Getting Started with pythODE++

Adam Preuss and Raymond J. Spiteri

September 27, 2016

1 Introduction

The pythODE++ problem-solving environment (PSE) is designed to evaluate permutations of numerical methods and initial-value problems (IVPs)[1]. It is a (mostly) stand-alone collection of scripts and programs that is heavily based on the functionality of a PSE that is entirely written in python, pythODE. The pythODE++ PSE is designed to be performance-focused with an emphasis on runtime measurements of numerical experimentation, whereas pythODE is geared more toward performing in-depth and highly customizable analysis.

The numerical methods and IVPs in pythODE++ are written entirely in C++. The supporting execution and analysis scripts are written in Python. A general overview of the software, motivation, and specific examples of application is presented in [?]. This tutorial aims to introduce a user to the basics of behind implementing IVPs and numerical methods in pythODE++. This tutorial is not complete documentation for each software component. However, there are many examples of IVPs and numerical methods already implemented in pythODE++ that can be used as a starting point for implementing new, more complicated problems and methods.

1.1 Requirements

1.1.1 System

The minimalistic version of pythODE++ requires a C++ compiler, Python version 2.7.3 with numpy and scipy modules, and the ability to link against the standard library. Therefore, pythODE++ should be supported by any UNIX-like platform (e.g., Linux, OpenBSD), Windows (via Cygwin), or Mac OS X. At present, the pythODE++ PSE is run exclusively from the command line.

The pythODE++PSE can optionally link libraries that are used to support the solution of sparse linear systems and (sparse) automatic differentiation. It is necessary to have ADOL-C installed if automatic differentiation is desired. To use sparse matrices, UMFPACK is required, along with its supporting library for sparsity, ColPack. Analysis graphs are

[1]Recall that an IVP is comprised of an ordinary differential equation and an initial condition.

generated using gnuplot; therefore, gnuplot should be installed. Specific instructions for installing these packages is generally operating-system dependent. Some common examples on how to install these optional libraries are given in Section 2.

1.1.2 Background Knowledge

To have a sense of the function of the supporting code in this PSE, the user should be relatively comfortable with general programming concepts such as virtual memory, lists, hashes, and function pointers, as well as object-oriented (OO) concepts including (abstract) classes, inheritance, polymorphism, and operator overloading. Further, an understanding the basic concepts of efficient programming with respect to the machine cache is highly beneficial. Ignoring caching effects can cause severe loss of performance.

As discussed in Section 1, the pythODE++ PSE is written in a combination of C++ and Python. This software makes use of many advanced C++ concepts. Although the implementation of numerical methods and IVPs is not generally complicated, much of the supporting code uses such concepts. Specific examples include templated classes/functions and exception handling.

1.2 Software Design

The software is organized into the following directories:

- `analysis` contains C++ code for loading problem runs from disk and performing analysis passes on them.
- `core` contains all supporting functions and classes for vectors, matrices, hashes, lists, file input/output, etc.
- `ivps` contains implementations for all IVPs in `pythODE++`.
- `loaders` contains supporting code that maintains a registry of all solvers, methods, and IVPs. This directory also contains the entry point for all components of the software such as the runner and the analysis tools.
- `methods` contains implementations for all methods in `pythODE++`.
- `runner` contains code for running a set of parameters.
- `scripts` contains Python scripts for each of the numerical experiments.
- `solvers` contains implementations for all solvers in `pythODE++`.
- `tutorial` contains the LaTeX source associated with this document.

2 Installation

The software can presently be accessed via the Numerical Simulation Lab's subversion server. On your local system, navigate to the location in which `pythODE++` is to be installed and execute the following command:

```
svn co svn+ssh://<username>@simcity.usask.ca/Users/svn/pythODE/pythODE++
```

2.1 Compiling

The `pythODE++` PSE is compiled using the `build` script located at the top level of the `pythODE++` repository. Simply invoke this script (by typing `./build`) to compile. Arguments to the builder script include `debug` to compile with special debug information and `clean` to completely remove all executables and intermediate build files.

By default, the software assumes that it will be built linking libraries for `ADOL-C` and `UMFPACK`. If these libraries are not installed, the build script should be called with arguments `noadolc` and/or `nosparsity` .

2.2 Optional Libraries

Installation instructions for the optional libraries are presented for Ubuntu Linux and for Mac OSX. The `pythODE++` PSE has been extensively tested on both of these operating systems. It is recommended that the latest version of these libraries be used; however, during the development of `pythODE++`, no issues involving the library versions were observed.

