bacoli_py - A Python Package for the Error Controlled Numerical Solution of 1D Time-Dependent PDEs*

Connor Tannahill†, Paul Muir‡

July 12, 2019

Abstract

Partial Differential Equations (PDEs) are widely used for modelling systems arising in many application domains. When considering a PDE-based model, one must typically make use of numerical methods to compute approximations to the solutions of the PDEs. When solving a problem numerically, the accuracy of the computed solution should, of course, be of substantial concern. Error control algorithms attempt to generate approximate solutions which are accurate to within a user-prescribed error tolerance. In this way, the user can be reasonably confident that the solution which is being returned will be accurate to within the requested tolerance. Additionally, the cost of the computation can be expected to be proportional to the requested tolerance.

In this report, we introduce *bacoli_py*, a Python 3 package for solving 1D time-dependent PDEs with error control. This package wraps modified versions of the Fortran 77 packages, BACOLI [1] and BACOLRI [2] so that these algorithms can be used in a far more user-friendly environment. This report provides a description of the components of this Python package and discusses several examples which demonstrate its usage. Some explanation of the underlying algorithms is also given in order to help in understanding many of the optional arguments which can be provided to the solver to enable more effective computations.

*This work was supported by the Mathematics of Information Technology and Complex Systems Network, the Natural Sciences and Engineering Research Council of Canada and Saint Mary's University.

†Saint Mary's University, Halifax, NS, Canada, B3H 3C3

‡Saint Mary's University, Halifax, NS, Canada, B3H 3C3

1 Introduction

Partial Differential Equations (PDEs) are fundamental tools in mathematical modelling. PDE models are frequently employed to model complex phenomena occurring in areas such as epidemiology [3], image processing [4], etc.. Numerical algorithms must typically be employed to compute numerical approximations to the solution. Error control algorithms compute approximate solutions which have an estimated error which is within a user-specified error tolerance. In this way, the user can be reasonably confident that they have obtained an approximate solution which has a level of accuracy appropriate for the application. In this report, we introduce *bacoli_py*, an open-source Python module which can be used to compute error controlled numerical solutions to one dimensional time-dependent PDEs of the form

$$\boldsymbol{u}_t(t,x) = \boldsymbol{f}(t,x,\boldsymbol{u}(t,x),\boldsymbol{u}_x(t,x),\boldsymbol{u}_{xx}(t,x)), \quad x \in [x_a,x_b], \quad t \in [t_0,t_{out}], \tag{1}$$

where $\boldsymbol{u} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^n$ and $\boldsymbol{f} : \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, with initial conditions

$$\boldsymbol{u}(t,x_0) = \boldsymbol{u}_0(x), \quad x \in [x_a,x_b], \tag{2}$$

where $\boldsymbol{u}_0 : \mathbb{R} \to \mathbb{R}^n$ and separated boundary conditions

$$\boldsymbol{b}_L(t,\boldsymbol{u}(x_a,t),\boldsymbol{u}_x(x_a,t)) = \boldsymbol{0}, \quad \boldsymbol{b}_R(t,\boldsymbol{u}(x_b,t),\boldsymbol{u}_x(x_b,t)) = \boldsymbol{0}, \quad t \in [t_0,t_{out}], \tag{3}$$

where $\boldsymbol{b}_L, \boldsymbol{b}_R : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ and $\boldsymbol{0} \in \mathbb{R}^n$, and where $n \equiv npde$ is the number of PDEs.

This solver wraps the Fortran packages BACOLI [1] and BACOLRI [2], the most recent members of a family of B-spline Gaussian collocation error control solvers for this problem class. These solvers have been shown to efficiently compute numerical solutions to general 1D PDEs of the form (1-3) which are typically accurate to within a user-specified error tolerance [5, 6]. In order to employ these solvers within this Python module, they were modified in two significant ways: *(i)* the Almost Block-Diagonal linear system solver COLROW [7] was replaced with the LAMPAK solver [8] to ensure copyright compliance and *(ii)* the calls to the function that defines (1) were modified such that the cross-language callbacks could be done more efficiently through vectorization using *numpy* [9] arrays. The purpose of this vectorization is to minimize the number of cross-language callbacks used by re-organizing the code such that repeated evaluations of the main user callback routine are instead done in one

larger call to a modified user routine. Alternatively, compiled Fortran subroutines can be provided The Python to Fortran interface required for this package is generated using *f2py* [10].

The advantage of being able to apply these solvers within the Python language primarily comes with the improvements which can be made in the user interface compared to what is necessary when using the low-level Fortran codes directly. Additionally, the user can take advantage of the many high-quality tools which are easily available within the Python ecosystem for analysis and visualization of results. These advantages do come at a cost to performance, with `bacoli_py` being substantially slower than its Fortran equivalents in its standard usage. The Python module is therefore primarily useful for initial prototyping and model exploration. For applications where performance is of concern, the authors recommend the use of the Fortran versions of this software, available for download at `http://cs.smu.ca/~muir/BACOLI-3_Webpage.htm`.

This report is structured as follows; Section 2 gives a description of the underlying BACOLI and BACOLRI solvers, which serves both to explain how `bacoli_py` works internally, as well as to give context for many of the optional arguments which can be provided to the solver. Section 3 gives an overview of the *bacoli_py* module. Section 4 provides some examples of the package applied to solve several test problems.

For more information and complete documentation, see `https://bacoli-py.readthedocs.io/en/latest/`. `bacoli_py` can be downloaded from PyPi at `http://pypi.python.org/pypi/bacoli-py`. Source code is available at `https://github.com/connortannahill/bacoli_py`.

2 Overview of BACOLI & BACOLRI

In this section, we provide an overview of BACOLI and BACOLRI, the solvers wrapped by `bacoli_py`. As mentioned previously, these solvers compute approximate solutions to 1D time-dependent PDEs using an error control algorithm. These solvers return an approximate solution which has an estimated error which is less than a user-provided error tolerance. The purpose of this section is to explain the underlying algorithms employed in these solvers. This knowledge is important for understanding many of the optional arguments of `bacoli_py` and provide some idea of how these can be chosen for more efficient computation. We do not intend for this to be a complete description of these algorithms, and indeed many important details are not included here. For more complete descriptions, see [1, 11] and the references within.

The BACOL family of software (to which BACOLI and BACOLRI belong) represent the approximate solution to a PDE a given point in time, t, as a linear combination of B-spline basis functions [12] of degree p. This representation leads to a time-dependent, C^1 continuous approximate solution in x.

Let $x_a = x_0 < x_1 < \cdots < x_{nint} = x_b$ be a mesh of *nint* subintervals partitioning the spatial domain $[x_a, x_b]$.

