py_bvp: A Universal Python Interface for BVP Codes

Jason J. Boisvert
Department of Computer Science
University of Saskatchewan
jjb701@mail.usask.ca

Paul H. Muir
Department of Mathematics and Computing Science
Saint Mary's University
muir@smu.ca

Raymond J. Spiteri
Department of Computer Science
University of Saskatchewan
spiteri@cs.usask.ca

Abstract

Boundary-value problems (BVPs) for ordinary differential equations arise in many important applications, and over the last few decades a number of high-quality software packages for this problem class have been developed. Some of these solvers have been designed for use on high-performance computers, and there is potential for further development in this direction. Unfortunately these codes, typically written in languages like Fortran or C, require complicated parameter lists and user-written subroutines. Moreover, researchers often want to try more than one solver on a given problem, and doing so can be challenging because the different interfaces to each package require distinctly different code to be written to describe the problem to be solved. In this paper, we present a Python environment called py_bvp that is specifically designed for the numerical solution of BVPs. In addition to providing a uniform interface that allows multiple BVP codes to be conveniently accessed, several tools are provided to support researchers who want to try to use these codes. Furthermore, the py_bvp environment is designed to allow additional BVP codes and tools to be easily added. We discuss the design decisions made to ensure that py_bvp is both user-friendly and easily expandable and give examples to illustrate its use.

1. INTRODUCTION

Boundary-value problems (BVPs) for ordinary differential equations (ODEs) arise frequently in scientific computing applications (see, e.g., [1], Chapter 1). Because the range of BVPs that arise is so diverse, no single numerical approach can be expected to perform well on all problems. As with other classes of problems, researchers must invest some time experimenting with different software packages with the hope of finding one that meets their needs for a particular problem. There are a number of high-quality BVP codes available. Some of the most popular codes are COLSYS [1], COLNEW [3], MIRKDC [7], TWPBVP [5], and TWPBVPC [4], all written in Fortran 77. However, using BVP software can be challenging because the different interfaces to each package are complex and require distinctly different code. For example, each BVP code mentioned above requires more than 15 parameters in the call to the primary solver routine, and most of these are unrelated to the mathematical specification of the problem. Therefore, researchers who wish to try to solve a given problem with multiple solvers essentially have to rewrite the problem description for each code. As well, there are a number of standard sub-tasks one often has to deal with when providing the input to a given solver or in processing the output from a given solver, and researchers usually have to write their own software or gather appropriate routines from various sources for these sub-tasks.

A simple and intuitive interface was a primary requirement for the Fortran 90/95 BVP code, BVP_SOLVER [26]. The BVP_SOLVER interface requires the use of a four-parameter initialization function and a three-parameter solver function. This is decidedly simpler than the interface of its predecessor, MIRKDC, which requires 20 parameters. To make this possible, numerous high-level features of Fortran 90/95 were exploited, such as dynamic memory allocation and optional parameters. The result is that BVP_SOLVER is not quite as fast as MIRKDC, but it is significantly faster than if it were written purely in a higher-level language such as Matlab or Python.

Matlab features two high-quality BVP codes, bvp4c [9] and bvp5c [10]. In both cases, the primary solver function requires only three parameters. Moreover, all parameters relate primarily to the mathematical specification of the BVP itself, thus leading to an intuitive interface. The interfaces also offer a good degree of portability between solvers; i.e., a problem written for bvp4c can generally be solved with bvp5c by simply changing the name of the primary solver function.

In this paper, we describe a new universal Python interface, called py_bvp, which provides a convenient environment specifically for the numerical solution of BVPs. Python has many high-level features that allow the creation of easy-to-use interfaces. This allows for the possibility of creating a simple Python interface to sit on top of high-quality numerical software. We take advantage of these features to provide an environment in which it is easy to use BVP codes; we discuss this further in Section 3 of the paper. py_bvp offers the user the opportunity to describe a given BVP in a straightforward manner so that it can be conveniently treated by any of the solvers available within the interface. py_bvp currently includes three

popular Fortran BVP codes, COLNEW, BVP_SOLVER, and TWPBVPC, but additional solvers can easily be added. Although we focus on Fortran BVP codes in this paper, we note that the Python C/API allows one to easily add C/C++ BVP codes as well. Examples of such codes of the latter type are nag_ode_bvp_fd_nonlin_gen (d02rac) from the NAG C library [17] and imsl_f_bvp_finite_difference from the Visual Numerics/IMSL C library [8]. In order to assist users in making use of the codes in py_bvp, the interface also provides several computational tools specific to the numerical solution of BVPs; currently py_bvp includes tools for computing numerical approximations to partial derivatives needed by some of the solvers and for converting one form of the initial solution approximation; we provide more details below. As well, the Python environment provides many useful general tools that can support a user trying to solve BVPs. An example of one such tool we employ later in this paper is matplotlib [13] from the Python graphing library. py_bvp can easily be extended in include other tools that might be useful for researchers attempting to solve BVPs. Tools planned for later addition to py_bvp are to provide support for error estimation, mesh refinement, discretization schemes, parameter continuation, etc., with the ultimate goal of providing a customized, extendable BVP-solving environment for the convenient, efficient numerical solution of BVPs by high-quality BVP codes, with support from a library of BVP specific tools.

