VBM and MCCC: Packages for Objected Oriented Visualization and Computation of Bifurcation Manifolds *

Randy C. Paffenroth†

Abstract

This case study describes two software packages, VBM (Visualization of Bifurcation Manifolds) and MCCC (Multi-parameter Computation and Continuation Code) which provide an object oriented (OO) framework for visualizing, steering and implementing multi-parameter continuation algorithms.

1 Introduction

The computation at the heart of many mathematical models of physical systems is the solution of a parameter-dependent system of nonlinear equations,

$$f(x) = 0,\ f : \mathbb{R}^m \to \mathbb{R}^n,\ m \geq n. \tag{1}$$

Generically, the solution set of such an equation is a collection of $m - n$ dimensional manifolds and their singular points, e.g. folds, bifurcation points, etc., embedded in m dimensional space. We call this set of manifolds and singular points a **bifurcation manifold**, and any representation or visualization thereof we call a **bifurcation diagram**. Often, some of the components of the argument of f are specially designated as **parameters** based upon the structure of the underlying problem. The computation of such bifurcation manifolds is the domain of numerical parameter continuation algorithms.

The two softwares described here are VBM (Visualization of Bifurcation Manifolds) and MCCC (Multi-parameter Computation and Continuation Code) which together provide an object oriented (OO) framework for visualizing, steering and implementing multi-parameter continuation algorithms.

VBM provides a user interface utilizing accelerated three-dimensional graphics that allows an interactive exploration of projections of the space in which bifurcation manifolds are naturally embedded, even when m is large. VBM can also be used to interactively steer the computation of the bifurcation manifold via continuation algorithms running on distributed CPUs. MCCC, on the other hand, is an OO toolkit for the implementation of multi-parameter continuation programs. It provides a high-level framework, written in Java, which utilizes both pure Java numerical routines and native methods. It is designed to allow the user to easily experiment with different discretization, projection, and continuation methods.

*This work has been supported by an AASERT award from the US Air Force.

†Institute for Physical Science and Technology, University of Maryland and Départment de Mathématique, EPFL

2 Continuation Methods

A continuation method consists of a procedure in which a known point on the bifurcation manifold is used to discover new points which also lie on the bifurcation manifold. There are various classes of such methods and in this paper we will restrict our attention to the class of **predictor corrector** or PC methods. In PC methods one uses some set of known solutions to problem (1) to generate a new point x^*. The point x^* is typically not on the bifurcation manifold so one then corrects it with a given tolerance, perhaps using a Newton like method, to a point x on the bifurcation manifold. One obvious issue, which we will not discuss here, is how one constructs the initial set of solutions to start such a procedure. There is a large body of literature on the subject of continuation methods and our intention is merely to motivate what follows. More information can be found in [1, 2, 5, 4].

The question which now arises is how does one easily and efficiently implement such algorithms in an OO framework. One way to do this is to decompose the PC continuation procedure, from a conceptual perspective, as shown in the class diagram in figure 1 [3, 9]. The main idea is to encapsulate the structure of the nonlinear function (1) into the **Function object**. If the structure of the Function object allows more efficient algorithms to be used, e.g. if the Jacobian of the nonlinear function is sparse, the Function object can communicate this fact by implementing various interfaces. Other objects may then inspect the Function object and decide whether to use standard algorithms or the methods provided by the Function object.

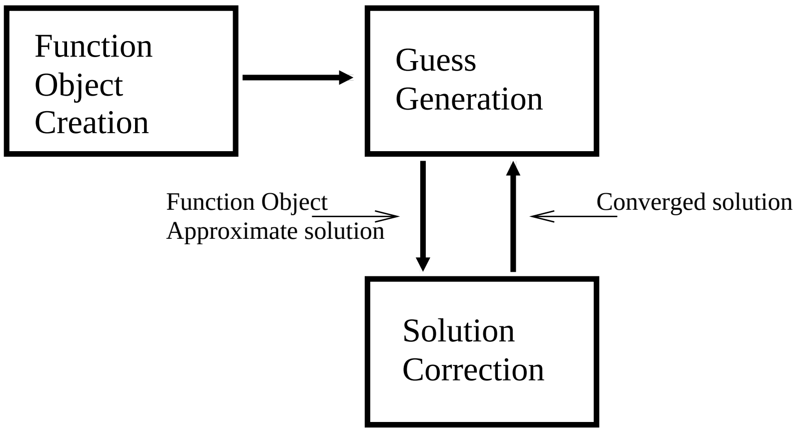

FIG. 1. *Conceptual perspective class diagram for the PC continuation algorithm. To start the process a given problem must be placed into the form of equation (1), which, in keeping with the theme of object oriented programming, we call a Function object. The Function object and some set of solutions to the problem are then passed to a Guess Generation object which constructs a new approximate solution to the problem. This guess and the original Function object are passed to a procedure which projects the guess onto the bifurcation manifold, and returns the converged solution to the Guess Generation object. The process can then be repeated.*

