Numerical Methods

PSLTDSim utilizes a variety of numerical methods to perform integration. Some of the methods are coded 'by hand', while others are included in Python packages. This appendix is meant to introduce some numerical integration techniques, provide basic information about two Python functions used to perform numerical integration, compare results of numerical methods to exact solutions via examples, and briefly explain how some dynamic agents utilize the explained techniques.

1.1 Integration Methods

The options included in PSLTDSim to solve the combined swing equation for a new system frequency are Euler, Adams-Bashforth, and Runge-Kutta. Each of these methods are numerical approximations that provide an *approximation* to the solution of an initial value problem. Method equations presented below were adapted from [**?**].

1.1.1 Euler Method

Of the integration methods available, the Euler method is the simplest. In general terms, the next y value associated with a given differential function $f(t, y)$ is

$$y_{n+1} = y_n + f(t_n, y_n)t_s \tag{1.1}$$

Euler Method

where t_s is desired time step. The y_{n+1} solution is simply a projection along a line tangent to $f(t_n, y_n)$. It should be noted that the accuracy of this approximation method, and others described, is often related to the time step size, or the distance between approximations.

1.1.2 Runge-Kutta Method

Improving on the Euler method, the Runge-Kutta method combines numerous projections through a weighted average to approximate the next y value. The fourth order four-stage Runge-Kutta method is defined as Equation Block 1.2.

$$\begin{aligned} k_1 &= f(t_n,\ y_n) \\ k_2 &= f(t_n + t_s/2,\ y_n + t_s k_1/2) \\ k_3 &= f(t_n + t_s/2,\ y_n + t_s k_2/2) \\ k_4 &= f(t_n + t_s,\ y_n + t_s k_3) \\ y_{n+1} &= y_n + t_s(k_1 + 2k_2 + 2k_3 + k_4)/6 \end{aligned} \tag{1.2}$$

Fourth Order Four-Stage Runge-Kutta

It can be seen that k_1 and k_4 are solutions on either side of the interval of approximation defined by the time step t_s, and that k_2 and k_3 represent midpoint estimations.

1.1.3 Adams-Bashforth Method

Unlike previously introduced methods, the Adams-Bashforth method requires data from previous solution steps. Methods of this nature are sometimes referred to as multistep methods. A two-step Adams-Bashforth method is described in Equation 1.3, however, larger step methods do exist.

$$y_{n+1} = y_n + t_s \left(1.5f(t_n, y_n) - 0.5f(t_{n-1}, y_{n-1})\right) \tag{1.3}$$

Two-Step Adams-Bashforth

Regardless of the number of steps, Adams-Bashforth methods utilize a weighted combination of values similar to the Runge-Kutta method, but using only previously known values instead of projected future values.

1.1.4 Trapezoidal Integration

To integrate known values generated each time step, PSLTDSim uses a trapezoidal integration method. Given some value $y(t)$, the trapezoidal method states that

$$\int_{t-t_s}^{t} y(t)\ \mathrm{d}t \approx t_s \left(y(t) + y(t-ts)\right)/2, \tag{1.4}$$

Trapezoidal Integration

where t_s is the time step used between calculated values of y. Visually, this method can be thought of connecting the two y values with a straight line, and then calculating the area of the trapezoid formed between them. As with previously described methods, the accuracy of this method depends on step size.

1.2 Python Functions

To allow for more robust solution methods, two Python functions were incorporated into PSLTDSim. The two functions are from the Scipy package for scientific computing. General information about these two functions is presented in this section.

1.2.1 scipy.integrate.solve_ivp

The Scipy solve_ipv function is capable of numerically integrating ordinary differential equations with initial values using a variety of techniques. A generic call to the function is shown in Figure 1.1. Required inputs include a multi-variable function of x and y (i.e. some $f(x, y)$), a tuple

describing the range of integration, and an initial value list. The output is an object with various collections of time points, solution points, and other information about the returned solution.

```
soln = scipy.integrate.solve_ivp(fp, (t0, t1), [initVal])
```

Figure 1.1: Generic call to solve_ivp.

The default integration method used by solve_ivp is an explicit Runge-Kutta of order 5(4). This method is similar to the previously discussed 4th order Runge-Kutta, but with an additional estimation factor. The four in parenthesis describes an approximation generated by a 4th order method which is used to calculate an error term between the 5th order solution and adjust the approximation time step accordingly. The exact execution of this process may be studied in the source code of the function itself. Other possible integration methods and function usage suggestions are described in [**?**].

1.2.2 scipy.signal.lsim

The Scipy function that simulates the output from a continuous-time linear system is called lsim. A general call to lsim is shown in Figure 1.2. The inputs include an 'lti' system, an input vector, a time vector, and an initial state vector.

```
tout, y, x = scipy.signal.lsim(system, [U,U], [t0,t1], initialStates)
```

Figure 1.2: Generic call to lsim.

Accepted lti systems passed into lsim may be transfer functions or state space systems created by the Scipy signal package. Function output includes a simulated time vector, system output, and state history. The computations performed by lsim utilize a state space solution centered around a matrix exponential that solves a system of first order differential equations. More complete information about the usage of lsim may be found in its source code or in [**?**].

1.3 Method Comparisons via Python Code Examples

Approximations from each method or function described above were compared to an exact solution by way of a Python script. This section includes full code from each test case, equations required to solve integrals exactly, and simulation results. Due to the lack of an accepted code listing format for this document, code is presented in figures that may span page breaks. Despite the breaks in code presentation, code line numbers are continuous where applicable.

1.3.1 General Approximation Comparisons

The code used to compare the Euler, Adams-Bashforth, and Runge-Kutta method to an exact solution is presented below. As most code does, the created script begins with package imports. Numpy was imported for its math capabilities, such as the exponential function, and Matplotlib was imported for its plotting functions.

```
"""
File meant to show numerical integration methods applied via python
Structured in a way that is related to the simulation method in PSLTDSim

NOTE: lambda is the python equivalent to matlab anonymous functions
"""
# Package Imports
import numpy as np
import matplotlib.pyplot as plt
```

Figure 1.3: Approximation comparison package imports.

Each approximation method described in Equation 1.1-1.4 was coded as a Python function. It should be noted that trapezoidal integration was intended to be performed after the simulation is run and full data is collected. This choice was made because of the various time steps involved with solution results.

```
# Function Definitions
def euler(fp, x0, y0, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    ts = time step
    Returns y1 using Euler or tangent line method
    """
    return y0 + fp(x0,y0)*ts

def adams2(fp, x0, y0, xN, yN, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    xN = Previous x value
    yN = Previous y value
    ts = time step
    Returns y1 using Adams-Bashforth two step method
```

```
    """
    return y0 + (1.5*fp(x0,y0) - 0.5*fp(xN,yN))*ts

def rk45(fp, x0, y0, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    ts = time step
    Returns y1 using Runge-Kutta method
    """
    k1 = fp(x0, y0)
    k2 = fp(x0 +ts/2, y0+ts/2*k1)
    k3 = fp(x0 +ts/2, y0+ts/2*k2)
    k4 = fp(x0 +ts, y0+ts*k3)
    return y0 + ts/6*(k1+2*k2+2*k3+k4)

def trapezoidalPost(x,y):
    """
    x = list of x values
    y = list of y values
    Returns integral of y over x.
    Assumes full lists / ran post simulation
    """
    integral = 0
    for ndx in range(1,len(x)):
        integral+= (y[ndx]+y[ndx-1])/2 * (x[ndx]-x[ndx-1])
    return integral
```

Figure 1.4: Approximation comparison function definitions.