Then the approximate solution is represented as

$$\boldsymbol{U}(t,x) = \sum_{i=1}^{NC_p} \boldsymbol{y}_{p,i}(t) B_{p,i}(x), \tag{4}$$

where $\boldsymbol{y}_{p,i}(t)$ is an unknown time-dependant vector coefficient, $B_{p,i}(x)$ is the ith B-spline basis function of degree p, and $NC_p = nint(p-1)+2$. The unknown coefficients in (4) are determined by requiring that the approximate solution exactly satisfy the PDE at certain points in space. These conditions are referred to as the collocation conditions, and the points at which they are imposed are called the collocation points. The number of collocation points, $kcol = p-1, \quad 3 \le kcol \le 10$, used in computation can set through the optional argument *kcol*. This determines the order of accuracy in the spatial discretization of the PDE, as the spatial error in the collocation solution (4) is $\mathcal{O}(h^{p+1})$, where h is the maximum subinterval size in the spatial mesh. Experimentation with *kcol* choices in BACOLI was performed in [5], and while there is no general rule for the choice in *kcol*, we see that, in general, for modest tolerance requests, e.g., 10^{-3}, *kcol* values of 3, 4 or 5 have the best performance, while for sharp tolerance requests, e.g., 10^{-6}, *kcol* values of 6 or 7 are better.

The BACOL family implements Gaussian collocation, which prescribes that (4) satisfy the collocation conditions

$$\frac{d}{dt}\boldsymbol{U}(t,\xi_l) = \boldsymbol{f}(t,\xi_l,\boldsymbol{U}(t,\xi_l),\boldsymbol{U}_x(t,\xi_l),\boldsymbol{U}_{xx}(t,\xi_l)), \tag{5}$$

$l = 2, ..., NC_p - 1$, where the collocation points ξ_l are

$$\xi_l = x_{i-1} + h_i \rho_j,$$
$$l = 1 + (i-1)(p-1) + j, \quad i = 1, \ldots, nint, \quad j = 1, \ldots, p-1,$$

and $\{\rho_i\}_{i=1}^{p-1}$ are the images of the order $p-1$ Gauss points on $[0,1]$ and, $h_i = x_i - x_{i-1}$. The additional points $\xi_a = x_a$, $\xi_b = x_b$ are where the approximate solution, $\boldsymbol{U}(t,x)$, is required to satisfy the BC's,

$$\boldsymbol{b}_L(t,\boldsymbol{U}(t,x_a),\boldsymbol{U}_x(t,x_a)) = \boldsymbol{0}, \quad \boldsymbol{b}_R(t,\boldsymbol{U}(t,x_b),\boldsymbol{U}_x(t,x_b)) = \boldsymbol{0}. \tag{6}$$

Note that (5) is a system of time-dependent ordinary differential equations, which when coupled with (6), forms

a system of Differential Algebraic Equations (DAEs) which can be solved to obtain the B-spline coefficients in (4) using standard error control solvers for DAEs.

It is at this point that BACOLI and BACOLRI algorithms diverge; BACOLI uses DASSL [13] for solving the DAE system (5-6), whereas BACOLRI uses RADAU5 [14]. These two solvers implement different classes of time integration formulas. DASSL makes use of a family of multi-step methods called Backwards Differentiation Formulas (BDFs). RADAU5 is based on an fifth order Implicit Runge Kutta (IRK) method of Radau IIA type. As mentioned in Chapter 3, BACOLRI out-performs BACOLI for certain classes of problems. This is due to instabilities seen in higher-order BDF methods such as some of those implemented in DASSL. In particular, BACOLI performs poorly for problems for which the DAE system (5-6) has a Jacobian with eigenvalues near the imaginary axis [11]. Restricting the maximum order of the BDF formulas used in DASSL by setting the optional arguments `maxord` to 2 or less has been seen to allow BACOLI to converge to a solution but at substantial cost to its performance. However, it has been seen that even with this restriction BACOLI struggles to reach its requested tolerance for these problems [15].

After the collocation solution has been obtained through the solution to (5-6), a global, spatial, error estimate is computed in order to determine whether the solution at the current step should be accepted. A scaled global error estimate is computed for each solution component, $U_s(t,x)$, and is given by

$$E_s(t) = \sqrt{\int_{x_a}^{x_b} \left(\frac{U_s(t,x) - \bar{U}_s(t,x)}{ATOL_s + RTOL_s|U_s(t,x)|} \right)^2 dx}, \quad s = 1, ..., npde, \tag{7}$$

where $\bar{\boldsymbol{U}}(t,x) = (\bar{U}_1(t,x), \ldots, \bar{U}_{npde}(t,x))^T$ is an approximate solution to the PDE at this point in time which has spatial error of a different order of accuracy than $\boldsymbol{U}(t,x)$ (more on this later). Additionally, a second set of error estimates is computed which are local to each subinterval in order to determine the distribution of the error in the spatial domain,

$$\hat{E}_i(t) = \sqrt{\sum_{s=1}^{npde} \int_{x_{i-1}}^{x_i} \left(\frac{U_s(t,x) - \bar{U}_s(t,x)}{ATOL_s + RTOL_s|U_s(t,x)|} \right)^2 dx}, \quad i = 1, \ldots, nint. \tag{8}$$

The solution at the current time step is accepted if the scaled error estimate (10) meets the tolerance for each solution component, i.e., if

$$\max_{1 \leq s \leq npde} E_s(t) \leq 1. \tag{9}$$

If condition (9) is met then the step is accepted and the solver will repeat this process at the next time step in order to advance the solution forward in time. Otherwise, the step is rejected and a spatial remeshing algorithm is applied which attempts to compute a new mesh such that the computed solution obtained on this new mesh will have a spatial error that satisfies the tolerances. This remeshing algorithm also attempts to keep *nint* as small as possible, as the cost of the algorithm scales primarily based on this arguments. This remeshing algorithm works by *(i)* adjusting the number of mesh subintervals based on the magnitude of the spatial error estimates (7) and *(ii)* using equidistribution to re-position the mesh points in regions where the error estimates are largest, as indicated by the per-subinterval estimates (8). Note also that spatial remeshing may reduce *nint* when the spatial error estimate is considerably beneath the tolerance. This is vital for efficient computation.

The error estimates (7) and (8) both depend on additional approximation to the solution of the PDE, $\bar{\boldsymbol{U}}_s(t,x)$. One of the primary innovations of BACOLI and BACOLRI is the use of inexpensive, interpolation-based error estimates. This error estimation can be accomplished in two ways based on how $\bar{\boldsymbol{U}}(t,x)$ is computed; these two methods are referred to as the SuperConvergent Interpolation (SCI) scheme and the Lower Order Interpolation (LOI) scheme. The choice of what scheme to use is provided by the user through the optional `s_est` arguments, which can take the values `'sci'` for the SCI scheme or `'loi'` for the LOI scheme.

The SCI scheme takes advantage of a known property of a solution approximation computed using Gaussian collocation. In particular, for each subinterval, the SCI interpolates known points of supercovergence, i.e., points where the error of solution and/or derivative values of the collocation solution have higher orders of accuracy than the global collocation solution. In this case, $\bar{\boldsymbol{U}}(t,x)$ is a C^1 continuous piecewise polynomial composed of Hermite-Birkhoff interpolating polynomials in each subinterval which interpolate a certain set of superconvergent values associated with the subinterval. An issue when applying the SCI scheme is that the error of $\bar{\boldsymbol{U}}(t,x)$, and hence the accuracy of the error estimates depends on the ratios of the mesh subinterval sizes, which can become large in some cases. While in most observed cases this does not cause any issues, it motivated the development of an alternative form of error estimation which did not suffer from this issue, namely the LOI scheme.

