Numerical Techniques

PSLTDSim utilizes a variety of numerical techniques to perform integration involved with the differential equations handled by dynamic agents. Some of the employed methods are coded 'by hand', while others utilize Python packages. This appendix is meant to introduce some standard numerical integration techniques, provide information about some Python functions that perform numerical integration, compare results of numerical methods via examples, and briefly explain how dynamic agents utilize the explained techniques.

1.1 Classical Techniques

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 functions presented below were adapted from [**?**].

1.1.1 Euler Method

Of the of integration methods available to solve system frequency, the Euler method is the simplest. In general terms, to find the next y value given some 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 shown as 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 midpoints.

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 approximation 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 previous data.

1.1.4 Trapezoidal Integration

To integrate known values generated each time step, a trapezoidal integration method is used. 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. This technique is used in a variety of agents, such as the BA agent for IACE, and the window integrator agent.

1.2 Python Functions

To allow for more robust solution methods, various Python functions were employed in PSTLDSim. 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
- outputs
- handling of 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
- outputs
- handling of outputs

1.3 Method Comparisons via Python Code Examples

To compare the resulting approximates from each method a Python script was created.

1.3.1 Classical Comparison Results

- Compare classical numerical techniques (euler, RK, AB, AND trapezoidal vs exact integration)

Sinusoidal Example

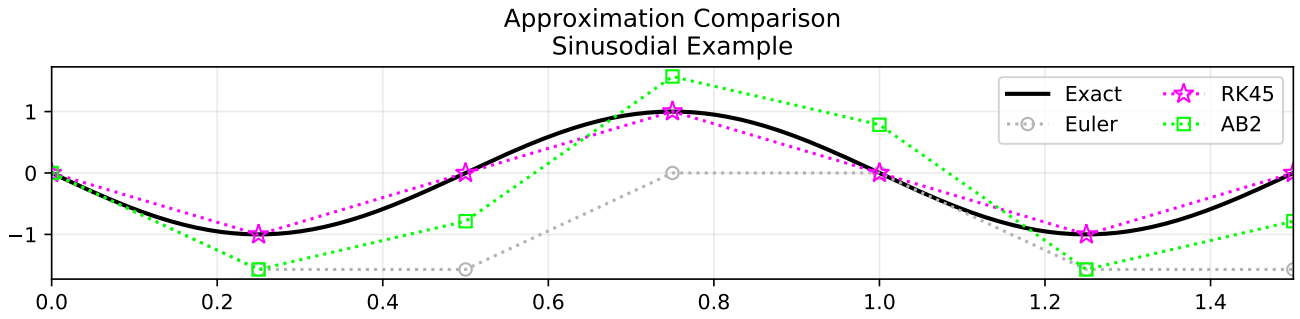

Figure 1.1: Approximation comparison of a sinusoidal function.

Exponential Example

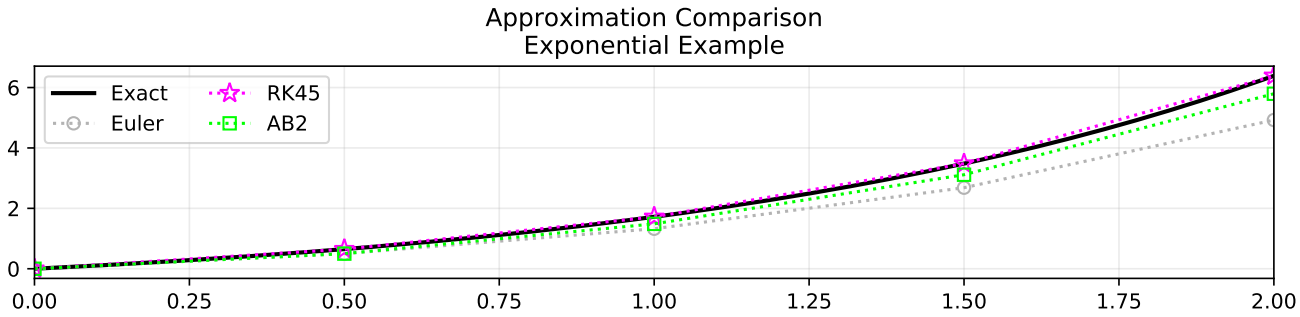

Figure 1.2: Approximation comparison of an exponential function.