py_bvp is also useful for BVP code developers. For example, the library of tools could be used by developers for the quick implementation and preliminary testing of prototype BVP solvers; i.e., py_bvp could allow a developer to explore new combinations of algorithms that are not available in current BVP solvers. The py_bvp environment could also be useful to BVP code developers because it would allow for convenient benchmarking of new codes against those already included in py_bvp.

py_bvp addresses the important question of how to effectively harness the computational power available through legacy BVP codes within a convenient, modern software development environment, specifically designed for the numerical solution of BVPs. High-performance computing can be included by interfacing with codes that take advantage of such environments. For example, the solution of BVPs on parallel computers could be implemented by extending py_bvp to provide interfaces to parallel BVP solvers, such as PMIRKDC [15]; using the facilities within py_bvp, it would be straightforward to develop an interface to this parallel code. Interfaces to tools that implement other high-performance computations such as the parallel solution of linear systems solvers — an important sub-task in the efficient numerical solution of BVPs — can also easily be added to py_bvp.

To our knowledge, py_bvp is the only BVP specific environment of its kind. Related work includes the much larger SciPy project [23], which is a Python-based environment for general scientific computing, and the massive Sage project [21], which is a Python-based environment that provides a uniform interface to access many open-source mathematical systems in a variety of areas including number theory, computer algebra, and numerical linear algebra. Also related is the PyIMSL Studio project [20], from Visual Numerics, which provides a Python-based environment for access to the algorithms in the IMSL C Library. The Common Component Architecture Project [6] is another example of a related project. The purpose of this project is to establish a definition of a standard component architecture for high-performance computing. Although the above software environments represent powerful frameworks for mathematical computations, they provide little that is specifically focused on the numerical solution of BVPs.

Python is a common and reasonable choice for the implementation of py_bvp. Two of the most obvious alternatives, Matlab [12] or Mathematica [11], are proprietary, and thus not necessarily available to everyone. Python is open source and freely available on a wide range of operating systems. Python is also is meant to be a much more general-purpose programming language, making it a better choice for software development.

This paper is organized as follows. We describe the three Fortran BVP codes presently provided in py_bvp in section 2. We discuss the design decisions made to ensure that py_bvp is both simple to use and easy to expand in section 3. We use py_bvp to solve a given BVP using the three codes currently featured in the environment in section 4. Finally, we provide some conclusions and a discussion of future work in section 5.

2. BVP CODES CURRENTLY INCLUDED IN PY_BVP

Here we briefly describe each of the three BVP codes currently available within the py_bvp interface. As we show below, it is straightforward to add other BVP codes, and we have plans to do so.

2.1. COLNEW: A Global Error Control, Collocation Code

The BVP code COLNEW is written in Fortran 77 and is a modification of the earlier BVP solver COLSYS. COLNEW computes an approximate solution to a system of m mixed-order ODEs

$$\mathbf{y}^{(\mathbf{d})}(x) = \mathbf{f}(x, \mathbf{z}(\mathbf{y}(x))), \quad a < x < b,$$

where

$$\mathbf{y}^{(\mathbf{d})}(x) = [y_1^{(d_1)}(x), \ldots, y_m^{(d_m)}(x)],$$

$$\mathbf{f}(x,\mathbf{z}(\mathbf{y}(x))) = [f_1(x,\mathbf{z}(\mathbf{y}(x))),\ldots,f_m(x,\mathbf{z}(\mathbf{y}(x))],$$

and

$$\mathbf{z}(\mathbf{y}(x)) = [y_1(x), y_1^{(1)}(x),\ldots,y_1^{(d_1-1)}(x),\ldots,$$
$$y_m(x), y_m^{(1)}(x),\ldots,y_m^{(d_m-1)}(x)],$$

with separated multi-point conditions

$$g_j(\zeta_j;\mathbf{z}(\mathbf{y}(\zeta_j))) = 0, \quad j = 1,\ldots,m^*,$$

where $\zeta_j \in [a,b]$ is the location of the jth condition and $m^* = \sum_{i=1}^{m} d_m$.

The numerical solution is obtained through the use of collocation. This involves representing the approximate solution as a piecewise polynomial; i.e., the approximate solution is a polynomial on each subinterval $[x_i, x_{i+1}]$ of a mesh

$$a = x_0 \le x_1 \le \cdots \le x_{N-1} \le x_N = b,$$

that partitions the problem domain. Each polynomial is of order $k+m$ (degree $< k+m$), where $k \ge m$ is the (user-specified) number of collocation points per subinterval and the piecewise polynomial is required to be C^{m-1} continuous at the internal mesh points. `COLNEW` uses a monomial basis proposed in [18] for the representation of the piecewise polynomial; i.e., the approximate solution is represented as a linear combination of monomial basis functions in which the unknown coefficients of the basis functions are determined by applying the collocation conditions and the multi-point and continuity conditions. Each collocation condition is obtained by requiring the approximate solution to exactly satisfy the BVP at a collocation point. In `COLNEW`, these points are chosen to be the images of the Gauss points on each subinterval of the mesh. The application of the above conditions leads to a system of nonlinear equations that is solved using a modified Newton iteration.