To enable one file to execute all desired tests, a for loop that cycles through a case number variable was created. Each case if statement contains definitions for case name, simulation start and stop times, number of points to plot, the initial value problem, the exact solution, and the exact integral solution. Equations from each case are further described in future sections. The Python `lambda` command was used to create temporary functions that are passed to other functions.

```
# Case Selection
for caseN in range(0,3):
    blkFlag = False # for holding plots open
    if caseN == 0:
        # Trig example
        caseName = 'Sinusoidal Example'
        tStart =0
        tEnd = 3
        numPoints = 6*2
```

```
67
68          ic = [0,0] # initial condition x,y
69          fp = lambda x, y: -2*np.pi*np.cos(2*np.pi*x)
70          f = lambda x,c: -np.sin(2*np.pi*x)+c
71          findC = lambda x,y: y+np.sin(2*np.pi*x)
72          c = findC(ic[0],ic[1])
73          calcInt = ( 1/(2*np.pi)*np.cos(2*np.pi*tEnd)+c*tEnd -
74                      1/(2*np.pi)*np.cos(2*np.pi*ic[0])-c*ic[0] )
75
76      elif caseN == 1:
77          # Exp example
78          caseName = 'Exponential Example'
79          tStart =0
80          tEnd = 3
81          numPoints = 3
82
83          ic = [0,0] # initial condition x,y
84          fp = lambda x, y: np.exp(x)
85          f = lambda x,c: np.exp(x)+c
86          findC = lambda x, y: y-np.exp(x)
87          c= findC(ic[0],ic[1])
88          calcInt = np.exp(tEnd)+c*tEnd-np.exp(ic[0])+c*ic[0]
89
90      elif caseN == 2:
91          # Log example
92          caseName = 'Logarithmic Example'
93          tStart =1
94          tEnd = 4
95          numPoints = 3
96          blkFlag = True # for holding plots open
97
98          ic = [1,1] # initial condition x,y
99          fp = lambda x, y: 1/x
100         f = lambda x,c: np.log(x)+c
101         findC = lambda x, y: y-np.log(x)
102         c= findC(ic[0],ic[1])
103         calcInt = (tEnd*np.log(tEnd)- tEnd +c*tEnd -
104                    ic[0]*np.log(ic[0])+ ic[0] -c*ic[0])
```

Figure 1.5: Approximation comparison case definitions.

After case selection, a current value dictionary `cv` was initialized to mimic how PSLTDSim stores current values. Unlike PSLTDSim, the lists used to store history values were not initialized to the full length they were expected to be. This required logged values to be appended to the list after each solution. The reasoning behind this choice was again due to the various time steps involved with solution results.

```
    # Initialize current value dictionary
    # Shown to mimic PSLTDSim record keeping
    cv={
        't' :ic[0],
        'yE': ic[1],
        'yRK': ic[1],
        'yAB': ic[1],
        }

    # Initialize running value lists
    t=[]
    yE=[]
    yRK =[]
    yAB = []

    t.append(cv['t'])
    yE.append(cv['yE'])
    yRK.append(cv['yRK'])
    yAB.append(cv['yAB'])
```

Figure 1.6: Approximation comparison variable initialization.

An exact solution was computed using a hand-derived exact function. The code then entered a while loop that solved the selected differential equation for the next y value using the Euler, Runge-Kutta, and Adams-Bashforth methods. It should be noted that Python enables negative indexing of lists. Intuitively, negative indexes step backwards through an iterable object. An if statement was required to handle the first step of the Adams-Bashforth method as a -2 index does not exist in a list of length 1. After each approximation method was executed, and the solution stored in the current value dictionary, all values were logged and simulation time increased.

```
    # Find C from integrated equation for exact soln
    c = findC(ic[0], ic[1])
    # Calculate time step
    ts = (tEnd-tStart)/numPoints
    # Calculate exact solution
    tExact = np.linspace(tStart,tEnd, 10000)
    yExact = f(tExact, c)

           # Start Simulation
    while cv['t']< tEnd:

        # Calculate Euler result
        cv['yE'] = euler( fp, cv['t'], cv['yE'],  ts )
        # Calculate Runge-Kutta result
        cv['yRK'] = rk45( fp, cv['t'], cv['yRK'],  ts )
```

```
139
140         # Calculate Adams-Bashforth result
141         if len(t)>=2:
142             cv['yAB'] = adams2( fp, cv['t'], cv['yAB'], t[-2], yAB[-2], ts )
143         else:
144             # Required to handle first step when a -2 index doesn't exist
145             cv['yAB'] = adams2( fp, cv['t'], cv['yAB'], t[-1], yAB[-1], ts )
146
147         # Log calculated results
148         yE.append(cv['yE'])
149         yRK.append(cv['yRK'])
150         yAB.append(cv['yAB'])
151
152         # Increment and log time
153         cv['t'] += ts
154         t.append(cv['t'])
```

Figure 1.7: Approximation comparison solution calculations.

Matplotlib functions were used to generate result plots after simulated time accumulated to a point that the while loop exited. Each line color, legend label, and various other superficial options were defined before global plot output options were configured and the plot displayed.

```
155     # Generate Plot
156     fig, ax = plt.subplots()
157     ax.set_title('Approximation Comparison\n' + caseName)
158
159     #Plot all lines
160     ax.plot(tExact,yExact,
161             c=[0,0,0],
162             linewidth=2,
163             label="Exact")
164     ax.plot(t,yE,
165             marker='o',
166             fillstyle='none',
167             linestyle=':',
168             c=[0.7,0.7,0.7],
169             label="Euler")
170     ax.plot(t,yRK,
171             marker='*',
172             markersize=10,
173             fillstyle='none',
174             linestyle=':',
175             c=[1,0,1],
176             label="RK4")
177     ax.plot(t,yAB,
```

```
178             marker='s',
179             fillstyle='none',
180             linestyle=':',
181             c =[0,1,0],
182             label="AB2")
183 
184     # Format Plot
185     fig.set_dpi(150)
186     fig.set_size_inches(9, 2.5)
187     ax.set_xlim(min(t), max(t))
188     ax.grid(True, alpha=0.25)
189     ax.legend(loc='best',  ncol=2)
190     ax.set_ylabel('y Value')
191     ax.set_xlabel('x Value')
192     fig.tight_layout()
193     plt.show(block = blkFlag)
194     plt.pause(0.00001)
```

Figure 1.8: Approximation comparison plotting.

After plotting, trapezoidal integration was performed on all results and compared to the calculated integral. It should be noted that the 'exact' result uses trapezoidal integration on 10,000 points while the calculated integral `calcInt` was computed via calculus. After code line 208 executes, the for loop that started on line 59 is restarted until all case numbers in the selected range are applied.

```
195     # Trapezoidal Integration
196     exactI = trapezoidalPost(tExact,yExact)
197     Eint = trapezoidalPost(t,yE)
198     RKint = trapezoidalPost(t,yRK)
199     ABint = trapezoidalPost(t,yAB)
200 
201     print("\n%s" % caseName)
202     print("time step:  %.2f" % ts)
203     print("Method: Trapezoidal Int\t Absolute Error from calculated")
204     print("Calc: \t%.9f\t%.9f" % (calcInt ,abs(calcInt-calcInt)))
205     print("Exact: \t%.9f\t%.9f" % (exactI ,abs(calcInt-exactI)))
206     print("RK4: \t%.9f\t%.9f" % (RKint,abs(calcInt-RKint)))
207     print("AB2: \t%.9f\t%.9f" % (ABint,abs(calcInt-ABint)))
208     print("Euler: \t%.9f\t%.9f" % (Eint,abs(calcInt-Eint)))
```

Figure 1.9: Approximation comparison trapezoidal integration and display.

Sinusoidal Example and Results

The first initial value example is presented as Equation Block 1.5.

$$\begin{aligned} \text{Given: } y(0) &= 0 \\ y'(x) &= 2\pi\cos(2\pi x) \end{aligned} \tag{1.5}$$

Sinusoidal Example

Two integrations of Equation 1.5 were performed to calculate the exact integral and plot the exact solution. This is shown in Equation Block 1.6.

$$\begin{aligned} \int y'(x)\ \mathrm{d}x = y(x) &= -\sin(2\pi x) + C_1 \\ C_1 &= y_0 + \sin(2\pi x_0) \\ \int_0^\tau y(x)\ \mathrm{d}x &= \frac{1}{2\pi}\cos(2\pi x) + C_1 x \Big|_0^\tau \end{aligned} \tag{1.6}$$

Sinusoidal Example Integration

Figure 1.10 shows that when using a 0.5 step size, the approximations of all methods do not accurately reflect the exact function. This example and step size were contrived to show such behavior. The explanation for such a result lies in the derivatives calculated at the points used to generate each approximation.