Logarithmic Example

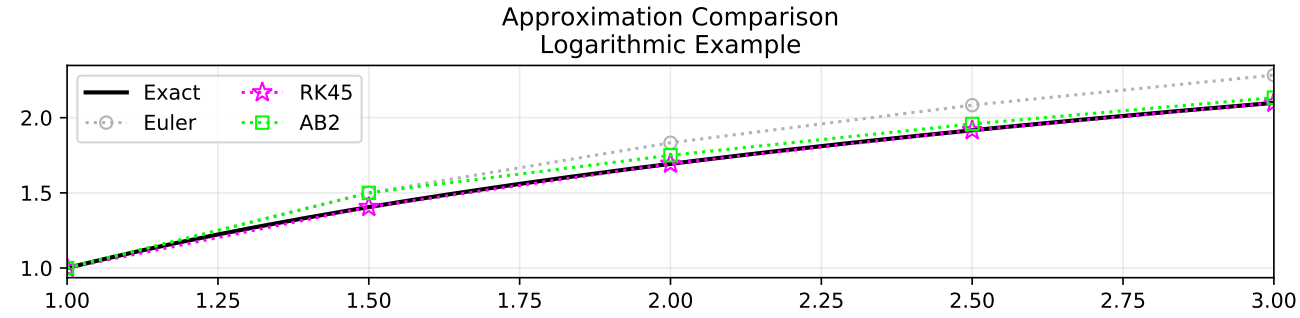

Figure 1.3: Approximation comparison of a logarithmic function.

1.3.2 Python Function Comparison Results

- Compare Python results to RK (again check trapezoidal integration)

Integrator Example

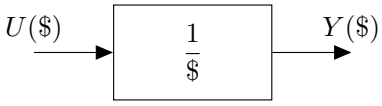

Figure 1.4: Integrator block.

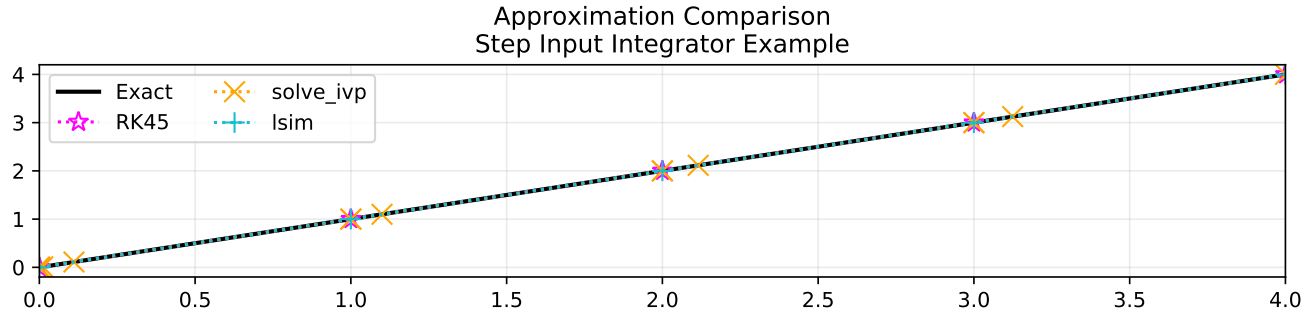

Figure 1.5: Approximation comparison of an integrator block.

Low Pass Example

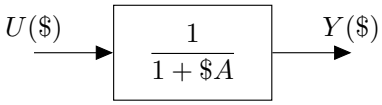

Figure 1.6: Low pass filter block.