The LOI scheme implements an alternative form of error estimation known as Local Extrapolation (LE) error estimation. In standard error estimation, after an approximate solution has been computed, another solution of a higher order of accuracy is generated to estimate the error of the computed solution, as in the SCI scheme. In LE error control, rather than computing a higher order approximation to estimate the error in the computed solution,

a solution of one order less is computed. The error in this lower order approximation is then estimated using the computed solution. While this clearly does not estimate the error in the computed solution, *LE error control provides a conservative upper bound for the error in the computed solution.* In the LOI scheme, this lower order approximation comes in the form of a Hermite-Birkhoff interpolate on each subinterval which has been constructed such that its interpolation error is, asymptotically, equivalent to the leading order error term in a collocation solution of one order of accuracy less than the collocation solution. See [1] for further details.

3 Description of bacoli_py

3.1 Basic Usage

bacoli_py provides a convenient, minimal, object-oriented programming interface. Standard usage of `bacoli_py` to solve a PDE consists of just a few steps. The user must first define the system of `npde` PDEs they wish to solve in terms of the Python callback functions `f`, `bndxa`, `bndxa`, `uinit`, which correspond to the PDE (1), left boundary condition and right boundary condition (3), and initial conditions (2), respectively. These are encapsulated within a `ProblemDefinition` object as

```
problem_definition = bacoli_py.ProblemDefinition(npde, f, bndxa, bndxb, uinit).
```

A `Solver` object which performs the main functionality of `bacoli_py` is then initialized. This can be done simply by

```
solver = bacoli_py.Solver().
```

The `Solver` object contains the method `solve`, which is used to solve the PDE defined by a `ProblemDefinition` object. The arguments to this method include a `ProblemDefinition object`, the initial point in time t_0, the spatial boundaries x_a, x_b, and the points in time and space at which the solution values are requested. A call to `solve` takes the form

```
evaluation = solver.solve(problem_definition, initial_time, [xa, xb], tspan, xspan).
```

This call returns an `Evaluation` object containing the computed solution information. In particular, the approximation solution is contained, as an attribute in this object, as a *numpy* array with dimensions `(npde, len(tspan), len(xspan))`,

```
u = evaluation.u
```

This summarizes the process of using `bacoli_py` in the majority of use cases. What follows in the next subsection is a more complete description of this module, including its overall structure and descriptions of the many arguments and settings which can be specified for more efficient computations.

3.2 Detailed Description

The `bacoli_py` module consists of the three main classes described previously, namely the `ProblemDefinition`, `Solver`, and `Evaluation` classes. The contents of the module is given in Figure 1.

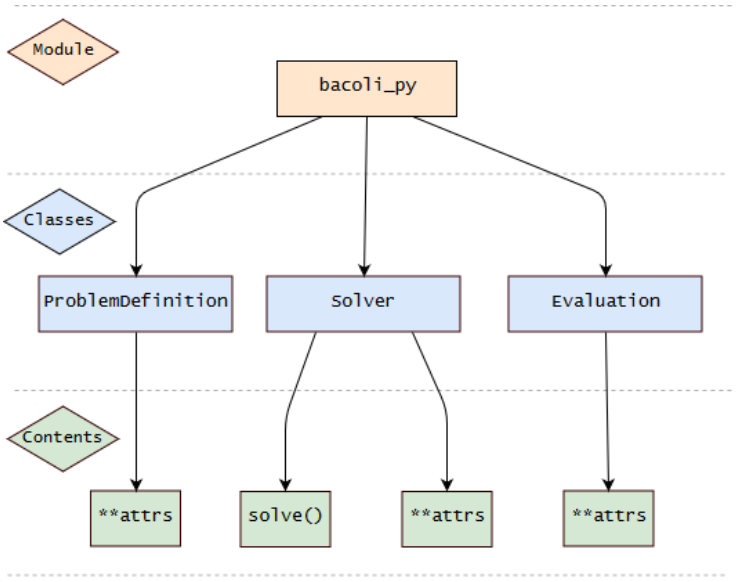

Figure 1: Contents of the `bacoli_py` module. In the diagram, `**attrs` refers to the attributes of the object, which are simply the arguments in the signatures of their constructors.

To define the PDE to be solved, the user provides callback functions to define (1-3), which are collected in a `ProblemDefinition` object, the full signature of which is

```
ProblemDefinition(npde, f, bndxa, bndxb, uinit, derivf = None, difbxa = None, difbxb =
    None).
```

`Solver` is the primary class, with methods related to solving a given PDE. The full signature of this class is

```
Solver(nint_max = 500, kcol = 5, t_int = 'b', s_est = 'loi', maxord = None, ini_ss =
    None).
```

Each of these objects must be initialized before a computation can be done. Definitions of the input arguments are given in Listings 1 and 2.

`nint_max (optional, default = 500)`:
- The maximum number of subintervals allowed for the spatial mesh during computation.

`kcol (optional, default = 4)`:
- The number of collocation points per mesh subinterval. Then the degree in space of the numerical solution is `kcol+1` and the order of accuracy of the spatial discretization scheme is `kcol` +2. $3 \leq$ `kcol` ≤ 10.

`t_int (optional, default = 'b')`:
- Determines which of the two adaptive error control time integration algorithms to be used. Choosing `t_int = 'b'` means that time integration will be based on an adaptive order BDF method. A choice of `t_int = 'r'` corresponds to the use of an algorithm based on a 5th order implicit Runge-Kutta method.

`s_est (optional, default = 'loi')`:
- Determines which of the two spatial error estimation schemes will be used. The choices are the Lower Order Interpolant (LOI) local extrapolation error estimation scheme (`s_est = 'loi'`) or the SuperConvergent Interpolant (SCI) standard error estimation scheme (`s_est = 'sci'`).

`maxord (optional, default = 5)`:
- The maximum order of the BDF method to be used in time integration. Only used if `t_int = 'b'`. $1 \leq$ `maxord` ≤ 5

`ini_ss (optional, default = None)`:
- Initial step size for time integration. If not provided this will be chosen automatically.

Listing 1: Arguments for `Solver`.

`npde`:
- The number of PDE's in the system to be solved.

`f`:
- System of `npde` PDE's to be solved.

`bndxa`:
- Left boundary conditions for the system of PDEs.

`bndxa`:
- Right boundary conditions for the system of PDEs.

`uinit`:
- Initial conditions for the system of PDEs.

`derivf (optional, default = None)`:
- Partial derivatives of the PDE system.

`difbxa (optional, default = None)`:
- Partial derivatives of the left boundary conditions.

`difbxb (optional, default = None)`:
- Partial derivatives of the right boundary conditions.

Listing 2: Arguments for `ProblemDefinition`.