Once the nonlinear system is solved and the coefficients for the piecewise polynomial are obtained, the global error of the solution is estimated through a relatively low-cost estimate of the discretization error. If the norm of the global error is greater than a user-defined tolerance, the code generates a new mesh that attempts to approximately equidistribute the error, and then the process repeats. If this estimate of the global error satisfies the user tolerance, a more expensive but more robust global error estimate is computed through the use of Richardson extrapolation (see section 5.5.2 of [2]), and the termination criterion for acceptance of the approximate solution is based on this global error estimate. If this estimate is satisfied, the continuous approximate solution is returned to the user. If not, a new mesh is determined and the process repeats.

`COLNEW` has the advantage that it directly solves systems of mixed-order ODEs, whereas the other codes we discuss here can only solve systems of first-order ODEs. Of course, any system of mixed-order ODEs can be converted to a first-order system in a straightforward manner, e.g., see [24]; however, the resulting system generally takes longer to solve.

The interface to `COLNEW` is complicated by `Fortran` 77 constraints. For example, because default parameters cannot be defined, the subroutine requires users to define 17 parameters. Also, because `Fortran` 77 does not allow dynamic memory allocation, users must provide workspaces in the form of pre-declared integer and floating-point arrays.

A `Python` interface [27] for `COLNEW` has already been developed and we have incorporated this interface into `py_bvp`.

2.2. `BVP_SOLVER`: A Defect Control, Runge–Kutta Code

`BVP_SOLVER` is a `Fortran` 90/95 code that solves BVPs of the form

$$\mathbf{y}'(x) = \frac{1}{x-a}\mathbf{S}\,\mathbf{y}(x) + \mathbf{f}(x,\mathbf{y}(x),\mathbf{p}), \quad a \le x \le b, \qquad (1)$$

with separated two-point boundary conditions

$$\mathbf{g}_a(\mathbf{y}(a),\mathbf{p}) = \mathbf{0}, \quad \mathbf{g}_b(\mathbf{y}(b),\mathbf{p}) = \mathbf{0}.$$

The vector functions, $\mathbf{y}(x)$ and $\mathbf{f}(x,\mathbf{y}(x),\mathbf{p})$, each have m components, where m is the number of ODEs, and are defined in a manner similar to $\mathbf{y}(x)$ and $\mathbf{f}(x,\mathbf{z}(\mathbf{y}(x)))$ of the previous subsection. $\mathbf{p}$ is a vector of m_p unknown parameters to be determined. As well, $\mathbf{g}_a(\mathbf{y}(a),\mathbf{p})$ is vector function representing m_a boundary conditions applied at $x = a$, and $\mathbf{g}_b(\mathbf{y}(b),\mathbf{p})$ is a vector function representing m_b boundary conditions applied at $x = b$, where $m_a + m_b = m + m_p$. Unlike many other BVP codes, `BVP_SOLVER` allows for the option to solve BVPs with singularities at $x = a$ by having users define a constant matrix $\mathbf{S} \in R^{m\times m}$.

For a given mesh that partitions the problem domain, the ODEs are discretized using mono-implicit Runge–Kutta (MIRK) formulas of either order 2, 4, or 6, and the resultant equations, together with the boundary conditions, gives a system of nonlinear equations for the discrete solution approximations, $\mathbf{y}_i$, $i = 0,1,\ldots,N$, at the mesh points. The MIRK formulas can be found in [16]. MIRK continuous extensions of the discrete solution are used to obtain a C^1 continuous solution approximation, $\mathbf{u}(x)$, over the problem domain.

Because the approximate solution is C^1 continuous, the code is able to use a backward error approach to estimate and control the residual of the numerical solution

$$\mathbf{r}(x) = \mathbf{u}'(x) - \left[\frac{1}{x-a}\mathbf{S}\,\mathbf{u}(x) + \mathbf{f}(x,\mathbf{u}(x),\mathbf{p})\right].$$

This is in contrast to the other codes discussed here that attempt to control an estimate of the global error. By using the

residual, BVP_SOLVER is able to provide a computationally inexpensive and robust method for control of the computation. The code proceeds through a sequence of meshes, adaptively chosen based on the residual, until a residual-controlled numerical solution that satisfies the user tolerance is obtained; once this occurs, BVP_SOLVER can (optionally) compute an estimate of the global error (using Richardson extrapolation).

As mentioned in Section 1, BVP_SOLVER uses several high-level features of Fortran 90/95 to greatly simplify the interface; examples include the use of dynamic memory allocation to avoid the requirement of user-provided work arrays and the use of pre-defined function parameters to keep the amount of required input from the user to a minimum.

BVP_SOLVER also provides a finite-difference routine to further simplify the use of the code. This routine returns a numerical approximation of the partial derivatives for both the ODEs and the boundary conditions; these are required because the code uses a Newton iteration to solve the nonlinear equations. The other codes mentioned in this section require users to provide their own routines to evaluate the derivatives of the ODES and boundary conditions.

