Numerical Methods

PSLTDSim utilizes a variety of numerical methods to perform integration. Some of the employed methods are coded 'by hand', while others utilize Python packages. This appendix is meant to introduce some numerical integration techniques, provide information about two Python functions used to perform numerical integration, compare results of numerical methods 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, to find the next y value given some 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 next value of y is simply a projection along a line tangent to f at time t. It should be noted that the accuracy of approximation methods is often related to the time step size.

1.1.2 Runge-Kutta Method

Improving on the Euler method, the Runge-Kutta method combines numerous projections as a weighted average to find the next y value. The fourth order four-stage Runge-Kutta method is described in Equation 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 on either side of the interval of approximation defined by the time step ts, and 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 time steps. Methods of this nature are sometimes referred to as multistep or predictor-corrector 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, the Adams-Bashforth methods utilize a weighted combination of values similar to the Runge-Kutta method, but using only previously known data.

1.1.4 Trapezoidal Integration

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

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

Trapezoidal Integration

where t_s is the time step used between calculated values of x. Visually, this method can be thought of connecting the two y values with a straight line, then calculating the area of the trapezoid formed between.

1.2 Python Functions

To allow for more robust solution methods, two Python functions were incorporated into PSLTDSim. The two used 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. 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 is an explicit Runge-Kutte of order 5(4). This method is similar to the previously discussed 4th order Runge-Kutta, but with an additional factor. The four in parenthesis describes another approximation generated by a 4th order method which is used to calculate an error term between the 5th order solution and adjust the integration step accordingly. The exact execution of this process may be studied in the source code of the function itself and other integration methods are listed [**?**].

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.

Systems passed into lsim may be transfer functions or state space systems. More complete information about the usage of lsim may be found in [**?**]. Function output includes the 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 the system of first order differential equations.

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 a 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. Numpy is imported for its math capabilities, such as the exponential function, and Matplotlib is imported to create the resulting plots.

```
"""
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: Code package imports.

Each function definition are created as presented in Equations 1.1-1.4. It should be noted that trapezoidal integration is 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.

```
# Method 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

```

```
33 def rk45(fp, x0, y0, ts):
34     """
35     fp = Some derivative function of x and y
36     x0 = Current x value
37     y0 = Current y value
38     ts = time step
39     Returns y1 using Runge-Kutta method
40     """
41     k1 = fp(x0, y0)
42     k2 = fp(x0 +ts/2, y0+ts/2*k1)
43     k3 = fp(x0 +ts/2, y0+ts/2*k2)
44     k4 = fp(x0 +ts, y0+ts*k3)
45     return y0 + ts/6*(k1+2*k2+2*k3+k4)
46 
47 def trapezoidalPost(x,y):
48     """
49     x = list of x values
50     y = list of y values
51     Returns integral of y over x.
52     Assumes full lists / ran post simulation
53     """
54     integral = 0
55     for ndx in range(1,len(x)):
56         integral+= (y[ndx]+y[ndx-1])/2 * (x[ndx]-x[ndx-1])
57     return integral
```

Figure 1.4: Code function definitions.

To only require one file to run all tests, a for loop that cycles through a case number variable was created. Each case defines the case name, simulation start and stop times, number of points to plot, the initial value problem, the exact solution, and the exact integral solution. These equations from each case are further described preceding case results.

```
58 # Case Selection
59 for caseN in range(0,3):
60     blkFlag = False # for holding plots open
61     if caseN == 0:
62         # Trig example
63         caseName = 'Sinusodial Example'
64         tStart =0
65         tEnd = 3
66         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
```

```
        findC = lambda x,y: y+np.sin(2*np.pi*x)
        c = findC(ic[0],ic[1])
        calcInt = ( 1/(2*np.pi)*np.cos(2*np.pi*tEnd)+c*tEnd -
                    1/(2*np.pi)*np.cos(2*np.pi*ic[0])-c*ic[0] )

    elif caseN == 1:
        # Exp example
        caseName = 'Exponential Example'
        tStart =0
        tEnd = 3
        numPoints = 3

        ic = [0,0] # initial condition x,y
        fp = lambda x, y: np.exp(x)
        f = lambda x,c: np.exp(x)+c
        findC = lambda x, y: y-np.exp(x)
        c= findC(ic[0],ic[1])
        calcInt = np.exp(tEnd)+c*tEnd-np.exp(ic[0])+c*ic[0]

