(py)oscode Documentation

Release 1.0

Fruzsina Agocs

Jun 30, 2023

CONTENTS:

CHAPTER

ONE

(PY)OSCODE: OSCILLATORY ORDINARY DIFFERENTIAL EQUATION SOLVER

1.1 About

`oscode` is a C++ tool with a Python interface that solves **osc**illatory **o**rdinary **d**ifferential **e**quations efficiently. It is designed to deal with equations of the form

$$\ddot{x}(t) + 2\gamma(t)\dot{x}(t) + \omega^2(t)x(t) = 0,$$

where $\gamma(t)$ and $\omega(t)$ can be given as arrays.

`oscode` makes use of an analytic approximation of $x(t)$ embedded in a stepping procedure to skip over long regions of oscillations, giving a reduction in computing time. The approximation is valid when the frequency $\omega(t)$ changes slowly relative to the timescales of integration, it is therefore worth applying when this condition holds for at least some part of the integration range.

For the details of the numerical method used by `oscode`, see the Citations section.

1.2 Installation

1.2.1 Dependencies

Basic requirements for using the C++ interface:

- C++11 or later
- Eigen (a header-only library included in this source)

The strictly necessary Python dependencies are automatically installed when you use *pip* or the *setup.py*. They are:

- numpy

The *optional* dependencies are:

- **for tests**
  - scipy
  - pytest
- **for examples/plotting**
  - matplotlib
  - scipy
- **for generating offline documentation**
  - sphinx
  - doxygen
  - breathe
  - exhale

1.2.2 Python

pyoscode can be installed via pip

```
pip install pyoscode
```

or via the setup.py

```
git clone https://github.com/fruzsinaagocs/oscode
cd oscode
python setup.py install --user
```

You can then import pyoscode from anywhere. Omit the `--user` option if you wish to install globally or in a virtual environment. If you have any difficulties, check out the *FAQs* section below.

You can check that things are working by running *tests/* (also ran by Travis continuous integration):

```
pytest tests/
```

1.2.3 C++

oscode is a header-only C++ package, it requires no installation.

```
git clone https://github.com/fruzsinaagocs/oscode
```

and then include the relevant header files in your C++ code:

```
#include "solver.hpp"
#include "system.hpp"
```

1.3 Quick start

Try the following quick examples. They are available in the examples.

1.3.1 Python

Introduction to pyoscode

Cosmology examples

1.3.2 C++

Introduction to oscode *examples/burst.cpp*

To plot results from *burst.cpp* *examples/plot_burst.py*

To compile and run:

```
g++ -g -Wall -std=c++11 -c -o burst.o burst.cpp
g++ -g -Wall -std=c++11 -o burst burst.o
./burst
```

1.4 Documentation

Documentation is hosted at readthedocs.

To build your own local copy of the documentation you can run:

```
cd pyoscode/docs
make html
```

1.5 Citation

If you use `oscode` to solve equations for a publication, please cite:

- Efficient method for solving highly oscillatory ordinary differential equations with applications to physical systems,
- Dense output for highly oscillatory numerical solutions

1.6 Contributing

Any comments and improvements to this project are welcome. You can contribute by:

- Opening and issue to report bugs and propose new features.
- Making a pull request.

1.7 Further help

You can get help by submitting an issue or posting a message on Gitter.

1.8 FAQs

1.8.1 Installation

1. **Eigen import errors:**

   ```
   pyoscode/_pyoscode.hpp:6:10: fatal error: Eigen/Dense: No such file or directory
    #include <Eigen/Dense>
             ^~~~~~~~~~~~~
   ```

   Try explicitly including the location of your Eigen library via the `CPLUS_INCLUDE_PATH` environment variable, for example:

```
CPLUS_INCLUDE_PATH=/usr/include/eigen3 python setup.py install --user
# or
CPLUS_INCLUDE_PATH=/usr/include/eigen3 pip install pyoscode
```

where `/usr/include/eigen3` should be replaced with your system-specific eigen location.