A Python interface [22] for BVP_SOLVER has already been developed, and we have incorporated this interface into py_bvp.

2.3. TWPBVPC: A Global Error Control, Runge–Kutta Code based on Deferred Correction

The code TWPBVPC is a modified version of the Fortran 77 code TWPBVP. It determines numerical solutions to BVPs of the form

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)), \quad a \le x \le b,$$

with separated two-point boundary conditions

$$\mathbf{g}_a(\mathbf{y}(a)) = \mathbf{0}, \quad \mathbf{g}_b(\mathbf{y}(b)) = \mathbf{0}.$$

Here the above vector functions are defined in a manner similar to those in the previous subsections.

TWPBVPC uses MIRK formulas of orders 4, 6, and 8 as potential methods of discretization. These formulas are employed within a deferred-correction approach; see [5] for additional details. This process provides discrete solution approximations on a mesh of points that partitions $[a,b]$ as well as an estimate of the global error associated with these solution approximations. If the user tolerance is satisfied, the code returns the discrete solution. If not, the code also estimates the conditioning of the BVP and uses a combination of a discretization error estimate and the conditioning estimate to determine the next mesh.

Despite TWPBVPC being a later version of TWPBVP, the interface remains fairly complicated. The solver subroutine requires over 40 parameters. These parameters include integer and floating-point workspaces. Further complicating the interface is the requirement of user-defined global variables.

We have developed a Python interface for TWPBVPC and describe it in the next section. To our knowledge, no such interface existed previously.

3. THE PY_BVP ENVIRONMENT

In this section, we describe py_bvp in detail. We began the development of py_bvp by first adding existing Python interfaces for COLNEW and BVP_SOLVER to py_bvp. We discuss the development of the Fortran to Python interfaces for the BVP codes in section 3.1. We describe how we have made it easy to add other BVP codes by careful implementation of the universal interface in section 3.2. We demonstrate how the user interface for py_bvp makes it easy to use in section 3.3.

3.1. Interfaces for the Fortran BVP codes

The Python C/API allows us to easily create our own interfaces. However, using the Python C/API requires us to write a large amount of code. Instead we chose to use of F2PY [19], which automatically generates the code for us. We used F2PY to create a Fortran to Python interface for TWPBVPC.

F2PY comes bundled with a command-line tool called f2py. The tool requires two sources of input from users.

First, the user must provide the Fortran source code; i.e., in this case, the original source code for TWPBVPC. However, in the case of the BVP_SOLVER interface, the code consists of an intermediate subroutine that processes the input from Python and then sends it to a BVP_SOLVER subroutine. The intermediate subroutine is used to work around a limitation of F2PY; user-created data types cannot be passed from Fortran to Python. BVP_SOLVER uses such a data type to store information about both the BVP and the solution. Therefore, the intermediate subroutine constructs the data type from input it receives from Python and then sends it to BVP_SOLVER subroutine; see [22] for additional details.

Along with the source code, we must also provide f2py with a *signature file*. This file tells Python the names of the Fortran functions accessible from Python as well as their respective parameter lists. This involves listing the name of each function as well as their parameters and respective types. As an example, we provide the signature file for the appsln subroutine for the COLNEW Python to Fortran interface [27] below.

```
subroutine appsln(x, z, fspace, ispace)
  double precision, intent(in) :: x
  double precision, dimension(ispace[3]),&
        intent(out) :: z
  double precision, dimension(*),&
```

```
        intent(in) :: fspace
    integer, dimension(*), intent(in) ::&
        ispace
end subroutine appsln
```

Although we can create our own signature file, the command-line tool can create a signature file for the Fortran code. The signature file is a plain text file written in a Fortran-like language specific to F2PY; see the F2PY user manual [19] for further details. The language has features to simplify the interface. For example, within the signature file we can assign default values for some parameters, e.g., sizes of work arrays. As a consequence, we can ignore such parameters when calling the codes from within py_bvp.

Further simplification of the interface for Fortran BVP codes can be done within Python; we discuss this further in the next section.

3.2. The Implementation of py_bvp

Although the initial implementation of py_bvp provides access to only three BVP solvers, we plan to add other solvers in future. With this in mind, implementation of py_bvp was built around a need for simple expandability. This was accomplished in two ways.

First, many object-oriented programming principles were used to simplify future expansion, most notably that of polymorphism. Figure 1 illustrates this relationship in py_bvp. Each interface class must be a child of the mostly abstract class, solverInterface. Also, the interface class must hold an instance of the class responsible for interacting with the Fortran routine for the BVP code. This means that expanding py_bvp is as simple as creating a child class to solverInterface and defining some required methods.

For example, adding BVP_SOLVER to py_bvp first requires users to create an interface class called pyBvp_SolverInterface, which must be a child of solverInterface. Within pyBvp_SolverInterface, multiple abstract methods of solverInterface are defined. These methods include the primary solve method. The class pyBvp_SolverInterface must also create and hold an instance of the interface class that is responsible for transferring data to and from BVP_SOLVER. For this example, this class is called f2py_bvp_solver_interface.

