ACSIJ Advances in Computer Science: an International Journal, Vol. 5, Issue 3, No.21 , May 2016
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

COPADS IV: Fixed Time-Step ODE Solvers for a System of Equations Implemented as a Set of Python Functions

Maurice HT Ling

Colossus Technologies LLP, Republic of Singapore
School of BioSciences, The University of Melbourne
Parkville, Victoria 3010, Australia
mauriceling@colossus-tech.com

Abstract

Ordinary differential equation (ODE) systems are commonly used many different fields. The *de-facto* method to implement an ODE system in Python programming using SciPy requires the entire system to be implemented as a single function, which only allow for inline documentation. Although each equation can be broken up into sub-equations, there is no compart-mentalization of sub-equations to its ODE. A better method will be to implement each ODE as a function. This encapsulates the sub-equations to its ODE, and allow for function and inline documentation, resulting in better maintainability. This study presents the implementation 11 ODE solvers that enable each ODE in a system to be implemented as a function. Three enhancements will be added. Firstly, the solvers will be implemented as generators to allow for virtually infinite simulation and returning a stream of intermediate results for analysis. Secondly, the solvers will allow for non-ODE-bounded variables or solution vector to improve code and results documentation. Lastly, a means to set upper and lower boundary of ODE solutions will be added. Validation testing shows that the enhanced ODE solvers give comparable results to SciPy's default ODE solver. The implemented solvers are incorporated into COPADS repository (https://github.com/copads/copads).

Keywords: *ODE solvers; system of ODEs; Python; Documentation; one-ODE-one-function*

1. Introduction

Ordinary differential equation (ODE) is an important mathematical tool in many fields [1], especially in modeling, as it describes the rate of change of one variable with respect to another variable. The integral of an ODE results in the analytical form of equation. However, symbolic integration of an ODE may be difficult and numerical integration is usually performed. An ODE solver is an algorithm to perform numerical integration on an ODE. When a set of ODEs describing different is related to a common underlying variable, such as time, it becomes a system of ODEs and relationship between different variables can be examined.

Almost all programming languages will have tools or libraries to implement ODE as a single equation or as a system and solving the ODE(s). In Python programming, the standard way of writing ODE systems in SciPy [2] and Odespy [3] is to implement one or more ODEs within a single function. Using a simple 3-equation ODE to model zombie invasion [4] as an example, the following are means of writing the ODEs in SciPy:

```
def example1(y, t):
    human = birth - \
       transmission*y[0]*y[1] - death*y[0]
    zombie = transmission*y[0]*y[1] + \
       resurect*y[2] - destroy*y[0]*y[1]
    dead = death*y[0] + \
       destroy*y[0]*y[1] -  resurect*y[2]
    return [human, zombie, dead]

def example2(y, t):
    f = [birth - transmission*y[0]*y[1] - \
         death*y[0],
         transmission*y[0]*y[1] + \
         resurect*y[2] - destroy*y[0]*y[1],
         death*y[0] + destroy*y[0]*y[1] - \
         resurect*y[2]]
    return f
```

However, standard means of writing ODEs in SciPy [2] as shown above present two issues. Firstly, it is common to have ODEs comprising of many constituent functions. This will result in equations spending multiple lines and possibly across multiple pages, which is difficult to read. Secondly, it does not allow for effective documentation. As a result, a large system of ODEs written in SciPy's method will be difficult to read and usually suffer from the lack of ample documentation.

An improved solution is to allow users to implement each ODE as a separate function, allowing for function-level documentation that is compatible to documentation generators, such as Epydoc (http://epydoc.sf.net) or Sphinx (http://sphinx-doc.org). This is followed by consolidating individual ODEs into a system of ODEs using a data