2.2.1 Ubuntu Linux

Installation of `gnuplot` and `UMFPACK` is simple on Ubuntu Linux because these libraries exist as precompiled packages. They can be installed as follows (when run as root):

```
apt-get install gnuplot libsuitesparse-dev
```

When you compile `pythODE++` make sure to include the libraries in your LD_LIBRARY path, the command is:

```
export LD_LIBRARY= \$ LD_LIBRARY:/usr/include/
```

3 Scripting

The scripts contain many examples that evaluate numerical methods on IVPs. In general, performing an experiment consists of two parts. First, the runner loops over a set of specified runs. Second, the analysis modules are used to gather runs and generate meaningful graphs.

Each numerical experiment can be contained in a single Python module. A numerical experiment can be invoked by using the `run-experiment.sh` script and specifying the module name (without the `.py`) as the argument. This module defining the numerical experiment must specify the following two global variables:

- `simname` is the name of the numerical simulation. It is used in directory names and can generally be thought of as a unique identifier for a set of similar numerical experiments. This name should probably not contain spaces, only because file management is more fragile when file names contain spaces.

- `simpath` is the path for the numerical simulation. All files associated with the simulation are stored in this path. The auto-runner creates new directories within this path each time a numerical simulation is conducted; therefore, the simulation path can shared for all `pythODE++` simulations. For clean file management, it might be useful (though not necessary) to add `simname` as a subdirectory `simpath`, as is done in virtually all of the examples in `pythODE++`.

There are two important functions required to specify a numerical simulation. The first construct a list of run parameters; the second specifies the analysis passes.

Run parameters are specified by the function `GenerateRunList()`, which returns a list of hashes. Each hash specifies the set of parameters for the run. What follows is a list of parameters that are commonly used. All of these are not required; however, common sense must be invoked for solving IVPs when deciding whether a given parameter combination is valid, e.g., specifying a constant solver with a predictive step controller is not valid. Additional method- or problem-specific parameters may be specified as well. The following example shows how to instruct `pythODE++` to solve two IVPs using two methods.

```
def GenerateRunList():
    runlist = []
    for ivp in ('Brusselator1D','Brusselator2D'):
        for method in ('RK4','DOPR54'):
            runlist.append({'ivp':ivp,
                            'method':method,
                            'solver':'ConstantSolver',
                            'dt':1e-2})
    return runlist
```

Required Parameters	
`ivp`	The (registered) name of the IVP that is to be solved, e.g., `Brusselator1D`.
`method`	The (registered) name of the method that is to be used, e.g., `RK4`.
`solver`	The solver that is to be used. This value can be one of `ConstantSolver`, `StepDoublingSolver`, or `EmbeddedSolver`.
(Generally) Optional Parameters	
`dt`	Initial timestep.
`atol`	Absolute tolerance for step control. The default is 10^{-5}.
`rtol`	Relative tolerance for step control. The default is 10^{-5}.
`newton tol`	Tolerance for Newton's method. The default is 10^{-8}.
`sparse`	Specifies whether to use sparsity when solving linear systems. This value can either be 0 or 1.
`jacobian`	The method of Jacobian calculation. The options for this value are `Forward`, `Centred`, `Autodiff`, or `Analytic`.
`jacobian splitting`	Specifies whether to apply Jacobian splitting to the IVP. This value can either be 0 or 1.
`max steps`	The maximum number of steps until simulation is stopped.
`min write time`	The minimum amount of elapsed simulation time before the next solution point can be written. A better approach is to use an interpolant that is associated with the numerical method. However, `pythODE++` does not presently support an interpolated output.
`timing group`	A given parameter set must be run multiple times to conduct accurate timings. This is accomplished by specifying the parameter set hash multiple times in the run list. Each group of identical parameter sets should have a unique timing group (unique with respect to other sets of identical parameters) so the analysis phase can appropriately group runs.