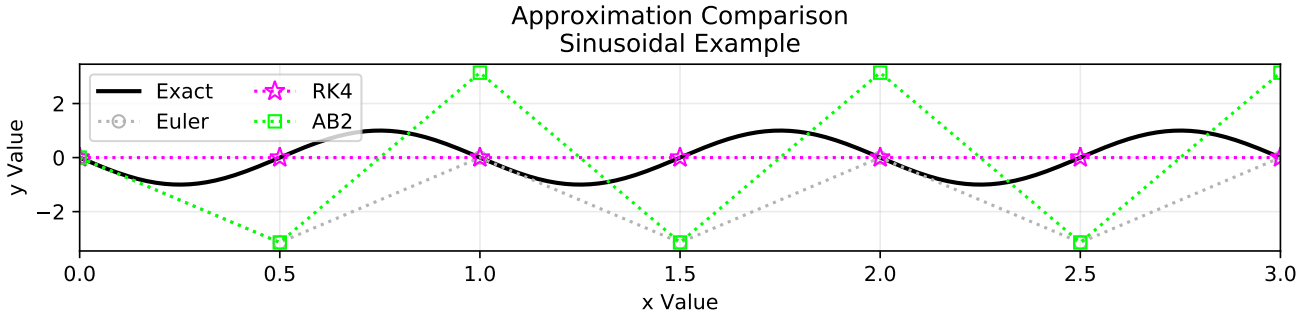

Figure 1.10: Approximation comparison of a sinusoidal function using a step of 0.5.

Using a smaller step size of 0.25, as shown in Figure 1.11, results with more accurate approximations. For all calculated points, the Runge-Kutta method matches the exact solution while the other two methods do not.

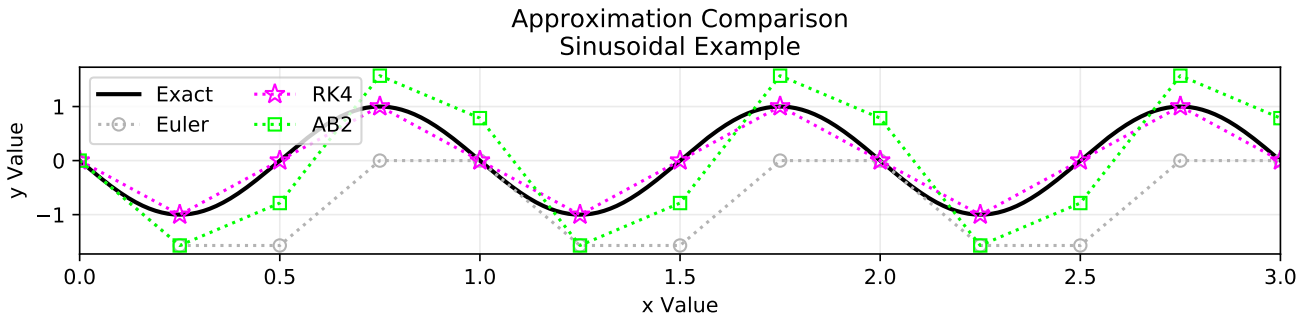

Figure 1.11: Approximation comparison of a sinusoidal function using a step of 0.25.

Table 1.1 shows the calculated integrals of the 0.25 step size example. It should be noted that because the integral is zero, any function may numerically match the calculated result if it is symmetrical about zero. The Runge-Kutta method meets this criteria despite representing more of a triangle wave instead of a sine wave. The Euler method has the largest error from exact integral as there are no approximated points above zero.

Table 1.1: Trapezoidal integration results of a sinusoidal function using an x step of 0.25.

Method	Result	Absolute Error
Calculated	0.000000000	0.000000000
Exact	0.000000000	0.000000000
RK4	-0.000000000	0.000000000
AB2	-0.098174770	0.098174770
Euler	-2.356194490	2.356194490

Exponential Example and Results

The second initial value example is presented as Equation Block 1.7.

$$\begin{aligned} \text{Given: } y(0) &= 0 \\ y'(x) &= e^x \end{aligned} \tag{1.7}$$

Exponential Example

The required integrations of Equation 1.7 are shown in Equation Block 1.8.

$$\begin{aligned} \int y'(x)\ \mathrm{d}x = y(x) &= e^x + C_1 \\ C_1 &= y_0 - e^x \\ \int_0^\tau y(x)\ \mathrm{d}x &= e^x + C_1 x|_0^\tau \end{aligned} \tag{1.8}$$

Exponential Example Integration

Figure 1.12 shows the resulting comparison plot using a step size of 1. The Runge-Kutta method matches the exact solution well while the other two approximation methods under-approximate. This is due to the lack of the Euler and Adams-Bashforth methods to accurately represent a constantly changing derivative.

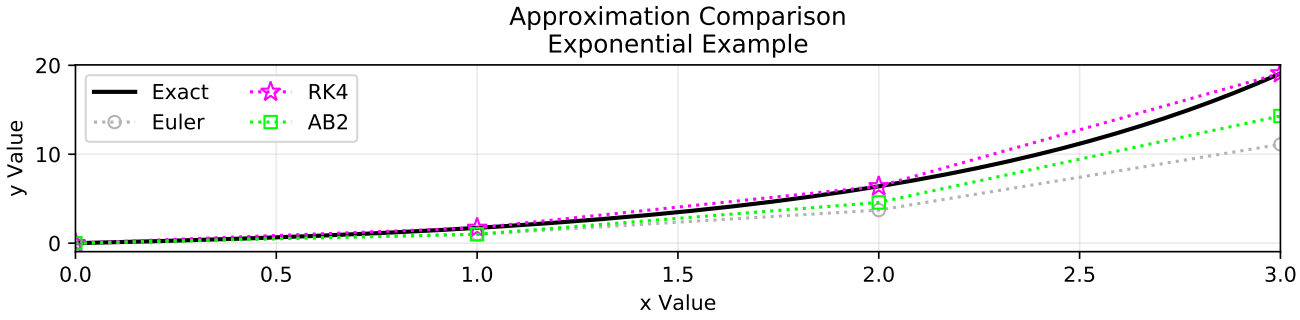

Figure 1.12: Approximation comparison of an exponential function.

Table 1.2 shows the trapezoidal integration of the exact function does not match the calculated integral. This is due to the exponential function not being well represented by trapezoids. Absolute error continued to increase with the Runge-Kutta, Adams-Bashforth, and Euler methods respectively.

Table 1.2: Trapezoidal integration results of an exponential function using an x step of 1.

Method	Result	Absolute Error
Calculated	16.085536923	0.000000000
Exact	16.085537066	0.000000143
RK4	17.656057171	1.570520247
AB2	12.728355731	3.357181192
Euler	10.271950792	5.813586131

Logarithmic Example and Results

The third initial value example is presented as Equation Block 1.9. Initial values are not zero as this would immediately lead to a divide by zero situation.

$$\begin{aligned} \text{Given: } y(1) &= 1 \\ y'(x) &= \frac{1}{x} \end{aligned} \tag{1.9}$$

Logarithmic Example

The required integrations of Equation 1.9 are shown in Equation Block 1.10.

$$\int y'(x)\ \mathrm{d}x = y(x) = \ln(x) + C_1$$
$$C_1 = y_0 - \ln(x_0) \tag{1.10}$$
$$\int_0^\tau y(x)\ \mathrm{d}x = x\ln(x) - x + C_1 x\Big|_0^\tau$$

Logarithmic Example Integration

Figure 1.13 shows the resulting comparison plot using a step size of 1. Again the Runge-Kutta method produces the best approximation while the Euler method has the worst. The Adam-Bashforth method appears to be converging to the exact solution. While the exponential function and logarithmic functions both contain constantly changing derivatives, the logarithmic derivative decreases with increasing x values. This produces an over-approximating situation where as the exponential function was generally under-approximated.