structure, such as a list, before solving. In this case, the 3 ODEs in Munz et al. [4] can be written as:

```
def human(t, y):
    '''Modeling the number of remaining
    humans. Infected represents the number
    of humans infected by existing zombies
    (zombified). Dead represents natural
    (non-zombified) deaths.'''
    infected = transmission * y[0] * y[1]
    dead = death * y[0]
    return birth - infected - dead
def zombie(t, y):
    '''Modeling the number of zombies.
    Newly infected represents the number of
    new additions as humans are infected by
    existing zombies (zombified). Resur-
    rected represents revival of the dead.
    Destroyed represents the number of
    zombies destroyed by humans.'''
    newly_infected = transmission * \
       y[0] *  y[1]
    resurrected = resurrect * y[2]
    destroyed = destroy * y[0] * y[1]
    return newly_infected + \
       resurrected - destroyed
def dead(t, y):
    '''Modeling the total number of dead.
    Natural death represents natural (non-
    zombified) deaths of humans. Destroyed
    zombies represent the number of zombies
    destroyed by humans. Created zombies
    represents revival of all dead, be it
    from natural death or destroyed
    zombies.'''
    natural_death = death * y[0]
    destroyed_zombies = destroy * \
       y[0] * y[1]
    created_zombies = resurrect * y[2]
    return natural_death + \
       destroyed_zombies - created_zombies

example4 = [human, zombie, dead]
```

However, this will not execute in SciPy. A compromise may be using functions within a function (as shown below). However, this results in type error in SciPy; hence, not suitable.

```
def example3(y, t):
    def human(y, t):
        infected = transmission * \
                   y[0] * y[1]
        dead = death * y[0]
        return birth - infected - dead
    def zombie(y, t):
        newly_infected = transmission * \
              y[0] * y[1]
        resurrected = resurrect * y[2]
        destroyed = destroy * y[0] * y[1]
        return newly_infected + \
                   resurrected - destroyed
    def dead(y, t):
        natural_death = death * y[0]
        destroyed_zombies = destroy * \
                            y[0] * y[1]
        created_zombies = resurrect * y[2]
        return natural_death + \
               destroyed_zombies - \
               created_zombies
    return [human, zombie, dead]
```

There are a number of Python implementations of ODE solvers available on the web. Senning [5] from Gordon College, USA, released an implementation based on the ODE solvers in Octave but required SciPy type implementation of ODE systems. Another released module (http://fisica.uc.pt/data/20032004/apontamentos/apnt_013_32.txt) presents 4th-order Runge-Kutta method for both single ODE equation and a system of ODEs. The 4th-order Runge-Kutta solver for a system of ODEs allows for individual ODE equations to be implemented as function, followed by consolidating the functions into a Python list. This is identical to the improved solution described above. However, other solvers implemented in http://fisica.uc.pt/data/20032004/apontamentos/apnt_013_32.txt, such as Euler method, only cater to single ODE equation and do not have the equivalent means for solving a system of ODEs.

This manuscript build on these ODE solvers to cater for a system of ODEs implemented as individual functions (the improved solution). Three enhancements will be added. Firstly, the solvers will be implemented as generators. ODE solvers are essentially iterative function and a naïve implementation is using an evaluation loop for equations, storing results from each iteration, followed by returning the stored results to the calling function. This will not be able to cater to virtually infinite simulations, such as the simulation of geomagnetic field reversal [6]. A generator will enable a stream of intermediate results for analysis. Secondly, the solvers will allow for non-ODE-bounded variables or solution vector. Commonly, there will be an ODE for each variable; thus, the number of variables or solutions and ODEs are the same. This will require all equation parameters needed in the ODEs to be implemented as a data structure or as separate global variables. By allowing for non-ODE-bounded variables will provide a means to place all equation parameters and ODE solutions together, which is likely to improve code maintainability. Lastly, a means to set upper and lower boundary of ODE solutions will be added. This study re-works 11 fixed time-step ODE solvers to allow for a system of ODEs to be implemented as a set of Python functions.

