Citation: Ling, MHT. 2018. **COPADS VI: Fixed Time-Step ODE Solvers with Mixed ODE and non-ODE Function, and Script Generator.** In Current STEM, Volume 1, pp. 173-212. Nova Science Publishers, Inc. ISBN 978-1-53613-416-2.

COPADS VI: FIXED TIME-STEP ODE SOLVERS WITH MIXED ODE AND NON-ODE FUNCTION, AND SCRIPT GENERATOR

Maurice HT Ling*
Colossus Technologies LLP, Singapore

ABSTRACT

Ordinary differential equations (ODEs) are commonly used in mathematical modelling. However, the standard means of implementing a system of ODEs in Python, using Scipy, does not allow for each ODE to be implemented as an individual Python function. This results in poor documentation and maintainability. We have re-implemented 11 fixed-steps ODE solvers to allow for an ODE system to be implemented as a set of Python functions. Here, the ODE solvers are enhanced to take on a non-ODE function, allowing for modification of the ODE result vector at each time step, which may be useful in cases where one or more results has to be calibrated using other results. In addition, a script generator is implemented to assist in the generation of a Python ODE script from a set of parameters. This module had been incorporated into the Collection of Python Algorithms and Data Structures (COPADS; https://github.com/mauriceling/copads), under Python Software Foundation License version 2.

Keywords: Fixed time step, ODE solver, Modifying function.

Date Submitted: November 9, 2017. **Date Accepted:** December 15, 2017.

* Corresponding Author address
Email: mauriceling@acm.org, mauriceling@colossus-tech.com

INTRODUCTION

Mathematical modelling is an important aspect of science (Otto and Day, 2007) and differential equation, especially ordinarily differential equation (ODE), is one of the two main paradigms in mathematical modelling (Ling, 2016a); the other being state transition modelling (Chay et al., 2016). As previously described (Ling, 2016b), the standard way of writing ODE systems in Python is via the use of SciPy (Oliphant, 2007) or Odespy (Langtangen and Wang, 2014) to implement one or more ODEs within a single function. Using a simple 3-equation ODE to model zombie invasion (Munz et al., 2009) 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 (Oliphant, 2007) as shown above present two issues. Firstly, it is common to have an ODE system comprising of many individual ODEs. This will result in function codes spanning multiple lines and possibly across multiple pages, which is difficult to read. Secondly, it does not allow for effective documentation – both at the equation level and the term (sub-equation) level. 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.

The ideal solution is to allow users to implement each ODE as a separate function. Besides being able to separate each term within an ODE into meaningful variable names (resulting in self-documenting code), this also allows 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 then 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. (2009) 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).
        Resurrected 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 solution may be using functions within a function as shown,

```
    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]
```

However, this results in type error in SciPy; hence, not suitable. Hence, a set of ODE solvers, which allows for each ODE to be implemented as a separate Python function (the ideal solution), has been implemented (Ling, 2016b).

This article documents the code implementation of the ODE solvers in Ling (2016b). At the same time, the ODE solvers are enhanced to take on a non-ODE function (see Appendix A for example code). This allows for modification of the ODE result vector at each time step, which may be useful in cases where one or more results has to be calibrated using other results. In addition, a script generator is implemented to assist in the generation of a Python ODE script from a set of parameters. This module had been incorporated into the Collection of Python Algorithms and Data Structures (COPADS; https://github.com/mauriceling/copads), under Python Software Foundation License version 2.

CODE DESCRIPTION

Eleven fixed time-step ODE solvers were implemented has been implemented in Ling (2016b); 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 (Cash and Karp, 1990) method (CK4),
9. 5th order Cash-Karp (Cash and Karp, 1990) method (CK5),
10. 4th order Dormand-Prince (Dormand and Prince, 1980) method (DP4), and
11. 5th order Dormand-Prince (Dormand and Prince, 1980) method (DP5).

Each ODE solver implemented in Ling (2016b) consists of a ODE solver core function (known as `solver` function) and a driver, where the ODE solver core function is implemented as a sub-function within the driver function. The ODE

solver core function is implemented from the respective Butcher Tableaues (Schober et al., 2014) of the ODE solvers. On the other hand, the driver function of each solver (named as their own solver names, such as `Heun` for Heun's method) is identical for all solvers as the following:

```
driver_function(funcs, x0, y0, step, xmax, nonODEfunc,
                lower_bound, upper_bound,
                overflow, zerodivision):
    yield [x0] + y0
    while x0 < xmax:
        y1 = solver(funcs, x0, y0, step)
        y1 = nonODEfunc(y1, step)
        y1 = boundary_checker(y1, lower_bound, 'lower')
        y1 = boundary_checker(y1, upper_bound, 'upper')
        y0 = y1
        x0 = x0 + step
        yield [x0] + y0
```

where

- funcs is system of differential equations in a Python list
- x0 is the initial value of x-axis, which is usually starting time
- y0 is a vector of initial float values for variables
- step is the step size on the x-axis (also known as step in calculus)
- xmax is the maximum value of x-axis, which is usually ending time
- nonODEfunc is a non-ODE Python function to modify the variable list (y0)
- lower_bound is set of values, as a Python dictionary, for lower boundary of variables (see Appendix B for example codes)
- upper_bound is set of values, as a Python dictionary for upper boundary of variables (see Appendix B for example codes)
- overflow is a value (usually a large value) to assign in event of over flow error (usually caused by a large number) during integration
- zerodivision is value (usually a large value) to assign in event of zero division error, which results in positive infinity, during integration

In essence, the solver core function performs a single time step solution. The result vector then underwent 3 post-processing steps. Firstly, a user-defined function (known as `nonODEfunc` function; see Appendix A for example code) will be executed. This function will not be part of the ODE system of equations or functions; hence, can be included to modify the result vector. For example, this function can be used to modify a specific value in the result vector with respect to other values in the result vector. Secondly, the values in the result vector are lower

bounded as values below the lower bound will be reset to an lower bound value. Lastly, the values in the result vector are upper bounded as values above the upper bound will be reset to an upper bound value. In both cases of the upper and lower bounding, each value in the result vector can be set to a different upper and/or lower bound value and the respective resetting value.

In order for each ODE to be used and substituted in various codes as seamlessly as possible, each driver function uses the same set of parameters. In addition, each driver function is implemented as a generator function to cater for long or virtually infinite simulation, which simulation results can be generated as a data stream for continual processing.