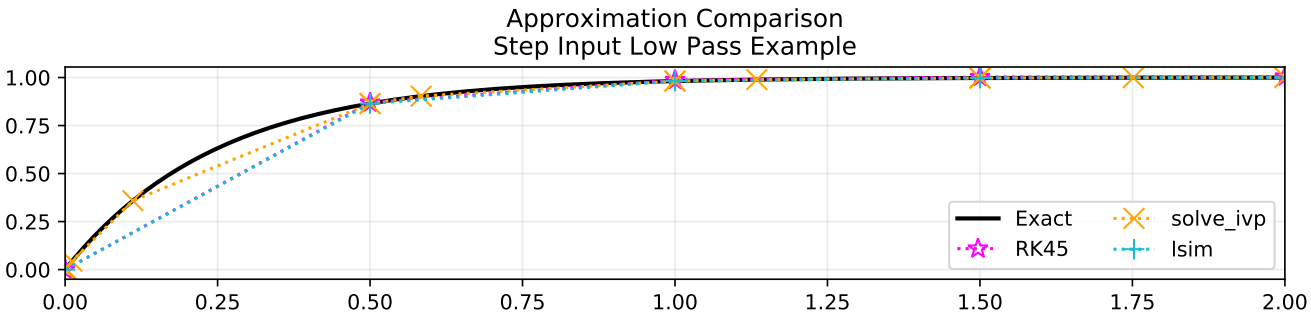

Figure 1.7: Approximation comparison of a low pass filter block.

Third Order System Example

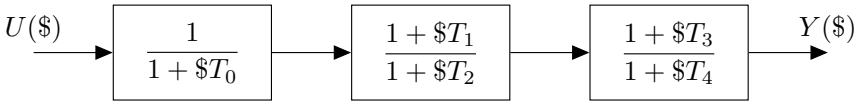

Figure 1.8: Third order system block diagram.

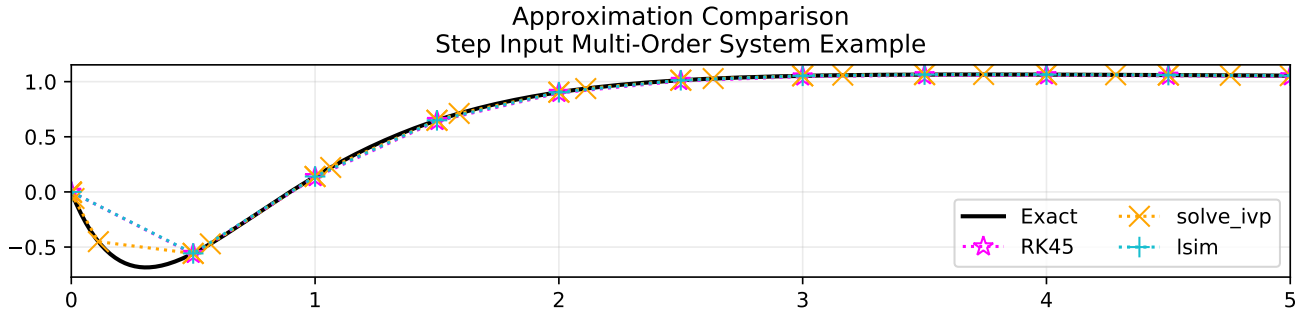

Figure 1.9: Approximation comparison of third order system.

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 1.10 and explained below.

The agent is initialized by any agent that is desired to perform window integration and supplied with a reference to the system mirror and window length in seconds. The reference to the system mirror is stored and then 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 is transformed 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 values 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.

```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 1.10: Window integrator definition.

1.4.2 Combined Swing Equation

The full code for the combined swing equation is presented in Figure 1.11 and described in this section.

```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 1.11: Combined swing equation definition.

utilize the hand written Euler or Adams-Bashforth methods, or the Runge-Kutta 4(5) method

supplied by solve_ivp. 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.

1.4.3 Governor and Filter Agents

The lsim function was decided upon for governor and filter dynamics. 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 to solution understanding, the input system to the lsim function is was chosen to be a known state space system. This avoided verifying the automatic transform from a transfer function to a state space system.

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

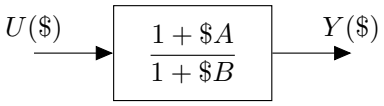

Figure 1.12: Lead-lag filter block.

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

Lead-Lag Transfer Function to State Space

PI Controller Example PI to SS

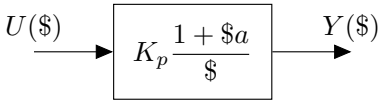

Figure 1.13: PI filter block.

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

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. The method used for handling this is to check specific state and output values and adjust as necessary using if statements.

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 block systems and simulating one using a combined multi-order transfer function.