2. Experimental Details

Eleven fixed time-step ODE solvers were implemented in this study; namely,

1. Euler method,
2. Heun's method,
3. 3rd order Runge-Kutta method (RK3),
4. 4th order Runge-Kutta method (RK4),
5. 4th order Runge-Kutta method with 3/8 rule (RK4 (3/8)),
6. 4th Runge-Kutta-Fehlberg method (RKF4),
7. 5th order Runge-Kutta-Fehlberg method (RKF5),
8. 4th order Cash-Karp [7] method (CK4),
9. 5th order Cash-Karp [7] method (CK5),
10. 4th order Dormand-Prince [8] method (DP4), and
11. 5th order Dormand-Prince [8] method (DP5).

Implementation. All ODE solvers are first implemented, based on publically available Butcher's tableaus, for solving single ODE equation. Each solver is then improved to cater for a system of ODEs by implementing a loop around the initial solver. For example, the single ODE version of RK4 can be implemented as

```
f1 = func(x, y)
f2 = func(x+0.5*step, y+0.5*step*f1)
f3 = func(x+0.5*step, y+0.5*step*f2)
f4 = func(x+step, y+step*f3)
x = x + step
y = y + step*(f1+2.0*f2+2.0*f3+f4)/6.0
```

which can be easily improved for solving a system of ODEs as shown,

```
for i in range(n):
    f1[i] = func[i](x, y)
for j in range(n):
    y1[j] = y[j] + (0.5*step*f1[j])
for i in range(n):
    f2[i] = func[i]((x+(0.5*step)), y1)
for j in range(n):
    y1[j] = y[j] + (0.5*step*f2[j])
for i in range(n):
    f3[i] = func[i]((x+(0.5*step)), y1)
for j in range(n):
    y1[j] = y[j] + (step*f3[j])
for i in range(n):
    f4[i] = func[i]((x+step), y1)
x = x + step
for i in range(n):
    y1[i] = y[i] + (step * \
            (f1[i] + (2.0*f2[i]) + \
            (2.0*f3[i]) + f4[i]) / 6.0)
```

Python programming language allows for easy conversion of a function into a generator using `yield` keyword, which satisfied the first enhancement. The second enhancement for non-ODE-bounded variables will result in an error, as there will not be equivalent ODE for the variables. This can be resolved by wrapping each operation with exception handling. The third enhancement of upper and lower boundaries is implemented by a checking function, which checks and value-bound each variable after each time-step.

Testing. Each ODE solver undergoes three rounds of testing. Firstly, the system-capable ODE solver is compared against the single ODE solver counterpart. A standard one-process radioactive decay [9], which consists of only one equation, is used. It is expected that the system-capable ODE solver yield the same result as the single ODE solver counterpart. Secondly, results from system-capable ODE solvers are compared to the analytical solution for one-process radioactive decay. Lastly, using the 3-equation ODE to model zombie invasion [4] , the results from system-capable ODE solvers are compared to that of SciPy's default ODE solver.

3. Results and Discussion

SciPy [2] can be considered as a *de-facto* scientific and numerical library in Python programming. However, SciPy does not allow a system of ODEs to be implemented as a set of functions, which improves documentation and maintainability. In this study, 11 ODE solvers are developed to enable a system of ODEs to be implemented as a set of functions. The implemented solvers are incorporated into COPADS repository (https://github.com/copads/copads).

Functions-enabled ODE solvers give comparable results to SciPy's default ODE solver. By comparing the solutions from system-capable ODE solver is compared against the single ODE solver counterpart, the results suggest that solutions from system-capable ODE solver is identical against the single ODE solver counterpart for each of the 11 ODE solvers (correlation = 0, sum of % error across 500 time-steps = 0%).

Comparing between the solutions from system-capable ODE solver and analytical equation for one-process radioactive decay, the results show that Euler method has the largest error (Table 1). This is expected since Euler method is the oldest and simplest method for numerical integration, which is used to derive higher order method with higher accuracy [10]. This suggests that Euler method is the least accurate of all numerical integration methods, which is supported by this study. However, despite its inaccuracy, solutions from Euler method to the 3-equation ODE to model zombie invasion [4] show consistency (Table 1, average correlation = 0.9469) with the solutions

from SciPy [2]. The results show that the other 10 ODE solvers show comparable solutions (average correlation > 0.998) to that of SciPy [2]. The average correlation from 3-equation ODE to model zombie invasion [4] is perfectly correlated (correlation = 1.0) to the sum of errors between ODE and analytical solutions. Given that SciPy [2] is the *de-facto* scientific and numerical library in Python programming, which can be used as the gold standard, these results suggest that the errors are likely due to floating point errors rather than implementation errors.

