas.

int MFAtlasAddChart(A,MFNVector,MFErrorHandler); – Adds a chart centered at the given point to the atlas. The point is assumed to be on the manifold, MFTangentSpace is used to get the tangent space, and MFScale to get the initial guess at the radius.

int MFAtlasAddChartWithAll(A,u,Phi,double R,MFErrorHandler);] – Adds a chart centered at the given point u with tangent space Phi, and radius R to the atlas.

MFImplicitMF MFAtlasMF(A,MFErrorHandler); – Returns the manifold corresponding to the atlas.

int MFAtlasNumberOfCharts(A,MFErrorHandler); – Returns the number of charts in the atlas.

double MFAtlasChartRadius(A,int chart,MFErrorHandler); – Returns the radius of a chart in the atlas.

MFNVector MFAtlasCenterOfChart(A,int chart,MFErrorHandler); – Returns the center of a chart in the atlas.

MFNKMatrix MFAtlasChartTangentSpace(A,int chart,MFErrorHandler); – Returns the tangent space of a chart in the atlas.

int MFAtlasIsPointInChart(A,int chart,MFKVector s,MFErrorHandler); – Tests to see if the point is in the domain of a chart in the atlas.

void MFAtlasEvaluateChart(A,int chart,MFKVector s,MFNVector u,MFErrorHandle – Projects a point in the domain of a chart onto the manifold. The NVector u must have been allocated by the user, and should have the same type as the chart center.

int MFAtlasNumberOfChartsWithBoundary(A,MFErrorHandler); – Returns the number of charts on the boundary of the atlas.

int MFAtlasChartWithBoundary(A,int,MFErrorHandler); – Returns the number of a chart on the boundary of the atlas.

int MFAtlasPointOnBoundaryInsideRegion(A,Omega,u,double *delta,MFErrorHandl – Finds a point on the boundary of the atlas. The chart it lies on is returned by the routine, as well as the distance between the point and the tangent space. The NVector u must have been allocated by the user, and should have the same type as the chart center.

void MFFreeAtlas(A,MFErrorHandler); – Release a reference to the Atlas. When the reference count goes to zero the storage associated with the object is free'd.

17 Error handling

These routines provide a way of finding out what errors have occured in the algorithm. The code attempts to indicate errors by returning results that are out of range (i.e. a negative number for the radius if a chart doesn't exist). It is safer however, to check the number of errors.

int MFNErrors(void,MFErrorHandler); – Returns the number of errors committed so far (or since the last ClearErrors).

int MFErrorSev(int error,MFErrorHandler); – Returns the severity of an error. 4=Warning, 8=severe, 12=terminal.

char *MFErrorRoutine(int error,MFErrorHandler); – Returns the name of the routine in which an error was issued.

char *MFErrorMsg(int error,MFErrorHandler); – Returns the message associated with an error.

int MFErrorLine(int error,MFErrorHandler); – Returns the line number at which an error was issued.

char *MFErrorFile(int,MFErrorHandler); – Returns the name of the source file containing the routine in which an error was issued.

int MFError(void,MFErrorHandler); – Returns 1 if an error has occured, 0 otherwise.

void MFClearErrors(void,MFErrorHandler); – Clear all errors.

18 Implementing an MFNVector

The MFNVector object is a base class for vectors in the embedding space. Since the implementation depends heavily on the solver being used I've allowed the user who is writing an interface to a solver to implement their own MFNVector. All n-vectors in the continuation are created by cloning the initial point or one of its clones (this is fun, eh?), so the solver can control what type of vector it sees by requiring the user to create the starting point as a particular type of vector.

MFNVector's (and the other base classes described below) provide a ctor for the base class, and a way of passing a data block (usually a C struct) to all of the member functions. So the user implements a member function that looks like the base class member function with the extra argument, and in the ctor for the new vector he creates the new vector with the "CreateBaseClass" ctor, and sets the data block and member functions by calling various "Set" routines.

The source for the dense vector class is a good place to look at an implementation (src/MFDenseNVector.c).

MFNVector MFCreateNVectorBaseClass(char *id,MFErrorHandler);
– This creates an empty MFNVector and returns it to the user.