Once the interface is created for a BVP code and added to py_bvp, users may access the primary solver function through the primary solver class; an example can be found in section 4.1. This allows for a level of abstraction, to the extent that is allowed by Python, to be placed over the BVP code interface classes.

To further make adding BVP codes easier, as mentioned previously, py_bvp currently comes packaged with two tools that are useful for BVP codes, and we plan to add a number of other tools in future. These tools are meant to help users add additional features to their interfaces without requiring them to write complex numerical routines.

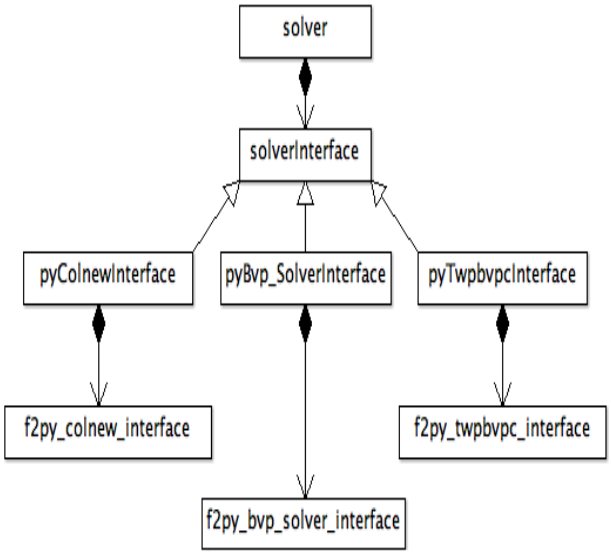

Figure 1. Class structure for py_bvp.

Two of these tools have been used with the TWPBVPC interface. One tool determines numerical partial derivatives by the use of finite differences. Unlike BVP_SOLVER, TWPBVPC does not provide users the option to determine numerical derivatives. Instead, users must provide derivative routines for both the ODEs and boundary conditions. The numerical partial derivatives tool for py_bvp provides users with an alternative. The second tool converts one form of initial guess to another. COLNEW and BVP_SOLVER allow users to provide an initial guess in the form of a continuous function. However, TWPBVPC requires the initial guess to be in the form of a vector of discrete solution values. The interface for TWPBVPC uses the initial-guess tool to convert a guess in the form of a solution function to a vector of solution values. Afterward, the interface passes the guess to the Fortran subroutine. By making use of this tool, the interface for TWPBVPC becomes more consistent with those of the other BVP codes.

We now describe py_bvp from the user perspective.

3.3. The py_bvp User Interface

The user interface to py_bvp is built to be as application specific as possible. That is, users must only provide information about the BVP itself. This is achieved in two ways.

First, py_bvp uses a Python dictionary as the primary means for users to communicate information about a BVP to the solver class of py_bvp. A dictionary is an array that links keys to various values of any type. The keys to Python

dictionaries can be strings. This allows required parameters to be given clear and intuitive labels.

Second, users need only provide information on the BVPs themselves. Many other BVP codes, including the original TWPBVPC, require users to provide information on items that are unrelated to the actual BVP. For example, users must provide workspace arrays and their sizes. In contrast, py_bvp uses the high-level features of Python, such as default parameters and dynamic memory allocation, to perform this task automatically.

Although py_bvp sets default values for many of the parameters unrelated to the BVP itself, e.g., the maximum amount of memory allocated to a BVP, users retain the option of adjusting those parameters, thus retaining the same level of control as directly calling the Fortran BVP code.

The addition of multiple BVP problem classes has the potential to greatly complicate the use of py_bvp. For example, TWPBVPC only allows two-point BVPs, whereas COLNEW allows multiple boundary points. As a result, users are required to provide different information depending on the BVP code they wish to use. Within py_bvp, this translates to different BVP codes having different dictionary entries. For example, both the interface for COLNEW and that for TWPBVPC allow users to specify whether the problem is linear or not. On the other hand, BVP_SOLVER does not.

For clarity and simplicity, py_bvp alerts users when they fail to provide or incorrectly provide a required dictionary entry. For example, if a user fails to provide the number of ODEs when using the TWPBVPC interface, py_bvp returns the following error:

```
py_bvp error --> problem dictionary error -->

missing value --> number of ODEs
```

These error messages are from a set of py_bvp exception types that are children of the main Python exception class. As a consequence, they function in the same way as all other Python exceptions. However, they provide information to the user that is unique to py_bvp. Figure 2 shows the class structure of py_bvp exception types. Users who add BVP code interfaces to py_bvp are encouraged to use these exceptions instead of their own. By doing so, users get the same feedback independent of the BVP code they are using. Moreover, this further adds to the intuitive nature of py_bvp.

4. SOLVING A SIMPLE BVP WITH PY_BVP

We illustrate the ideas of the previous sections by using py_bvp to solve a simple BVP with BVP_SOLVER as the basic solver in section 4.1. We demonstrate how simple it is to then use py_bvp to solve the same BVP with COLNEW and TWPBVPC in section 4.2.