Ling (2016b) also demonstrated that self-documenting ODE solution results can be achieved by non-ODE-bounded variables. That is, one or more parameters be implemented within the variable vector; thereby, resulting in more elements in the variable vector compared to the number of ODE equations (see Appendix C for example code). This will result in the parameters being "printed" along with the simulation results; thus, resulting in self-documentation as the ODE function becomes an operational scaffold. Using non-ODE-bounded variables can be useful in event whereby many sets of simulation results are to be generated, such as in the case of sensitivity analysis (Bao and Zhang, 2014).

Another enhancement to the ODE system is the ODE script generator. A system of ODE can be specified as a Python dictionary where each key-value pair represents an ODE. The key is the name of the ODE function while the value is a Python list where term in the ODE is an element in the list. For example, the following ODE to describe an enzymatic reaction: $A \xrightarrow{enzyme} P$, where molecule A (the reactant) is converted into molecule P (the product) via an enzyme as the catalyst. The formation of molecule P can be modelled as the following ODE,

$$\frac{d\,P}{dt} = (7 * enzyme * A) - (1e^{-6} * P)$$

which can be specified as

```
{'P': ['7 * enzyme * A', '- (1e-6 * P)']}
```

and this will generate the following Python function suitable for ODE solver

```
def P(t, y):
    exp_1 = (7 * y[0] * y[1])
    exp_2 = -(1e-6 * y[2])
    return exp_1 + exp_2
```

where `y[0]`, `y[1]`, and `y[2]` represent the concentration of enzyme, A, and P, respectively.

Hence, the availability of this script generator will allow for the generation of an executable Python ODE script file from the textual specification of a system of ODEs. This will enable ODE scripts to be generated during runtime.

As an example, given the following parameters:

- Rate of new births ($rate_{birth}$) = 0 % / day
- Rate of "zombie virus" transmission ($rate_{transmission}$) = 0.95 % / day
- Rate of natural human deaths ($rate_{death}$) = 0.01 % / day
- Rate of zombies resurrected from dead ($rate_{resurrect}$) = 0.02 % / day
- Rate of zombies destroyed by humans ($rate_{destroy}$) = 0.03 % / day
- Influx of 5 new humans per day ($rate_{influx}$)
- Maximum number of zombies = 700
- Minimum number of humans = 0

A system of ODEs describing zombie apocalypse (Munz et al., 2009) can be written as,

$$\begin{aligned}\frac{d\,human}{dt} &= influx + birth - zombied - death \\ &= \left(rate_{influx} + rate_{birth}\right) - \left(rate_{transmit} \times human \times zombie\right) \\ &\qquad - \left(rate_{death} \times human\right) \\ &= (5.0 + 0.0) - (0.0095 \times human \times zombie) - (0.0001 \times human)\end{aligned}$$

$$\begin{aligned}\frac{d\,zombie}{dt} &= zombied + resurrected - destroyed \\ &= \left(rate_{transmit} \times human \times zombie\right) + \left(rate_{resurrect} \times dead\right) \\ &\qquad - \left(rate_{destroy} \times human \times zombie\right) \\ &= (0.0095 \times human \times zombie) + (0.0002 \times dead) \\ &\qquad - (0.0003 \times human \times zombie)\end{aligned}$$

$$\begin{aligned}\frac{d\,dead}{dt} &= death + destroyed - resurrected \\ &= \left(rate_{death} \times human\right) + \left(rate_{destroy} \times human \times zombie\right) \\ &\qquad - \left(rate_{resurrect} \times dead\right) \\ &= (0.0001 \times human) + (0.0003 \times human \times zombie) - (0.0002 \times dead)\end{aligned}$$

can be specified in the following format (file name = zombie_construct.py),

```
from ode import ODE_constructor

scriptfile = 'zombie_attack.py'
resultsfile = 'zombie_data.csv'
```

```
time = (0.0, 0.1, 100.0)
ODE_solver = 'RK4'
expressions = {'human': ['birth_rate',
                         '- (transmission_rate * human * zombie)',
                         '- (death_rate * human)'],
               'zombie': ['(transmission_rate * human *
                           zombie)',
                         '(resurrection_rate * dead)',
                         '- (destroy_rate * human * zombie)'],
               'dead': ['(death_rate * human)',
                        '(destroy_rate * human * zombie)',
                        '- (resurrection_rate * dead)']}
parameters = {'birth_rate': 0.0,
              'transmission_rate': 0.0095,
              'death_rate': 0.0001,
              'resurrection_rate': 0.0002,
              'destroy_rate':0.0003}
initial_conditions = {'human': 500.0,
                      'zombie': 0.0,
                      'dead': 0.0}
modifying_expressions = ['human = human + (5*step)']
lower_bound = {'human': [0.0, 0.0]}
upper_bound = {'zombie': [700, 700]}
overflow = 1e100
zerodivision = 1e100

s = ODE_constructor(scriptfile, resultsfile, time, ODE_solver,
                    expressions, parameters,
                    initial_conditions,
                    modifying_expressions,
                    lower_bound, upper_bound,
                    overflow, zerodivision)
```

for simulating an initial population of 500 humans with no zombies (hence, initial "zombification" is spontaneous) over 100 days. The above specification will generate the following Python script file (file name = zombie_attack.py) when executed,

```
import ode

y = range(3)
y[0] = 0.0
y[1] = 0.0
y[2] = 500.0


def zombie(t, y):
    exp_1 = (0.0095 * y[2] * y[0])
    exp_2 = (0.0002 * y[1])
```

```
    exp_3 = - (0.0003 * y[2] * y[0])
    return exp_1 + exp_2 + exp_3
def dead(t, y):
    exp_1 = (0.0001 * y[2])
    exp_2 = (0.0003 * y[2] * y[0])
    exp_3 = - (0.0002 * y[1])
    return exp_1 + exp_2 + exp_3
def human(t, y):
    exp_1 = 0.0
    exp_2 = - (0.0095 * y[2] * y[0])
    exp_3 = - (0.0001 * y[2])
    return exp_1 + exp_2 + exp_3


def modifying_expression(y, step):
    y[2] = y[2] + (5*step)
    return y

ODE = range(3)
ODE[0] = zombie
ODE[1] = dead
ODE[2] = human

stime = 0.0
step = 0.1
etime = 100.0
lower_bound = {'2': [0.0, 0.0]}
upper_bound = {'0': [700, 700]}
overflow = 1e+100
zerodivision = 1e+100

f = open('zombie_data.csv', 'w')
f.write('time,zombie,dead,human' + '\n')

for x in ode.RK4(ODE, stime, y, step, etime,
        modifying_expression,
        lower_bound, upper_bound,
        overflow, zerodivision):
    f.write(','.join([str(item) for item in x]) + '\n')
f.close()
```