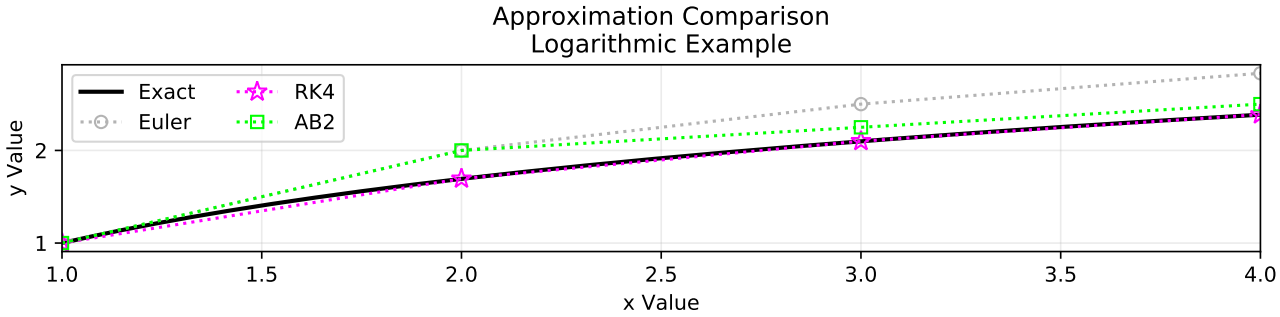

Figure 1.13: Approximation comparison of a logarithmic function.

Table 1.3 shows the integration results have a similar trend as seen in Table 1.2 where the exact trapezoidal method doesn't match the calculated integral, and absolute error gradually increases using the Runge-Kutta, Adams-Bashforth, and Euler methods respectively.

Table 1.3: Trapezoidal integration results of logarithmic function using an x step of 1.

Method	Result	Absolute Error
Calulated	5.545177444	0.000000000
Exact	5.545177439	0.000000006
RK4	5.488293651	0.056883794
AB2	6.000000000	0.454822556
Euler	6.416666667	0.871489222

General Approximation Result Summary

The chosen examples showed that the Runge-Kutta method typically produces better results than the simpler Euler or Adams-Bashforth methods. Step size is an important factor to consider

when using approximation methods as phenomena may be ignored or reported in error elsewise. Depending on step size, trapezoidal integration can produce results that are reasonable approximations of calculated integrals.

1.3.2 Python Function Comparisons

Code used to compare the Python lsim and solve_ivp functions to the exact solution and fourth order Runge-Kutta approximation is presented below. The code is very similar to the previously discussed approximation comparison code and again begins with package imports and function definitions. The solve_ivp function was imported from the integrate methods of Scipy, while the lsim function is part of the signal collection of functions. Only the Runge-Kutta and trapezoidal methods are defined as functions in this code example.

```
# Package Imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy import signal

# Function Definitions
def rk45(fp, x0, y0, ts):
    """
    fp = Some derivative function of x and y
    x0 = Current x value
    y0 = Current y value
    ts = time step
    Returns y1 using Runge-Kutta method
    """
    k1 = fp(x0, y0)
    k2 = fp(x0 +ts/2, y0+ts/2*k1)
    k3 = fp(x0 +ts/2, y0+ts/2*k2)
    k4 = fp(x0 +ts, y0+ts*k3)
    return y0 + ts/6*(k1+2*k2+2*k3+k4)

def trapezoidalPost(x,y):
    """
    x = list of x values
    y = list of y values
    Returns integral of y over x.
    Assumes full lists / ran post simulation
    """
    integral = 0
    for ndx in range(1,len(x)):
        integral+= (y[ndx]+y[ndx-1])/2 * (x[ndx]-x[ndx-1])
    return integral
```

Figure 1.14: Python function comparison imports and definitions.

Case definitions were similar to the previous example with the addition of an lti system definition. For simplicity, a transfer function style system was used as input to create each lti system. More specifically, this input consisted of the numerator and denominator of the transfer function as lists of descending powers $ (the Laplace 's'). Numerous transforms and calculus based mathematical methods found in [?, ?] and [?] were employed to calculate the exact functions and integrals which are described in more detail after this code discussion.

```
33 # Case Selection
34 for caseN in range(0,3):
35     blkFlag = False # for holding plots open
36 
37     if caseN == 0:
38         # step input Integrator example
39         caseName = 'Step Input Integrator Example'
40         tStart =0
41         tEnd = 4
42         numPoints = 4
43 
44         U = 1
45         initState = 0
46         ic = [0,initState] # initial condition x,y
47         fp = lambda x, y: 1
48         f = lambda x, c: x+c
49         findC = lambda x, y: y-x
50         system = signal.lti([1],[1,0])
51         calcInt = 0.5*(tEnd**2) # Calculated integral
52 
53     elif caseN == 1:
54         # step input Low pass example
55         caseName = 'Step Input Low Pass Example'
56         tStart =0
57         tEnd = 2
58         numPoints = 4
59 
60         A = 0.25
61         U = 1.0
62         initState = 0
63         ic = [0,initState] # initial condition x,y
64         fp = lambda x, y: 1/A*np.exp(-x/A)# via table
65         f = lambda x, c: -np.exp(-x/A) +c
66         findC = lambda x, y : y+np.exp(-x/A)
67         system = signal.lti([1],[A,1])
68         calcInt = tEnd + A*np.exp(-tEnd/A)-A # Calculated integral
69 
70     else:
```

```
        # step multi order system
        caseName = 'Step Input Third Order System Example'
        tStart =0
        tEnd = 5
        numPoints = 5*2
        blkFlag = True # for holding plots open

        U = 1
        T0 = 0.4
        T2 = 4.5
        T1 = 5
        T3 = -1
        T4 = 0.5

        alphaNum = (T1*T3)
        alphaDen = (T0*T2*T4)
        alpha = alphaNum/alphaDen

        num = alphaNum*np.array([1, 1/T1+1/T3, 1/(T1*T3)])
        den  = alphaDen*np.array([1, 1/T4+1/T0+1/T2, 1/(T0*T4)+1/(T2*T4)+1/(T0*T2),
                        1/(T0*T2*T4)])
        system = signal.lti(num,den)

        # PFE
        A = ((1/T1-1/T0)*(1/T3-1/T0))/((1/T2-1/T0)*(1/T4-1/T0))
        B = ((1/T1-1/T2)*(1/T3-1/T2))/((1/T0-1/T2)*(1/T4-1/T2))
        C = ((1/T1-1/T4)*(1/T3-1/T4))/((1/T0-1/T4)*(1/T2-1/T4))

        initState = 0 # for steady state start
        ic = [0,0] # initial condition x,y
        fp = lambda x, y: alpha*(A*np.exp(-x/T0)+B*np.exp(-x/T2)+C*np.exp(-x/T4))
        f = lambda x, c: alpha*(-T0*A*np.exp(-x/T0)-T2*B*np.exp(-x/T2)-T4*C*np.exp(-x/T4))+c
        findC = lambda x, y : alpha*(A*T0+B*T2+C*T4)
        c = findC(ic[0], ic[1])
        calcInt = (
            alpha*A*T0**2*np.exp(-tEnd/T0) +
            alpha*B*T2**2*np.exp(-tEnd/T2) +
            alpha*C*T4**2*np.exp(-tEnd/T4) +
            c*tEnd -
            alpha*(A*T0**2+B*T2**2+C*T4**2)
            )# Calculated integral
```

Figure 1.15: Python function comparison case definitions.

Initial conditions and list initializations were performed in a similar manner as the previous example. An additional `xLS` variable was required to track the states associated with the lsim function.

```
    # Initialize current value dictionary
    # Shown to mimic PSLTDSim record keeping
    cv={
        't' :ic[0],
        'yRK': ic[1],
        'ySI': ic[1],
        'yLS': ic[1],
        }

    # Initialize running value lists
    t=[]
    yRK =[]
    # solve ivp
    ySI = []
    tSI = []
    # lsim
    yLS = []
    xLS = [] # required to track state history

    t.append(cv['t'])
    yRK.append(cv['yRK'])
    yLS.append(cv['yLS'])
    xLS.append(cv['yLS'])
```

Figure 1.16: Python function comparison variable initializations.