The numerical solution computed when using `bacoli_py` is encapsulated in a `Evaluation` object, which has the signature

```
Evaluation(tspan, xspan, u, ux = None),
```

where `u` is a `numpy` array with `u.shape = (npde, len(tspan), len(xspan))` containing the solution evaluations. If the first spatial derivative values of the numerical solution are required, `ux` is an array of the same shape as `u` containing the derivative

evaluations at these points. The user accesses the solution information by simply using the attributes of the `Evaluation` object, which are the arguments to its constructor.

The user can solve the PDE defined in `ProblemDefinition` using the `Solver.solve` method.

```
Solver.solve(problem_definition, initial_time, initial_mesh, tspan, xspan, atol = 1e-4,
    rtol = 1e-4, dirichlet = False, tstop = None, vec = True, deriv = False).
```

A call to this method returns the solution at the requested `tspan` and `xspan` values, encapsulated in a `Evaluation` object. The input arguments are described in Listing 3.

`problem_definition`:
- `ProblemDefinition` object for the PDE to be solved.

`initial_time`:
- Initial point in the temporal domain.

`initial_mesh`:
- The initial spatial mesh. Optionally, an array containing only $[x_a, x_b]$ (in that order) can be given. In this case, a mesh is automatically generated based on the initial conditions.

`tspan`:
- Vector or scalar containing times at which the solution will be output.

`xspan`:
- Vector or scalar containing points at which the solution will be output.

`atol (optional, default = 1e-4)`:
- Absolute error tolerance. Can be either a scalar or a numpy array of size `npde`.

`rtol (optional, default = 1e-4)`:
- Relative error tolerance. Can be either a scalar or a numpy array of size `npde`.

`dirichlet (optional, default = False)`:
- Should be set to `True` whenever all of the boundary conditions are Dirichlet (more on this later.) Otherwise set to `False`.

`tstop (optional, default = None)`:
- Indicates the absolute end of the temporal domain. Used by the underlying time integrator. If not set, the time integrator may step past the point in time at which a solution is requested and use interpolation to evaluate the solution at the requested point. Only used if `t_int = 'b'`.

`vec (optional, default = True)`:
- Boolean value indicating whether vectorization should be used to increase efficiency. Should only be set to False when compiled Fortran subroutines are being provided for the callback functions in the `ProblemDefinition` object (as in Section 4.4). Otherwise, `vec` should be set to `True`.

`deriv (optional, default = False)`:
- Indicates whether solve should compute the first spatial partial derivative $\underline{u}_x(t, x)$ at each of the requested output points. If set to `True` then the `Evaluation` object returned will have the attribute `Evaluation.ux`, a numpy array of the same dimension as `Evaluation.u`.

Listing 3: Arguments for `Solver.solve`.

4 Examples

In this section, we provide some examples of *bacoli_py* applied to several PDEs while demonstrating the different modes of operation available when using this package.

4.1 One Layer Burgers Equation

The first example is the One Layer Burgers Equation, given by the single PDE,

$$u_t(t,x) = \epsilon u_{xx}(t,x) - u(t,x)u_x(t,x), \quad \epsilon \in \mathbb{R}, \tag{10}$$

with the initial conditions

$$u(t,x_0) = \frac{1}{2} - \frac{1}{2}\tanh\left(\frac{x-\frac{1}{4}}{4\epsilon}\right), \quad x \in [0,1],$$

and the Dirichlet boundary conditions

$$u(t,x_a) = \frac{1}{2} - \frac{1}{2}\tanh\left(\frac{-\frac{1}{2}t-\frac{1}{4}}{4\epsilon}\right), \quad u(x_b,t) = \frac{1}{2} - \frac{1}{2}\tanh\left(\frac{\frac{3}{4}-\frac{1}{2}t}{4\epsilon}\right), \quad t \in [0,1].$$

In this case the parameter ϵ is chosen to be 10^{-3}. We look to solve this problem to obtain a numerical solution which is accurate to within the error tolerance 10^{-6}. We first describe this system in terms of Python callback functions, first globally defining *npde*, the number of PDEs to be solved, and the problem-dependent parameter ϵ. These functions are then placed within a `ProblemDefinition` object.

```
import bacoli_py
import numpy
from numpy import tanh, array

# Specify the number of PDE's in this system.
npde = 1

# Initialize problem-dependent parameters.
eps = 1.0e-3

# Function defining the PDE system.
def f(t, x, u, ux, uxx, fval):
    fval[0] = eps*uxx[0] - u[0]*ux[0]

    return fval

# Function defining the left spatial boundary condition.
def bndxa(t, u, ux, bval):
```

```
    bval[0] = u[0] - 0.5 + 0.5*tanh( (-0.5*t-0.25) / (4.0*eps) )

    return bval

# Function defining the right spatial boundary condition.
def bndxb(t, u, ux, bval):
    bval[0] = 0.5*tanh((0.75-0.5*t)/(4.0*eps)) - 0.5 + u[0]

    return bval

# Function defining the initial condition.
def uinit(x, u):
    u[0] = 0.5 - 0.5 * tanh((x - 0.25) / (4.0*eps))

    return u

# Pack all of these callbacks and the number of PDE's into a
# ProblemDefinition object.
problem_definition = bacoli_py.ProblemDefinition(npde, f=f,
                                                 bndxa=bndxa,
                                                 bndxb=bndxb,
                                                 uinit=uinit)
```

Once the `ProblemDefinition` is created, using *bacoli_py* to solve the PDE is fairly straightforward, requiring only

- The creation of a `Solver` object
- Specification of the initial time t_0
- An array $[x_a, x_b]$ containing the spatial boundary points.
- The points at which the solution will be evaluated. This is done by providing a list of x points, `xspan`, and t points, `tspan`, at which the evaluations of the numerical solution will be provided.
- The absolute (`atol`) and relative (`rtol`) error tolerances, i.e., how accurate we would like the solution to the PDE to be. Here we set `atol` = `rtol` = 10^{-6}.

```
# Initialize the Solver object.
solver = bacoli_py.Solver()

# Set t0.
initial_time = 0.0

# Define the spatial boundaries.
initial_mesh = numpy.array([0.0, 1.0])

# Choose output times and points. Here our final time t_end = 1.
tspan = numpy.linspace(0.001, 1, 100)
xspan = numpy.linspace(0, 1, 100)

# Solve this problem.
evaluation = solver.solve(problem_definition, initial_time, initial_mesh,
                          tspan, xspan, atol=1e-6, rtol=1e-6,
                          dirichlet=True)
```

The solution obtained from this call is plotted in Figure 2

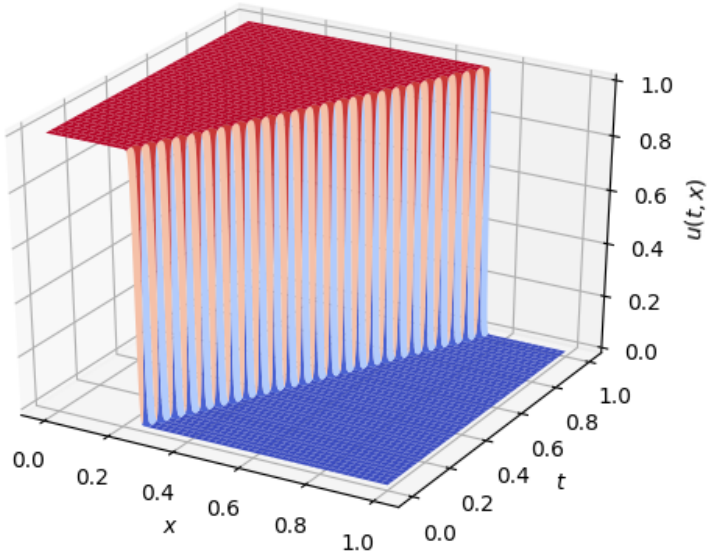

Figure 2: Numerical solution to One Layer Burgers Equation $\epsilon = 10^{-3}$, computed with `atol` = `rtol` = 10^{-6}.