    elif caseN == 2:
        # Log example
        caseName = 'Logarithmic Example'
        tStart =1
        tEnd = 4
        numPoints = 3
        blkFlag = True # for holding plots open

        ic = [1,1] # initial condition x,y
        fp = lambda x, y: 1/x
        f = lambda x,c: np.log(x)+c
        findC = lambda x, y: y-np.log(x)
        c= findC(ic[0],ic[1])
        calcInt = (tEnd*np.log(tEnd)- tEnd +c*tEnd -
                   ic[0]*np.log(ic[0])+ ic[0] -c*ic[0])
```

Figure 1.5: Case definitions.

A current value dictionary `cv` was created to mimic how PSLTDSim stores current values. Unlike PSLTDSim, the lists used to store values are not initialized to the full length they are expected to be. This requires 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: Creation of current value dictionary and logging lists.

The entire exact solution is then computed using the calculated 'f' function. The code enters a while loop that solves the differential equation for the next y value using the Euler, Runge-Kutta, and Adams-Bashforth methods. Resulting values are logged and 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 )

        # Calculate Adams-Bashforth result
        if len(t)>=2:
            cv['yAB'] = adams2( fp, cv['t'], cv['yAB'], t[-2], yAB[-2], ts )
        else:
            # Required to handle first step when a -2 index doesn't exist
            cv['yAB'] = adams2( fp, cv['t'], cv['yAB'], t[-1], yAB[-1], ts )

        # Log calculated results
```

```
        yE.append(cv['yE'])
        yRK.append(cv['yRK'])
        yAB.append(cv['yAB'])

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

Figure 1.7: Solution calculations.

When time progresses to the point that the while loop exits, a plot is generated that allows for comparison of the solution approximations. Each line color, legend label, and various other superficial options are defined before global plot output options are configured.

```
    # 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,yE,
            marker='o',
            fillstyle='none',
            linestyle=':',
            c=[0.7,0.7,0.7],
            label="Euler")
    ax.plot(t,yRK,
            marker='*',
            markersize=10,
            fillstyle='none',
            linestyle=':',
            c=[1,0,1],
            label="RK45")
    ax.plot(t,yAB,
            marker='s',
            fillstyle='none',
            linestyle=':',
            c =[0,1,0],
            label="AB2")