The exact solution and Runge-Kutta methods were handled as before, but Python function inputs required slightly different input. The lsim and solve_ivp outputs also required slightly different handling as their output was not just a single value. Again, negative indexing is used to access the last value in an iterable object.

```
    # Calculate time step
    ts = (tEnd-tStart)/numPoints
    # Find C from integrated equation for exact soln
    c = findC(ic[0], ic[1])
    # Calculate exact solution
    tExact = np.linspace(tStart,tEnd, 1000)
    yExact = f(tExact, c)

    # Start Simulation
    while cv['t']< tEnd:

        # Calculate Runge-Kutta result
        cv['yRK'] = rk45( fp, cv['t'], cv['yRK'],  ts )

        # Runge-Kutta 4(5) via solve IVP.
```

```
        soln = solve_ivp(fp, (cv['t'], cv['t']+ts), [cv['ySI']])

        # lsim solution
        if cv['t'] > 0:
            tout, ylsim, xlsim = signal.lsim(system, [U,U], [0,ts], xLS[-1])
        else:
            tout, ylsim, xlsim = signal.lsim(system, [U,U], [0,ts], initState)

        # Log calculated results
        yRK.append(cv['yRK'])

        # handle solve_ivp output data
        ySI += list(soln.y[-1])
        tSI += list(soln.t)
        cv['ySI'] = ySI[-1] # ensure correct cv

        # handle lsim output data
        cv['yLS']=ylsim[-1]
        yLS.append(cv['yLS'])
        xLS.append(xlsim[-1]) # this is the state

        # Increment and log time
        cv['t'] += ts
        t.append(cv['t'])
```

Figure 1.17: Python function comparison solution calculations.

Once the simulation is complete, plotting and trapezoidal integration was carried out in the same manner as previously discussed before the for loop restarts.

```
    # Generate Plot
    fig, ax = plt.subplots()
    ax.set_title('Approximation Comparison\n' + caseName)

    #Plot all lines
    ax.plot(tExact,yExact,
            c=[0,0,0],
            linewidth=2,
            label="Exact")
    ax.plot(t,yRK,
            marker='*',
            markersize=10,
            fillstyle='none',
            linestyle=':',
            c=[1,0,1],
            label="RK45")
    ax.plot(tSI,ySI,
```

```
            marker='x',
            markersize=10,
            fillstyle='none',
            linestyle=':',
            c=[1,.647,0],
            label="solve_ivp")
    ax.plot(t,yLS,
            marker='+',
            markersize=10,
            fillstyle='none',
            linestyle=':',
            c ="#17becf",
            label="lsim")

    # Format Plot
    fig.set_dpi(150)
    fig.set_size_inches(9, 2.5)
    ax.set_xlim(min(t), max(t))
    ax.grid(True, alpha=0.25)
    ax.legend(loc='best',  ncol=2)
    ax.set_ylabel('y Value')
    ax.set_xlabel('Time [seconds]')
    fig.tight_layout()
    plt.show(block = blkFlag)
    plt.pause(0.00001)

    # Trapezoidal Integration
    exactI = trapezoidalPost(tExact,yExact)
    SIint = trapezoidalPost(tSI,ySI)
    RKint = trapezoidalPost(t,yRK)
    LSint = trapezoidalPost(t,yLS)

    print("\n%s" % caseName)
    print("time step:  %.2f" % ts)
    print("Method: Trapezoidal Int\t Absolute Error from calculated")
    print("Calc: \t%.9f\t%.9f" % (calcInt ,abs(calcInt-calcInt)))
    print("Exact: \t%.9f\t%.9f" % (exactI ,abs(calcInt-exactI)))
    print("RK4: \t%.9f\t%.9f" % (RKint,abs(calcInt-RKint)))
    print("SI: \t%.9f\t%.9f" % (SIint,abs(calcInt-SIint)))
    print("lsim: \t%.9f\t%.9f" % (LSint,abs(calcInt-LSint)))
```

Figure 1.18: Python function comparison plotting and integration code.

Integrator Example and Results

The first example is the Laplace domain integrator block shown in Figure 1.19.

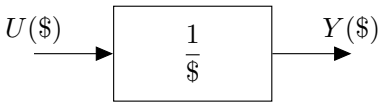

Figure 1.19: Integrator block.

Transformation of the block into a time domain derivative function is shown in Equation Block 1.11. As step input is a given, this results in a very simple differential equation.

$$\begin{aligned} &\text{Given: Step input, } y(0) = 0 \\ F(\$) &= \frac{Y(\$)}{U(\$)} = \frac{1}{\$} \\ F(\$) &= Y(\$)\$ = U(\$) \\ \mathscr{L}^{-1}\{F(\$)\} &\longrightarrow y'(t) = u(t) = 1 \end{aligned} \tag{1.11}$$

Integrator Transform Example

The required integrations are shown in Equation Block 1.12.

$$\begin{aligned} \int y'(t)\ \mathrm{d}t &= y(t) = t + C_1 \\ C_1 &= y_0 - t_0 \\ \int_0^\tau y(t)\ \mathrm{d}t &= \frac{1}{2}t^2 + C_1 t \Big|_0^\tau \end{aligned} \tag{1.12}$$

Integrator Example Integration

The resulting approximation comparisons are plotted in Figure 1.20. While all methods produce the same result, it is worth noting the extra approximations generated by the solve_ivp function near the beginning of each approximation interval.

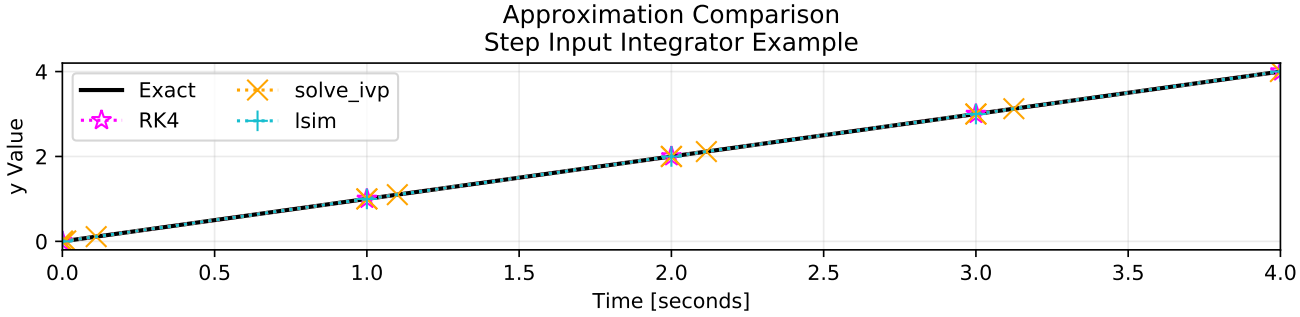

Figure 1.20: Approximation comparison of an integrator block.

Table 1.4 shows that all methods match the calculated integral. Obviously, this particular linear function can be accurately represented by trapezoids.

Table 1.4: Trapezoidal integration results of integral function using a t step of 1.

Method	Result	Absolute Error
Calculated	8.000000000	0.000000000
Exact	8.000000000	0.000000000
RK4	8.000000000	0.000000000
solve_ivp	8.000000000	0.000000000
lsim	8.000000000	0.000000000

Low Pass Example and Results

A slightly more interesting example consists of the Laplace low pass filter block shown in Figure 1.21.

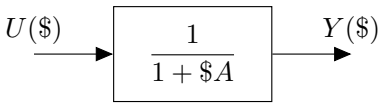

Figure 1.21: Low pass filter block.

Equation Block 1.13 shows the manipulation of $F(\$)$ to match a common Laplace form so that a conversion table could be used to easily convert the equation from the frequency domain to the time domain.

$$\begin{aligned} &\text{Given: Step input, } A = 0.25, y(0) = 0 \\ &F(\$) = \frac{Y(\$)}{U(\$)} = \left(\frac{1}{A}\right)\left(\frac{1}{\$ + 1/A}\right) \\ &\mathscr{L}^{-1}\{F(\$)\} \longrightarrow y'(t) = \frac{e^{-t/A}}{A} \end{aligned} \tag{1.13}$$

Low Pass Transform Example

Required integration is shown in Equation Block 1.14.

$$\begin{aligned} \int y'(t)\ \mathrm{d}t = y(t) &= -e^{-t/A} + C_1 \\ C_1 &= y_0 + e^{-t_0/A} \\ \int_0^\tau y(t)\ \mathrm{d}t &= Ae^{-t_0/A} + C_1 t\Big|_0^\tau \end{aligned} \tag{1.14}$$

Low Pass Example Integration

The resulting approximation comparisons are shown in Figure 1.22. All methods produce approximations that are very close to the exact solution. The solve_ivp function again produces more approximations between the defined step range.

