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

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

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.

1.2.1 scipy.integrate.solve_ivp

The Scipy solve_ipv function is capable of numerically integrating ordinary differential equations using a variety of techniques.

- inputs
- operations
- outputs

1.2.2 scipy.signal.lsim

The Scipy function that simulates the output from a continuous-time linear system is called lsim. Input systems include Laplace transfer functions, zero pole gain form, and state space forms. Regardless of system input, the computation performed utilizes the state space solution that is centered around a matrix exponential.

- inputs
- operations
- outputs

1.3 Method Comparisons via Python Code Examples

To compare the resulting approximates from each method or function described above, a Python script was created. Full code is presented with explanations throughout.

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. Due to the lack of an accepted code listing format for this document, code is presented in figures that may span page breaks.

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

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

Figure 1.1: Code package imports.

Each function definition is 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.

```python
# 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

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 1.2: 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.

```
# Case Selection
for caseN in range(0,3):

    if caseN == 0:
        # Trig example
        caseName = 'Sinusodial Example'
        tStart =0
        tEnd = 1.5
        numPoints = 6
        blkFlag = False # for holding plots open

        ic = [0,0] # initial condition x,y
        fp = lambda x, y: -2*np.pi*np.cos(2*np.pi*x)
        f = lambda x,c: -np.sin(2*np.pi*x)+c
        findC = lambda x,y: y+2*np.pi*np.sin(2*np.pi*x)

        calcInt = 1/(2*np.pi)*np.cos(2*np.pi*1.5)-1/(2*np.pi)

    elif caseN == 1:
        # Exp example
        caseName = 'Exponential Example'
        tStart =0
        tEnd = 2
        numPoints = 4
        blkFlag = False # for holding plots open

        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)

        calcInt = np.exp(2)-3 # Calculated integral

    elif caseN == 2:
        # Log example
        caseName = 'Logarithmic Example'
        tStart =1
        tEnd = 3
        numPoints = 4
        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)

        calcInt = 3*np.log(3) # Calculated integral
```

Figure 1.3: 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],
        }

    # Calculate time step
    ts = (tEnd-tStart)/numPoints

    # 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 1.4: 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 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 1.5: 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.

```
    # 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)
    ax.set_xlim(min(t), max(t))
    ax.grid(True, alpha=0.25)
    ax.legend(loc='best',  ncol=2)
    fig.tight_layout()
    plt.show(block = blkFlag)
    plt.pause(0.00001)
```

Figure 1.6: Result Plotting

Finally, trapezoidal integration is performed on all results and compared to the calculated integral. It should be noted that the exact result 'only' uses 10,000 points.

```
    # Trapezoidal Integration
    exactI = trapezoidalPost(tExact,yExact)
    Eint = trapezoidalPost(t,yE)
    RKint = trapezoidalPost(t,yRK)
    ABint = trapezoidalPost(t,yAB)