Table 1: Testing of System-Enabled ODE Solvers. Sum of %Errors between ODE and analytical solution is calculated as the summation of %Errors across 500 time-steps.

Solver	Sum of % Error (ODE vs Analytical)	Average Correlation (n=3) with Output from SciPy's Default ODE Solver
Euler	2455.1818	0.94693390
Heun	16.95633237	0.99841780
RK3	8.48470E-02	0.99991199
RK4	3.39614E-04	0.99996421
RK4 (3/8)	3.39614E-04	0.99996426
RKF4	5.35594E-05	0.99996847
RKF5	7.37329E-07	0.99996802
CK4	1.19722E-07	0.99996823
CK5	1235.207898	0.99996816
DP4	3.30892E-05	0.99996853
DP5	2.30707E-07	0.99996825

Non-ODE-bounded Solution / Variable Vector. It is common to keep equation parameters and variable (ODE solutions) separate. However, this is usually a prescriptive rule rather than suggested implementation, as most ODE solvers will require an ODE for each of the variable.

Consider the case whereby the same system of ODEs is being studied using different sets of equation parameters, which is common to examine the effects of parameters on the solutions, also known as sensitivity analysis [11, 12]. When large numbers of combinations of equation parameters are being examined, documentation becomes increasingly difficult, as the equation parameters are not coupled with the ODE solutions. Hence, it may be useful in this case to have the equation parameters as part of the entire vector of ODE solutions. A means to achieve this will be to keep equation parameters as part of the solution vector, which requires the ODE solver to cater for "solutions" (which are actually equation parameters) that do not have ODE attached. An enhancement is made to the 11 ODE solvers in this study to cater for variables without ODE attached for use as equation parameters.

For example, to the 3-equation ODE to model zombie invasion [4],

```
import ode

# birth rate
birth = 0
# natural death percent (per day)
death = 0.0001
# transmission percent  (per day)
transmission = 0.0095
# resurect percent (per day)
resurect = 0.0001
# destroy percent  (per day)
destroy = 0.0001

def human(t, y):
    infected = transmission * y[0] * y[1]
    dead = death * y[0]
    return birth - infected - dead
def zombie(t, y):
    newly_infected = transmission * \
        y[0] * y[1]
    resurrected = resurrect * y[2]
    destroyed = destroy * y[0] * y[1]
    return newly_infected + resurrected - \
           destroyed
def dead(t, y):
    natural_death = death * y[0]
    destroyed_zombies = destroy * \
        y[0] * y[1]
    created_zombies = resurrect * y[2]
    return natural_death + \
           destroyed_zombies - \
           created_zombies

# system of ODEs
f = [human, zombie, dead]

# initial human, zombie, death population
# respectively
y = [500.0, 0, 0]
print('''Solving using 5th order Dormand-
Prince method ......''')
for i in [x for x in
          ode.DP5(f, 0.0, y, 0.1, 50.0)]:
    print(','.join([str(z) for z in i]))
```

can be implemented as

```
import ode

def human(t, y):
    infected = y[5] * y[0] * y[1]
    dead = y[4] * y[0]
    return y[3] - infected - dead
def zombie(t, y):
    newly_infected = y[5] * y[0] * y[1]
    resurrected = y[6] * y[2]
    destroyed = y[7] * y[0] * y[1]
    return newly_infected + resurrected - \
           destroyed
def dead(t, y):
```