4.2 Catalytic Surface Reaction Model

For our second example, we consider a Catalytic Surface Reaction Model (CSRM), given by

$$
\begin{aligned}
(u_1)_t &= -(u_1)_x + n(D_1 u_3 - A_1 u_1 \gamma) + (u_1)_{xx}/Pe_1, \\
(u_2)_t &= -(u_2)_x + n(D_2 u_4 - A_2 u_2 \gamma) + (u_2)_{xx}/Pe_1, \\
(u_3)_t &= A_1 u_1 \gamma - D_1 u_3 - R u_3 u_4 \gamma^2 + (u_3)_{xx}/Pe_2, \\
(u_4)_t &= A_2 u_2 \gamma - D_2 u_4 - R u_3 u_4 \gamma^2 + (u_4)_{xx}/Pe_2,
\end{aligned} \tag{11}
$$

where $\gamma = 1 - u_3 - u_4$ and $n, r, Pe_1, Pe_2, D_1, D_2, R, A_1$, and A_2 are problem-dependent parameters. The initial conditions at $t = 0$ are

$$
u_1(0, x) = 2 - r, \quad u_2(0, x) = r, \quad u_3(0, x) = u_4(0, x) = 0, \quad x \in [0, 1],
$$

and the mixed boundary conditions are

$$(u_1)_x(t,0) = -Pe_2(2 - r - u_1(t,0)), \quad (u_2)_x(t,0) = -Pe_1(r - u_2(t,0)),$$

$$(u_3)(t,0) = (u_4)(t,0) = 0,$$

$$(u_1)_x(t,1) = (u_2)_x(t,1) = (u_3)_x(t,1) = (u_4)_x(t,1) = 0, \quad t \in [0,18].$$

The values of the problem-dependent parameters are $Pe_1 = Pe_2 = 10000, D_1 = 1.5, D_2 = 1.2, R = 1000, r = 0.96, n = 1$, and $A_1 = A_2 = 30$.

This system was written in terms of the appropriate Python callback functions and used to construct a `ProblemDefinition` object as follows

```
npde = 4

# Initialize problem-dependent parameters.
a1 = 30.0
a2 = 30.0
d1 = 1.50
d2 = 1.20
r = 1000.0
c = .96
n = 1.0
pe1 = 1.0 * 10**4
pe2 = 1.0 * 10**4

# Set t0.
initial_time = 0.0

# Function defining the PDE system.
def f(t, x, u, ux, uxx, fval):
    fval[0] = -ux[0]+n*(d1*u[2]-a1*u[0]*(1.0-u[2]-u[3])) + (1.0/pe1)*uxx[0]
    fval[1] = -ux[1]+n*(d2*u[3]-a2*u[1]*(1.0-u[2]-u[3]))+(1.0/pe1)*uxx[1]
    fval[2] = a1*u[0]*(1.0-u[2]-u[3])-d1*u[2]-r*u[2]*u[3]*(1.0-u[2]-u[3])**2 \
              +(1.0/pe2)*uxx[2]
    fval[3] = a2*u[1]*(1.0-u[2]-u[3])-d2*u[3]-r*u[2]*u[3]*(1.0-u[2]-u[3])**2 \
              +(1.0/pe2)*uxx[3]
    return fval

# Function defining the left spatial boundary condition.
def bndxa(t, u, ux, bval):
    bval[0] = ux[0]+pe1*(2.0-c-u[0])
    bval[1] = ux[1]+pe1*(c-u[1])
    bval[2] = ux[2]
    bval[3] = ux[3]
    return bval

# Function defining the right spatial boundary condition.
def bndxb(t, u, ux, bval):
    bval[0] = ux[0]
    bval[1] = ux[1]
    bval[2] = ux[2]
    bval[3] = ux[3]
    return bval
```

```
# Function defining the initial conditions.
def uinit(x, u):
    u[0] = 2.0-c
    u[1] = c
    u[2] = 0.0
    u[3] = 0.0
    return u

# Instantiate problem definition object.
problem_definition = bacoli_py.ProblemDefinition(npde, f=f,
                                                 bndxa=bndxa,
                                                 bndxb=bndxb,
                                                 uinit=uinit)
```

The CSRM system can then be solved; here we use absolute and relative error tolerances of 10^{-6}.

```
solver = bacoli_py.Solver()

# Initial time and uniform mesh.
initial_time = 0
initial_mesh = numpy.linspace(0, 1, 101)

# Output points.
tspan = numpy.linspace(0.001, 18, 1001)
xspan = numpy.linspace(0, 1, 1001)

# Set a high level of error control.
atol = 1.0e-6
rtol = atol

evaluation = solver.solve(problem_definition, initial_time, initial_mesh,
                          tspan, xspan, atol, rtol)
```