The "id" is a character string that will be returned by the MFNVGetId routine, and can be used to check the type of a vector for example, before casting the data block to the struct used for this class.

char *MFNVGetId(MFNVector,MFErrorHandler); – Returns the identification string of a vector (i.e. the string passed to the base class ctor).

void MFNVectorSetData(MFNVector,void*,MFErrorHandler); – Sets the data pointer.

void MFNVectorSetWriteData(MFNVector,void (*)(FILE*,void*),MFErrorHandler);
– Sets the routine used to write a vector to file.

void MFNVectorSetFreeData(MFNVector,void (*)(void*),MFErrorHandler);
– Sets the routine that is called when the last reference to the vector is Free'd. Note that the CreateBaseClass returns a vector with one reference.

void *MFNVectorGetData(MFNVector,MFErrorHandler); – returns the data pointer of a vector.

void MFNVectorSetClone(MFNVector,MFNVector (*)(void*),MFErrorHandler);
– Sets the routine used by the MFCloneNVector routine. I'd suggest that the routine extract data from the data pointer and invoke one of the ctors.

void MFNVectorSetGetNC(MFNVector,int (*)(void*),MFErrorHandler);
– Sets the routine used to retrieve the dimension of the vector. The continuation does not use this routine.

void MFNVectorSetGetC(MFNVector,double (*)(int,void*),MFErrorHandler);
– Sets the routine used to retrieve a coordinate. This is meant to be a fallback in case a routine has to deal with a vector of unknown type. (Performance suffers if used to access long vectors.)

void MFNVectorSetSetC(MFNVector,void (*)(int,double,void*),MFErrorHandler);
– Sets the routine used to change a coordinate. This is meant to be a fallback in case a routine has to deal with a vector of unknown type. (Performance suffers if used to access long vectors.)

void MFNVectorSetAdd(MFNVector,void (*)(void*,void*,void*),MFErrorHandler);
– Sets the routine used to add two vectors.

void MFNVectorSetDiff(MFNVector,void (*)(void*,void*,void*),MFErrorHandler);
– Sets the routine used to multiply a vector by a scalar.

void MFNVectorSetPrint(MFNVector,void (*)(FILE*,void*),MFErrorHandler);
– Sets the routine used to print a readable version of a vector.

19 Implementing an MFNSpace

The MFNSpace base class represents the embedding space. The ctor for an MFImplicitMF usually creates the space.

The MFNSpace is similar in design to the MFNVector described above, but with different access routines.

MFNSpace MFCreateNSpaceBaseClass(char *id,MFErrorHandler);
– This creates an empty MFNSpace and returns it to the user.

The "id" is a character string that will be returned by the MFNSpaceGetId routine, and can be used to check the type of a vector for example, before casting the data block to the struct used for this class.

char *MFNSpaceGetId(MFNSpace,MFErrorHandler); – Returns the identification string of a vector (i.e. the string passed to the base class ctor). base class ctor).

void MFNSpaceSetData(MFNSpace,void*,MFErrorHandler); – Sets the data pointer.

void *MFNSpaceGetData(MFNSpace,MFErrorHandler); – Sets the routine that is called when the last reference to the vector is Free'd. Note that the CreateBaseClass returns a vector with one reference.

void MFNSpaceSetWriteData(MFNSpace,writedata,MFErrorHandler);
– Sets the routine used to write a vector to file. The routine has the signature

```
void writedata(FILE*,MFNSpace,void*,MFErrorHandler);
```

void MFNSpaceSetFreeData(MFNSpace,void (*freedata)(void *),MFErrorHandler);
– Sets the routine that is called when the last reference to the vector is Free'd. Note that the CreateBaseClass returns a vector with one reference.

void *MFNVectorGetData(MFNVector,MFErrorHandler); – returns the data pointer of a vector.

void MFNSpaceSetDistance(MFNSpace,distance,MFErrorHandler);
– Sets the routine that computes the distance between two vectors in the space. The routine has the signature:

```
double distance(MFNSpace,MFNVector,MFNVector,void*,MFErrorHandler);
```

void MFNSpaceSetInnerProduct(MFNSpace,inner,MFErrorHandler);
– Sets the routine that computes the inner product of two vectors in the space. The routine has the signature:

```
double inner(MFNSpace,MFNVector,MFNVector,void*,MFErrorHandler);
```

void MFNSpaceSetDirection(MFNSpace,direction,MFErrorHandler);
– Sets the routine that computes the direction from one vector to another. The routine has the signature:

```
void direction(MFNSpace,MFNVector,MFNVector,MFNVector,void*,MFErrorHandler);
```