    # Format Plot
    fig.set_dpi(150)
    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     fig.tight_layout()
191     plt.show(block = blkFlag)
192     plt.pause(0.00001)
```

Figure 1.8: Result Plotting

After plotting, trapezoidal integration is 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.

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

Figure 1.9: Trapezoidal integration and result printing.

Sinusoidal Example and Results

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

Sinusoidal Example

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

Sinusoidal Example Integration

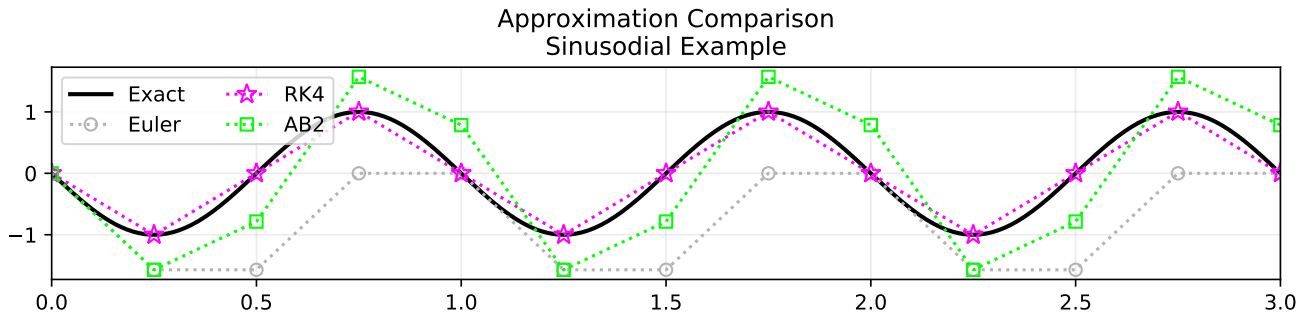

Figure 1.10: Quarter second approximation comparison of a sinusoidal function.

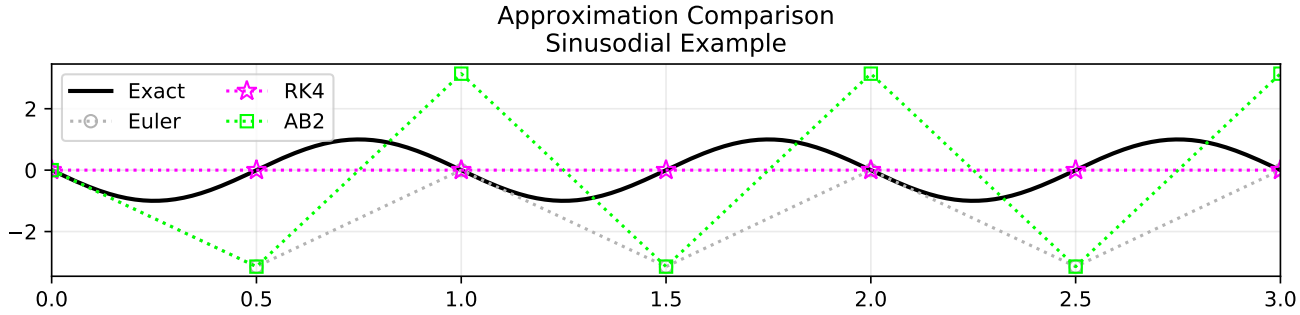

Figure 1.11: Half second approximation comparison of a sinusoidal function.

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

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

Exponential Example

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

Exponential Example Integration

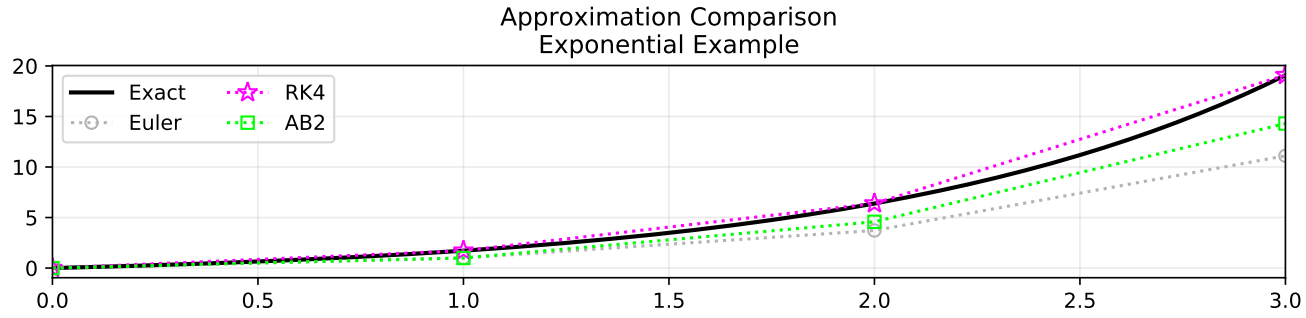

Figure 1.12: Approximation comparison of an exponential function.

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

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

Logarithmic Example

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

Logarithmic Example Integration

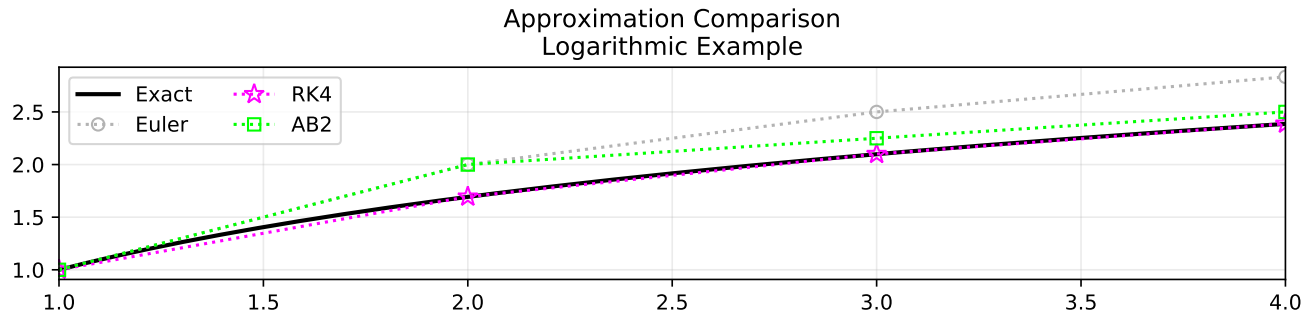

Figure 1.13: Approximation comparison of a logarithmic function.

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

Stuff is good, other stuff is bad - overall things are okay.

1.3.2 Python Approximation Comparisons

Code used to compare the Python lsim and solve_ivp method approximations to the exact and fourth order Runge-Kutta method is presented below. The code is very similar to the previously discussed comparison code and begins with package imports and method definitions. The solve_ivp function is imported from the integrate methods of Scipy, while the lsim function in part of the signal collection of functions.

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

# Method 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: Package imports and method 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 an lti system. Specifically, this input consisted of the numerator and denominator lists of descending powers $. Numerous transforms and calculus based mathematical methods found in [?, ?] and [?] were employed to calculate the exact function and integral. In the third order system case, partial fraction expansion was applied for a 'simpler' equation. Specific steps are described in the related case result sections.