Not all problems that we wish to solve naturally have the form (1). Often it is necessary to recast the problem in some way to achieve a form such as (1). For example, in the work of [11, 15] the problem of interest is a 14 dimensional 2 point boundary value problem for a set of Hamiltonian ODEs which describes a continuum rod model of DNA. In such a case we discretize the ODEs using the Collocation method to achieve a problem of the appropriate type [5, 4].

Mappings of the problem from its natural form to form (1) often imposes some type of structure on the discretized equations (e.g. a banded Jacobian). We now find ourselves with two contradictory goals. First, we would like our implementation to be as general as possible; it should be applicable to problems with many different structures. Second, we would like our implementation to be as efficient as possible; it should use the structure of the problem to accelerate the calculations and facilitate visualization (e.g. through the use of distinguished diagrams [14]). These two goals have driven the design of VBM and MCCC.

3 VBM

The unifying theme in VBM is to use visualization and special projections of the bifurcation manifold to facilitate efficient computation of the solution set, organize and allow easy access to the large data sets that arise, and most of all to generate mathematical understanding of the features and properties of the solution set.

The core of VBM consists of data management and manipulation routines for bifurcation diagrams. At this level, bifurcation diagrams are stored as vectors of floating point numbers along with connectivity information and user defined tags. These user defined tags are one of the key features of VBM and they are where arbitrary problem dependent information may be stored. To maintain flexibility, the tags are generally not interpreted or processed by the core VBM code, but are only stored for later retrieval by user defined routines. In addition, VBM defines three application programming interfaces (API) which can be used to add custom visualization and computation modules, see figure 2.

The **bifurcation visualization** API allows the user to write modules to visualize and interact with the bifurcation diagrams. The bifurcation visualization API is important to VBM in that most of the user interaction with the bifurcation diagram, such as attaching probes and computations, occurs through it. The bifurcation API is complicated because of the above interactivity requirements, and the fact that the data structure being visualized is internal to VBM. On the other hand, if the bifurcation diagram has an especially complicated form, such as three or higher dimensional manifolds, other visualization tools, such as GeomView [17] and VtK [19], can be made to be compliant with the bifurcation visualization API and can be used within VBM. Examples of bifurcation visualization modules can be seen in figures 3, 4, and 5.

While bifurcation visualization modules are used to visualize the bifurcation diagram as a whole, it is often useful to have additional, more detailed, information about individual points of the diagram. This is where the **data probe** API is used. The user can interactively attach markers to points on a bifurcation diagram. Data probes can then be attached to the markers to show more detailed information about the designated points, and the markers can then be moved about the diagram to explore the relationship between the structure of the bifurcation diagram and the information presented in the data probe. The data probe API is designed to be simple and concise since it is envisioned that users will often want to add modules which perform problem-specific visualization functions. Examples of data probe modules can be seen in figures 3 and 4.

Both the bifurcation visualization modules and the data probe modules work with static representations of the bifurcation diagram. The compute engine API, on the other hand, allows the user to make interactive changes to the bifurcation manifold. Points can be marked on a bifurcation diagram and compute engines can be attached to the marker which allow interactive extension of the data. Current compute engines include AUTO