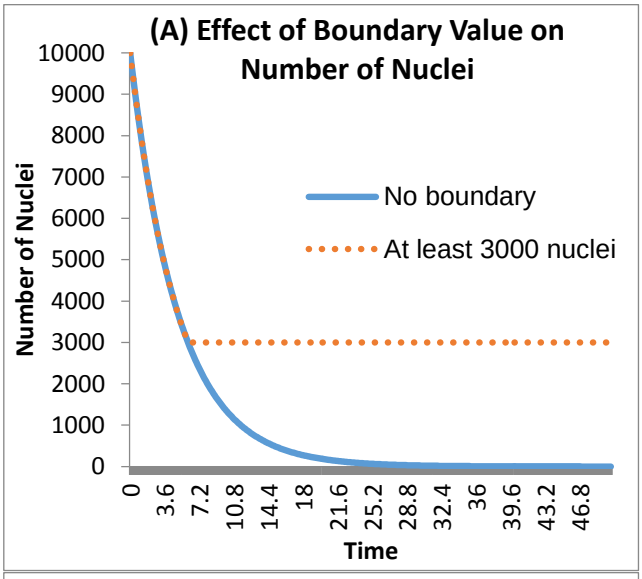
```
    natural_death = y[4] * y[0]
    destroyed_zombies = y[7] * y[0] * y[1]
    created_zombies = y[6] * y[2]
    return natural_death + \
           destroyed_zombies - \
           created_zombies

f = range(8)
f[0] = human
f[1] = zombie
f[2] = dead

y = range(8)
y[0] = 500.0 # initial human population
y[1] = 0.0   # initial zombie population
y[2] = 0.0   # initial death population
y[3] = 0     # birth rate
y[4] = 0.0001 # natural death percent / day
y[5] = 0.0095 # transmission percent  / day
y[6] = 0.0001  # resurrect percent / day
y[7] = 0.0001  # destroy percent / day

print('''Solving using 5th order Dormand-
Prince method ......''')
for i in [x for x in
          ode.DP5(f, 0.0, y, 0.1, 50.0)]:
    print(','.join([str(z) for z in i]))
```

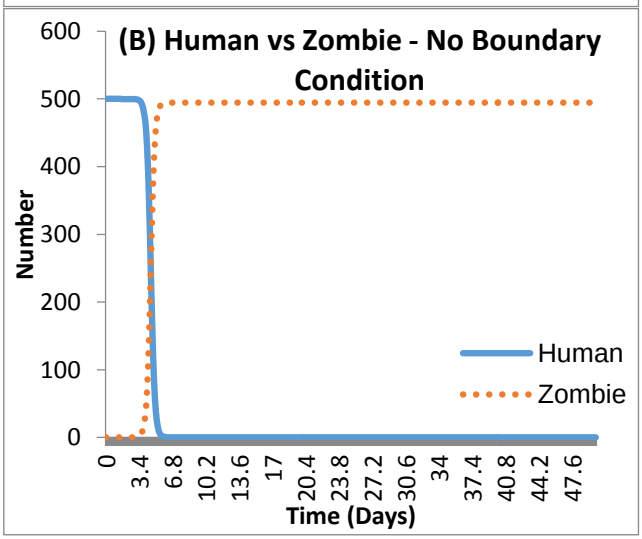
The solution vector will be an array of 500 time-steps of 9 data elements, inclusive of time as the first element. However, only the 2nd, 3rd, and 4th data elements are the actual ODE solutions, which are number of humans, zombies, and dead respectively. The last 4 data elements are the equation parameters. These equation parameters will not change throughout the simulation, as there are no ODEs attached to change these parameters. Although this will result in a larger result file, if saved, due to repeated equation parameters; there can be no error as to the set of equation parameters is used to generate the corresponding ODE solutions. Hence, the saved result file is self-documenting, which improves results documentation.