```
# Case Selection
for caseN in range(0,3):
    blkFlag = False # for holding plots open

    if caseN == 0:
        # step input Integrator example
        caseName = 'Step Input Integrator Example'
        tStart =0
        tEnd = 4
        numPoints = 4

        U = 1
        initState = 0
        ic = [0,initState] # initial condition x,y
        fp = lambda x, y: 1
```

```
        f = lambda x, c: x+c
        findC = lambda x, y: y-x
        system = signal.lti([1],[1,0])
        calcInt = 0.5*(tEnd**2) # Calculated integral

    elif caseN == 1:
        # step input Low pass example
        caseName = 'Step Input Low Pass Example'
        tStart =0
        tEnd = 2
        numPoints = 4

        A = 0.25
        U = 1.0
        initState = 0
        ic = [0,initState] # initial condition x,y
        fp = lambda x, y: 1/A*np.exp(-x/A)# via table
        f = lambda x, c: -np.exp(-x/A) +c
        findC = lambda x, y : y+np.exp(-x/A)
        system = signal.lti([1],[A,1])
        calcInt = tEnd + A*np.exp(-tEnd/A)-A # Calculated integral

    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)])

        # 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)
        system = signal.lti(num,den)
        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: Comparison case definitions.

Initial conditions and log 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: Current and logging value initializations.

The exact solution and Runge-Kutta methods were handled as before, but Python function inputs require slightly different input. The lsim and solve_ivp outputs also require slightly different handling as their output is not just a single value. It should be noted that Python allows negative indexing of lists to return values at the end of a list.

```
135     # Calculate time step
136     ts = (tEnd-tStart)/numPoints
137     # Find C from integrated equation for exact soln
138     c = findC(ic[0], ic[1])
139     # Calculate exact solution
140     tExact = np.linspace(tStart,tEnd, 1000)
141     yExact = f(tExact, c)
142 
143     # Start Simulation
144     while cv['t']< tEnd:
145 
146         # Calculate Runge-Kutta result
147         cv['yRK'] = rk45( fp, cv['t'], cv['yRK'],  ts )
148 
149         # Runge-Kutta 4(5) via solve IVP.
150         soln = solve_ivp(fp, (cv['t'], cv['t']+ts), [cv['ySI']])
151 
152         # lsim solution
153         if cv['t'] > 0:
154             tout, ylsim, xlsim = signal.lsim(system, [U,U], [0,ts], xLS[-1])
155         else:
156             tout, ylsim, xlsim = signal.lsim(system, [U,U], [0,ts], initState)
157 
158         # Log calculated results
159         yRK.append(cv['yRK'])
160 
161         # handle solve_ivp output data
162         ySI += list(soln.y[-1])
163         tSI += list(soln.t)
164         cv['ySI'] = ySI[-1] # ensure correct cv
165 
166         # handle lsim output data
167         cv['yLS']=ylsim[-1]
168         yLS.append(cv['yLS'])
169         xLS.append(xlsim[-1]) # this is the state
170 
171         # Increment and log time
172         cv['t'] += ts
173         t.append(cv['t'])
```

Figure 1.17: Exact and approximate solution computations.