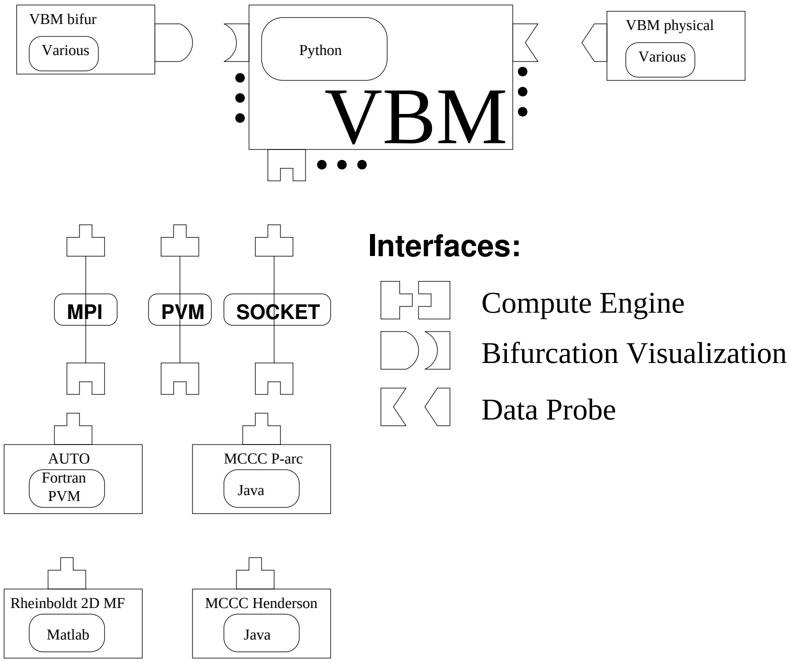

FIG. 2. *Block diagram of the VBM APIs. The interface blocks show where external programs may be used with VBM. Of special interest are the compute engines. The names of the modules are shown in the top of each box, while the implementation language is shown in the rounded box inside each module. The lines with compute engine interfaces at each end represent networking interfaces, which allow any compute engine to be executed on a computer other than the one on which VBM itself is running (e.g. a remote supercomputer). The module named "AUTO" uses [6], "MCCC P-arc" is based on [5, 4]. "Rheinboldt 2D MF" is based on [18], and "MCCC Henderson" is based on [10].*

[6], a Matlab implementation of the Rheinboldt moving frame algorithm [18], and various algorithms implemented in MCCC (e.g. a two parameter algorithm of Henderson [10]). Any continuation code which can be made to comply with the compute engine API can be utilized from within VBM using a single coherent interface. VBM also provides the ability to run any compliant compute engine on remote computing resources, while having the results automatically transmitted and displayed on the local workstation. Examples of 1D, 2D, and 3D continuations can be seen in figure 5.

The core of VBM is written entirely in Python [12], while the current set of modules are written in a variety of languages (see table 1). From the inception of the project one of the central themes has been to use as much pre-existing software as possible. The use of a high level scripting language, such as Python, has been very useful in this respect. Often wrappers must be written to make a given software package compatible with the VBM APIs. Python, with its easy access to system resources and native C and C++ functions has made this much easier.

In many of the modules developed in our group we gain access to accelerated 3D graphics via OpenGL through a locally developed Python/TK/OpenGL/C++ package called DataViewer. While DataViewer forms an integral part of VBM, it is also available separately as a stand-alone program under the GNU library general public license. DataViewer has both standard and user definable OO data-structures, provides generic