The above generated code is logically identical to the manually written ODE code script and produces identical simulation results,

```
import ode

birth = 0               # birth rate
death = 0.0001          # natural death percent (per day)
transmission = 0.0095   # transmission percent  (per day)
resurect = 0.0002       # resurect percent (per day)
```

```
destroy = 0.0003        # destroy percent (per 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

def modifying_expression(y, step):
    y[0] = y[0] + (5 * step)
    return y

lower_bound = {'2': [0.0, 0.0]}
upper_bound = {'0': [700, 700]}

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

y = [500.0, 0, 0]  # initial human, zombie, death population
print('Solving using 4th order Runge Kutta method ......')
for i in [x for x in ode.RK4(f, 0.0, y, 0.1, 100.0,
          modifying_expression, lower_bound, upper_bound)]:
    print(','.join([str(z) for z in i]))
print('')
```

CODE FILE FOR ODE SOLVER (ODE.PY)

```
'''
Ordinary Differential Equation (ODE) Solvers.
Copyright (c) Maurice H.T. Ling <mauriceling@acm.org>
Date created: 20th December 2014'''
import re

def boundary_checker(y, boundary, type):
    '''Private function - called by ODE solvers to perform
    boundary checking of variable values and reset them to
    specific values if the original values fall out of the
    boundary values.

    Boundary parameter takes the form of a dictionary with
    variable number as key and a list of [<boundary value>,
```

```
        <value to set if boundary is exceeded>]. For example, the
        following dictionary for lower boundary (type = 'lower')
        {'1': [0.0, 0.0], '5': [2.0, 2.0]} will set the lower
        boundary of variable y[0] and [5] to 0.0 and 2.0
        respectively. This also allows for setting to a different
        value - for example, {'1': [0.0, 1.0]} will set variable
        y[0] to 2.0 if the original y[0] value is negative.

        @param y: values for variables
        @type y: list
        @param boundary: set of values for boundary of variables
        @type boundary: dictionary
        @param type: the type of boundary to be checked, either
        'upper' (upper boundary) or 'lower' (lower boundary)
        '''
        for k in list(boundary.keys()):
            if y[int(k)] < boundary[k][0] and type == 'lower':
                y[int(k)] = boundary[k][1]
            if y[int(k)] > boundary[k][0] and type == 'upper':
                y[int(k)] = boundary[k][1]
        return y

    def Euler(funcs, x0, y0, step, xmax, nonODEfunc=None,
              lower_bound=None, upper_bound=None,
              overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y0' = funcs(x0,
        y0), using Euler method.

        A function (as nonODEfunc parameter) can be included to
        modify one or more variables (y0 list). This function will
        not be an ODE (not a dy/dt). This can be used to
        consolidate the modification of one or more variables at
        each ODE solving step. For example, y[0] = y[1] / y[2]
        can be written as

        C{
        def modifying_function(y, step):
            y[0] = y[1] / y[2]
            return y
        }

        This function must take 'y' (variable list) and 'step'
        (time step) as parameters and must return 'y' (the modified
        variable list). This function will execute before boundary
        checking at each time step.

        Upper and lower boundaries of one or more variable can be
        set using upper_bound and lower_bound parameters
        respectively. These parameters takes the form of a
        dictionary with variable number as key and a list
```

```
of [<boundary value>, <value to set if boundary is
exceeded>]. For example, the following dictionary for
lower boundary {'1': [0.0, 0.0], '5': [2.0, 2.0]} will set
the lower boundary of variable y[0] and y[5] to 0.0 and
2.0 respectively. This also allows for setting to a
different value - for example, {'1': [0.0, 1.0]} will set
variable y[0] to 2.0 if the original y[0] value is
negative.

@param funcs: system of differential equations
@type funcs: list
@param x0: initial value of x-axis, which is usually
starting time
@type x0: float
@param y0: initial values for variables
@type y0: list
@param step: step size on the x-axis (also known as step
in calculus)
@type step: float
@param xmax: maximum value of x-axis, which is usually
ending time
@type xmax: float
@param nonODEfunc: a function to modify the variable list
(y0)
@type nonODEfunc: function
@param lower_bound: set of values for lower boundary of
variables
@type lower_bound: dictionary
@param upper_bound: set of values for upper boundary of
variables
@type upper_bound: dictionary
@param overflow: value (usually a large value) to assign
in event of over flow error (usually caused by a large
number) during integration. Default = 1e100.
@type overflow: float
@param zerodivision: value (usually a large value) to
assign in event of zero division error, which results in
positive infinity, during
integration. Default = 1e100.
@type zerodivision: float
'''
yield [x0] + y0
def solver(funcs, x0, y0, step):
    n = len(funcs)
    y1 = [0]*n
    for i in range(n):
        try: y1[i] = y0[i] + (step*funcs[i](x0, y0))
        except TypeError: pass
        except ZeroDivisionError: y0[i] = zerodivision
        except OverflowError: y0[i] = overflow
```

```
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def Heun(funcs, x0, y0, step, xmax, nonODEfunc=None,
             lower_bound=None, upper_bound=None,
             overflow=1e100, zerodivision=1e100):
        '''
        Generator to integrate a system of ODEs, y0' = funcs(x0,
        y0), using Heun's method, which is also known as Runge-
        Kutta 2nd method or Trapezoidal method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1 = [0]*n
            y1, y2 = [0]*n, [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + step*f1[i]
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y2[i] = overflow
            for i in range(n):
                try: y2[i] = y0[i] + 0.5*step*(f1[i] + \
                             funcs[i](x0+step, y1))
                except TypeError: pass
                except ZeroDivisionError: y2[i] = zerodivision
                except OverflowError: y2[i] = overflow
            return y2
        while x0 < xmax:
            y2 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
```

```
                y2 = boundary_checker(y2, lower_bound, 'lower')
            if upper_bound:
                y2 = boundary_checker(y2, upper_bound, 'upper')
            y0 = y2
            x0 = x0 + step
            yield [x0] + y0

    def RK3(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y0' = funcs(x0,
        y0), using third order Runge-Kutta method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3 = [0]*n, [0]*n, [0]*n
            y1, y2 = [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = step * funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for i in range(n):
                y1[i] = y0[i] + 0.5*f1[i]
            for i in range(n):
                try: f2[i] = step * funcs[i](x0 + 0.5*step, y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
                except OverflowError: f2[i] = overflow
            for i in range(n):
                y1[i] = y0[i] - f1[i] + 2*f2[i]
            for i in range(n):
                try: f3[i] = step * funcs[i](x0 + step, y1)
                except TypeError: pass
                except ZeroDivisionError: f3[i] = zerodivision
                except OverflowError: f3[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + (f1[i] + 4*f2[i] + f3[i])/6.0