void MFNSpaceSetAdd(MFNSpace,add,MFErrorHandler); – Sets the routine that computes the sum oof two vectors in the space. The routine has the signature:

```
void add(MFNSpace,MFNVector,MFNVector,MFNVector,void*,MFErrorHandler);
```

void MFNSpaceSetScale(MFNSpace,scale,MFErrorHandler); – Sets the routine that multiplies a vector in the space by a scalar. The routine has the signature:

```
void scale(MFNSpace,double,MFNVector,MFNVector,void*,MFErrorHandler);
```

20 Implementing an MFNRegion

The MFNRegion represents a subset of an n-dimensional space. The only real function is supplies is a ”test”.

MFNRegion MFNRegionCreateBaseClass(char*,MFErrorHandler);
– creates an empty MFNRegion and returns it to the user.

The ”id” is a character string that will be returned by the MFNSpaceGetId routine, and can be used to check the type of a vector for example, before casting the data block to the struct used for this class.

void MFNRegionSetTest(MFNRegion,test,MFErrorHandler); – sets the routine used to test if a vector is inside the region. The routine has the signature

```
int test(MFNVector,void*,MFErrorHandler);
```

and should return 1 if the point is in the region, and 0 if it is not. The data pointer is passed as the third argument.

void MFNRegionSetData(MFNRegion,void*,MFErrorHandler); – Sets the data pointer.

void *MFNRegionGetData(MFNRegion,MFErrorHandler); – Returns the data pointer.

void MFNRegionSetFreeData(MFNRegion,void (*)(void*),MFErrorHandler);
– Sets the routine that is called when the last reference to the vector is Free’d. Note that the CreateBaseClass returns a region with one reference.

void MFNRegionSetWriteData(MFNRegion,writedata,MFErrorHandler);
– Sets the routine used to write a region’s data to file. The routine has the signature

```
void writedata(FILE*,void*,MFErrorHandler);
```

21 Implementing an Implicitly Defined Manifold

An Implicity Defined Manifold is a definition of a set of points. In principle the points satisfy a continuous equation, but we only require a way of projecting points onto the manifold, a procedure for computing an orthonormal basis for the tangent space, and a local scale.

To implement your own you write a constructor (ctor), and provide certain basic routines. The ctor calls MFIMFCreateBaseClass, and then the MFIMFSet

MFImplicitMF MFIMFCreateBaseClass(n,k,char *id,MFErrorHandler); – creates an empty MFImplicitMF and returns it to the user. The integers n and k are the dimension of the embedding space and the dimension of the manifold respectively.

The "id" is a character string that will be returned by the MFIMFGetId routine, and can be used to check the type of a manifold for example, before casting the data block to the struct used for this class.

void MFIMFSetSpace(M,MFNSpace,MFErrorHandler); – Sets the embedding space.

void MFIMFSetData(M,void*,MFErrorHandler); – Sets the data pointer.

void MFIMFSetFreeData(MF,void (*)(void*),MFErrorHandler); – Sets the routine that is called when the last reference to the vector is Free'd.

void MFIMFSetProject(M,projectPoint,MFErrorHandler); – Sets the routine used to project a point in n-space onto the manifold orthogonal to a k dimensional linear subspace. The project routine must have the signature

```
int ProjectPoint(int n,int k,MFNVector u,MFNKMatrix Phi,MFNVector v,void*
```

The data pointer is passed as the last argument.

void MFIMFSetTangent(M,tangent,MFErrorHandler); – Sets the routine used to compute an orthonormal basis for the tangent space of the manifold at a point on the manifold. The tangent routine must have the signature

```
    void tangent(int n,int k,MFNVector u,MFNKMatrix Phi,void*,MFErrorHandler)
```

The data pointer is passed as the last argument. PhiG, and Phi will have been allocated before the routine is called.

void MFIMFSetTangentWithGuess(M,tangentWG,MFErrorHandler); – Sets the routine used to compute an orthonormal basis for the tangent space of the manifold at a point on the manifold. The tangent routine must have the signature

```
   void tangentWG(int n,int k,MFNVector u,MFNKMatrix PhiG,MFNKMatrix Phi,voic
```

The data pointer is passed as the last argument. The guess is in PhiG, and Phi will have been allocated before the routine is called.