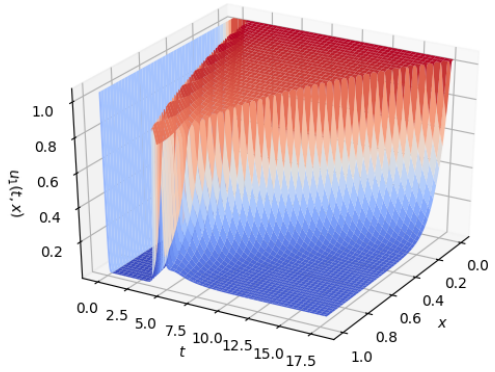

Figure 3: CSRM - $u_1(t, x)$

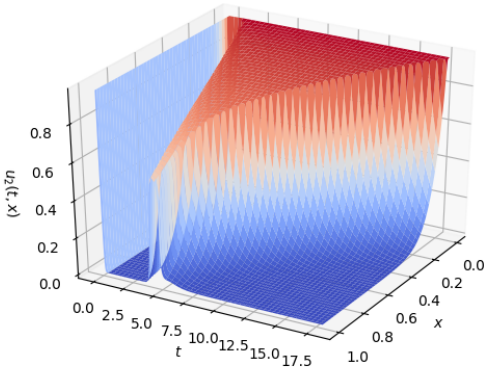

Figure 4: CSRM - $u_2(t, x)$

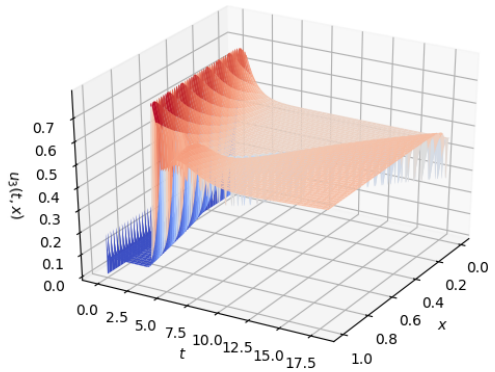

Figure 5: CSRM - $u_3(t, x)$

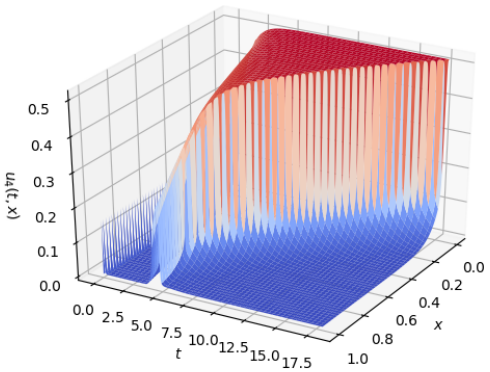

Figure 6: CSRM - $u_4(t, x)$

The solution components are plotted in Figures 3-6.

4.3 Nonlinear Schrödinger System

Up to this point we have given examples using `t_int = 'b'`, which invokes the DAE solver that uses BDF time integration. In the `bacoli_py` context this is generally the better choice. (In the Fortran context, the performance of the two codes are comparable.) This is due to how the Python code interfaces with the underlying solvers. In BACOLI a family of BDF time integration methods is used which performs fewer evaluations of the user-provided callback functions in comparison to BACOLRI, which uses a high-order Runge-Kutta method for time integration. The cost of Fortran-to-Python invocations of these callback functions was the primary performance bottleneck when implementing `bacoli_py`. While vectorization of the

primary user callback alleviated this issue to some extent, the Runge-Kutta method used in BACOLRI uses comparatively more callback calls, which puts it at a disadvantage in this context.

However, there are examples of problems which BACOLI will, without some intervention within the underlying numerical algorithm, fail to converge to a solution. Even after the intervention, the code does not obtain an error controlled solution. This issue is related to stability issues associated with the BDFs [15]. BACOLRI was developed as a solution to this issue, as its underlying numerical algorithms are well-suited for solving the problems for which BACOLI fails.

In this section we give an example which applies `bacoli_py` to one of these problems by employing the switch which prescribes the use of a Runge-Kutta time integration method (by setting `t_int = 'r'`).

We consider the nonlinear Schrödinger system, given by

$$
\begin{aligned}
(u_1)_t &= i\left(\frac{1}{2}(u_1)_{xx} + \eta(u_1)_x + (|u_1|^2 + \rho|u_2|^2)u_1\right), \\
(u_2)_t &= i\left(\frac{1}{2}(u_2)_{xx} - \eta(u_2)_x + (\rho|u_1|^2 + |u_2|^2)u_2\right),
\end{aligned} \tag{12}
$$

where $i = \sqrt{-1}$. The Neumann boundary conditions are given by

$$
(u_1)_x(t,a) = (u_2)_x(t,a) = 0, \quad (u_1)_x(t,b) = (u_2)_x(t,b) = 0,
$$

where $a \to -\infty, b \to \infty$ and the initial conditions are,

$$
\begin{aligned}
u_1(0,x) &= \sqrt{\frac{2\kappa}{1+\rho}}\,\mathrm{sech}\left(\sqrt{2\kappa}x\right)e^{i\left(\left(\phi-\eta\right)x\right)}, \\
u_2(0,x) &= \sqrt{\frac{2\kappa}{1+\rho}}\,\mathrm{sech}\left(\sqrt{2\kappa}x\right)e^{i\left(\left(\phi+\eta\right)x\right)}, \quad t \in [0,1],
\end{aligned}
$$

where $\phi = 1, \eta = 0.5, \rho = 2/3$ and $\kappa = 1$.

As the solver cannot solve complex-valued PDEs, this system is separated into real and complex parts, which can be combined after the computation has been performed to obtain the solution to this problem. Additionally, we choose $x_a = -30$ and $x_b = 90$, as the code can not treat boundary conditions imposed at $\pm\infty$. The source code implementing this problem is given below.

```
# The number of PDEs in this system.
npde = 4

# Initialize problem-dependent parameters.
tempt1 = numpy.sqrt(6.0/5.0)
tempt2 = numpy.sqrt(2.0)
```

```
# Function defining the PDE system.
def f(t, x, u, ux, uxx, fval):

    fval[0] = -0.5*ux[0] - 0.5*uxx[1] - u[1]      \
        * ((u[0] * u[0] + u[1] * u[1]) + 2.0/3.0  \
        * ((u[2] * u[2] + u[3] * u[3])))

    fval[1] = - 0.5 * ux[1] + 0.5 * uxx[0] + u[0] \
        * ((u[0] * u[0] + u[1] * u[1]) + 2.0/3.0  \
        * ((u[2] * u[2] + u[3] * u[3])))

    fval[2] = 0.5 * ux[2] - 0.5 * uxx[3] - u[3]   \
        * ((u[2] * u[2] + u[3] * u[3]) + 2.0/3.0  \
        * ((u[0] * u[0] + u[1] * u[1])))

    fval[3] = 0.5 * ux[3] + 0.5 * uxx[2] + u[2]   \
        * ((u[2] * u[2] + u[3] * u[3]) + 2.0/3.0  \
        * ((u[0] * u[0] + u[1] * u[1])))

    return fval

# Function defining the left spatial boundary condition.
def bndxa(t, u, ux, bval):
    bval[0] = ux[0]
    bval[1] = ux[1]
    bval[2] = ux[2]
    bval[3] = ux[3]
    return bval

# Function defining the right spatial boundary condition.
def bndxb(t, u, ux, bval):
    bval[0] = ux[0]
    bval[1] = ux[1]
    bval[2] = ux[2]
    bval[3] = ux[3]
    return bval

# Function defining the initial conditions.
def uinit(x, u):
    u[0] = tempt1/numpy.cosh(tempt2*x)*numpy.cos(0.5*x)
    u[1] = tempt1/numpy.cosh(tempt2*x)*numpy.sin(0.5*x)
    u[2] = tempt1/numpy.cosh(tempt2*x)*numpy.cos(1.5*x)
    u[3] = tempt1/numpy.cosh(tempt2*x)*numpy.sin(1.5*x)

    return u

# Instantiate problem definition object.
problem_definition = bacoli_py.ProblemDefinition(npde, f=f,
                                                 bndxa=bndxa,
                                                 bndxb=bndxb,
                                                 uinit=uinit)

# Create Solver object. Here use the Runge-Kutta method for time
# integration and allow a large number of spatial subintervals to be
# used.
solver = bacoli_py.Solver(t_int='r', nint_max=2000)

# Set t_0.
```

```
initial_time = 0

# Initial spatial mesh.
initial_mesh = numpy.linspace(-30, 90, 101)

# Output points
tspan = numpy.linspace(0.001, 50, 1001)
xspan = numpy.linspace(-30, 90, 1001)

# Set a high level of error control.
atol = 1.0e-6
rtol = atol

# Solve the nonlienar Schrodinger system.
evaluation = solver.solve(problem_definition, initial_time, initial_mesh,
                          tspan, xspan, atol, rtol)
```

The solution components to the nonlinear Schrödinger system are plotted below in Figures 7-10.