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y1[i] = overflow
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
```

```
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def RK4(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fourth order Runge-Kutta method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3, f4 = [0]*n, [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.5*step*f1[j])
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(0.5*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
                except OverflowError: f2[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.5*step*f2[j])
            for i in range(n):
                try: f3[i] = funcs[i]((x0+(0.5*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f3[i] = zerodivision
                except OverflowError: f3[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (step*f3[j])
            for i in range(n):
                try: f4[i] = funcs[i]((x0+step), y1)
                except TypeError: pass
                except ZeroDivisionError: f4[i] = zerodivision
                except OverflowError: f4[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + (step * \
                        (f1[i] + (2.0*f2[i]) + (2.0*f3[i]) + \
```

```
                    f4[i]) / 6.0)
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y1[i] = overflow
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def RK38(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fourth order Runge-Kutta method, 3/8 rule.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3, f4 = [0]*n, [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (step*f1[j]/3.0)
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(step/3.0)), y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
                except OverflowError: f2[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (-1*step*f1[j]/3.0) + (step*f2[j])
            for i in range(n):
                try: f3[i] = funcs[i]((x0+(2*step/3.0)), y1)
                except TypeError: pass
                except ZeroDivisionError: f3[i] = zerodivision
                except OverflowError: f3[i] = overflow
            for j in range(n):
```

```
                y1[j] = y0[j] + (step*f1[j]) + (-step*f2[j]) + \
                        (step*f3[j])
            for i in range(n):
                try: f4[i] = funcs[i]((x0+step), y1)
                except TypeError: pass
                except ZeroDivisionError: f4[i] = zerodivision
                except OverflowError: f4[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + (step * \
                         (f1[i] + (3.0*f2[i]) + (3.0*f3[i]) + \
                         f4[i]) / 8.0)
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y1[i] = overflow
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def CK4(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fifth order Cash-Karp method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3, f4, f5, f6 = [0]*n, [0]*n, [0]*n, \
                                     [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.2*step*f1[j])
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(0.2*step)), y1)
```

```
            except TypeError: pass
            except ZeroDivisionError: f2[i] = zerodivision
            except OverflowError: f2[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (0.075*step*f1[j]) + \
                    (0.225*step*f2[j])
        for i in range(n):
            try: f3[i] = funcs[i]((x0+(0.3*step)), y1)
            except TypeError: pass
            except ZeroDivisionError: f3[i] = zerodivision
            except OverflowError: f3[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (0.3*step*f1[j]) + \
                    (-0.9*step*f2[j]) + (1.2*step*f3[j])
        for i in range(n):
            try: f4[i] = funcs[i]((x0+(0.6*step)), y1)
            except TypeError: pass
            except ZeroDivisionError: f4[i] = zerodivision
            except OverflowError: f4[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (-11*step*f1[j]/54.0) + \
               (2.5*step*f2[j]) + (-70*step*f3[j]/27.0) + \
               (35*step*f4[j]/27.0)
        for i in range(n):
            try: f5[i] = funcs[i](x0+step, y1)
            except TypeError: pass
            except ZeroDivisionError: f5[i] = zerodivision
            except OverflowError: f5[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (1631*step*f1[j]/55296) + \
               (175*step*f2[j]/512) + (575*step*f3[j]/13824)\
              + (44275*step*f4[j]/110592) + \
               (253*step*f5[j]/4096)
        for i in range(n):
            try: f6[i] = funcs[i](x0+(0.875*step), y1)
            except TypeError: pass
            except ZeroDivisionError: f6[i] = zerodivision
            except OverflowError: f6[i] = overflow
        for i in range(n):
            try: y1[i] = y0[i] + (step * \
                    ((2825*f1[i]/27648) + \
                     (18575*f3[i]/48384) + \
                     (13525*f4[i]/55296) + \
                     (277*f5[i]/14336) + \
                     (0.25*f6[i])))
            except TypeError: pass
            except ZeroDivisionError: y1[i] = zerodivision
            except OverflowError: y1[i] = overflow
        return y1
    while x0 < xmax:
```

```
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def CK5(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fifth order Cash-Karp method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3, f4, f5, f6 = [0]*n, [0]*n, [0]*n, \
                                     [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.2*step*f1[j])
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(0.2*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
                except OverflowError: f2[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.075*step*f1[j]) + \
                        (0.225*step*f2[j])
            for i in range(n):
                try: f3[i] = funcs[i]((x0+(0.3*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f3[i] = zerodivision
                except OverflowError: f3[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.3*step*f1[j]) + \
                    (-0.9*step*f2[j]) + (1.2*step*f3[j])
            for i in range(n):
                try: f4[i] = funcs[i]((x0+(0.6*step)), y1)
```

```
                except TypeError: pass
                except ZeroDivisionError: f4[i] = zerodivision
                except OverflowError: f4[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (-11*step*f1[j]/54.0) + \
                  (2.5*step*f2[j]) + (-70*step*f3[j]/27.0) + \
                  (35*step*f4[j]/27.0)
            for i in range(n):
                try: f5[i] = funcs[i](x0+step, y1)
                except TypeError: pass
                except ZeroDivisionError: f5[i] = zerodivision
                except OverflowError: f5[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (1631*step*f1[j]/55296.0) + \
                        (175*step*f2[j]/512.0) + \
                        (575*step*f3[j]/13824.0) + \
                        (44275*step*f4[j]/110592.0) + \
                        (253*step*f5[j]/4096.0)
            for i in range(n):
                try: f6[i] = funcs[i](x0+(0.875*step), y1)
                except TypeError: pass
                except ZeroDivisionError: f6[i] = zerodivision
                except OverflowError: f6[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + (step * \
                        ((37*f1[i]/378.0) + (250*f3[i]/621.0) + \
                         (125*f4[i]/594.0) + (512*f6[i]/1771.0)))