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

```
174     # Generate Plot
175     fig, ax = plt.subplots()
176     ax.set_title('Approximation Comparison\n' + caseName)
177 
178     #Plot all lines
179     ax.plot(tExact,yExact,
180             c=[0,0,0],
181             linewidth=2,
182             label="Exact")
183     ax.plot(t,yRK,
184             marker='*',
185             markersize=10,
186             fillstyle='none',
187             linestyle=':',
188             c=[1,0,1],
189             label="RK45")
190     ax.plot(tSI,ySI,
191             marker='x',
192             markersize=10,
193             fillstyle='none',
194             linestyle=':',
195             c=[1,.647,0],
196             label="solve_ivp")
197     ax.plot(t,yLS,
198             marker='+',
199             markersize=10,
200             fillstyle='none',
201             linestyle=':',
202             c ="#17becf",
203             label="lsim")
204 
205     # Format Plot
206     fig.set_dpi(150)
207     fig.set_size_inches(9, 2.5)
208     ax.set_xlim(min(t), max(t))
209     ax.grid(True, alpha=0.25)
210     ax.legend(loc='best',  ncol=2)
211     fig.tight_layout()
212     plt.show(block = blkFlag)
213     plt.pause(0.00001)
```

Figure 1.18: Result plotting.

```
214     # Trapezoidal Integration
215     exactI = trapezoidalPost(tExact,yExact)
216     SIint = trapezoidalPost(tSI,ySI)
217     RKint = trapezoidalPost(t,yRK)
218     LSint = trapezoidalPost(t,yLS)
219
220     print("\n%s" % caseName)
221     print("time step:  %.2f" % ts)
222     print("Method: Trapezoidal Int\t Absolute Error from calculated")
223     print("Calc: \t%.9f\t%.9f" % (calcInt ,abs(calcInt-calcInt)))
224     print("Exact: \t%.9f\t%.9f" % (exactI ,abs(calcInt-exactI)))
225     print("RK4: \t%.9f\t%.9f" % (RKint,abs(calcInt-RKint)))
226     print("SI: \t%.9f\t%.9f" % (SIint,abs(calcInt-SIint)))
227     print("lsim: \t%.9f\t%.9f" % (LSint,abs(calcInt-LSint)))
```

Figure 1.19: Trapezoidal integration comparison calculations.

Integrator Example and Results

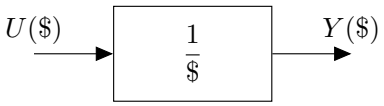

Figure 1.20: Integrator block.

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

Integrator Transform Example

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

Integrator Example Integration

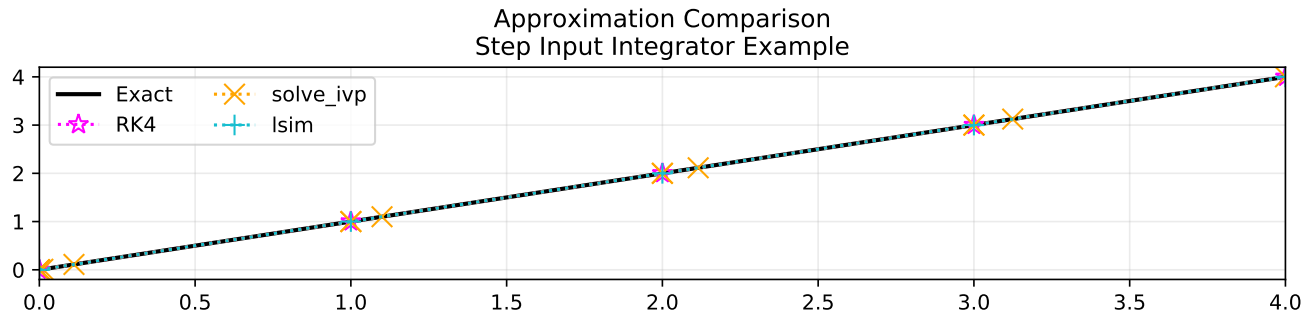

Figure 1.21: Approximation comparison of an integrator block.

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

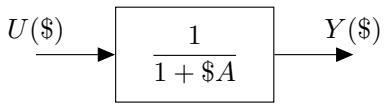

Figure 1.22: Low pass filter block.