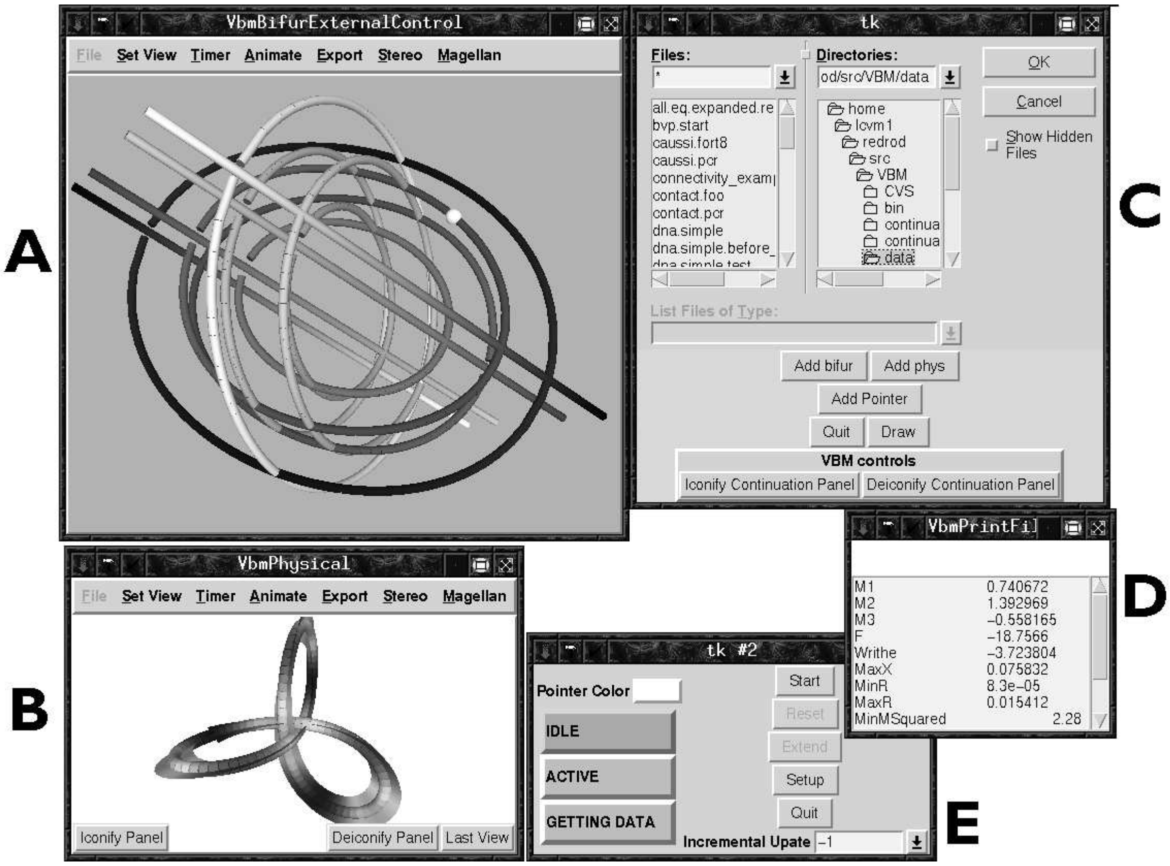

FIG. 3. *Pictures of some of the VBM interface elements and modules. The example problem shown here is a continuum rod model of DNA. The window marked A is a bifurcation visualization module written in DataViewer. The window marked B is a data probe showing the solution corresponding to the point in the bifurcation visualization window marked with the white ball. The window marked C is the main VBM control panel . The window marked D is a data probe displaying numerical data and the window marked E is a compute engine, both of which are also attached to the white marker.*

viewing facilities (rotation, zoom, etc. through either mouse, key-bindings or 6D joystick), hardcopy output (PPM), VRML output, animation, stereo viewing, etc. DataViewer is analogous to the widely distributed GeomView program, but the data structures and implementation are more oriented toward interactive high-performance computing.

4 MCCC

Now that we have introduced the overall structure of VBM, we will closely examine the compute engine modules. Some VBM compute engine modules are drawn directly from the work of other groups, for example [6], and are used in the VBM framework with very little modification, often with just a thin interface layer. On the other hand, we have also written a framework for developing continuation methods which is called MCCC. It can be used both as a stand-alone package and as a simple framework in which to develop VBM compute engines. At its core, MCCC is an implementation of the conceptual class diagram in figure 1 as a set of Java classes and interfaces. The unifying themes in MCCC are to use