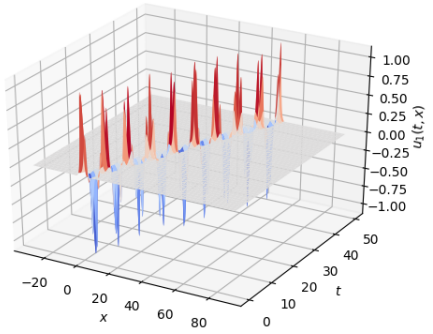

Figure 7: Schrödinger System - $u_1(t,x)$

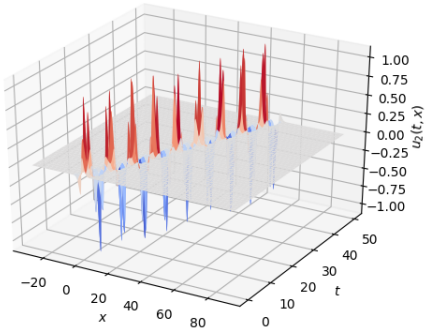

Figure 8: Schrödinger System - $u_2(t,x)$

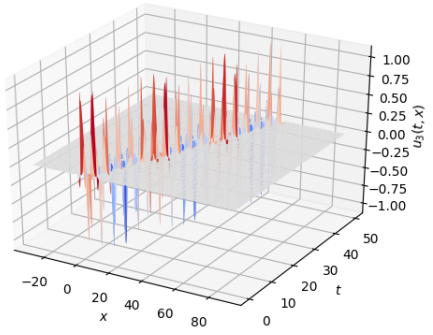

Figure 9: Schrödinger System - $u_3(t,x)$

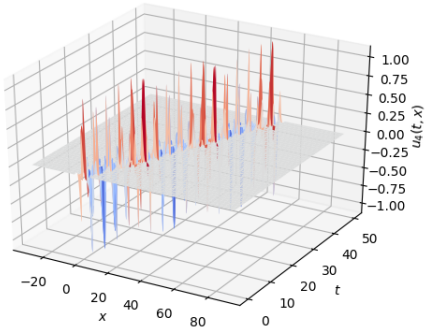

Figure 10: Schrödinger System - $u_4(t,x)$

4.4 Two Layer Burgers Equation

To increase the performance of `bacoli_py`, it is possible to define the callback functions in a `ProblemDefinition` as compiled Fortran subroutines. In this example, we demonstrate how this can be done in the context of solving the Two Layer Burgers Equation (TLBE). This PDE is given by

$$u_t(t,x) = \epsilon u_{xx}(t,x) - u(t,x)u_x(t,x), \tag{13}$$

with initial and (Dirichlet) boundary conditions chosen such that (13) has the exact solution

$$u(t,x) = \frac{0.1e^{-A} + 0.5e^{-B} + e^{-C}}{e^{-A} + e^{-B} + e^{-C}}, \quad t \in [0,1], \quad x \in [0,1],$$

where,

$$A = \frac{0.05}{\epsilon}(x - 0.5 + 4.95t), \quad B = \frac{0.25}{\epsilon}(x - 0.5 + 0.75t), \quad C = \frac{0.5}{\epsilon}(x - 0.375), \quad \epsilon \in \mathbb{R}.$$

We first define Fortran 95 callback routines for each of the callback functions representing the TLBE, its initial conditions, and its boundary conditions. The `numpy.f2py` module is used to build an extension module containing these Fortran subroutines, callable from Python.

```
import numpy.f2py as f2py

# String defining the Fortran 95 callback subroutines.
prob_def_f = """
      subroutine f(t, x, u, ux, uxx, fval)
          integer           npde
          parameter        (npde=1)
          double precision t, x, u(npde), ux(npde)
          double precision uxx(npde), fval(npde)

          double precision eps
          parameter        (eps=1d-4)

          fval(1) = eps*uxx(1) - u(1)*ux(1)
      return
      end

      subroutine bndxa(t, u, ux, bval)
          integer           npde
          parameter        (npde=1)
          double precision t, u(npde), ux(npde), bval(npde)
          double precision eps
          parameter        (eps=1d-4)
          double precision a1, a2, a3, expa1, expa2, expa3, temp
```

```
      a1 = (0.5d0 - 4.95d0 * t) * 0.5d-1 / eps
      a2 = (0.5d0 - 0.75d0 * t) * 0.25d0 / eps
      a3 = 0.1875d0 / eps
      expa1 = 0.d0
      expa2 = 0.d0
      expa3 = 0.d0
      temp = max(a1, a2, a3)
      if ((a1-temp) .ge. -35.d0) expa1 = exp(a1-temp)
      if ((a2-temp) .ge. -35.d0) expa2 = exp(a2-temp)
      if ((a3-temp) .ge. -35.d0) expa3 = exp(a3-temp)

      bval(1) = u(1) - (0.1d0*expa1+0.5d0*expa2+expa3)  &
                / (expa1+expa2+expa3)
      return
      end

      subroutine bndxb(t, u, ux, bval)
      integer           npde
      parameter        (npde=1)
      double precision t, u(npde), ux(npde), bval(npde)
      double precision eps
      parameter        (eps=1d-4)
      double precision a1, a2, a3, expa1, expa2, expa3, temp

      a1 = (-0.5d0 - 4.95d0 * t) * 0.5d-1 / eps
      a2 = (-0.5d0 - 0.75d0 * t) * 0.25d0 / eps
      a3 = - 0.3125d0 / eps
      expa1 = 0.d0
      expa2 = 0.d0
      expa3 = 0.d0
      temp = max(a1, a2, a3)
      if ((a1-temp) .ge. -35.d0) expa1 = exp(a1-temp)
      if ((a2-temp) .ge. -35.d0) expa2 = exp(a2-temp)
      if ((a3-temp) .ge. -35.d0) expa3 = exp(a3-temp)

      bval(1) = u(1) - (0.1d0*expa1+0.5d0*expa2+expa3)  &
                / (expa1+expa2+expa3)
      return
      end

      subroutine uinit(x, u)
      integer           npde
      parameter        (npde=1)
      double precision x, u(npde)
      double precision eps
      parameter        (eps=1d-4)
      double precision a1, a2, a3, expa1, expa2, expa3, temp

      a1 = (-x + 0.5d0) * 0.5d-1 / eps
      a2 = (-x + 0.5d0) * 0.25d0 / eps
      a3 = (-x + 0.375d0) * 0.5 / eps
      expa1 = 0.d0
      expa2 = 0.d0
      expa3 = 0.d0
      temp = max(a1, a2, a3)
      if ((a1-temp) .ge. -35.d0) expa1 = exp(a1-temp)
      if ((a2-temp) .ge. -35.d0) expa2 = exp(a2-temp)
```

```
          if ((a3-temp) .ge. -35.d0) expa3 = exp(a3-temp)

          u(1) = (0.1d0*expa1+0.5d0*expa2+expa3) / (expa1+expa2+expa3)
      return
      end
"""

# Build extension module containing these callbacks.
f2py.compile(prob_def_f, modulename='problemdef', verbose=0, extension='.f95')
```