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

Low Pass Transform Example

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

Low Pass Example Integration

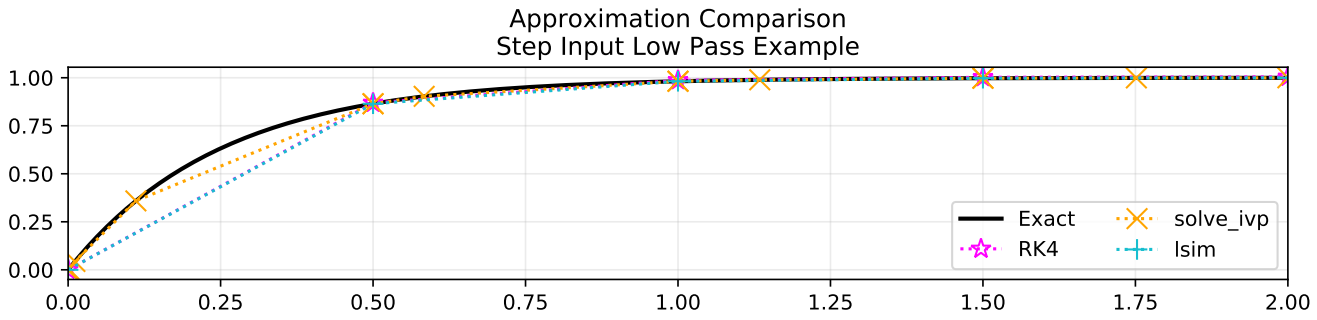

Figure 1.23: Approximation comparison of a low pass filter block.

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

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

Third Order Transform Example

$$\begin{aligned}\int y'(t)\ \mathrm{d}t = y(t) &= -\alpha\left(AT_0e^{-t/T_0} + BT_2e^{-t/T_2} + CT_4e^{-t/T_4}\right) + C_1 \\ C_1 &= y_0 + \alpha\left(AT_0e^{-t_0/T_0} + BT_2e^{-t_0/T_2} + CT_4e^{-t_0/T_4}\right) \\ \int_0^\tau y(t)\ \mathrm{d}t &= \alpha\left(AT_0^2e^{-t/T_0} + BT_2^2e^{-t/T_2} + CT_4^2e^{-t/T_4}\right) + C_1t\Big|_0^\tau\end{aligned} \tag{1.16}$$

Third Order Example Integration

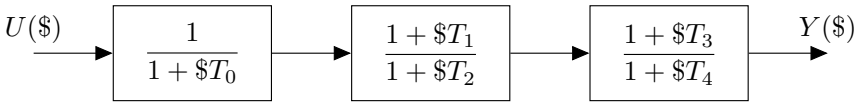

Figure 1.24: Third order system block diagram.

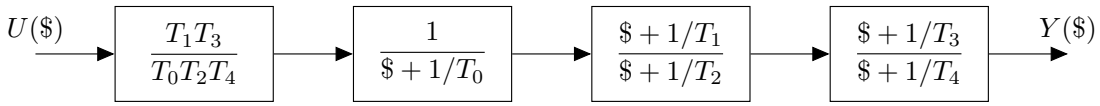

Figure 1.25: Modified third order system block diagram.

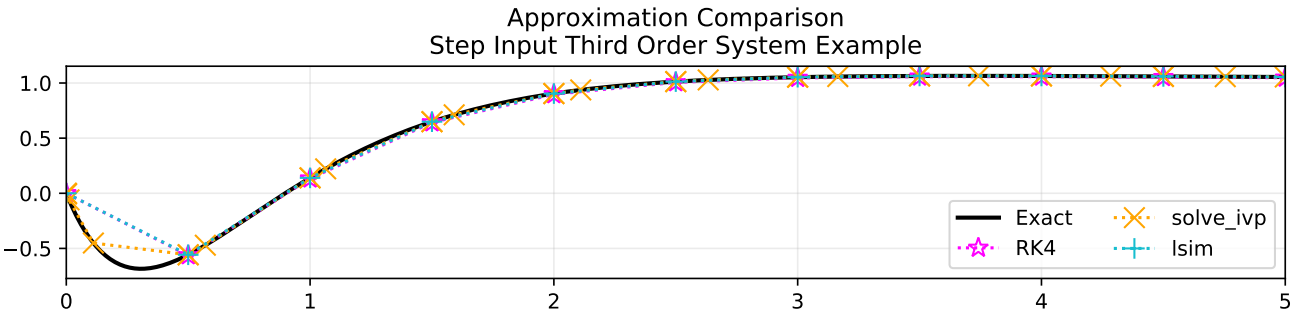

Figure 1.26: Third order approximation comparison using half second time step.