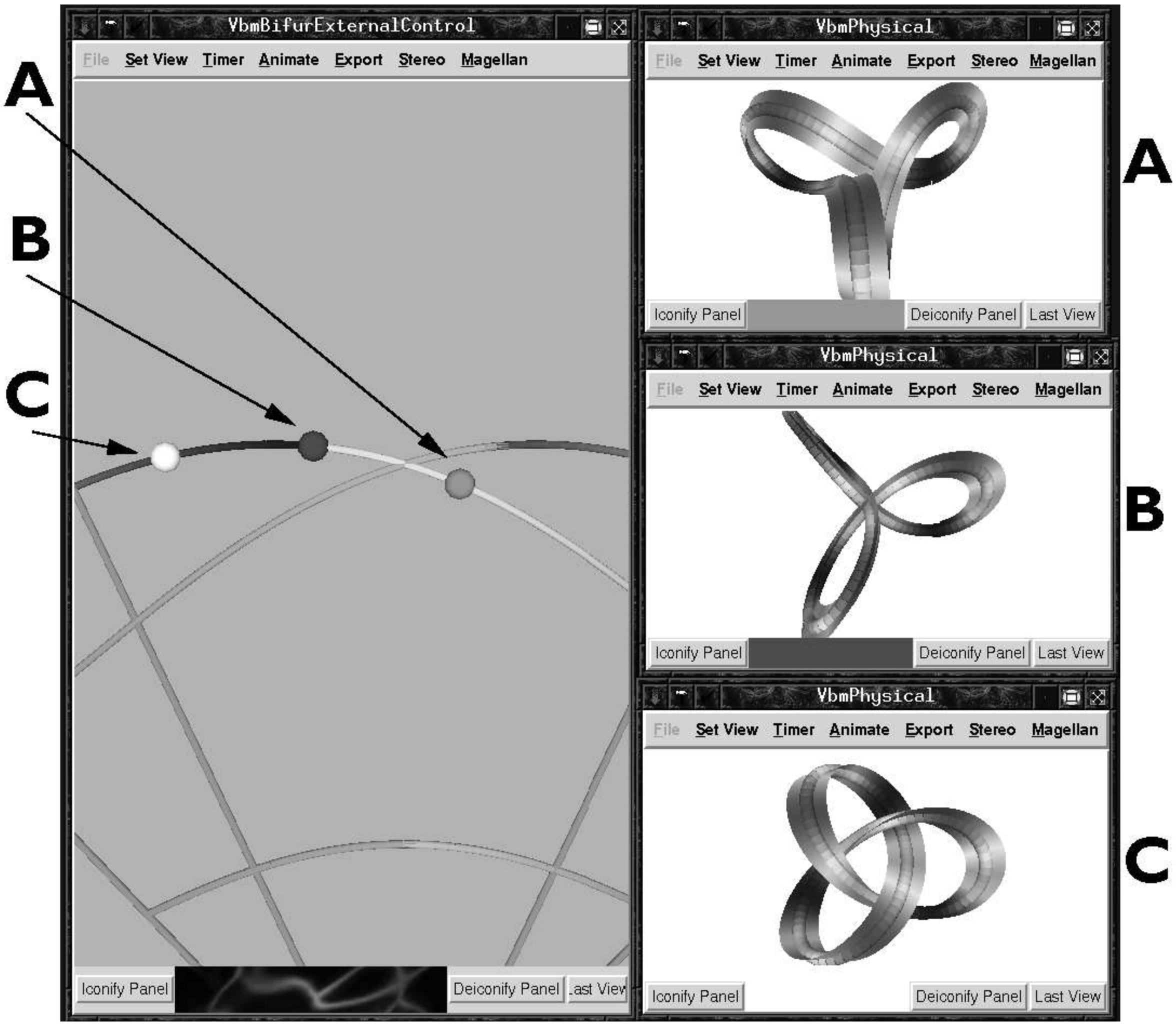

FIG. 4. *The left hand window is a bifurcation visualization window showing a bifurcation diagram for the continuum rod model of DNA with 3 markers attached to it. Each of the windows on the right is a data probe showing the solution to the problem at the correspondingly marked point. Using such a visualization one can observe that as the markers move from right to left the solution goes from an unknotted configuration, through a self contact, into a trefoil knot.*

an OO approach to encapsulate the interrelationship between the various components of a numerical continuation algorithm and to make the implementation of such algorithms easy and efficient. The implementation class hierarchy of MCCC is shown in figure 6.

The most important element in the class hierarchy is the Function interface, because it encapsulates the implementation of (1). Function objects can be constructed in several different ways; directly, in the case of inherently algebraic problems, or as discretizations of continuous problems. Some methods of constructing functions impose structure on the Function objects, and the use of this structure is the key to efficient computations. MCCC implements this in a general way by defining other interfaces, such as "RootFinder" and "HasJacobian", which Function objects can also implement. If a Function object implements one of these interfaces it implies that it provides a good algorithm for the procedure specified in the interface. Higher level objects can use Java's introspection