void MFIMFSetScale(M,scale,MFErrorHandler); – Sets the routine used to compute an estimate of the radius to be used for a chart centered at a point on the manifold with tangent space Phi. The scale routine must have the signature

```
    double scale(int n,int k,MFNVector u,MFNKMatrix Phi,void*,MFErrorHandler)
```

The data pointer is passed as the last argument.

void MFIMFSetProjectForSave(M,project,MFErrorHandler); – Sets the routine used to project a point for saving to disk. The project routine must have the signature

```
   int project(MFNVector u,double*,void*,MFErrorHandler);
```

The data pointer is passed as the last argument. If u or the Pu is passed as NULL the routine is expected to return the required length of Pu. This is used to allocate Pu before calling the project routine with a non-NULL u.

void MFIMFSetProjectForDraw(M,project,MFErrorHandler); – Sets the routine used to project a point for plotting. The project routine must have the signature

```
int project(MFNVector u,double *Pu,void*,MFErrorHandler);
```

The data pointer is passed as the last argument. If u or the Pu is passed as NULL the routine is expected to return the required length of Pu. This is used to allocate Pu before calling the project routine with a non-NULL u.

void MFIMFSetProjectForBB(M,project,MFErrorHandler); – Sets the routine used to project a point for the hierarchical bounding box used to find the charts near a new point. The projection need not be linear, but the dimension should probably be at least k. The project routine must have the signature

```
int project(MFNVector u,double *Pu,void*,MFErrorHandler);
```

The data pointer is passed as the last argument. If u or the Pu is passed as NULL the routine is expected to return the required length of Pu. This is used to allocate Pu before calling the project routine with a non-NULL u.

void MFIMFSetWriteData(M,writedata,MFErrorHandler); – Sets a routine to write the manifold's data to a disk file. The routine must have the signature

```
void writedata(FILE*, void*,MFErrorHandler);
```

The data pointer is passed as the second argument.

void MFIMFSetEvaluate(M,eval,MFErrorHandler); – Sets a routine to evaluate the function "F" that defines the manifold. This isn't needed explicitly by the continuation algorithm, but it seems like a reasonable thing to want to do. The routine must have the signature

```
void eval(int n,MFNVector u, MFNVector f,void*,MFErrorHandler);
```

The data pointer is passed as the last argument.

void MFIMFSetApplyJacobian(M,apply,MFErrorHandler); – Sets a routine to apply the Jacobian of the function "F" that defines the manifold to a set of vectors (passed as a matrix). This isn't needed explicitly by the continuation algorithm, but it seems like a reasonable thing to have. The routine must have the signature to want to do.

```
void apply(int n,int k,MFNVector u,MFNKMatrix Phi,MFNKMatrix FuPhi,void*,MFEr
```

The data pointer is passed as the last argument.

void MFIMFSetApplySecDer(M,apply,MFErrorHandler); – Sets a routine to apply the second derivative of the function "F" that defines the manifold to a pair of vectors. This isn't needed explicitly by the continuation algorithm, but it seems like a reasonable thing to have. The routine must have the signature to want to do.

```
void apply(int n,int k,MFNVector u,MFNVector phi0,MFNVector phi1,MFNVector ps
```

The data pointer is passed as the last argument.

void MFIMFSetStop(M,stop,MFErrorHandler); `int stop(MFImplicitMF,MFNVector,MFN`

The data pointer is passed as the last argument.

void MFIMFSetR(M,double,MFErrorHandler); – Sets a radius that is used if the scale routine (provided in MFIMFSetScale) is not present.

void MFIMFSetSingular(M,singular,MFErrorHandler); – Sets a routine to compute the $(k+1)$st Null vector of the Jacobian F_u at a singular point u. pair of vectors.

```
int singular(int n,int k,u,Phi,phi1,void*,MFErrorHandler);
```

Phi is the tangent space a little away from the singular point. The new Null vector phi1 is expected to be outside the span of the columns of Phi. The data pointer is passed as the last argument.

void MFIMFSetSetStability(M,stability,MFErrorHandler); – Sets a routine to assign an index to a point on the manifold. In addition to the Stop routine described above, if the index changes between two points a singular point is located by bisection.

```
void stability(M,u,Phi,void*,MFErrorHandler);
```

The data pointer is passed as the last argument.