1.9 Thanks

Many thanks to **Will Handley**, **Lukas Hergt**, **Anthony Lasenby**, and **Mike Hobson** for their support and advice regarding the algorithm behind *oscode*. There are many packages without which some part of *oscode* (e.g. testing and examples) wouldn't run as nicely and smoothly, thank you all developers for making and maintaining these open-source projects. A special thanks goes to the devs of exhale for making the beautiful C++ documentation possible.

1.10 Changelog

- **1.1.2: current version**
  - Dense output bug fix at the C++ interface
- **1.1.1:**
  - Support for mac and Windows OS at CI.
- **1.1.0:**
  - Users can now define w, g as functions in Python (pyoscode) and call the solver via pyoscode.solve_fn(...)
- **1.0.6:**
  - Fix issues related to dense output not being correctly generated, e.g. when timepoints at which dense output was asked for are in descending order, etc.
- **1.0.5:**
  - Fixes related to dense output generation
  - Support for w, g to be given as class member functions in C++
  - Switched to GH actions for continuous integration, and fixed code such that unit tests would run again
  - Minor tweaks
- **1.0.4:**
  - set minimally required numpy version: numpy>=1.20.0
  - drop Python 2.7 support, instead support 3.8 and 3.9 in addition to 3.7
- **1.0.3:**
  - paper accepted to JOSS
- **1.0.2:**
  - Fixed getting correct numpy include directories
- **1.0.1:**
  - Added *pyproject.toml* to handle build dependencies (numpy)

- **1.0.0:**
  - Dense output
  - Arrays for frequency and damping term need not be evenly spaced
  - Automatic C++ documentation on readthedocs
  - Eigen included in source for pip installability
  - First pip release :)
- **0.1.2:**
  - Bug that occurred when beginning and end of integration coincided corrected
- **0.1.1:**
  - Automatic detection of direction of integration
- **0.1.0:**
  - Memory leaks at python interface fixed
  - C++ documentation added

CHAPTER

TWO

PYOSCODE

pyoscode.**solve**(*ts*, *ws*, *gs*, *ti*, *tf*, *x0*, *dx0*, *t_eval=[]*, *logw=False*, *logg=False*, *order=3*, *rtol=0.0001*, *atol=0.0*, *h=None*, *full_output=''*, *even_grid=False*, *check_grid=False*)

Solve a differential equation with the RKWKB method.

Parameters

ts: numpy.ndarray [float] or list [float] An array of real numbers representing the values of the independent variable at which the frequency and friction term are evaluated.

ws: numpy.ndarray [complex] or list [complex] An array-like object of real or complex numbers, representing the values of frequency w at the points given in ts.

gs: numpy.ndarray [complex] or list [complex] An array-like object of real or complex numbers representing the values of the friction term g at the points given in ts.

ti,tf: float Start and end of integration range.

x0, dx0: complex Initial values of the dependent variable and its derivative.

t_eval: numpy.ndarray [float] or list [float] An array of times where the solution is to be returned.

logw, logg: boolean, optional If true, the array of frequencies and friction values, respectively, will be exponentiated (False, False by default).

order: int, optional Order of WKB approximation to use, 3 (the highest value) by default.

rtol, atol: float, optional Relative and absolute tolerance of the solver, 1e-4 and 0 by default. Note that atol at the moment is not implemented.

h: float, optional Size of the initial step, 1 by default.

full_output: str , optional If given, the return dictionary will be written to a file with the supplied name.

even_grid: boolean, optional False by default. Set this to True if the ts array is evenly spaced for faster interpolation.

check_grid: boolean, optional False by default. If True, the fineness of the ws, gs grids will be checked based on how accurate linear interpolation would be on them, and a warning will be issued if this accuracy is deemed too low. It's a good idea to set this to True when solving an equation for the first time.