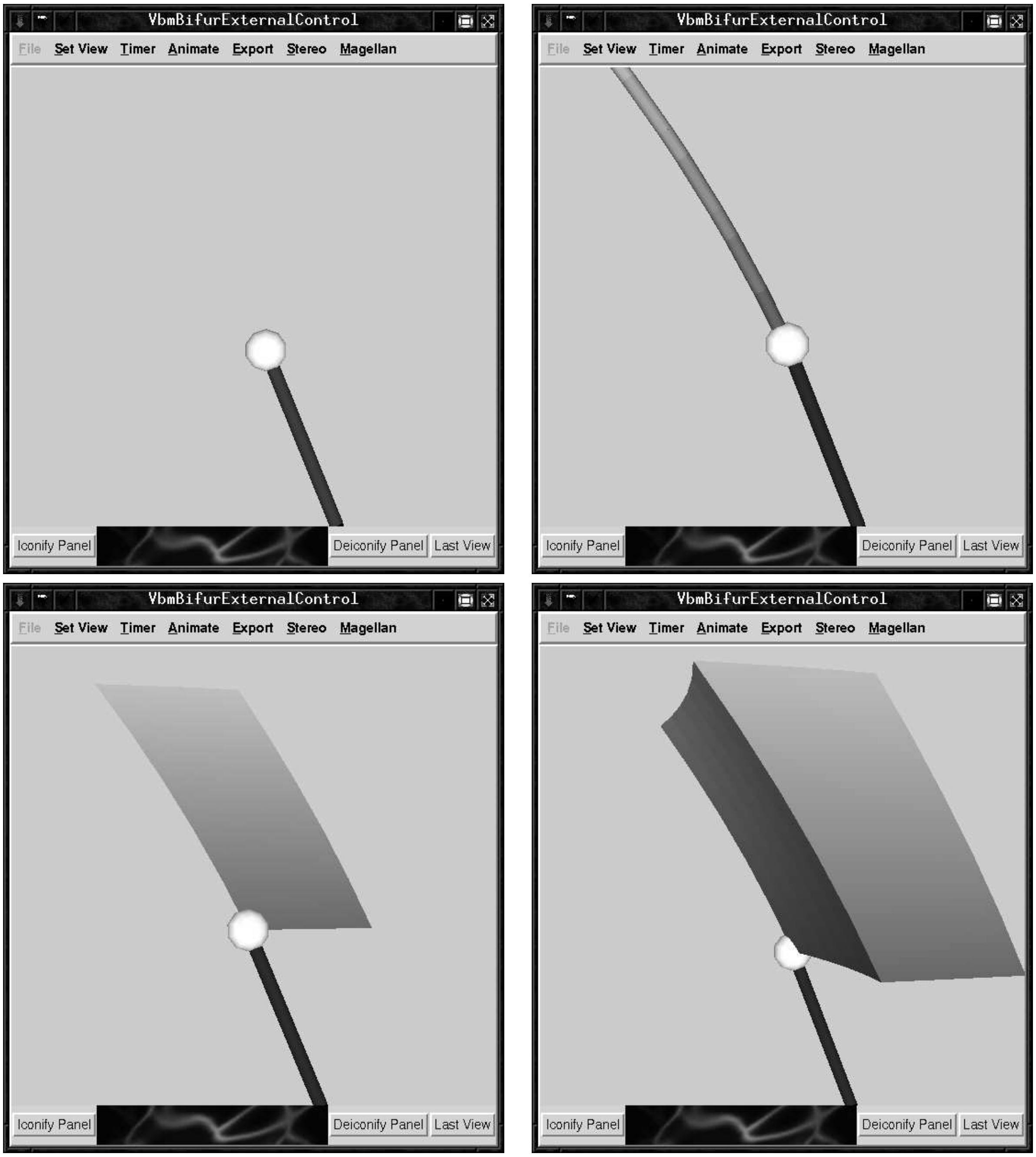

FIG. 5. *The image in the top-left shows the original data set, before any parameter continuation. The other 3 image show the same data set after a 1D, 2D, and 3D continuation, respectively.*

services (through the Class class) to examine objects which are passed to them and then decide if they will use standard algorithms or the methods offered by the Function object. This allows structures such as sparse Jacobians to be used for those Function objects for which it is appropriate, while standard methods can be applied as a fall-back. We also note that, as of this writing, Java is somewhat slower than native languages. Consequently, another way in which to accelerate computations is the judicious use of native methods.

Table 1
All of the currently developed VBM modules and the languages in which they are implemented.

Bifurcation modules	VBMBifur	DataViewer, C++, Python
	VBMBifurSimple	Python
Data probe modules	VBMPhysical	DataViewer, C++, Python
	VBMGeneral	Python
	VBMKnotPhysical	DataViewer, C++, Python
	ViewAUTO	Python, DataViewer, C++, Fortran
	ViewAUTOIVP	Python, DataViewer, C++, Fortran
Compute Engines	AUTO	Python, Fortran
	AUTOIVP	Python, Fortran
	Rheinboldt	Python, Matlab
	Henderson	Java (MCCC)

Currently, the main example of this is in the "Solution Correction" block of figure 1. While there are Java implementations of this block, there is a version which consists of a thin Java wrapper over a Fortran Powell-Hybrid routine from MinPack [16]. For high-dimensional problems the added speed of the Fortran routines is very useful, but there is a tradeoff. You gain speed by using native code, but you lose optimizations based on the structure of the problem, as well as portability.