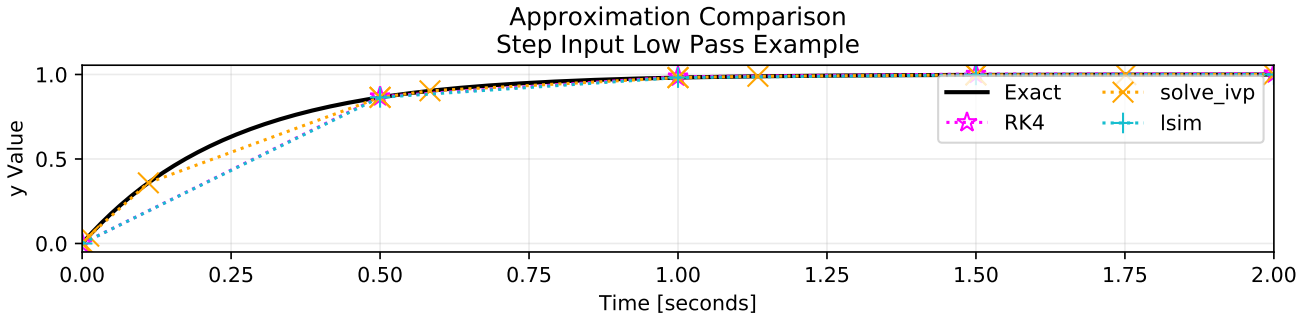

Figure 1.22: Approximation comparison of a low pass filter block.

Table 1.5 shows the integration results of the low pass example. The exact solution has slight error from the calculated integral as trapezoidal integration provides only an approximate solution. The solve_ivp result has the next smallest error due to the added points between defined approximation steps. Runge-Kutta and lsim results were very similar.

Table 1.5: Trapezoidal integration results of low pass function using a t step of 0.5.

Method	Result	Absolute Error
Calculated	1.750083866	0.000000000
Exact	1.750082530	0.000001336
RK4	1.680138966	0.069944900
solve_ivp	1.719657220	0.030426646
lsim	1.671851297	0.078232568

Third Order System Example and Results

A third order system that resembles the one used in the genericGov is shown in Figure 1.23. As the previous example showed, manipulation of Laplace transfer function blocks with poles may be useful when it comes time to convert to the time domain. The resulting modified block diagram is shown in Figure 1.24.

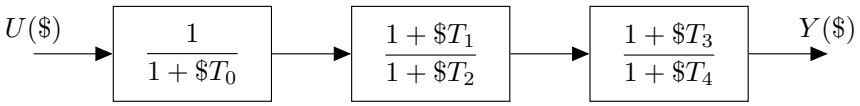

Figure 1.23: Third order system block diagram.

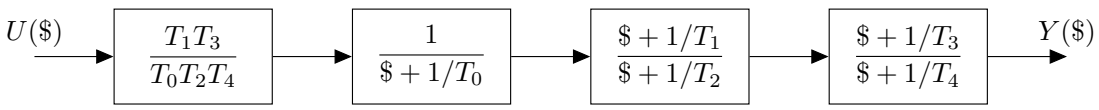

Figure 1.24: Modified third order system block diagram.

Example givens and algebraic simplifications are listed at the top of Equation 1.15. The time constants chosen were those of the generic hydro governor with the exception of T_2, which was reduced by an order of magnitude so that a steady state was reached within a reasonable amount of time. Partial fraction expansion was used to express the third order equation as sum of first order terms. The rational behind this action was to enable a simpler inverse Laplace transform.

$$\begin{aligned}
\text{Given: Step input, } T_0 &= 0.4, T_1 = 5.0, T_2 = 4.5, \\
T_3 &= -1.0, T_4 = 0.5, y(0) = 0 \\
\text{Let } \alpha &= \frac{T_1 T_3}{T_0 T_2 T_4} \\
F(\$) = \alpha \frac{(\$ + 1/T_1)(\$ + 1/T_3)}{(\$ + 1/T_0)(\$ + 1/T_2)(\$ + 1/T_4)} &= \alpha \left(\frac{A}{\$ + 1/T_0} + \frac{B}{\$ + 1/T_2} + \frac{C}{\$ + 1/T_4} \right) \\
F(\$)(\$ + 1/T_0)\big|_{\$=-1/T_0} = A &= \frac{\left(1/T_1 - 1/T_0\right)\left(1/T_3 - 1/T_0\right)}{\left(1/T_2 - 1/T_0\right)\left(1/T_4 - 1/T_0\right)} \\
F(\$)(\$ + 1/T_2)\big|_{\$=-1/T_2} = B &= \frac{\left(1/T_1 - 1/T_2\right)\left(1/T_3 - 1/T_2\right)}{\left(1/T_0 - 1/T_2\right)\left(1/T_4 - 1/T_2\right)} \\
F(\$)(\$ + 1/T_4)\big|_{\$=-1/T_4} = C &= \frac{\left(1/T_1 - 1/T_4\right)\left(1/T_3 - 1/T_4\right)}{\left(1/T_0 - 1/T_4\right)\left(1/T_2 - 1/T_4\right)} \\
\mathscr{L}^{-1}\{F(\$)\} \longrightarrow y'(t) &= \alpha \left(A e^{-t/T_0} + B e^{-t/T_2} + C e^{-t/T_4} \right)
\end{aligned} \tag{1.15}$$

Third Order Transform Example

The relatively straight forward integrations required for an exact solution and integral are shown in Equation Block 1.16.

$$\begin{aligned}
\int y'(t)\ \mathrm{d}t = y(t) &= -\alpha \left(A T_0 e^{-t/T_0} + B T_2 e^{-t/T_2} + C T_4 e^{-t/T_4} \right) + C_1 \\
C_1 &= y_0 + \alpha \left(A T_0 e^{-t_0/T_0} + B T_2 e^{-t_0/T_2} + C T_4 e^{-t_0/T_4} \right) \\
\int_0^\tau y(t)\ \mathrm{d}t &= \alpha \left(A T_0^2 e^{-t/T_0} + B T_2^2 e^{-t/T_2} + C T_4^2 e^{-t/T_4} \right) + C_1 t \Big|_0^\tau
\end{aligned} \tag{1.16}$$

Third Order Example Integration

Figure 1.25 shows the approximation comparison results of the third order system. Using a half second time step produces results that are fairly similar to the exact method. As previously seen, the solve_ivp solution produces more approximations between defined time steps.

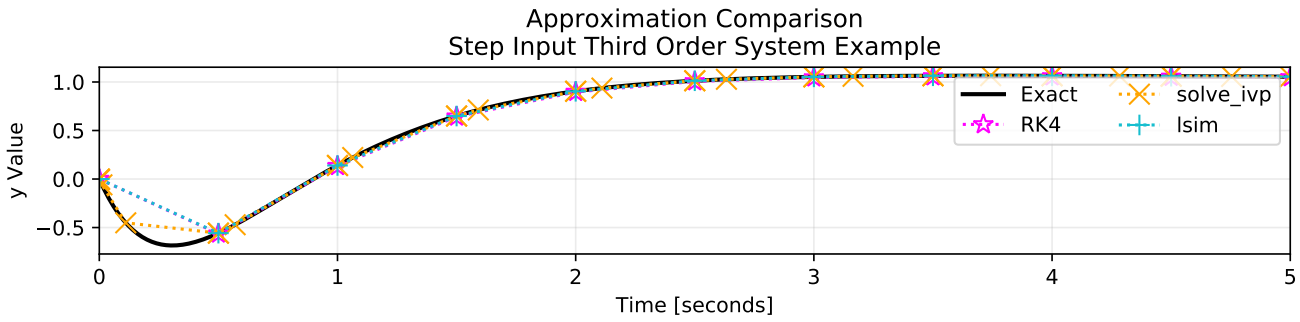

Figure 1.25: Third order approximation comparison using half second time step.

