Non-Linear Elements

Assignment 8

Due: 2021/04/19

1 Introduction

In this final assignment, we explore simulations of non-linear devices and the critical damping adjustment (CDA) technique to eliminate numerical oscillations that can occur with the trapezoidal discretization method. Numerical oscillations occur with this discretization technique when it encounters discontinuous inputs into its system, as we've seen throughout this class. CDA resolves this issue by dynamically changing the discretization method when a discontinuity is detected to a more stable one, and then reverting back to trapezoidal. To demonstrate its effectiveness, a non-linear inductance with saturating current is modeled and simulated in this assignment. The flux/current relationship of the non-linear inductance is approximated using a piecewise function which has discontinuities where its curves connect. A trapezoidal discretization with and without CDA will be performed to see how it improves the performance of the simulation.

2 Setup

Shown in Figure 1 is the flux linkage to current relationship that is used in both the hand-implemented and PSCAD simulations. It emulates a continuous curve which describes how inductance changes value as the current through it increases. This is realistic, non-linear behaviour that occurs in inductors which have cores. The core material reaches a point where it can no longer linearly increase the amount of the magnetic flux generated by the current passing through the inductor's coil.

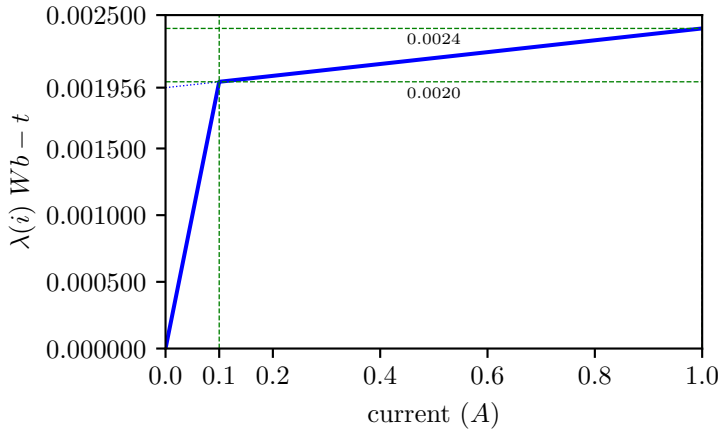

Figure 1: Piecewise flux linkage vs. current relationship for simulated non-linear, saturating inductance

Due to this piecewise approximation, the flux linkage current relationship features a sharp change in slope at $0.1A$. Trapezoidal discretization methods can struggle with discontinuities like this if the step size is not sufficiently small. The CDA method addresses this by changing the discretization method to backward Euler for two half steps. Trapezoidal discretization resumes after these two half steps.

The following sections will discuss both the hand-implemented nodal analysis simulation and the PSCAD implementation of the same circuit. Efforts were made to ensure that both PSCAD and hand-implemented simulations matched as closely as possible. This required careful alignment on things such as the main switch resistance as well as initial conditions to the circuit components.

2.1 Hand-Implemented Solution

As was implemented back in assignment 2, this assignment's hand-implemented simulation is derived using nodal analysis of the circuit. Shown in Figure 2 is the assignment's circuit schematic, with the nodes used for nodal analysis. The switch is turned into a variable resistance R_{sw} which the nodal analysis solution changes to a very small value when the switch closes to match PSCAD's switch implementation. The non-linear inductance L is converted into its equivalent resistance R_L as well as a current source h_L representing its discretization history.

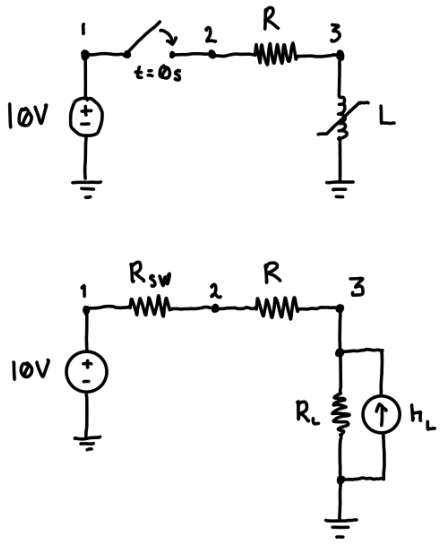

Figure 2: Schematic for circuit used in this assignment and its discretized equivalent