Returns

A dictionary with the following keywords and values:

sol: list [complex] A list containing the solution evaluated at timepoints listed under the 't' keyword.

dsol: list [complex] A list containint the first deriv ative of the solution evaluated at timepoints listed under the 't' keyword.

t: list [float] Contains the values of the independent variable where the solver stepped, i.e. evaluated the solution at. This is not determined by the user, rather these are the internal steps the solver naturally takes when integrating.

types: list [float] A list of True/False values corresponding to the step types the solver chose at the timepoints listed under the keyword 't'. If True, the step was WKB, and RK otherwise.

x_eval: list [complex] Values of the solution at the points specified in t_eval.

dx_eval: list [complex] Values of the derivative of the solution at the points specified in t_eval.

cts_rep: list [list [complex]] List containing a list of the polynomial coefficients needed to construct a continuous representation of the solution within each internal step the algorithm takes.

pyoscode.**solve_fn**(*w*, *g*, *ti*, *tf*, *x0*, *dx0*, *t_eval=[]*, *order=3*, *rtol=0.0001*, *atol=0.0*, *h=None*, *full_output=''*)

Solves the differential equation x" + 2g(t)x' + w^2(t)y = 0 for y(t) and y'(t) on an interval (ti, tf) given initial conditions [x(ti), x'(ti)] and function handles w(t), g(t).

Parameters

w: callable(t) Computes the frequency at point t, may return a complex number.

g: callable(t) Computes the damping term at point t, may return a complex number.

ti,tf: float Start and end of integration range.

x0, dx0: complex Initial values of the dependent variable and its derivative.

t_eval: numpy.ndarray [float] or list [float] An array of times where the solution is to be returned.

order: int, optional Order of WKB approximation to use, 3 (the highest value) by default.

rtol, atol: float, optional Relative and absolute tolerance of the solver, 1e-4 and 0 by default. Note that atol at the moment is not implemented.

h: float, optional Size of the initial step, 1 by default.

full_output: str , optional If given, the return dictionary will be written to a file with the supplied name.

Returns

A dictionary with the following keywords and values:

sol: list [complex] A list containing the solution evaluated at timepoints listed under the 't' keyword.

dsol: list [complex] A list containint the first deriv ative of the solution evaluated at timepoints listed under the 't' keyword.

t: list [float] Contains the values of the independent variable where the solver stepped, i.e. evaluated the solution at. This is not determined by the user, rather these are the internal steps the solver naturally takes when integrating.

types: list [float] A list of True/False values corresponding to the step types the solver chose at the timepoints listed under the keyword 't'. If True, the step was WKB, and RK otherwise.

x_eval: list [complex] Values of the solution at the points specified in t_eval.

dx_eval: list [complex] Values of the derivative of the solution at the points specified in t_eval.

cts_rep: list [list [complex]] List containing a list of the polynomial coefficients needed to construct a continuous representation of the solution within each internal step the algorithm takes.

CHAPTER

THREE

1 USING THE C++ INTERFACE (OSCODE)

3.1 1.1 Overview

This documentation illustrates how one can use `oscode` via its C++ interface. Usage of `oscode` involves

- defining an equation to solve,
- solving the equation,
- and extracting the solution and other statistics about the run.

The next sections will cover each of these. For a complete reference, see the *C++ interface reference* page, and for examples see the examples directory on GitHub.

3.2 1.2 Defining an equation

The equations `oscode` can be used to solve are of the form

$$\ddot{x}(t) + 2\gamma(t)\dot{x}(t) + \omega^2(t)x(t) = 0,$$

where $x(t)$, $\gamma(t)$, $\omega(t)$ can be complex. We will call t the independent variable, x the dependent variable, $\omega(t)$ the frequency term, and $\gamma(t)$ the friction or first-derivative term.

Defining an equation is via

- giving the frequency $\omega(t)$,
- giving the first-derivative term $\gamma(t)$,

Defining the frequency and the first-derivative term can either be done by giving them as **functions explicitly**, or by giving them as **sequences** evaluated on a grid of t.