    print("\nMethod: Trapezoidal Int\t Absolute Error from calculated")
    print("Exact: \t%.9f\t%.9f" % (exactI ,abs(calcInt-exactI)))
    print("RK4: \t%.9f\t%.9f" % (RKint,abs(calcInt-RKint)))
    print("AB2: \t%.9f\t%.9f" % (ABint,abs(calcInt-ABint)))
    print("Euler: \t%.9f\t%.9f" % (Eint,abs(calcInt-Eint)))
```

Figure 1.7: Trapezoidal integration and result printing.

Sinusoidal Results

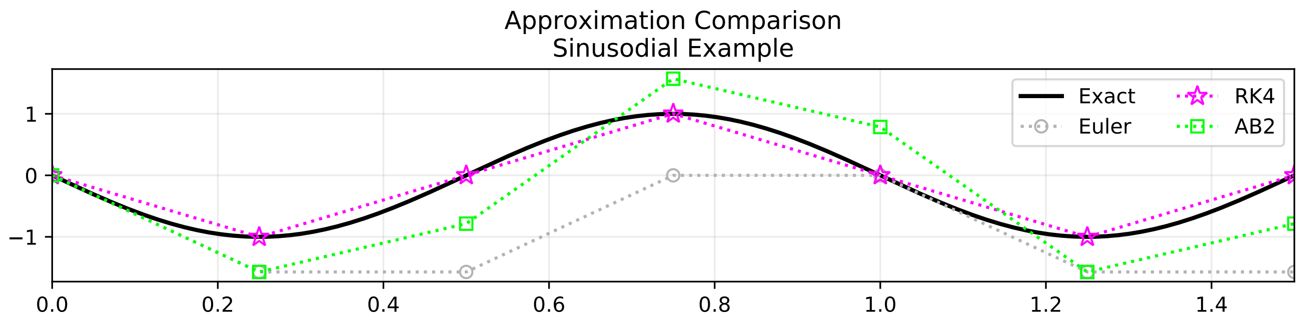

Figure 1.8: Approximation comparison of a sinusoidal function.

Exponential Results

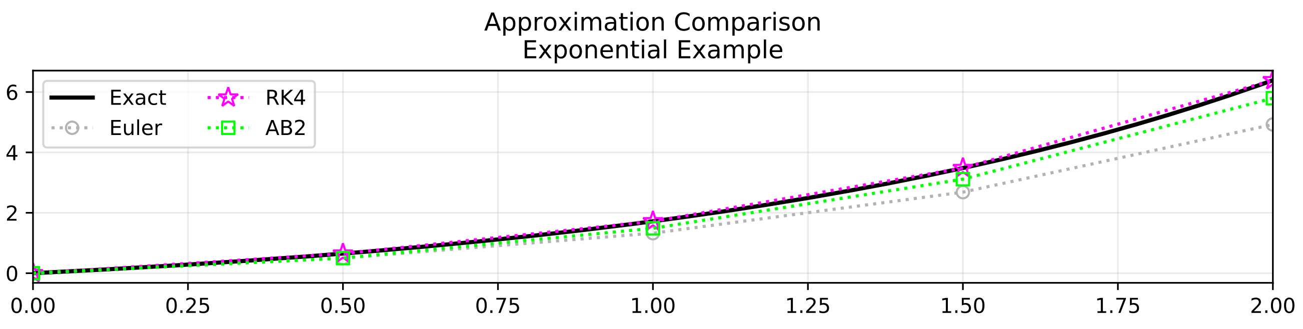

Figure 1.9: Approximation comparison of an exponential function.

Logarithmic Results

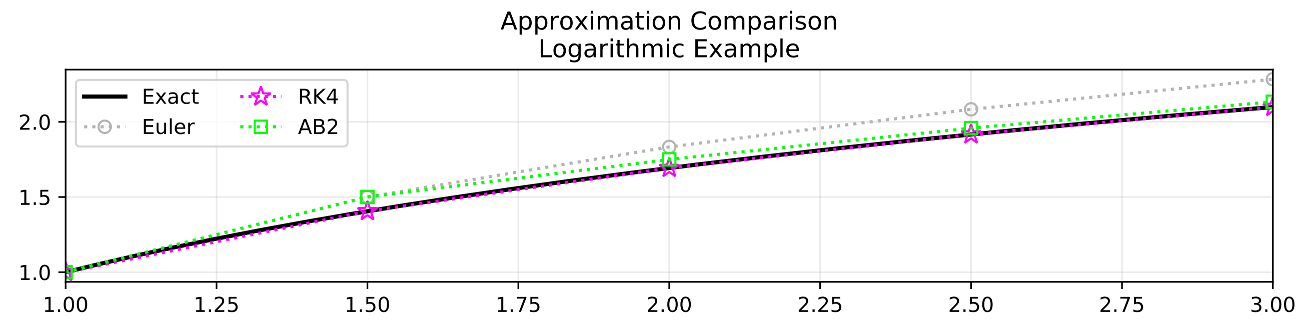

Figure 1.10: Approximation comparison of a logarithmic function.

1.3.2 Python Approximation Comparisons

The code used to compare the Python lsim and solve_ivp methodsto 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.

```python
# 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 1.11: Package imports and method definitions.

Case definitions were similar to the previous example with the addition of an lti system definition. The transfer function was used as input to create an lti system. Specifically, this input consisted of the numerator and denominator of descending $ powers. Numerous transforms and calculus based mathematical methods were employed to calculate the exact function and exact integral. In the third order system case, partial fraction expansion was required for a ‘simpler’ equation. Specific steps are described in the following case result sections.

```

# Case Selection
for caseN in range(0,3):

    if caseN == 0:
        # step input Integrator example
        caseName = 'Step Input Integrator Example'
        tStart =0
        tEnd = 4
        numPoints = 4
        blkFlag = False # for holding plots open

        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
```

```python

        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
        blkFlag = False # for holding plots open

        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 = 10
        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 1.12: 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],
        }

    # Calculate time step
    ts = (tEnd-tStart)/numPoints

    # 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 1.13: Current and logging value initializations.

The exact solution and Runge-Kutta methods were handled as before, but the Python function inputs require slightly different function input. The lsim and solve_ivp outputs also require slightly different handling as their output is not just a single value.

```
    # Find C from integrated equation for exact soln
    c = findC(ic[0], ic[1])

    # Calculate exact solution
    tExact = np.linspace(tStart,tEnd, 10000)
    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 1.14: Exact and approximate solution computations.