Since the CDA technique requires that the discretization method fall back to backward Euler at discontinuities, the hand-implemented simulation implements both the trapezoidal and backward Euler discretization methods. The following steps demonstrate how each discretization was applied to the non-linear inductor's voltage and flux linkage relationship:

$$v(t) = \frac{\lambda(t)}{dt} \tag{1}$$

$$\int_{t-\Delta t}^{t} v(t)dt = \int_{t-\Delta t}^{t} d\lambda(t) \tag{2}$$

$$\text{approximated area of integration} \simeq \lambda(t) - \lambda(t-\Delta t) \tag{3}$$

$$\tag{4}$$

Trapezoidal:

$$\frac{v(t) + v(t-\Delta t)}{2} = \lambda(t) - \lambda(t-\Delta t) \tag{5}$$

$$v(t) = \frac{2}{\Delta t}\left[\lambda_{kn} + L_n i(t)\right] - v(t-\Delta t) - \frac{2}{\Delta t}\lambda(t-\Delta t) \tag{6}$$

$$\boxed{v(t) = \frac{2L_n}{\Delta t} i(t) + \left[-v(t-\Delta t) - \frac{2}{\Delta t}\lambda(t-\Delta t) + \frac{2}{\Delta t}\lambda_{kn}\right]} \tag{7}$$

Backward Euler:

$$v(t)\Delta t = \lambda(t) - \lambda(t-\Delta t) \tag{8}$$

$$v(t) = \frac{1}{\Delta t}\left[\lambda_{kn} + L_n i(t)\right] - \frac{1}{\Delta t}\lambda(t-\Delta t) \tag{9}$$

$$\boxed{v(t) = \frac{L_n}{\Delta t} i(t) + \left[-\frac{1}{\Delta t}\lambda(t-\Delta t) + \frac{1}{\Delta t}\lambda_{kn}\right]} \tag{10}$$

In both derivations, the flux linkage knee value λ_{kn} and inductance L_n are obtained from the flux linkage current plot shown in Figure 1. The simulation detects which region the inductor is currently in and will choose the values for each accordingly.

Using the derived equations above, the equivalent resistance R_L and voltage source eh_L can be obtained to represent the equivalent discretized circuit. Source transformation is used to change the equivalent voltage source to a current source. The following equations represent the equivalent resistances and history sources (using the node naming convention shown in Figure 2:

$$R_{L_{trap}} = \frac{2L_n}{\Delta t} \tag{11}$$

$$R_{L_{euler}} = \frac{L_n}{\Delta t} \tag{12}$$

$$eh_{L_{trap}}(t) = -v_3(t) - \frac{2}{\Delta t}\lambda(t) + \frac{2}{\Delta t}\lambda_{kn} \tag{13}$$

$$eh_{L_{euler}}(t) = -\frac{1}{\Delta t}\lambda(t) + \frac{1}{\Delta t}\lambda_{kn} \tag{14}$$

$$h_{L_{trap}}(t) = \frac{eh_{L_{trap}}(t)}{R_{L_{trap}}} \tag{15}$$

$$h_{L_{euler}}(t) = \frac{eh_{L_{euler}}(t)}{R_{L_{euler}}} \tag{16}$$

Once the current source histories are determined, the branch current, which is required for calculating the next nodal voltage values, can be calculated using the following relationship (where the appropriate discretization resistance and current source history is substituted):

$$i_L(t) = v_3(t)/R_L - h_L(t) \tag{17}$$

For this exercise, the convention used for currents leaving the node marks them as positive and currents entering the node marks them as negative. Therefore the resulting branch currents are the nodal voltage divided by the branch resistance, minus the discretized component current sources for all branch currents in this circuit.

Finally, we can now calculate the nodal voltage equations using some clever matrix manipulation taught in class. This technique subdivides the usual nodal analysis conductance matrix into four sections, with nodes connected to known voltage sources occupying the outside matrices $\mathbf{G_{AB}}$, $\mathbf{G_{BA}}$, and $\mathbf{G_{BB}}$. This allows us to perform the following operation to obtain our nodal voltage equation vector:

$$[\mathbf{V_A(t)}] = [\mathbf{G_{AA}}]^{-1}[\mathbf{h_A(t)}] - [\mathbf{G_{AA}}]^{-1}[\mathbf{G_{AA}}][\mathbf{V_B(t)}] \tag{18}$$

For this implementation, the main switch closing and opening has the effect of changing the resistance of R_{sw}. Shown below are both conductance and history matrices used for this circuit's solution:

$$[\mathbf{G}] = \left[\begin{array}{c:c} \mathbf{G_{AA}} & \mathbf{G_{AB}} \\ \hdashline \mathbf{G_{BA}} & \mathbf{G_{BB}} \end{array}\right] = \left[\begin{array}{cc:c} \frac{1}{R_{sw}} & -\frac{1}{R_{sw}} & 0 \\ -\frac{1}{R_{sw}} & \frac{1}{R_{sw}} + \frac{1}{R} & -\frac{1}{R} \\ \hdashline 0 & -\frac{1}{R} & \frac{1}{R} + \frac{1}{R_L} \end{array}\right] \tag{19}$$

$$[\mathbf{H}] = \left[\begin{array}{c} \mathbf{H_A} \\ \hdashline \mathbf{H_B} \end{array}\right] = \left[\begin{array}{c} 0 \\ h_L(t) \\ \hdashline 0 \end{array}\right] \tag{20}$$

The resulting nodal voltage equations were generated using MATLAB, whose source can be viewed in Listing 5.4. The resulting equations were then ported to the final Python script which can be viewed in Listing 5.1.