3.2.1 1.2.1 ω and γ as explicit functions

If ω and γ are closed-form functions of time, then define them as

```
#include "solver.hpp" // de_system, Solution defined in here

std::complex<double> g(double t){
    return 0.0;
};
```

(continues on next page)

(continued from previous page)

```
std::complex<double> w(double t){
    return std::pow(9999,0.5)/(1.0 + t*t);
};
```

Then feed them to the solver via the de_system class:

```
de_system sys(&w, &g);
Solution solution(sys, ...) // other arguments left out
```

3.2.2 1.2.2 ω and γ as time series

Sometimes ω and γ will be results of numerical integration, and they will have no closed-form functional form. In this case, they can be specified on a grid, and `oscode` will perform linear interpolation on the given grid to find their values at any timepoint. Because of this, some important things to **note** are:

- `oscode` will assume the grid of timepoints ω and γ are **not evenly spaced**. If the grids are evenly sampled, set `even=true` in the call for `de_system()`, this will speed linear interpolation up significantly.
- The timepoints grid needs to be **monotonically increasing**.
- The timepoints grid needs to **include the range of integration** (t_i,:math:t_f).
- The grids for the timepoints, frequencies, and first-derivative terms have to be the **same size**.
- The speed/efficiency of the solver depends on how accurately it can carry out numerical integrals of the frequency and the first-derivative terms, therefore the **grid fineness** needs to be high enough. (Typically this means that linear interpolation gives a $\omega(t)$ value that is accurate to 1 part in 10^6 or so.) If you want *oscode* to check whether the grids were sampled finely enough, set `check_grid=true` in the call for `de_system()`.

To define the grids, use any array-like container which is **contiguous in memory**, e.g. an `Eigen::Vector`, `std::array`, `std::vector`:

```
#include "solver.hpp" // de_system, Solution defined in here

// Create a fine grid of timepoints and
// a grid of values for w, g
N = 10000;
std::vector<double> ts(N);
std::vector<std::complex<double>> ws(N), gs(N);

// Fill up the grids
for(int i=0; i<N; i++){
    ts[i] = i;
    ws[i] = std::sqrt(i);
    gs[i] = 0.0;
}
```

They can then be given to the solver again by feeding a pointer to their underlying data to the `de_system` class:

```
de_system sys(ts.data(), ws.data(), gs.data());
Solution solution(sys, ...) // other arguments left out
```

Often ω and γ are much easier to perform linear interpolation on once taken natural log of. This is what the optional `islogw` and `islogg` arguments of the overloaded `de_system::de_system()` constructor are for:

```
#include "solver.hpp" // de_system, Solution defined in here

// Create a fine grid of timepoints and
// a grid of values for w, g
N = 10000;
std::vector<double> ts(N);
std::vector<std::complex<double> logws(N), gs(N); // Note the log!

// Fill up the grids
for(int i=0; i<N; i++){
    ts[i] = i;
    logws[i] = 0.5*i;
    gs[i] = 0.0; // Will not be logged
}

// We want to tell de_system that w has been taken natural log of, but g
// hasn't. Therefore islogw=true, islogg=false:
de_system sys(ts.data(), logws.data(), gs.data(), true, false);
Solution solution(sys, ... ) // other arguments left out
```

1.2.2.1 DIY interpolation

For some problems, linear interpolation of ω and γ (or their natural logs) might simply not be enough.

For example, the user could carry out cubic spline interpolation and feed ω and γ as functions to `de_system`.