Increasing the time step to one second, as shown in Figure 1.26, highlights more differences between the methods. The solve_ivp soultion still tracks the exact solution well because of the additional approximations between time steps. Approximations of lsim match the exact solution, however, dynamics between time steps are not represented at all. The Runge-Kutta method also ignores dynamics between approximation results and appears to under-approximate steady state behavior.

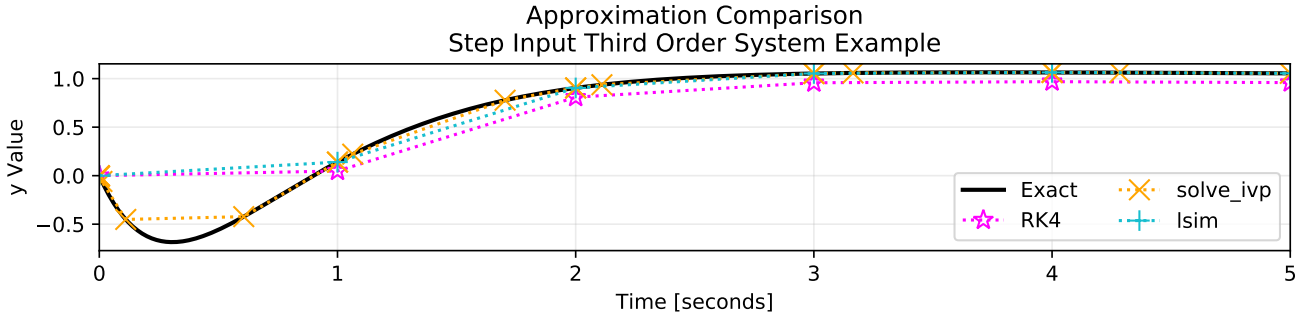

Figure 1.26: Third order approximation comparison using one second time step.

Table 1.6 lists the integration results from the half second time step test. The exact solution is near the calculated solution, but they do not match completely. The solve_ivp absolute error is the next smallest due to the additional data points generated between set time steps. While the absolute error from the Runge-Kutta solution is calculated as slightly less than the lsim result, this is due to the large negative area both solutions ignore between $t = 0$ and $t = 1$ and the continuous under-approximation by the Runge-Kutta method.

Table 1.6: Trapezoidal integration results of a third order function using a t step of 0.5.

Method	Result	Absolute Error
Calculated	3.351959451	0.000000000
Exact	3.351971025	0.000011574
RK4	3.425878989	0.073919538
solve_ivp	3.385138424	0.033178973
lsim	3.458377872	0.106418421

Python Approximation Result Summary

As previously stated, distance between approximations dictates much of what one can glean from resulting solutions. As such, the full resolution solve_ivp solution provided the most detail of the tested examples. However, the data at defined time steps were essentially the same between the lsim and solve_ivp results. With a small enough time step, the Runge-Kutta method approximations was also similar to the Python function approximations. When a larger time step was used, the Runge-Kutta method did not match the exact solution in cases where the other methods did. Through these experiments and comparisons, it was shown that the lsim and solve_ivp methods are comparable, and in some ways, better than the hand coded Runge-Kutta method. Additionally, trapezoidal integration was shown to produce adequate results depening on the input data.

1.4 Dynamic Agent Numerical Utilizations

This section is meant to better describe the handling of numerical methods by specific agents in PSLTDSim. Specifically, window integration and the combined swing equation function are described in detail before governor and filter agent considerations about integrator wind up and dynamic staging are presented.

1.4.1 Window Integrator

The window integrator agent used by balancing authority agents that integrate ACE applies the trapezoidal integration technique. As this agent is relatively simple, the full Python definition is shown in Figure 1.27. While attempts were made to create readable code, window integrator actions are also explained below.

```
1  class WindowIntegratorAgent(object):
2      """A window integrator that initializes a history of window
3      values, then updates the total window area each step."""
4
5      def __init__(self, mirror, length):
6          # Retain Inputs / mirror reference
7          self.mirror = mirror
8          self.length = length # length of window in seconds
9
10         self.windowSize = int(self.length / self.mirror.timeStep)
11
12         self.window = [0.0]*self.windowSize
13         self.windowNDX = -1 # so first step index points to 0
14
15         self.cv = {
16             'windowInt' : 0.0,
17             'totalInt' : 0.0,
```

```
            }

    def step(self, curVal, preVal):
        # calculate current window Area, return value
        self.windowNDX += 1
        self.windowNDX %= self.windowSize

        oldVal = self.window[self.windowNDX]
        newVal = (curVal + preVal)/ 2.0 * self.mirror.timeStep

        self.window[self.windowNDX] = newVal
        self.cv['windowInt'] += newVal - oldVal
        self.cv['totalInt'] += newVal

        return self.cv['windowInt']
```

Figure 1.27: Window integrator definition.

The agent is initialized by any agent that is desired to perform window integration. Required input parameters are a reference to the system mirror and window length in seconds. The reference to the system mirror is stored and a list of place holder values is created that is the length of the integration window in seconds, divided by the selected time step. This division result is cast into an integer as lists cannot have float value lengths. This list of history values is not required for integration, but it can be used to verify the correct operation of the integrator. A window index is created with an initial value of negative one so that during the first step, the index correctly points to list item zero. A current value dictionary `cv` is created to keep track of the most recent window integration and total integration values.

The parent agent is responsible for calling the window integrator step function each time step with current and previous values of integration focus. The window index variable is incremented by one, and then the modulo operator is used to ensure the index always points to a location that exists inside the list of history values. The value located at the current index value is stored as `oldVal` and later subtracted from the current window integration value. The integral between the two passed in values is calculated using the trapezoidal method and stored as `newVal`. This `newVal` is then stored in the window integrator history value list at the current index, and added to both the current value for window and total integration. The agent step ends by returning the current value of the window integrator.

1.4.2 Combined Swing Equation

The full code for the combined swing equation is presented in Figure 1.28. The function first checks if frequency effects should be accounted for, and then calculates the PU values required for computation of $\dot{\omega}_{sys}$ (`fdot` in the code). The calculated `fdot` is used by the Adams-Bashforth

and Euler solution methods if specified by the user. If the chosen integration method is 'rk45', the Runge-Kutta 4(5) method included in solve_ivp is used instead. While the Euler and Adams-Bashforth methods return only the next y value, the solve_ivp method returns more output variables that must be properly handled. The combined swing equation returns nothing and makes any required changes only to the system mirror.

```
def combinedSwing(mirror, Pacc):
    """Calculates fdot, integrates to find next f, calculates deltaF.
    Pacc in MW, f and fdot are PU
    """

    # Handle frequency effects option
    if mirror.simParams['freqEffects'] == 1:
        f = mirror.cv['f']
    else:
        f = 1.0

    PaccPU = Pacc/mirror.Sbase # for PU value
    HsysPU = mirror.cv['Hsys']/mirror.Sbase # to enable variable inertia
    deltaF = 1.0-mirror.cv['f'] # used for damping

    # Swing equation numerical solution
    fdot = 1/(2*HsysPU)*(PaccPU/f - mirror.Dsys*deltaF)
    mirror.cv['fdot'] = fdot

    # Adams Bashforth
    if mirror.simParams['integrationMethod'].lower() == 'ab':
        mirror.cv['f'] = f + 1.5*mirror.timeStep*fdot -
        ↪  0.5*mirror.timeStep*mirror.r_fdot[mirror.cv['dp']-1]

    # scipy.integrate.solve_ivp
    elif mirror.simParams['integrationMethod'].lower() == 'rk45':
        tic = time.time() # begin dynamic agent timer

        c = [HsysPU, PaccPU, mirror.Dsys, f] # known variables in swing eqn
        cSwing = lambda t, y: 1/(2*c[0])*(c[1]/y - c[2]*(1-c[3]))
        soln = solve_ivp(cSwing, [0, mirror.timeStep], [f])
        mirror.cv['f'] = float(soln.y[-1][-1]) # set current freq to last value

        mirror.IVPTime += time.time()-tic # accumulate and end timer

    # Euler method - chosen by default
    else:
        mirror.cv['f'] = mirror.cv['f'] + (mirror.timeStep*fdot)

    # Log values
    # NOTE: deltaF changed 6/5/19 to more useful 1-f
```

```
41      deltaF = 1.0 - mirror.cv['f']
42      mirror.cv['deltaF'] = deltaF
```

Figure 1.28: Combined swing function definition.