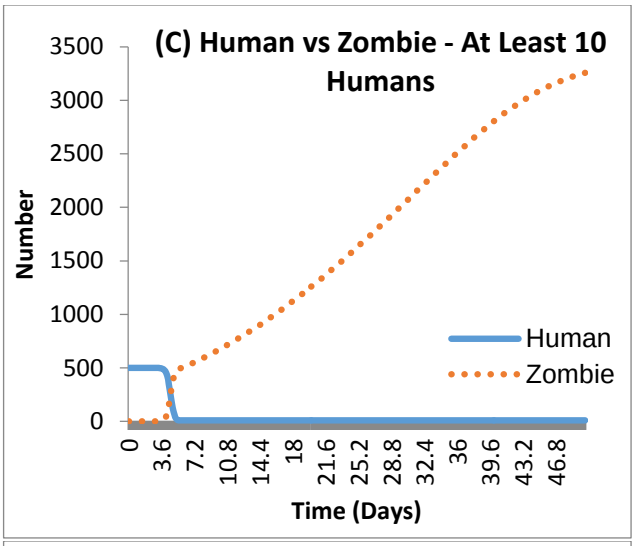
Upper and/or Lower Boundaries. Establishing boundaries for ODE solutions is regularly seen [13, 14]; hence, it will be useful to have this feature in the implemented solvers. Upper and lower boundaries are implemented separately, as different parameters to the ODE solver. This allows for ODEs to have either upper or lower boundaries or both. Moreover, boundaries are implemented as non-mandatory options in the solvers. This allow for maximum flexibility.

The boundaries are passed to the solver as a Python dictionary where the key is the array position of the ODE, which also corresponds to the position in the solution vector; and the value is a 2-element list of boundary value and reset. For example, lower boundary dictionary, `{2: [5.0, 5.1]}`, will reset the solution of the third ODE (Python language uses zero-based list index numbering) to 5.1 when the solution is lower than 5.0.

A lower boundary of at least 3000 radioactive nuclei is compared against that without boundary. The result show that the number of radioactive nuclei does not drop below 3000 when lower boundary is used (Figure 1A), suggesting that the implementation is functional.







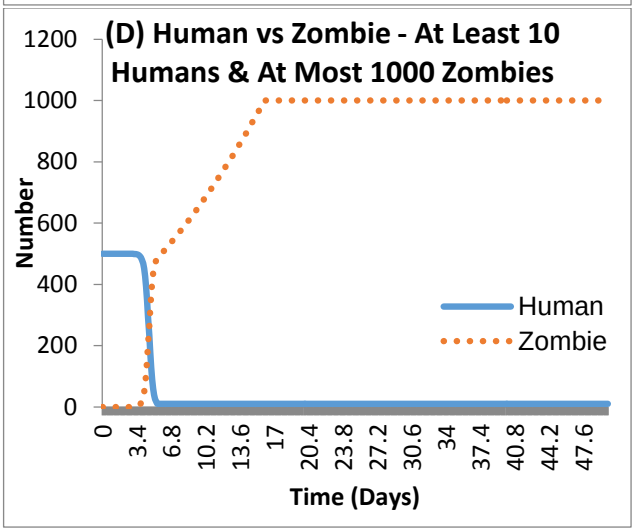


Fig.1 Effects of Boundaries on ODE Systems.

An additional example shows the 3-equation model of zombie invasion [4] with no boundary, lower boundary to have at least 10 humans, and both upper (maximum of 1000 zombies) and lower (at least 10 humans) boundaries:

```
import ode

birth = 0       # birth rate
# natural death percent / day
death = 0.0001
# transmission percent / day
transmission = 0.0095
# resurrect percent / day
resurect = 0.0001
destroy = 0.0001 # destroy percent / day
```

```
def human(t, y):
    infected = transmission*y[0]*y[1]
    dead = death*y[0]
    return birth - infected - dead
def zombie(t, y):
    newly_infected = transmission*y[0]*y[1]
    resurrected = resurect*y[2]
    destroyed = destroy*y[0]*y[1]
    return newly_infected + resurrected - \
           destroyed
def dead(t, y):
    natural_death = death*y[0]
    destroyed_zombies = destroy*y[0]*y[1]
    created_zombies = resurect*y[2]
    return natural_death + \
           destroyed_zombies - \
           created_zombies

# system of ODEs
f = [human, zombie, dead]

# initial human, zombie, death population
# respectively
y = [500.0, 0, 0]
lower_bound = {0: [10.0, 10.0]}
upper_bound = {1: [1000.0, 1000.0]}

print('''Solving using 5th order Dormand-
Prince method ......''')
nobound = [x for x in
           ode.DP5(f, 0.0, y, 0.1, 50.0)]
lowerbound = [x for x in
              ode.DP5(f, 0.0, y, 0.1, 50.0,
                      lower_bound)]
doublebound = [x for x in
               ode.DP5(f, 0.0, y, 0.1, 50.0,
               lower_bound, upper_bound)]
for i in range(len(nobound)):
    consolidated = nobound[i] + \
                   lowerbound[i][1:] + \
                    doublebound[i][1:]
    print ','.join([str(x)
                    for x in consolidated])
```