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y1[i] = overflow
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def RKF4(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fourth order Runge-Kutta_Fehlberg method.

        See Euler method documentation for description on
```

```
    parameters list.'''
    yield [x0] + y0
    def solver(funcs, x0, y0, step):
        n = len(funcs)
        f1, f2, f3, f4, f5, f6 = [0]*n, [0]*n, [0]*n, \
                                 [0]*n, [0]*n, [0]*n
        y1 = [0]*n
        for i in range(n):
            try: f1[i] = funcs[i](x0, y0)
            except TypeError: pass
            except ZeroDivisionError: f1[i] = zerodivision
            except OverflowError: f1[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (0.25*step*f1[j])
        for i in range(n):
            try: f2[i] = funcs[i]((x0+(0.25*step)), y1)
            except TypeError: pass
            except ZeroDivisionError: f2[i] = zerodivision
            except OverflowError: f2[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (3*step*f1[j]/32.0) + \
                    (9*step*f2[j]/32.0)
        for i in range(n):
            try: f3[i] = funcs[i]((x0+(3*step/8.0)), y1)
            except TypeError: pass
            except ZeroDivisionError: f3[i] = zerodivision
            except OverflowError: f3[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (1932*step*f1[j]/2197.0) + \
                    (-7200*step*f2[j]/2197.0) + \
                    (7296*step*f3[j]/2197.0)
        for i in range(n):
            try: f4[i] = funcs[i]((x0+(12*step/13.0)), y1)
            except TypeError: pass
            except ZeroDivisionError: f4[i] = zerodivision
            except OverflowError: f4[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (439*step*f1[j]/216.0) + \
                (-8.0*step*f2[j]) + (3680*step*f3[j]/513.0) +\
                (-845*step*f4[j]/4104.0)
        for i in range(n):
            try: f5[i] = funcs[i](x0+step, y1)
            except TypeError: pass
            except ZeroDivisionError: f5[i] = zerodivision
            except OverflowError: f5[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (-8*step*f1[j]/27.0) + \
              (2.0*step*f2[j]) + (-3544*step*f3[j]/2565.0) +\
              (1859*step*f4[j]/4104.0) + \
              (-11*step*f5[j]/40.0)
```

```
            for i in range(n):
                try: f6[i] = funcs[i](x0+(0.5*step), y1)
                except TypeError: pass
                except ZeroDivisionError: f6[i] = zerodivision
                except OverflowError: f6[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + (step * \
                       ((25*f1[i]/216.0) + (1408*f3[i]/2565.0) +\
                        (2197*f4[i]/4104.0) + (-0.2*f5[i])))
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y1[i] = overflow
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def RKF5(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fifth order Runge-Kutta_Fehlberg method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3, f4, f5, f6 = [0]*n, [0]*n, [0]*n, \
                                     [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.25*step*f1[j])
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(0.25*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
```

```
            except OverflowError: f2[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (3*step*f1[j]/32.0) + \
                    (9*step*f2[j]/32.0)
        for i in range(n):
            try: f3[i] = funcs[i]((x0+(3*step/8.0)), y1)
            except TypeError: pass
            except ZeroDivisionError: f3[i] = zerodivision
            except OverflowError: f3[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (1932*step*f1[j]/2197.0) + \
                     (-7200*step*f2[j]/2197.0) + \
                     (7296*step*f3[j]/2197.0)
        for i in range(n):
            try: f4[i] = funcs[i]((x0+(12*step/13.0)), y1)
            except TypeError: pass
            except ZeroDivisionError: f4[i] = zerodivision
            except OverflowError: f4[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (439*step*f1[j]/216.0) + \
              (-8.0*step*f2[j]) + (3680*step*f3[j]/513.0) + \
              (-845*step*f4[j]/4104.0)
        for i in range(n):
            try: f5[i] = funcs[i](x0+step, y1)
            except TypeError: pass
            except ZeroDivisionError: f5[i] = zerodivision
            except OverflowError: f5[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (-8*step*f1[j]/27.0) + \
              (2.0*step*f2[j]) + (-3544*step*f3[j]/2565.0) +\
              (1859*step*f4[j]/4104.0) + \
              (-11*step*f5[j]/40.0)
        for i in range(n):
            try: f6[i] = funcs[i](x0+(0.5*step), y1)
            except TypeError: pass
            except ZeroDivisionError: f6[i] = zerodivision
            except OverflowError: f6[i] = overflow
        for i in range(n):
            try: y1[i] = y0[i] + (step * \
                 ((16*f1[i]/135.0) + (6656*f3[i]/12825.0) + \
                  (28561*f4[i]/56430.0) + (-9*f5[i]/50) + \
                  (2*f6[i]/55)))
            except TypeError: pass
            except ZeroDivisionError: y1[i] = zerodivision
            except OverflowError: y1[i] = overflow
        return y1
    while x0 < xmax:
        y1 = solver(funcs, x0, y0, step)
        if nonODEfunc:
            y1 = nonODEfunc(y1, step)
```

```
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def DP4(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fourth order Dormand-Prince method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3 = [0]*n, [0]*n, [0]*n
            f4, f5, f6, f7 = [0]*n, [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.2*step*f1[j])
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(0.2*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
                except OverflowError: f2[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (3*step*f1[j]/40.0) + \
                        (9*step*f2[j]/40.0)
            for i in range(n):
                try: f3[i] = funcs[i]((x0+(0.3*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f3[i] = zerodivision
                except OverflowError: f3[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (44*step*f1[j]/45.0) + \
                   (-56*step*f2[j]/15.0) + (32*step*f3[j]/9.0)
            for i in range(n):
                try: f4[i] = funcs[i]((x0+(0.8*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f4[i] = zerodivision
                except OverflowError: f4[i] = overflow
```

```
        for j in range(n):
            y1[j] = y0[j] + (19372*step*f1[j]/6561.0) + \
                    (-25360*step*f2[j]/2187.0) + \
                    (64448*step*f3[j]/6561.0) + \
                    (-212*step*f4[j]/729.0)
        for i in range(n):
            try: f5[i] = funcs[i](x0+(8*step/9.0), y1)
            except TypeError: pass
            except ZeroDivisionError: f5[i] = zerodivision
            except OverflowError: f5[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (9017*step*f1[j]/3168.0) + \
                    (-355*step*f2[j]/33.0) + \
                    (46732*step*f3[j]/5247.0) + \
                    (49*step*f4[j]/176.0) + \
                    (-5103*step*f5[j]/18656.0)
        for i in range(n):
            try: f6[i] = funcs[i](x0+step, y1)
            except TypeError: pass
            except ZeroDivisionError: f6[i] = zerodivision
            except OverflowError: f6[i] = overflow
        for j in range(n):
            y1[j] = y0[j] + (35*step*f1[j]/384.0) + \
                    (500*step*f3[j]/1113.0) + \