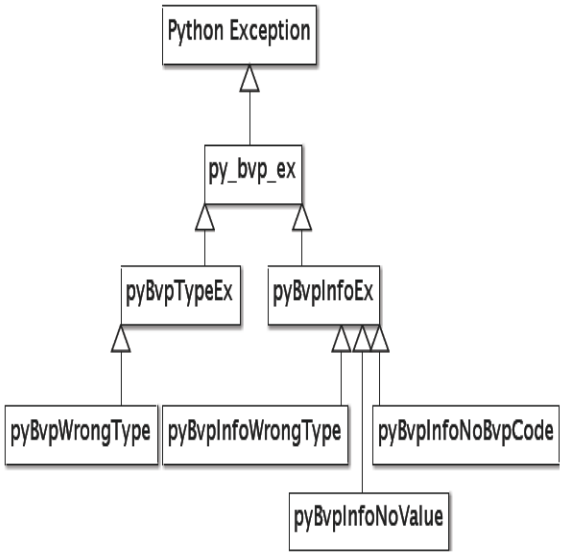

Figure 2. Exception class structure for py_bvp. The exception class pyBvpWrongType alerts the user when a wrong type is sent to a class method. The parent exception class pyBvpInfoEx is responsible for all errors associated with the problem dictionary. The class pyBvpInfoWrongType alerts the user when they enter a wrong type for a dictionary entry. The class pyBvpInfoNoValue is an exception raised when the user fails to enter a required dictionary entry. The exception class pyBvpInfoNoBvpCode indicates that the user has chosen a BVP code that is not supported by py_bvp.

4.1. py_bvp/BVP_SOLVER

We demonstrate the use of py_bvp by solving Bratu's problem; see example 3.5.2 in [24]. The problem is specified by the second-order ODE

$$y''(x) + \lambda \exp(y(x)) = 0, \quad 0 < x < 1, \tag{2a}$$

with boundary conditions

$$y(0) = y(1) = 0. \tag{2b}$$

The problem is solved with $\lambda = 1$ and is converted to the first-order system

$$y_1'(x) = y_2(x), \;\; y_2'(x) = -\lambda \exp(y_1(x)) = 0, \;\; 0 < x < 1, \tag{3a}$$

with boundary conditions

$$y_1(0) = y_1(1) = 0. \tag{3b}$$

The first step is to add py_bvp functionality to the Python code by importing the module:

```
import py_bvp
```

Once py_bvp is imported, the BVP is defined by creating two functions, one that defines the ODEs and one that defines the boundary conditions:

```
def fsub(x,y):
        Lambda = 1.0
        f=numpy.zeros(2)
        f[0] = y[1]
        f[1] = -Lambda*exp(y[0])
        return f

def BC(ya,yb):
        ga = numpy.array([ya[0]])
        gb = numpy.array([yb[0]])
        return ga,gb
```

In the above code, fsub returns a numpy array. py_bvp uses numpy arrays instead of other Python data structures [19] for two reasons. First, numpy allows quick and efficient computations. Second, f2py, the software used to create the interface between Python and the BVP code, easily supports passing numpy arrays between Python and Fortran.

For this problem, an initial guess in the form of a function is used:

```
def guess(x):
        y=numpy.zeros(2)
        y[0] = 0.0
        y[1] = 0.0
        return y
```

The next step involves passing parameters into the solver for py_bvp. As in the previous section, this involves creating a Python dictionary and defining a number of keys with associated values. Again, the use of Python dictionaries makes the meaning of each parameter clear.

```
info={}
info['number of ODEs'] = 2
info['number of left BCs'] = 1
info['ODEs'] = fsub
info['boundary points']=[0.0,1.0]
info['BCs'] = gsub
info['initial guess from function'] = guess
```

To solve the BVP, an instance of the solver class is created:

```
solve = py_bvp.Solver(info)
```

In the above code, the dictionary is passed into the constructor for the solver class. Next, we specify which BVP code we wish to apply to solve the BVP.

```
info['BVP code'] = 'BVP_SOLVER'
```

Finally, the solve method for the solver class is run.

```
solve.solve()
```

Once the problem is solved, the instance of the class holds the solution in a form that can be easily passed to matplotlib; Figure 3 shows the resulting graph for $y(x)$.

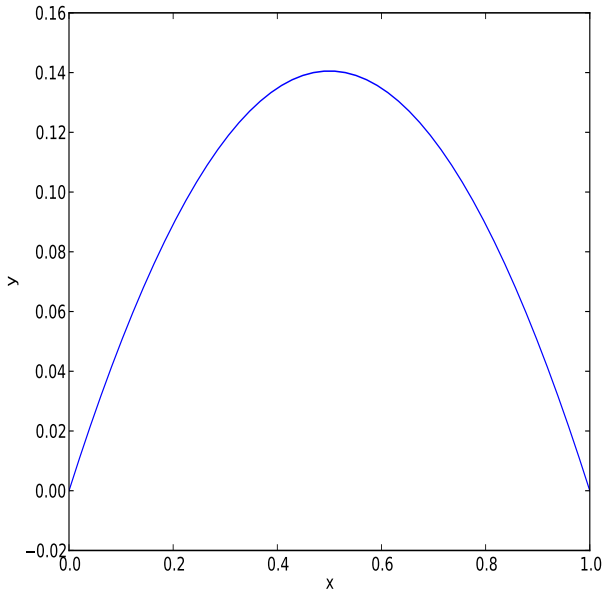

Figure 3. Solution for Bratu's problem (2) from BVP_SOLVER.