The computational capabilities of VBM and MCCC are designed to be highly scalable. For smaller problems the entire program, both graphics and compute engine, can be run on a moderate performance PC. For larger problems the program can make full use of a high-performance graphics workstation, and for the largest problems VBM includes a menu driven networking interface to allow MCCC to run on remote supercomputer facilities, with the graphics front-end running on a local workstation. In particular VBM and MCCC are designed to exploit a two-tier parallelism available in parameter continuation problems. First there is a standard, closely coupled, fine grain parallelization of the computation of a solution at a fixed set of parameter values (which can, for example, be implemented using threads and a parallelizing compiler on a shared memory architecture). Second there is a coarse grain, loosely coupled parallelization allowing many different parameter continuation steps within the same bifurcation manifold to be carried out simultaneously (which can be efficiently implemented via sockets, MPI or PVM).

5 Conclusions

In the design and implementation of VBM and MCCC several lessons have been learned. We have found that having the top-level program written in an interpreted scripting language, such as Python, is very useful. It shortens the development cycle by allowing rapid prototyping and testing of algorithms. It also provides a very good "glue" for putting together modules, written in a variety of languages, into a coherent whole. The OO nature of Python has been particularly useful since the interfaces to external modules can be very concisely written in terms of inheritance. Java, on the other hand, with its strong typing, explicit data encapsulation, and object introspection utilities, allows numerical algorithms to be efficiently implemented, and, if speed is of the utmost importance, wrappers for native language routines can also be used.

The target user group for VBM and MCCC are academic researchers using computa-

- class java.lang.Object
  - interface CH.epfl.lcvm.mccc.Continuation1D
  - interface CH.epfl.lcvm.mccc.Continuation2D
  - class CH.epfl.lcvm.mccc.ContinuationNatural (implements CH.epfl.lcvm.mccc.Continuation1D)
  - class CH.epfl.lcvm.mccc.ContinuationPseudoArclength (implements CH.epfl.lcvm.mccc.Continuation1D)
  - class CH.epfl.lcvm.mccc.ContinuationRheinbolt2D
  - class CH.epfl.lcvm.mccc.FDDiscretizationContact (implements CH.epfl.lcvm.mccc.Function)
  - interface CH.epfl.lcvm.mccc.Function
  - class CH.epfl.lcvm.mccc.FunctionBase (implements CH.epfl.lcvm.mccc.Function)
    - class CH.epfl.lcvm.mccc.FunctionConcatenation
    - class CH.epfl.lcvm.mccc.FunctionFixed
    - class CH.epfl.lcvm.mccc.FunctionSphere (implements CH.epfl.lcvm.mccc.HasJacobian)
  - interface CH.epfl.lcvm.mccc.HasJacobian
  - interface CH.epfl.lcvm.mccc.IVPSolver
  - class CH.epfl.lcvm.mccc.IVPSolverBase (implements CH.epfl.lcvm.mccc.IVPSolver)
    - class CH.epfl.lcvm.mccc.IVPEuler
    - class CH.epfl.lcvm.mccc.IVPRungeKutta2
  - interface CH.epfl.lcvm.mccc.Matrix
  - class CH.epfl.lcvm.mccc.MatrixArray (implements CH.epfl.lcvm.mccc.Matrix, java.lang.Cloneable)
  - class CH.epfl.lcvm.mccc.NewtonRaphson (implements CH.epfl.lcvm.mccc.RootFinder)
  - class CH.epfl.lcvm.mccc.NumericalJacobian (implements CH.epfl.lcvm.mccc.HasJacobian)
  - interface CH.epfl.lcvm.mccc.ODEFunction
  - class CH.epfl.lcvm.mccc.ODEFunctionBase (implements CH.epfl.lcvm.mccc.ODEFunction)
    - class CH.epfl.lcvm.mccc.ODEFunctionCircle (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionDNA1ParNative (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionDNA6ParNative (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionRod (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
  - interface CH.epfl.lcvm.mccc.ODEFunctionWithBC (extends CH.epfl.lcvm.mccc.ODEFunction)
  - class CH.epfl.lcvm.mccc.OneDDiscretization (implements CH.epfl.lcvm.mccc.Function)
    - class CH.epfl.lcvm.mccc.CollocationDiscretization
    - class CH.epfl.lcvm.mccc.FDDiscretization
  - interface CH.epfl.lcvm.mccc.OneDMesh
  - class CH.epfl.lcvm.mccc.OneDMeshUniform (implements CH.epfl.lcvm.mccc.OneDMesh)
  - class CH.epfl.lcvm.mccc.PowellHybridNative (implements CH.epfl.lcvm.mccc.RootFinder)
  - interface CH.epfl.lcvm.mccc.RootFinder
  - class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
      - class CH.epfl.lcvm.mccc.MCCCException
        - class CH.epfl.lcvm.mccc.NoConvergenceException
        - class CH.epfl.lcvm.mccc.SolverException