                    (125*step*f4[j]/192.0) + \
                    (-2187*step*f5[j]/6784.0) + \
                    (11*step*f6[j]/84.0)
        for i in range(n):
            try: f7[i] = funcs[i](x0+step, y1)
            except TypeError: pass
            except ZeroDivisionError: f7[i] = zerodivision
            except OverflowError: f7[i] = overflow
        for i in range(n):
            try: y1[i] = y0[i] + (step * \
                   ((5179*f1[i]/57600.0) + \
                   (7571*f3[i]/16695.0) + (393*f4[i]/640.0) \
                   + (-92097*f5[i]/339200.0) + \
                   (187*f6[i]/2100.0) + (f7[i]/40.0)))
            except TypeError: pass
            except ZeroDivisionError: y1[i] = zerodivision
            except OverflowError: y1[i] = overflow
        return y1
    while x0 < xmax:
        y1 = solver(funcs, x0, y0, step)
        if nonODEfunc:
            y1 = nonODEfunc(y1, step)
        if lower_bound:
            y1 = boundary_checker(y1, lower_bound, 'lower')
        if upper_bound:
            y1 = boundary_checker(y1, upper_bound, 'upper')
```

```
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def DP5(funcs, x0, y0, step, xmax, nonODEfunc=None,
            lower_bound=None, upper_bound=None,
            overflow=1e100, zerodivision=1e100):
        '''Generator to integrate a system of ODEs, y' = f(x, y),
        using fifth order Dormand-Prince method.

        See Euler method documentation for description on
        parameters list.'''
        yield [x0] + y0
        def solver(funcs, x0, y0, step):
            n = len(funcs)
            f1, f2, f3 = [0]*n, [0]*n, [0]*n
            f4, f5, f6, f7 = [0]*n, [0]*n, [0]*n, [0]*n
            y1 = [0]*n
            for i in range(n):
                try: f1[i] = funcs[i](x0, y0)
                except TypeError: pass
                except ZeroDivisionError: f1[i] = zerodivision
                except OverflowError: f1[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (0.2*step*f1[j])
            for i in range(n):
                try: f2[i] = funcs[i]((x0+(0.2*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f2[i] = zerodivision
                except OverflowError: f2[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (3*step*f1[j]/40.0) + \
                        (9*step*f2[j]/40.0)
            for i in range(n):
                try: f3[i] = funcs[i]((x0+(0.3*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f3[i] = zerodivision
                except OverflowError: f3[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (44*step*f1[j]/45.0) + \
                  (-56*step*f2[j]/15.0) + (32*step*f3[j]/9.0)
            for i in range(n):
                try: f4[i] = funcs[i]((x0+(0.8*step)), y1)
                except TypeError: pass
                except ZeroDivisionError: f4[i] = zerodivision
                except OverflowError: f4[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (19372*step*f1[j]/6561.0) + \
                        (-25360*step*f2[j]/2187.0) + \
                        (64448*step*f3[j]/6561.0) + \
```

```
                    (-212*step*f4[j]/729.0)
            for i in range(n):
                try: f5[i] = funcs[i](x0+(8*step/9.0), y1)
                except TypeError: pass
                except ZeroDivisionError: f5[i] = zerodivision
                except OverflowError: f5[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (9017*step*f1[j]/3168.0) + \
                        (-355*step*f2[j]/33.0) + \
                        (46732*step*f3[j]/5247.0) + \
                        (49*step*f4[j]/176.0) + \
                        (-5103*step*f5[j]/18656.0)
            for i in range(n):
                try: f6[i] = funcs[i](x0+step, y1)
                except TypeError: pass
                except ZeroDivisionError: f6[i] = zerodivision
                except OverflowError: f6[i] = overflow
            for j in range(n):
                y1[j] = y0[j] + (35*step*f1[j]/384.0) + \
                        (500*step*f3[j]/1113.0) + \
                        (125*step*f4[j]/192.0) + \
                        (-2187*step*f5[j]/6784.0) + \
                        (11*step*f6[j]/84.0)
            for i in range(n):
                try: f7[i] = funcs[i](x0+step, y1)
                except TypeError: pass
                except ZeroDivisionError: f7[i] = zerodivision
                except OverflowError: f7[i] = overflow
            for i in range(n):
                try: y1[i] = y0[i] + (step * \
                     ((35*f1[i]/384.0) + (500*f3[i]/1113.0) + \
                     (125*f4[i]/192.0) + (-2187*f5[i]/6784.0) + \
                     (11*f6[i]/84.0)))
                except TypeError: pass
                except ZeroDivisionError: y1[i] = zerodivision
                except OverflowError: y1[i] = overflow
            return y1
        while x0 < xmax:
            y1 = solver(funcs, x0, y0, step)
            if nonODEfunc:
                y1 = nonODEfunc(y1, step)
            if lower_bound:
                y1 = boundary_checker(y1, lower_bound, 'lower')
            if upper_bound:
                y1 = boundary_checker(y1, upper_bound, 'upper')
            y0 = y1
            x0 = x0 + step
            yield [x0] + y0

    def _equation_constructor(expressions={},
```

```
                                parameters={},
                                variables=[]):
        '''Private function to support ODE_constructor to generate
        ODE function for each ODE.

        @param expressions: dictionary of expressions for ODE(s).
        Please see above documentation.
        @param parameters: dictionary of parameter values to be
        substituted into the ODE equations
        @param variables: list of additional variables to be
        tagged for substitution(s)
        @return: tuple of (<list of generated ODE functions>,
        <list of tagged variables>) '''
        statements = []
        # Generate ODE function, one at a time
        for name in expressions.keys():
            # Generate ODE function definition
            stmt = '\ndef %s(t, y):' % str(name)
            expression = expressions[name]
            if type(expression) == type(''):
                expression = [expression]
            count = 1
            # Generate list of expression(s) for current ODE
            exp_list = []
            variables = list(set(variables + expressions.keys()))
            for exp in expression:
                # Substitute parameter/variable values
                for k in parameters.keys():
                    exp = exp.replace(str(k), str(parameters[k]))
                for v in variables:
                    exp = exp.replace(str(v), 'v@'+str(v))
                # Generate expression codes
                stmt = stmt + '\n    exp_%s = %s' % (str(count),
                                                     str(exp))
                exp_list.append('exp_%s' % str(count))
                count = count + 1
            # Compile generated expressions into return value
            return_stmt = '\n    return ' + ' + '.join(exp_list)
            stmt = stmt + return_stmt
            statements.append(stmt)
        return (statements, variables)

    def _modifying_constructor(modifying_expressions):
        '''Private function to support ODE_constructor to generate
        function code for modifying expressions.

        @param modifying_expressions: list of expressions to
        modify the variables
        @return: generated function code'''
        stmt = '\ndef modifying_expression(y, step):'
```

```
        for exp in modifying_expressions:
            stmt = stmt + '\n    %s' % str(exp)
        stmt = stmt + '\n    return y'
        return stmt

    def ODE_constructor(scriptfile,
                        resultsfile,
                        time=(0.0, 0.1, 100.0),
                        ODE_solver='RK4',
                        expressions={},
                        parameters={},
                        initial_conditions={},
                        modifying_expressions=[],
                        lower_bound=None,
                        upper_bound=None,
                        overflow=1e100,
                        zerodivision=1e100):
        '''Function to construct an ODE simulation script file
        from given definitions.

        For example, the following system of ODEs

        M{
        d(human)/dt = birth - zombied - death
        d(zombie)/dt = zombied + resurrected - destroyed
        d(dead)/dt = death + destroyed - resurrected

        birth = 0
        zombied = 0.0095 * human * zombie
        death = 0.0001 * human
        resurrected = 0.0002 * dead
        destroyed = 0.0003 * human * zombie

        where initially (t0),

        number of humans = 500
        number of zombies = 0
        number of dead = 0

        and with the following boundaries

        number of humans > 0
        number of zombies < 10000
        number of dead < 10000

        and there is an influx of 5 humans per day into the
        infected village}

        can be specified as
```

```
C{
scriptfile = 'zombie_attack.py'
resultsfile = 'zombie_data.csv'
time = (0.0, 0.1, 100.0)
ODE_solver = 'RK4'
expressions = \
{'human': ['birth_rate',
           '- (transmission_rate * human * zombie)',
           '- (death_rate * human)'],
 'zombie': ['(transmission_rate * human * zombie)',
            '(resurrection_rate * dead)',
            '- (destroy_rate * human * zombie)'],
 'dead': ['(death_rate * human)',
          '(destroy_rate * human * zombie)',
          '- (resurrection_rate * dead)']}
parameters = {'birth_rate': 0.0,
              'transmission_rate': 0.0095,
              'death_rate': 0.0001,
              'resurrection_rate': 0.0002,
              'destroy_rate':0.0003}
initial_conditions = {'human': 500.0,
                      'zombie': 0.0,
                      'dead': 0.0}
modifying_expression = ['human = human + (5 * step)']
lower_bound = {'human': [0.0, 0.0]}
upper_bound = {'zombie': [10000.0, 10000.0],
               'dead': [10000.0, 10000.0]}
overflow = 1e100
zerodivision = 1e100
}

@param scriptfile: name of Python file for the generated
ODE script file
@type scriptfile: string
@param resultsfile: name of ODE simulation results file
(CSV file), which will be included into the generated
simulation script file
@type resultsfile: string
@param time: tuple of time parameters for simulation in
the format of (<start time>, <time step>, <end time>).
Default = (0.0, 0.1, 100.0)
@param ODE_solver: name of ODE solver to use. Default =
RK4
@param expressions: dictionary of expressions for ODE(s).
Please see above documentation.
@param parameters: dictionary of parameter values to be
substituted into the ODE equations
@param modifying_expressions: list of expressions to
modify the variables
@param initial_conditions: dictionary of initial
```

```
conditions for each ODE
@param lower_bound: set of values for lower boundary of
variables
@type lower_bound: dictionary
@param upper_bound: set of values for upper boundary of
variables
@type upper_bound: dictionary
@param overflow: value (usually a large value) to assign
in event of over flow error (usually caused by a large
number) during integration. Default = 1e100.
@type overflow: float
@param zerodivision: value (usually a large value) to
assign in event of zero division error, which results in
positive infinity, during integration. Default = 1e100.
@type zerodivision: float
@return: generated ODE codes for the entire script
@rtype: list'''
statements = []
initial_conditions_list = initial_conditions.keys()
# Construct ODE functions
(ODE_functions,
 variables) = _equation_constructor(expressions,
                 parameters, initial_conditions_list)
# Construct modifying expression
mexpression = \
    _modifying_constructor(modifying_expressions)
# Generate ODE functions/equations table
table = {}
count = 0
for v in initial_conditions_list:
    table[str(v)] = str(count)
    count = count + 1
# Generate lower boundary table
if lower_bound != None:
    lbound = {}
    for k in lower_bound.keys():
        lbound[table[k]] = lower_bound[k]
else:
    lbound = None
# Generate upper boundary table
if upper_bound != None:
    ubound = {}
    for k in upper_bound.keys():
        ubound[table[k]] = upper_bound[k]
else:
    ubound = None
# Write ode module import
statements.append('import ode\n\n')
# Generate variable array and statements
statements.append('y = range(%s)\n' % \
```

```
                    str(len(variables)))
        count = 0
        for k in initial_conditions_list:
            statements.append('y[%s] = %s\n' % \
                    (str(count), str(initial_conditions[k])))
            count = count + 1
        statements.append('\n')
        # Perform variable replacements in ODE functions, and
        # write functions
        for funct in ODE_functions:
            for v in table.keys():
                funct = funct.replace('v@'+v, 'y[%s]' % table[v])
            statements.append(funct)
        statements.append('\n\n')
        # Perform variable replacements in modifying expression,
        # and write out
        for v in table.keys():
            mexpression = mexpression.replace(v, 'y[%s]' % \
                                              table[v])
        statements.append(mexpression)
        statements.append('\n\n')
        # Generate ODE assignment array and statements
        statements.append('ODE = range(%s)\n' % \
                          str(len(variables)))
        count = 0
        for k in initial_conditions_list:
            statements.append('ODE[%s] = %s\n' % \
                              (str(count), str(k)))
            count = count + 1
        statements.append('\n')
        # Generate ODE execution codes
        statements.append('stime = %s \n' % str(time[0]))
        statements.append('step = %s \n' % str(time[1]))
        statements.append('etime = %s \n' % str(time[2]))
        statements.append('lower_bound = %s \n' % str(lbound))
        statements.append('upper_bound = %s \n' % str(ubound))
        statements.append('overflow = %s \n' % str(overflow))
        statements.append('zerodivision = %s \n' % \
                          str(zerodivision))
        statements.append('\n')
        statements.append("f = open('%s', 'w')\n" % \
                          str(resultsfile))
        headers = ['time'] + \
                  [str(k) for k in initial_conditions.keys()]
        headers = ','.join(headers)
        statements.append("f.write('%s' + '\\n') \n" % headers)
        statements.append('\n')
        statements.append('for x in ode.%s(ODE, stime, y, step,
etime, \n' % str(ODE_solver))
        statements.append('        modifying_expression, \n')
```

```
        statements.append('        lower_bound, upper_bound, \n')
        statements.append('        overflow, zerodivision): \n')
        statements.append("      f.write(','.join([str(item)  for
item in x]) + '\\n')")
        statements.append('\n')
        statements.append('f.close()')
        # Write generated codes into script file
        sfile = open(scriptfile, 'w')
        sfile.writelines(statements)
        sfile.close()
        return statements
```