4.2. py_bvp/COLNEW and py_bvp/TWPBVPC

As mentioned previously, users are often interested in applying more than one code to solve a given BVP. However, a significant difficulty with this task is a lack of "problem portability" for Fortran BVP codes. Typically, the problem file must be modified substantially in order for multiple codes to be used to solve the same problem. py_bvp has been designed to overcome this difficulty. To illustrate the effectiveness of py_bvp, we solve (2) with COLNEW and TWPBVPC.

In this section we assume that a problem file was originally created to solve the problem with BVP_SOLVER, as was indeed done in the previous subsection. Our goal is to solve the problem with the other two BVP codes with minimal code modification. A significant obstacle is that different codes solve different classes of BVPs, e.g., COLNEW solves multi-point BVPs whereas TWPBVPC solves two-point BVPs. Codes require different forms of the functions for the ODEs, the boundary conditions, and the initial guess. To address this issue, we use first-order ODEs with separated two-point boundary conditions as the universal problem class for py_bvp. To use the problem class for every code, the user must add two lines of code.

```
info['use universal BC'] = True
info['use universal guess'] = True
```

After doing so, the user can use the same ODE function, boundary-condition function, and initial-guess function for

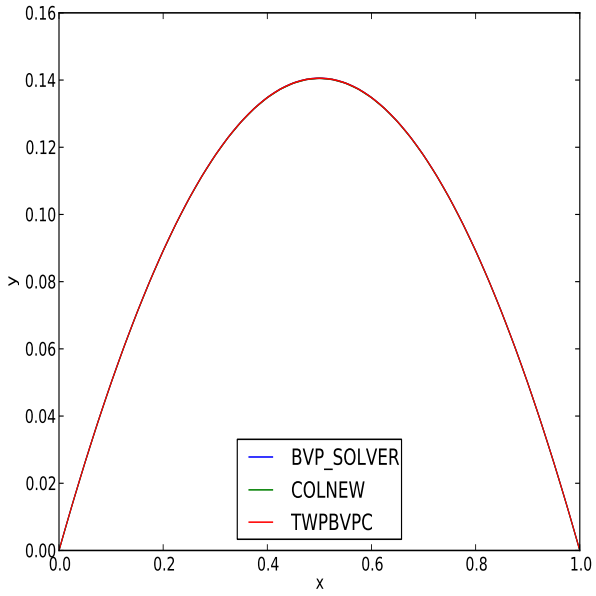

Figure 4. Solution for problem 2 from all three codes.

all BVP codes featured in `py_bvp`. Of course, we still allow users to define problems in the specific form intended for a given BVP code. For example, users can still define multipoint BVPs for `COLNEW`; however this would require users to write a new boundary-condition function.

Once those two lines are added, we simply modify the `'BVP code'` entry in the dictionary and run the solve routine again. The following code must be added to solve the problem with `COLNEW`:

```
info['BVP code'] = 'COLNEW'
solve.solve()
```

Similarly, the following code must be added to solve the problem with `TWPBVPC`:

```
info['BVP code'] = 'TWPBVPC'
solve.solve()
```

The solutions for all three codes can be easily extracted and graphed with `matplotlib`. All three codes obtain numerical solutions that are equivalent to within "graphical accuracy"; Figure 4 shows the resulting graph for $y(x)$ as obtained by the three BVP codes.

5. CONCLUSIONS AND FUTURE WORK

With `py_bvp`, we have created an environment for solving BVPs that facilitates the inclusion of different BVP codes and support tools. This is done by exploiting high-level features of `Python`, with expandability as a key design goal. In order to extend `py_bvp`, users must simply create a child of a mostly abstract class. We plan to add additional BVP codes to `py_bvp`. Users also have access to some built-in tools to assist in the process of using solvers contained in the interface. Two tools discussed in this paper facilitate the provision of numerical approximations for partial derivatives and initial solution guesses in an appropriate form; a number of additional tools are planned for later inclusion in `py_bvp`.

Despite offering convenient access to different BVP codes, the universal interface remains easy to use. This was accomplished by using a `Python` dictionary as the primary method to pass parameters to the `solver` class. Because the dictionary has keys that can be expressed in plain language, the meaning of each parameter is clear to users. As well, all parameters unrelated to the actual BVP itself have been eliminated by the use of high-level `Python` features. If the user forgets to enter a parameter or enters the wrong type of parameter for a given BVP code, `py_bvp` raises an easy-to-read exception. Finally, a universal BVP problem class was created for `py_bvp` so all codes can use the same functions to define the ODEs, the boundary conditions, and the initial guess. This allows users to solve a problem with multiple codes with little extra effort.