Initial human population limits the final zombie population in no boundary scenario (Figure 1B), as birth rate is set to zero. This is expected as human is the only source for creating more zombies. Once the human population collapses, there is no additional human to be infected; hence, the zombie population stagnates. When the number of human is set to at least 10, the number of zombies increases to nearly 3500 before showing signs of plateauing (Figure 1C). The increase in zombie population can be attributed to a constant lower limit of human population to be infected, suggesting that the dynamics of the system can be significantly different with non-zero lower boundary. Similarly, both upper and lower boundaries can be established (Figure 1D) to bring an artificial limit to the number of zombies. Nevertheless, by comparing the results between with and without

boundar(ies) suggests that boundary implementation is functional.

4. Conclusion

Eleven ODE solvers are implemented using Python programming language in this study, which allows for a system of ODEs to be implemented as a set of Python functions. This allow for better code documentation and maintenance. Three enhancements are implemented to improve the functionality and flexibility of the solvers. The implemented solvers are incorporated into COPADS repository (https://github.com/copads/copads).

Acknowledgement

The author will like to thank HJ Wang (Nanyang Technological University, Singapore) for her discussion and comments on the initial drafts.

References

[1] C. C. Chicone, "Ordinary Differential Equations with Applications", Springer, 1999.
[2] T. E. Oliphant, "Python for Scientific Computing", Computing in Science & Engineering, 9, 2007, 10-20.
[3] H. P. Langtangen and L. Wang, "Odespy software package", 2014. URL: https://github.com/hplgit/odespy.
[4] P. Munz, I. Hudea, J. Imad and R. J. Smith, "When Zombies Attack!: Mathematical Modelling of Outbreak of Zombie Infection", Infectious Disease Modelling Research Progress, 4, 2009, 133-150.
[5] J. Senning, "www.math-cs.gordon.edu/courses/ma342/python/diffeq.py", 2008.
[6] G. A. Glatzmaiers and P. H. Roberts, "A Three-Dimensional Self-Consistent Computer Simulation of a Geomagnetic Field Reversal", Nature, 377, 1995, 203-209.
[7] J. R. Cash and A. H. Karp, "A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides", ACM Transactions on Mathematical Software, 16, 1990, 201-222.
[8] J.R. Dormand and P. J Prince, "A Family of Embedded Runge-Kutta Formulae", Journal of Computational and Applied Mathematics, 6, 1980, 19–26.
[9] S. B. Patel, "Nuclear Physics: An Introduction", New Delhi: New Age International, 2000.
[10] Z. Memon, S. Qureshi, A. A. Shaikh and M. S. Chandio, "A Modified ODE Solver for Autonomous Initial Value Problems", Mathematical Theory and Modeling, 4, 2014, 80-85.
[11] G. Bao and H. Zhang, "Sensitivity Analysis of an Inverse Problem for the Wave Equation with Caustics", Journal of the American Mathematical Society, 27, 2014, 953-981.
[12] Z. Zi, "Sensitivity Approached Applied to System Biology Models", IET Systems Biology, 5, 2011, 336-346.
[13] F. Brauer, "Bounds for Solutions of Ordinary Differential Equations", Proceedings of the American Mathematical Society, 14, 1963, 36-43.
[14] J. K. Scott and P. I. Barton, "Improved Relaxations for Parametric Solutions of ODEs using Differential Inequalities", Journal of Global Optimization, 57, 2013, 143-176.