After the extension module has been built, the compiled callback functions can be used with `bacoli_py`. To do this, a `ProblemDefinition` object is defined which contains pointers to these compiled subroutines in place of the usual Python functions.

```
import bacoli_py
import numpy

# Import Fortran callbacks from extension module.
from problemdef import f, bndxa, bndxb, uinit

# Specify the number of PDE's in this system.
npde = 1

# Pack all of these callbacks and the number of PDE's into a
# ProblemDefinition object.
problem_definition = bacoli_py.ProblemDefinition(npde, f=f._cpointer,
                                                 bndxa=bndxa._cpointer,
                                                 bndxb=bndxb._cpointer,
                                                 uinit=uinit._cpointer)
```

`bacoli_py` can then be used in almost entirely the same way as we saw in the previous examples. The only exception to this begin that the flag `vec` in the call to the `Solver.solve` method must be set to `False`, indicating that the usual vector optimizations used in `bacoli_py` should not be used.

```
# Initialize the Solver object.
solver = bacoli_py.Solver()

# Specify initial time, boundary points, output_points and output_times.

# Set t0.
initial_time = 0.0

# Define the spatial boundaries.
initial_mesh = [0, 1]

# Choose output times and points.
tspan = numpy.linspace(0.001, 1, 100)
xspan = numpy.linspace(0, 1, 100)

# Solve this problem.
evaluation = solver.solve(problem_definition, initial_time, initial_mesh,
                          tspan, xspan, vec=False, dirichlet=True)
```

The computed solution is plotted in Figure 11.

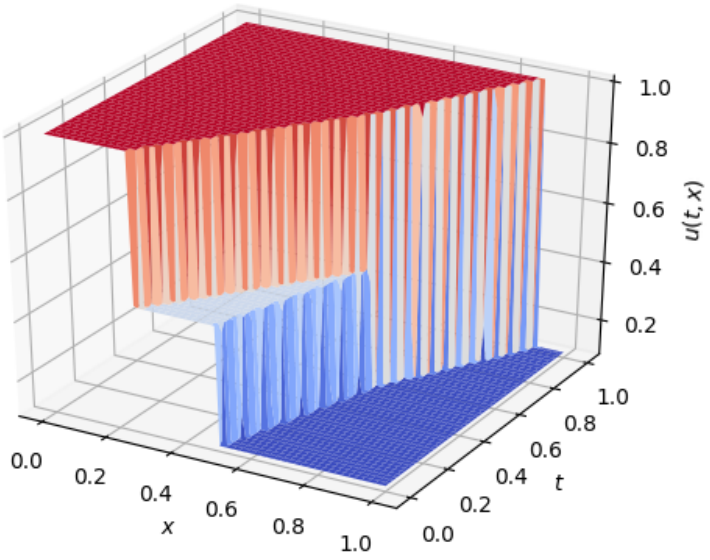

Figure 11: Numerical solution to Two Layer Burgers Equation $\epsilon = 10^{-4}$ using compiled Fortran callback functions.

References

[1] J. Pew, Z. Li, and P. Muir, "Algorithm 962: BACOLI: B-spline adaptive collocation software for PDEs with interpolation-based spatial error control," *ACM Transactions on Mathematical Software*, vol. 42, no. 3, p. 25, 2016.

[2] J. Pew, C. Tannahill, T. Murtha, and P. Muir, "Gaussian collocation/Runge-Kutta PDE software with interpolation-based error control," *Submitted to ACM Transactions on Mathematical Software*, 2019.

[3] B. Keyfitz and N. Keyfitz, "The McKendrick partial differential equation and its uses in epidemiology and population study," *Mathematical and Computer Modelling*, vol. 26, no. 6, pp. 1–9, 1997.

[4] T. Chan and J. Shen, *Image processing and analysis: variational, PDE, wavelet, and stochastic methods*. Society for Industrial and Applied Mathematics, 2005.

[5] J. Pew, Z. Li, C. Tannahill, P. Muir, and G. Fairweather, "Performance analysis of error-control b-spline gaussian collocation software for pdes," *Computers & Mathematics with Applications*, vol. 77, no. 7, pp. 1888–1901, 2019.

[6] Adams, M and Tannahill, C and Muir, P, "Error control Gaussian collocation software for boundary value ODEs and 1D time-dependent PDEs," *Numerical Algorithms*, pp. 1–15, 2019.

[7] J. Díaz, G. Fairweather, and P. Keast, "Algorithm 603: COLROW and ARCECO: FORTRAN packages for solving certain almost block diagonal linear systems by modified alternate row and column elimination," *ACM Transactions on Mathematical Software*, vol. 9, no. 3, pp. 376–380, 1983.

[8] P. Keast, "LAMPAK: a Fortran package for solving certain almost block diagonal matrices," *Unpublished Software*.

[9] T. Oliphant, "NumPy: A guide to NumPy." USA: Trelgol Publishing, 2006–. [Online; accessed June 5, 2019].

[10] P. Peterson, "F2py: a tool for connecting Fortran and Python programs," *International Journal of Computational Science and Engineering*, vol. 4, no. 4, pp. 296–305, 2009.

[11] J. Pew, T. Murtha, C. Tannahill, and P. Muir, "Error control B-spline Gaussian collocation/Runge-Kutta PDE software with interpolation-based spatial error estimation," *Technical Report 2018_002 Dept. of Mathematics and Computing Science Technical Report Series*, 2018.

[12] C. De Boor, *A practical guide to splines*, vol. 27. Springer-Verlag New York, 1978.

[13] L. Petzold, "Description of DASSL: a differential/algebraic system solver," tech. rep., Sandia National Labs., Livermore, CA (USA), 1982.

[14] E. Hairer and G. Wanner, *Solving ordinary differential equations II: stiff and differential-algebraic problems*, vol. 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, Germany, 1988.

[15] J. Pew, T. Murtha, C. Tannahill, and P. Muir, "Performance Analysis of Interpolation-based Spatial Error Control B-spline Gaussian Collocation PDE Software: BDF Time Integration vs. IRK Time Integration," *Technical Report 2019_001 Dept. of Mathematics and Computing Science Technical Report Series*, 2019.