1.4.3 Governor and Filter Agent Considerations

The lsim function was chosen for governor and filter dynamic calculations. This was meant to enable a consistent solution method for these agent types. However, lsim only performs linear simulation and non-linear actions, such as limiting, must be handled manually. Futher, to simplify model creation and allow non-linear action, governor models were created as multiple dynamic stages that pass values to each other. Both of these lsim specific areas are covered in this section.

Integrator Wind Up

Non-linear system behavior must be handled outside of, or in between, an lsim solution as lsim only handles linear simulation. A common non-linear action is limiting. An issue may arise when limiting a pure integrator and not addressing integrator wind up. A simple example demonstrating integrator wind up is shown in Figure 1.29. The system used is the same as shown in Figure 1.19 but with an output limiter set at ± 2, and the input is depicted in Figure 1.29.

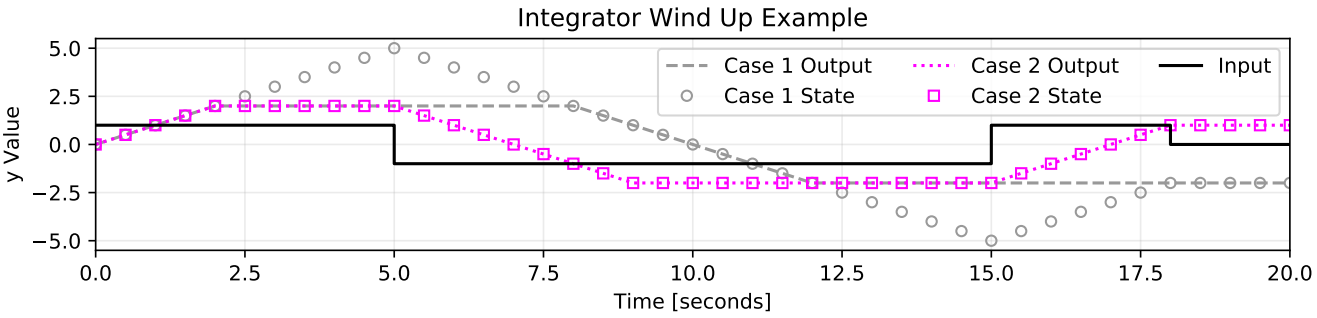

Figure 1.29: Effect of integrator wind up.

Results from Case 1 include only an output value limiter, while Case 2 also limits the integrator state. Limiting the state prevents integrator wind up which can be seen in Case 1 between $t = 2.5$ and $t = 7.5$ and again between $t = 12.5$ and $t = 17.5$. The execution of such limiting could be done multiple ways. In this case, a simple if statement was placed after the solution that checks output and state values. The if statement, if executed, adjusts the output and/or state values accordingly.

Combined System Comparisons

To allow for a variety of governor models without rewriting code and enable non-linear action, the technique of using a sequence of individual blocks for each part of a specific model was

employed in current PSLTDSim governor models. Modeling differences due to interaction of states in multi-order systems represented by a series of single order systems was explored by simulating equivalent systems consisting of various dynamic stages. For example, the block diagram shown in Figure 1.23 is mathematically equivalent to the block diagrams shown in Figures 1.30 and 1.31, however, the computation of each system may not be equivalent.

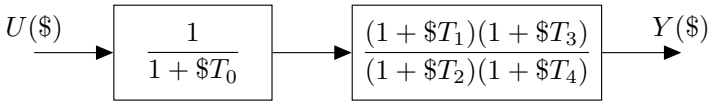

Figure 1.30: Third order system as two stages.

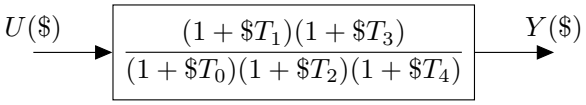

Figure 1.31: Third order system as single stage.

Figure 1.30 shows the output of a third order system as calculated by various dynamic stage models. The interaction of states affects the resulting output and it can be seen that the three stage system does not capture system dynamics well. A two stage calculation produces output closer to the single stage system, but some dynamics are not represented.

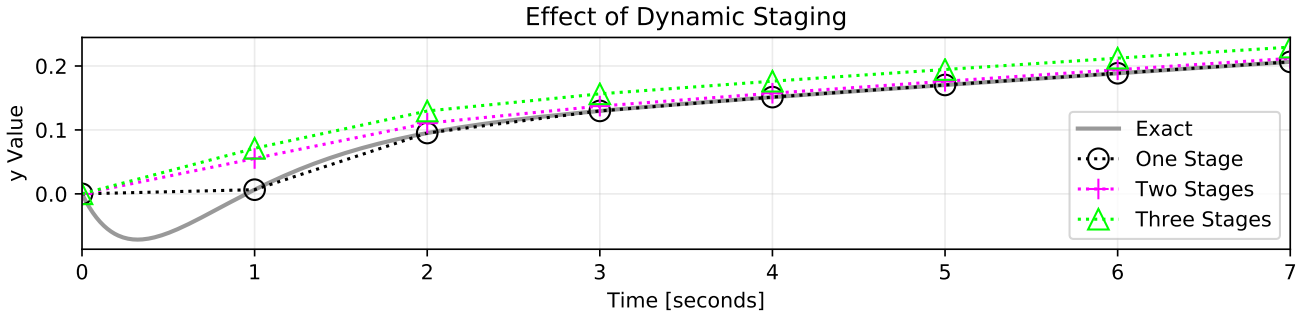

Figure 1.32: Effect of dynamic staging using one second time step.

Reducing step size, as shown in Figure 1.33, produces similar behavior where the three stage output is most different from the single stage model.

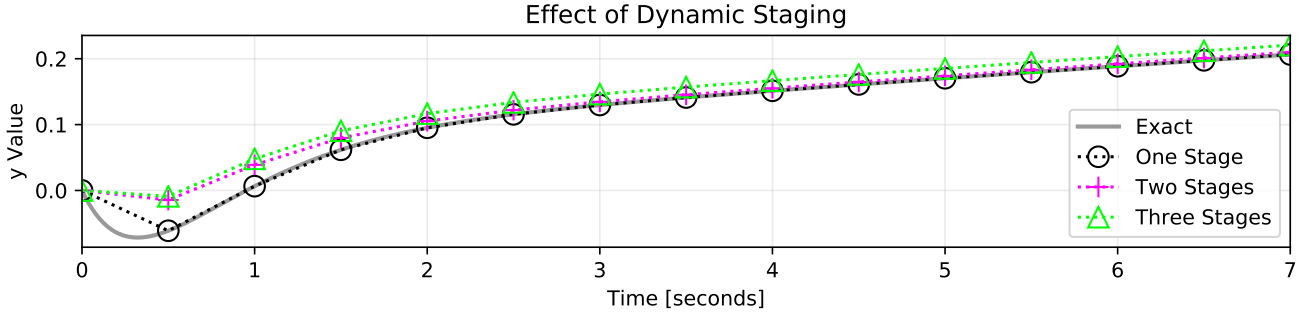

Figure 1.33: Effect of dynamic staging using half second time step.

1.4.4 Numerical Utilization Summary

PSLTDSim uses various numerical methods to achieve satisfactory results. However, PSLTDSim was designed to be a customizable simulation environment, and as more use cases arise, previously accepted solution methods may no longer be deemed as such. While there is no currently employed method for integrator wind up prevention, it is certainly possible. Likewise, experiments have shown there is a noticeable reduction in output definition when dynamic models are separated into multiple states. Of course, modifying or creating new, dynamic models to better meet changing user needs is possible. Such modifications require some understanding of the actual code. While documentation such as this can provide some assistance to such an endeavor, actual understanding can be best gained through actual study of the available source code.