Another example for wanting to do (linear) interpolation outside of `oscode` is when `Solution.solve()` is ran in a loop, and for each iteration a large grid of ω and γ is required, depending on some parameter. Instead of generating them over and over again, one could define them as functions, making use of some underlying vectors that are independent of the parameter we iterate over:

```
// A, B, and C are large std::vectors, same for each run
// k is a parameter, different for each run
// the grid of timepoints w, g are defined on starts at tstart, and is
// evenly spaced with a spacing tinc.

// tstart, tinc, A, B, C defined here

std::complex<double> g(double t){
    int i;
    i=int((t-tstart)/tinc);
    std::complex<double> g0 = 0.5*(k*k*A[i] + 3.0 - B[i] + C[i]*k;
    std::complex<double> g1 = 0.5*(k*k*A[i+1] + 3.0 - B[i+1] + C[i+1]*k);
    return (g0+(g1-g0)*(t-tstart-tinc*i)/tinc);
};
```

3.3 1.3 Solving an equation

Once the equation to be solver has been defined as an instance of the `de_system` class, the following additional information is necessary to solve it:

- initial conditions, $x(t_i)$ and $\dot{x}(t_f)$,
- the range of integration, from t_i and t_f,
- (optional) set of timepoints at which dense output is required,
- (optional) order of WKB approximation to use, `order=3`,
- (optional) relative tolerance, `rtol=1e-4`,
- (optional) absolute tolerance `atol=0.0`,
- (optional) initial step `h_0=1`,
- (optional) output file name `full_output=""`,

Note the following about the optional arguments:

- `rtol`, `atol` are tolerances on the local error. The global error in the solution is not guaranteed to stay below these values, but the error per step is. In the RK regime (not oscillatory solution), the global error will rise above the tolerance limits, but in the WKB regime, the global error usually stagnates.
- The initial step should be thought of as an initial estimate of what the first stepsize should be. The solver will determine the largest possible step within the given tolerance limit, and change `h_0` if necessary.
- The full output of `solve()` will be written to the filename contained in `full_output`, if specified.

Here's an example to illustrate usage of all of the above variables:

```
#include "solver.hpp" // de_system, Solution defined in here

// Define the system
de_system sys(...) // For args see previous examples

// Necessary parameters:
// initial conditions
std::complex<double> x0=std::complex<double>(1.0,1.0), dx0=0.0;
// range of integration
double ti=1.0, tf=100.0;

// Optional parameters:
// dense output will be required at the following points:
int n = 1000;
std::vector t_eval(n);
for(int i=0; i<n; i++){
    t_eval[i] = i/10.0;
}
// order of WKB approximation to use
int order=2;
// tolerances
double rtol=2e-4, atol=0.0;
// initial step
double h0 = 0.5;
// write the solution to a file
```

(continues on next page)

(continued from previous page)

```
std::string outfile="output.txt";

Solution solution(sys, x0, dx0, ti, tf, t_eval.data(), order, rtol, atol, h0, outfile);
// Solve the equation:
solution.solve()
```

Here, we've also called the `solve()` method of the `Solution` class, to carry out the integration. Now all information about the solution is in `solution` (and written to `output.txt`).

3.4 1.4 Using the solution

Let's break down what `solution` contains (what `Solution.solve()` returns). An instance of a `Solution` object is returned with the following attributes:

- `times` [std::list of double]: timepoints at which the solution was determined. These are **not** supplied by the user, rather they are internal steps that the solver has takes. The list starts with t_i and ends with t_f, these points are always guaranteed to be included.
- `sol` [std::list of std::complex<double>]: the solution at the timepoints specified in `times`.
- `dsol` [std::list of std::complex<double>]: first derivative of the solution at timepoints specified in `times`.
- `wkbs` [std::list of int/bool]: types of steps takes at each timepoint in `times`. **1** if the step was WKB, **0** if it was RK.
- `ssteps` [int]: total number of accepted steps.
- `totsteps` [int]: total number of attempted steps (accepted + rejected).
- `wkbsteps` [int]: total number of successful WKB steps.
- `x_eval` [std::list of std::complex<double>]: dense output, i.e. the solution evaluated at the points specified in the `t_eval` optional argument
- `dx_eval` [std::list of std::complex<double>]: dense output of the derivative of the solution, evaluted at the points specified in `t_eval` optional argument.

CHAPTER

FOUR

OSCODE

4.1 Full API

PYTHON MODULE INDEX

INDEX