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

```
    # 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)
    fig.tight_layout()
    plt.show(block = blkFlag)
    plt.pause(0.00001)
```

Figure 1.15: Result plotting.

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

    print("\nMethod: Trapezoidal Int\t Absolute Error from calculated")
    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 1.16: Trapezoidal integration comparison calculations.

Integrator Results

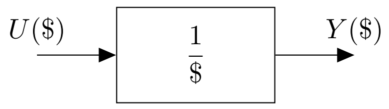

Figure 1.17: Integrator block.

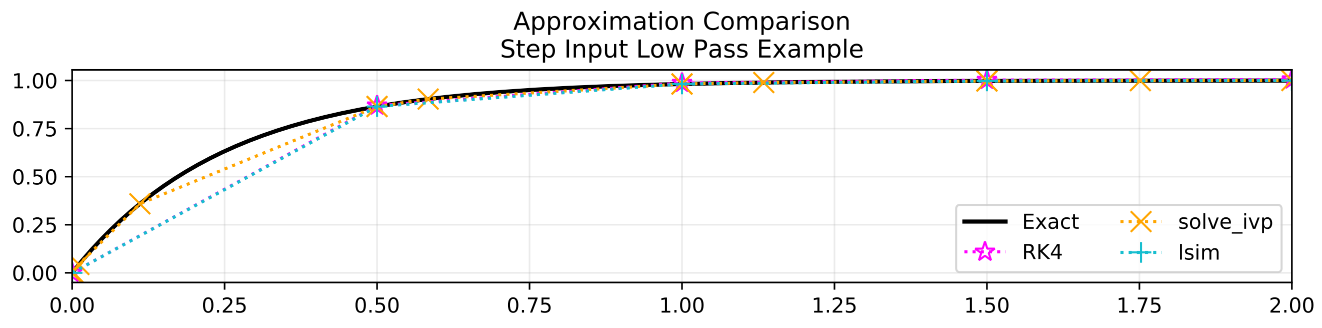

Figure 1.18: Approximation comparison of an integrator block.

Low Pass Results

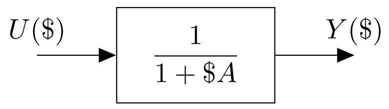

Figure 1.19: Low pass filter block.

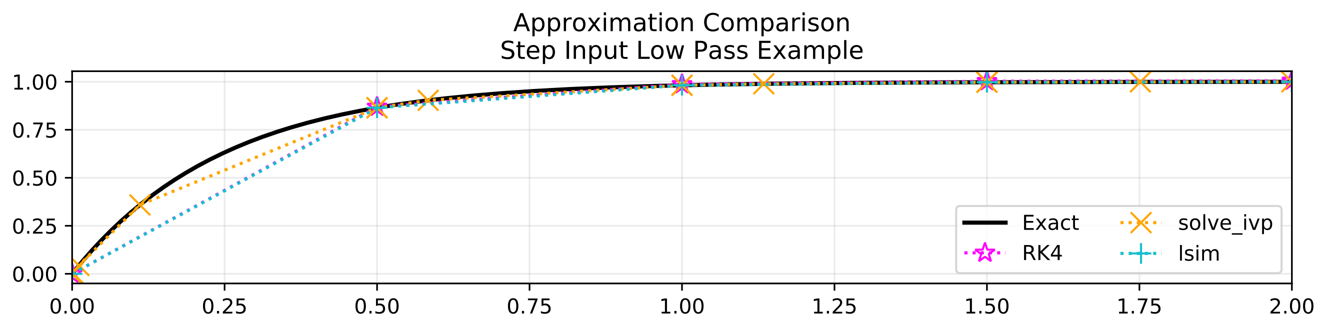

Figure 1.20: Approximation comparison of a low pass filter block.

Third Order System Results

$$\begin{tikzcd} U(\$) \arrow[r] & \boxed{\dfrac{1}{1+\$T_0}} \arrow[r] & \boxed{\dfrac{1+\$T_1}{1+\$T_2}} \arrow[r] & \boxed{\dfrac{1+\$T_3}{1+\$T_4}} \arrow[r] & Y(\$) \end{tikzcd}$$

Figure 1.21: Third order system block diagram.

$$\begin{tikzcd} U(\$) \arrow[r] & \boxed{\dfrac{T_1T_3}{T_0T_2T_4}} \arrow[r] & \boxed{\dfrac{1}{\$+1/T_0}} \arrow[r] & \boxed{\dfrac{\$+1/T_1}{\$+1/T_2}} \arrow[r] & \boxed{\dfrac{\$+1/T_3}{\$+1/T_4}} \arrow[r] & Y(\$) \end{tikzcd}$$

Figure 1.22: Modified third order system block diagram.