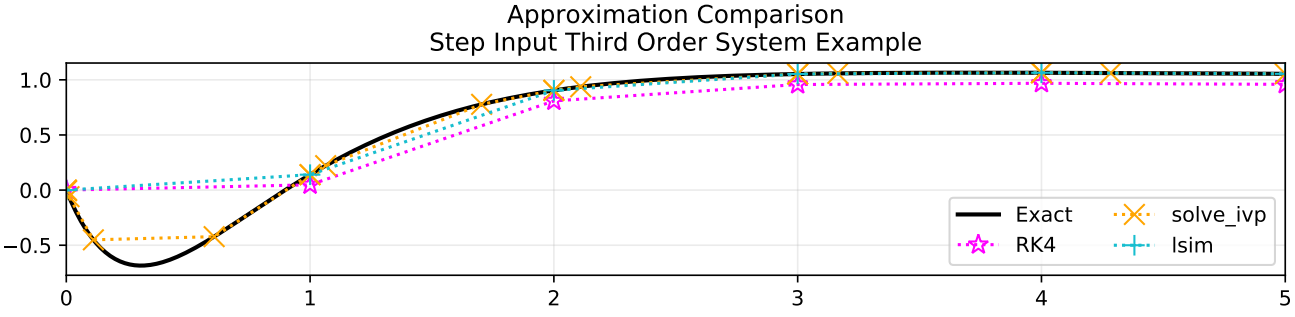

Figure 1.27: Third order approximation comparison using one second time step.

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

Stuff is good, other stuff is bad - overall things are okay.

1.4 Dynamic Agent Numerical Utilizations

This section is meant to better describe the handling of numerical methods by specific agents in PSLTDSim.

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, a full python definition shown in Figure 1.28 and explained below.

```
class WindowIntegratorAgent(object):
    """A window integrator that initializes a history of window
    values, then updates the total window area each step."""

    def __init__(self, mirror, length):
        # Retain Inputs / mirror reference
        self.mirror = mirror
        self.length = length # length of window in seconds

        self.windowSize = int(self.length / self.mirror.timeStep)

        self.window = [0.0]*self.windowSize
        self.windowNDX = -1 # so first step index points to 0

        self.cv = {
            'windowInt' : 0.0,
            'totalInt' : 0.0,
            }

```

```
20      def step(self, curVal, preVal):
21          # calculate current window Area, return value
22          self.windowNDX += 1
23          self.windowNDX %= self.windowSize
24
25          oldVal = self.window[self.windowNDX]
26          newVal = (curVal + preVal)/ 2.0 * self.mirror.timeStep
27
28          self.window[self.windowNDX] = newVal
29          self.cv['windowInt'] += newVal - oldVal
30          self.cv['totalInt'] += newVal
31
32          return self.cv['windowInt']
```

Figure 1.28: 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 index 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 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 1.29. 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', a 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.

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

Figure 1.29: Combined swing function definition.

1.4.3 Governor and Filter Agent Considerations

The lsim function was chosen for governor and filter dynamic calculation. This was meant to enable a consistent solution method for these agent types.

Integrator Wind Up

Non-linear system behavior must be handled outside of, or in between, the lsim solution as lsim only handles linear simulation. A common non-linear action is output limiting. An issue may arise when limiting a pure integrator and not addressing integrator wind up. The method for handling wind up is to check specific state and output values, then adjust any required variables.

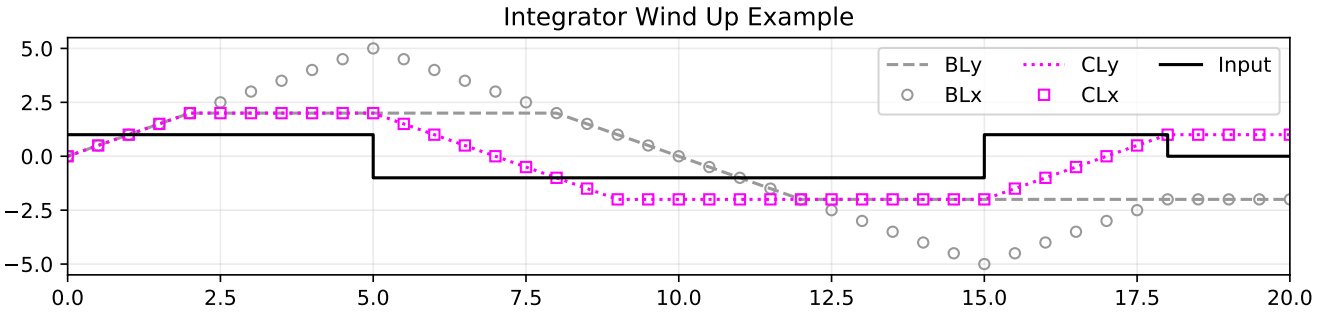

Figure 1.30: Effect of integrator wind up.