FIG. 6. *All of the Java classes and interfaces along with their inheritance hierarchy.*

tional bifurcation codes either as a research topic in itself, or for use in other research and applications areas. VBM and MCCC are available under the GNU general public license and GNU library general public license respectively.

More information on the software developed at the Laboratory for Computation and Visualization in Mathematics and Mechanics ($LCVM^2$) can be found at the URL `http://lcvmwww.epfl.ch`.

References

[1] E. L. Allgower and K. Georg, *Numerical Continuation Methods*, Springer-Verlag, 1990.

[2] ——, *Numerical path following*, in Handbook of Numerical Analysis, P. G. Ciarlet and J. L. Lions, eds., vol. 5, Elsevier Science B. V., 1997, pp. 3–208.

[3] S. Cook and J. Daniels, *Designing Object Systems: Object-oriented Modeling with Syntropy*, Prentice Hall International, 1994.

[4] E. J. Doedel, H. B. Keller, and J. P. Kernévez, *Numerical analysis and control of bifurcation problems, part II*, International Journal of Bifuration and Chaos, 4 (1991), pp. 745–772.

[5] E. J. Doedel, H. B. Keller, and J. P. Kernévez, *Numerical analysis and control of bifurcation problems, parts I and II*, Int. J. of Bif. and Chaos, 3, 4 (1991), pp. 493–520, 745–772.

[6] E. J. Doedel and X. Wang, *AUTO 94, Software for Continuation and Bifurcation Problems in Ordinary Differential Equations*, Caltech, Dept of Applied Mathematics, 1994.

[7] G. Domokos and R. C. Paffenroth, *Case study: Visualization for boundary value problems*, in IEEE Visualization '94 Conference Proceedings, R. D. Bergeron and A. E. Kaufman, eds., IEEE Computer Society Press, 1994, pp. 345–348.

[8] ——, *PCR - a visualization tool for multi-point boundary value problems*, Tech. Rep. BN-1167, Institute for Physical Science and Technology, University of Maryland, College Park, 1994.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patters: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

[10] M. E. Henderson, *Computing implicitly defined surfaces: Two parameter continuation*, tech. rep., IBM Research Division, T. J Watson Research Center, 1993.

[11] Y. Li and J. H. Maddocks, *On the computation of equilibria of elastic rods, Part I: Integrals, symmetry and a Hamiltonian formulation*, J. Comp. Phys., (to appear).

[12] M. Lutz, *Programming Python*, O'Reilly and Associates, 1996.

[13] J. Maddocks, R. Manning, R. Paffenroth, K. Rogers, and J. Warner, *Interactive computation, parameter contiuation, and visualization*, Int J. Bifurcation and Chaos, 7 (1997), p. 1699.

[14] J. H. Maddocks, *Stability of nonlinearly elastic rods*, Arch. Rational Mech. Anal., 85 (1984), pp. 180–198.

[15] R. S. Manning, J. H. Maddocks, and J. D. Kahn, *A continuum rod model of sequence-dependent DNA structure*, J. Chem. Phys., 105 (1996), pp. 5626–5646.

[16] J. Moré, B. Garbow, and K. Hillstrom, *Minpack*, Argonne National Laboratory.

[17] T. Munzner, S. Levy, and M. Phillips, *Geomview Users Guide*, Geometry Center, University of Minnesota.

[18] W. C. Rheinboldt, *On the computation of multi-dimensional solution manifolds of parameterized equations*, Numer. Math., 53 (1988), pp. 165–181.

[19] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 2nd ed., 1997.