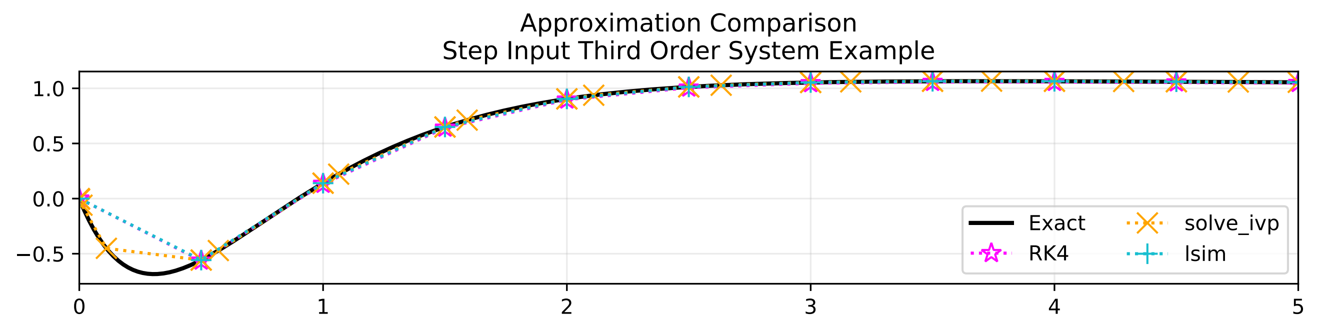

Figure 1.23: Approximation comparison of third order system.

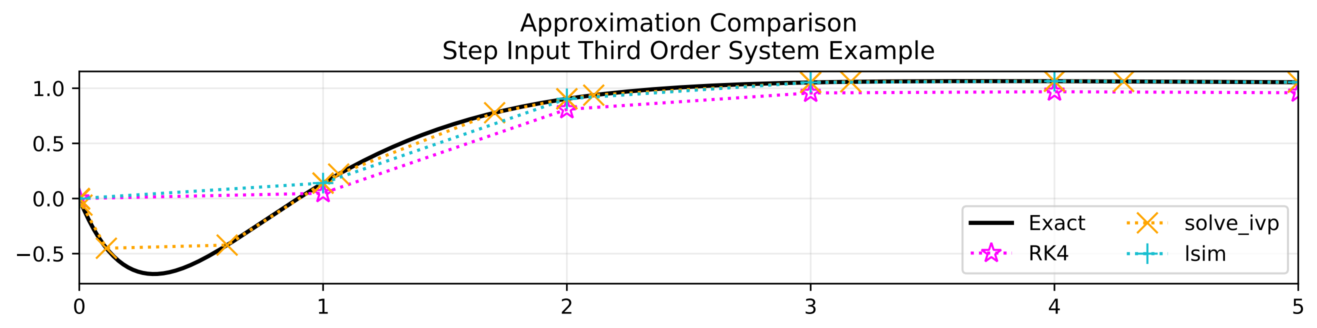

Figure 1.24: Third order system using 1 second time step.

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 1.25 and explained below.

```python
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,
            }

    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 1.25: 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 1.26. 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.

```python
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 1.26: Combined swing function definition.

1.4.3 Governor and Filter Agents

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

Laplace to State Space Transforms

As state space systems are not unique, and the handling of states is of vital importance when using state space methods, the input system to the lsim function was chosen to be a known state space system. This avoided verifying the automatic transform from a transfer function to a state space system. Examples of how PSLTDSim converted various tranfer functions to state space models are presented below. While only first order models are shown, automatic transformation may include gains that change how states are handled.

Integrator Example Basic integrator to SS

$$\frac{Y(\$)}{U(\$)} = \frac{1}{\$} \tag{1.5}$$

Integrator Transfer Function to State Space

Low Pass Filter Example low pass to SS

$$\frac{Y(\$)}{U(\$)} = \frac{1}{1+\$A} \tag{1.6}$$

Low Pass Transfer Function to State Space

Lead-Lag Example lead-lag to SS

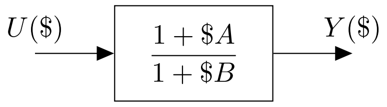

Figure 1.27: Lead-lag filter block.

$$\frac{Y(\$)}{U(\$)} = \frac{1+\$A}{1+\$B} \tag{1.7}$$

Lead-Lag Transfer Function to State Space

PI Controller Example PI to SS

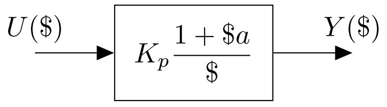

Figure 1.28: PI filter block.

$$\frac{Y(\$)}{U(\$)} = K_p\frac{1+\$a}{\$} \tag{1.8}$$

PI Transfer Function to State Space

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.

Pure Time Delay

To Achieve a pure time delay....

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.