Analysis passes are specified by the function `GenerateAnalysisPasses()`, which similarly returns a list of hashes. Each hash specifies the set of parameters to be used in conducting the analysis pass. For example, to print reference solutions, perform time versus accuracy comparisons, and perform steps versus accuracy comparisons, the analysis function might look like:

```
def GenerateAnalysisPasses():
    passes = []
```

```python
for ivp in ivps:
    # Perform solution plots
    passes.append({ 'mode': 'Solutions',
                    'title': ivp + ' Solutions',
                    'filename': ivp + '-solutions',
                    'xlabel': 'Time (s)',
                    'ylabel': 'Solution',
                    'legend': SolutionLegendName,
                    'match': {'ivp': ivp,
                              'method': methods,
                              'atol': tolerances[-1][0] } })
    # Perform time versus accuracy plots
    passes.append({ 'mode': 'Accuracy',
                    'title': ivp + ' CPU Time vs. Accuracy',
                    'filename': ivp + '-cputime',
                    'xlabel': 'Accuracy',
                    'ylabel': 'CPU Time (ms)',
                    'legend': AccuracyLegendName,
                    'reference solution': reference_solutions[ivp],
                    'match': {'ivp': ivp},
                    'comparison': 'time',
                    'group': ['method','solver','jacobian'] })
    # Perform steps versus accuracy plots
    passes.append({ 'mode': 'Accuracy',
                    'title': ivp + ' Steps vs. Accuracy',
                    'filename': ivp + '-steps',
                    'xlabel': 'Accuracy',
                    'ylabel': 'Steps',
                    'legend': AccuracyLegendName,
                    'reference solution': reference_solutions[ivp],
                    'match': {'ivp': ivp},
                    'comparison': 'steps',
                    'group': ['method','solver','jacobian'] })
    return passes
```

4 Implementing a Problem

The implementation of an IVP consists of defining the right-hand side and the initial condition. All IVPs inherit from the base-class `BaseIVP`. It is effective to learn by example. There are many IVPs provided with `pythODE++` upon which to base future implementations.

This section gives a brief overview of the basics for implementing IVPs in `pythODE++`.

For the simple ODE $y' = -y, y(0) = 10$ where the final simulation time is 5, an implementation might look like:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H
// TestEquation is inheriting from BaseIVP
class TestEquation : public BaseIVP {
protected:
    // Definition of the right-hand side.
    // The function name and parameters must match
    // this format exactly.
    void RHS(const FP t, const Vec<FP>& y, Vec<FP>& yp) {
        yp(0) = -y(0);
    }

public:
    // Definition of the constructor.
    // Once again, the parameters must match exactly, and
    // this function must always pass params to the BaseIVP
    TestEquation(Hash<ParamValue>& params) : BaseIVP(params) {
        // Set the (default) final time
        SetDefaultFP(params, "tf", 5.);
         // Set the problem size to 1
        _initialCondition.Resize(1);
        // Set the initial condition to 10
        _initialCondition[0] = 10;
    }

    IVP_NAME("Nonstiff_A1") // Macro to define IVP name
};
#endif
```

The implementation of an IVP class (as in the above example) should be placed into a header file; this header file must be included in `loaders/ivploader.cpp`. Lastly, for the IVP to be usable, `IVPCASE(NewClassName)` must be inserted in `AllocIVP()` (with `NewClassName` replaced by the name of the class). Note that this line must come before the `throw` statement.

The right-hand side of the IVP can be interpreted as additively split when it is comprised of the sum of two or more contributing factors. A specific class `TwoSplittingIVP` inherits from `BaseIVP` to make implementations of 2-additive IVPs easy. In such cases, the user can simply define:

- `void Split1(const FP t, const Vec<FP>& y, Vec<FP>& yp) { ... }`
- `void Split2(const FP t, const Vec<FP>& y, Vec<FP>& yp) { ... }`

For many numerical methods, the Jacobian is required. Jacobian matrices generated by forward or centred differences do not require any additional work to implement; ones generated by automatic differentiation or manually (e.g., analytical) do. The class `BaseIVP` contains a virtual function `JacAnalytic` (or `JacAnalyticSparse` when using sparsity) that can be overloaded to provide the analytic Jacobian. See the example contained in `ivps/zbinden/advection1d.h` for an IVP that is 2-additive, supports automatic differentiation, and defines a manual analytic Jacobian.

5 Implementing a Method

The implementation of a method defines how to take a step from one state to another. The following example show how to create a single RK method and an IMEX method. The `Runge2` is a second-order, explicit RK method, which is implemented as

```
// Inherit from the class of ERK methods
class Runge2 : public ERK {
public:
    // Constructor that specifies a Butcher tableau
    // of size 2
    Runge2(Hash<ParamValue>& params, BaseIVP* ivp)
        : ERK(params, ivp, 2) {
        _a(1,0) = 1./2;
        _b(1) = 1.;
        // Fill up the C values to make the method
        // consistent
        FillC();
    }

    // The name of method
    const char* GetName() const {
        return "Runge 2";
    }

    // Specify the order for step control
    long GetOrder() const {
        return 2;
    }
};
```

The IMEX method specifies two Butcher tableaux, where `_a` and `_b` refer to the implicit tableau, and `_a2` and `_b2` refer to the explicit tableau. Examples are given for any method in the folder `methods/ark`.