Combined System Comparisons

To allow for a variety of governor models without rewriting code, the technique of using a sequence of individual blocks for each part of a specific model was employed in the current governor models. Modeling differences due to interaction of states in multi-order systems represented by a series of single order systems was explored by creating two equivalent systems and simulating one using a combined multi-order transfer function, and the other a series of single blocks.

$$U(\$) \rightarrow \boxed{\frac{1}{1+\$T_0}} \rightarrow \boxed{\frac{(1+\$T_1)(1+\$T_3)}{(1+\$T_2)(1+\$T_4)}} \rightarrow Y(\$)$$

Figure 1.31: Third order system as two stages.

$$U(\$) \rightarrow \boxed{\frac{(1+\$T_1)(1+\$T_3)}{(1+\$T_0)(1+\$T_2)(1+\$T_4)}} \rightarrow Y(\$)$$

Figure 1.32: Third order system as single stage.

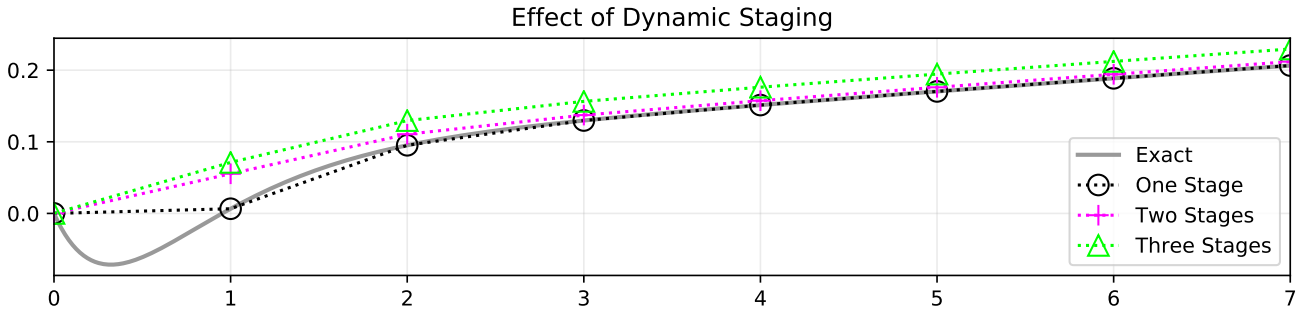

Figure 1.33: Effect of dynamic staging using one second time step.

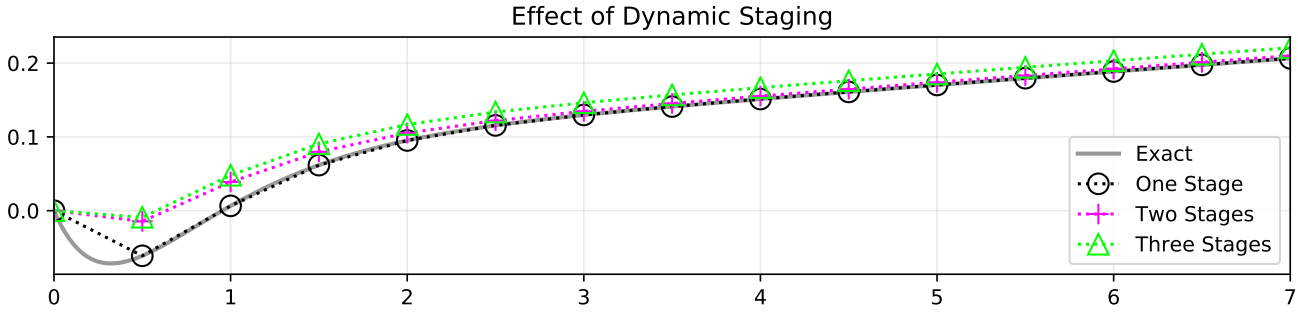

Figure 1.34: Effect of dynamic staging using half second time step.

1.4.4 Numerical Utilization Summary

Methods are used, some may be better than others, it is what it is.