ACKNOWLEDGEMENT

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

REFERENCES

Bao, G., and Zhang, H. (2014) Sensitivity Analysis of an Inverse Problem for the Wave Equation with Caustics. Journal of the American Mathematical Society **27**, 953-981.

Cash, J. R., and Karp, A. H. (1990) A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides. ACM Transactions on Mathematical Software 16, 201-222.

Chay, Z.E., Goh, B.F., and Ling, M.H.T. (2016) PNet: A Python Library for Petri Net Modeling and Simulation. Advances in Computer Science: an international journal 5(4): 24-30.

Dormand, J.R., and Prince, P. J (1980) A Family of Embedded Runge-Kutta Formulae. Journal of Computational and Applied Mathematics 6, 19–26.

Langtangen, H.P., and Wang, L. (2014) Odespy software package. URL: https://github.com/hplgit/odespy.

Ling, M.H.T. (2016a) Of (Biological) Models and Simulations. MOJ Proteomics & Bioinformatics 3(4): 00093.

Ling, M.H.T. (2016b) COPADS IV: Fixed Time-Step ODE Solvers for a System of Equations Implemented as a Set of Python Functions. Advances in Computer Science: an international journal 5(3): 5-11.

Munz, P., Hudea, I., Imad, J., and Smith, R. J. (2009) When Zombies Attack!: Mathematical Modelling of Outbreak of Zombie Infection. Infectious Disease Modelling Research Progress 4, 133-150.

Oliphant, T.E. (2007) Python for Scientific Computing. Computing in Science & Engineering 9, 10-20.

Otto, S.P., and Day, T. (2007) A Biologist's Guide to Mathematical Modeling in Ecology and Evolution (Vol. 13). Princeton University Press.

Schober, M., Duvenaud, D. K., and Hennig, P. (2014) Probabilistic ODE solvers with Runge-Kutta means. In Advances in neural information processing systems, pp. 739-747.

APPENDIX A
(EXAMPLE CODE ON NON-ODE / MODIFYING FUNCTION)

```
import random
import ode

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

def influx(y, step):
    y[0] = y[0] + (5*step)
    return y

f = [human, zombie, dead]   # system of ODEs
# initial human, zombie, death population, and total
y = [500.0, 0, 0]

print('Solving using 5th order Dormand-Prince method ......')
noinflux = [x for x in ode.DP5(f, 0.0, y, 0.1, 50.0)]
influx = [x for x in ode.DP5(f, 0.0, y, 0.1, 50.0, influx)]

for i in range(len(noinflux)):
    consolidated = noinflux[i] + influx[i][1:]
    print ','.join([str(x) for x in consolidated])
```

APPENDIX B
(EXAMPLE CODE ON BOUNDARY VALUES)

```
import ode

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

f = [human, zombie, dead]   # system of ODEs
# initial human, zombie, death population, and total
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, None,
                                 lower_bound)]
doublebound = [x for x in ode.DP5(f, 0.0, y, 0.1, 50.0, None,
                                  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])
```

APPENDIX C
(EXAMPLE CODE ON BOUNDED VERSUS UNBOUNDED VARIABLES)

```
import ode

# --------------------------
# Case 1: Bounded Variables
# --------------------------

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

def human1(t, y):
    infected = transmission * y[0] * y[1]
    dead = death * y[0]
    return birth - infected - dead
def zombie1(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 dead1(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
bounded = [human1, zombie1, dead1]

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

# ----------------------------
# Case 2: Un-Bounded Variables
# ----------------------------
```

```
def human2(t, y):
    infected = y[5] * y[0] * y[1]
    dead = y[4] * y[0]
    return y[3] - infected - dead
def zombie2(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 dead2(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

unbounded = range(8)
unbounded[0] = human2
unbounded[1] = zombie2
unbounded[2] = dead2

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 \
with un-bounded variables......')
for i in [x for x in
          ode.DP5(unbounded, 0.0, y, 0.1, 50.0)]:
    print(','.join([str(z) for z in i]))
```