As mentioned above, we plan to make multiple additions to `py_bvp`. For example, we expect to include additional BVP codes in `py_bvp`, most notably the `Fortran TWPBVPLC` code [14]. This code combines the error estimation/conditioning estimation-based mesh selection algorithm of `TWPBVPC` with fully implicit Lobatto type Runge–Kutta formulas. We also expect to incorporate additional tools into the universal interface, e.g., a tool for automatic differentiation. This would allow partial derivatives to be evaluated that are accurate to machine precision without the effort or potential human error associated with hand coding. The use of analytical partial derivatives allows BVP codes to determine solutions for certain BVPs faster and in a more robust fashion than with the use of approximations of partial derivatives based on finite differences; see [25] for additional details. Therefore, users stand to gain the benefit of exact derivatives without needing to code them manually.

Acknowledgments The authors wish to thank the referees for a number of helpful comments.

REFERENCES

[1] U. M. Ascher, J. Christiansen, and R. D. Russell. COLSYS–a collocation code for boundary-value problems. In *Proceedings of a Working Conference on Codes for Boundary-Value Problems in Ordinary Differential Equations*, pages 164–185, London, UK, 1979. Springer-Verlag.

[2] U. M. Ascher, R. M. M. Mattheij, and R. D. Russell. *Numerical solution of boundary value problems for ordinary differential equations*, volume 13 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. Corrected reprint of the 1988 original.

[3] G. Bader and U. M. Ascher. A new basis implementation for a mixed order boundary value ODE solver. *SIAM J. Sci. Statist. Comput.*, 8(4):483–500, 1987.

[4] J. R. Cash and F. Mazzia. A new mesh selection algorithm, based on conditioning, for two-point boundary value codes. *J. Comput. Appl. Math.*, 184(2):362–381, 2005.

[5] J. R. Cash and M. H. Wright. A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation. *SIAM J. Sci. Statist. Comput.*, 12(4):971–989, 1991.

[6] cca forum. The Common Component Architecture Forum. `www.cca-forum.org/index.html`.

[7] W. H. Enright and P. H. Muir. Runge–Kutta software with defect control for boundary value ODEs. *SIAM J. Sci. Comput.*, 17(2):479–497, 1996.

[8] IMSL. imsl_f_bvp_finite_difference. `www.vni.com/products/imsl/documentation/CNL700_Docs/Cmath.pdf`.

[9] J. Kierzenka and L. F. Shampine. A BVP solver based on residual control and the MATLAB PSE. *ACM Trans. Math. Software*, 27(3):299–316, 2001.

[10] J. Kierzenka and L. F. Shampine. A BVP solver that controls residual and error. *JNAIAM J. Numer. Anal. Ind. Appl. Math.*, 3(1-2):27–41, 2008.

[11] Mathematica. `www.wolfram.com/products/mathematica/index.html`.

[12] MATLAB. `www.mathworks.com/products/matlab/`.

[13] matplotlib. `matplotlib.sourceforge.net`.

[14] F. Mazzia and D. Trigiante. A hybrid mesh selection strategy based on conditioning for boundary value ODE problems. *Numer. Algorithms*, 36(2):169–187, 2004.

[15] P. Muir, R. Pancer, and K. Jackson. Pmirkdc: a parallel mono-implicit runge-kutta code with defect control for boundary value odes. *Parallel Comput.*, 29:711–741, 2003.

[16] P. H. Muir. Optimal discrete and continuous mono-implicit Runge-Kutta schemes for BVODEs. *Adv. Comput. Math.*, 10(2):135–167, 1999.

[17] NAG. Numerical Algorithms Group: nag_ode_bvp_fd_nonlin_gen(d02rac). `www.nag.co.uk/numeric/CL/nagdoc_cl08/pdf/D02/d02rac.pdf`.

[18] M. R. Osborne. Collocation, difference equations, and stitched function representations. In *Topics in numerical analysis, II (Proc. Roy. Irish Acad. Conf., Univ. College, Dublin, 1974)*, pages 121–132. Academic Press, London, 1975.

[19] P. Peterson. F2PY: Fortran to Python Interface Generator. `cens.ioc.ee/projects/f2py2e/`.

[20] PyIMSL. `www.vni.com/products/imsl/pyimslstudio/overview.php`.

[21] Sage. An open-source mathematics software system. `www.sagemath.org/`.

[22] J. Salvatier. Scikits. `pypi.python.org/pypi/scikits.bvp_solver/0.2.5`.

[23] SciPy. Scientific Tools for Python. `www.scipy.org`.

[24] L. F. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, Cambridge, 2003.

[25] L. F. Shampine, R. Ketzscher, and S. A. Forth. Using AD to solve BVPs in Matlab. *ACM Trans. Math. Software*, 31(1):79–94, 2005.

[26] L. F. Shampine, P. H. Muir, and H. Xu. A user-friendly Fortran BVP solver. *JNAIAM J. Numer. Anal. Ind. Appl. Math.*, 1(2):201–217, 2006.

[27] P. Virtanen. bvp. `www.elisanet.fi/ptvirtan/software/bvp/index.html`.