CDA is implemented in the hand-implemented solution by switching between the trapezoidal and backward Euler next voltage equations at the appropriate moment. A function named `get_flux(current_input)` was created to facilitate this. Current is passed into the `current_input` parameter, and the function returns a tuple with the associated flux linkage value, flux linkage knee value, and inductance for the region. This allows for each iteration of the nodal analysis solution to use an up-to-date value as needed (based on the previous iteration's current). When a change of region is detected (by keeping track of the inductance from iteration to iteration), the backward Euler discretization equations are used for two half Δt steps. After these two steps, the trapezoidal discretization equations are used once again.

2.2 PSCAD Simulation

The schematic diagram for the PSCAD solution can be found in Figure 3. Its main component is the variable inductance component, which is able to accept as an input an inductance value. By changing the inductance

value with respect to the current going through the inductor, a non-linear saturating inductance can be emulated for the simulation.

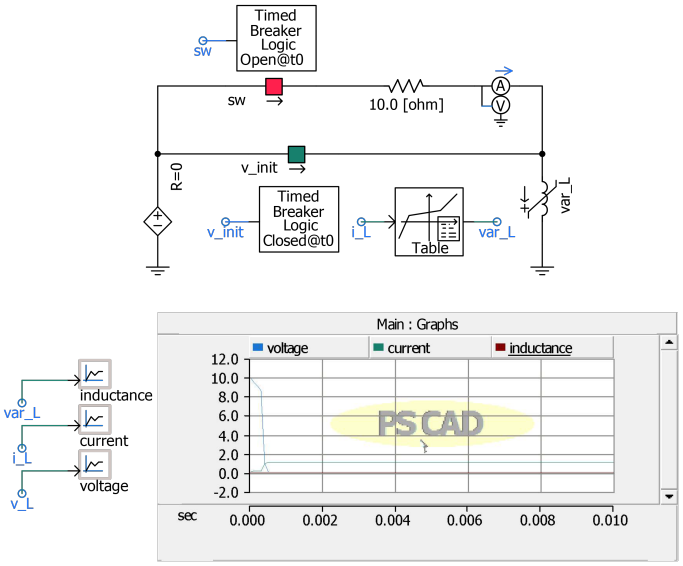

Figure 3: Schematic for PSCAD implementation of the assignment circuit

To implement the flux linkage current relationship, PSCAD's X-Y transfer function component is used to relate the current input to an inductance output. Conveniently, one can derive the inductance from the flux linkage current relationship as the inductance is simply the slope of the curve. Since we only have two linear piecewise components, this translates to two inductances: from 0 to 0.1, 0.02 Henries and from 0.1 onwards, 4.444e-4 Henries. Using this relationship, the inductance will track the current passing through the inductor as outlined by the flux linkage current plot.

In order to set the initial voltage on the inductor to be 10V, a connection is made from the 10V source to a breaker and then to the inductor. The breaker is set to open at t = 0s, which tells PSCAD that only at t = 0s does the inductor have 10V applied to it, giving it an initial value of 10V. The main switch is also used in a similar way, except that at t = 0s it is set to close instead of open.

As with the previous assignments using PSCAD, the required values are tapped off of their variable names and plotted directly on a single plot within the schematic. Right-clicking the plot allows one to export the data to the clipboard, which is in the form of comma separated values. For this assignment, this data was pasted into a dedicated `.csv` file which can be found in Listings 5.2 and 5.3.

2.3 Plot Generation

The plots were all generated using the matplotlib python library. The python script listed in Listing 5.1 is responsible for generating all of the plots in this assignment. In order to get a good understanding of how CDA affected the results, the python script plots the following curves:

- Continuous solution (ideal inductance)
- Continuous solution (non-linear inductance)
- Trapezoidal discretized solution (ideal inductance)

- Trapezoidal discretized solution (non-linear inductance)
- Trapezoidal discretized solution + CDA (non-linear inductance)

The continuous solutions were based on the ones from the first assignment. The non-linear inductance version of the continuous solution uses the flux linkage current relationship to dynamically change the inductor value during the simulation. The trapezoidal solution for the ideal inductance also uses the same implementation from the first assignment. For this assignment, however, the trapezoidal solutions differ from the first assignment as they implement nodal analysis. For the nodal analysis trapezoidal solutions, both model a non-linear inductance. The analysis is done twice, before and after applying CDA; this is to observe the effects the CDA has on the solution generated by the simulation.

3 Results

The following section contains the plots generated for this assignment. Two values of Δt were chosen for plotting because $\Delta t = 0.0001s$ did not show much numerical instability. By increasing the time step to $\Delta t = 0.0003s$, the numerical instability becomes much more noticeable, and therefore the CDA improvement becomes much clearer. The plots are also zoomed in in order to appreciate the detail that occurs when the discontinuity is encountered in the flux linkage current function.

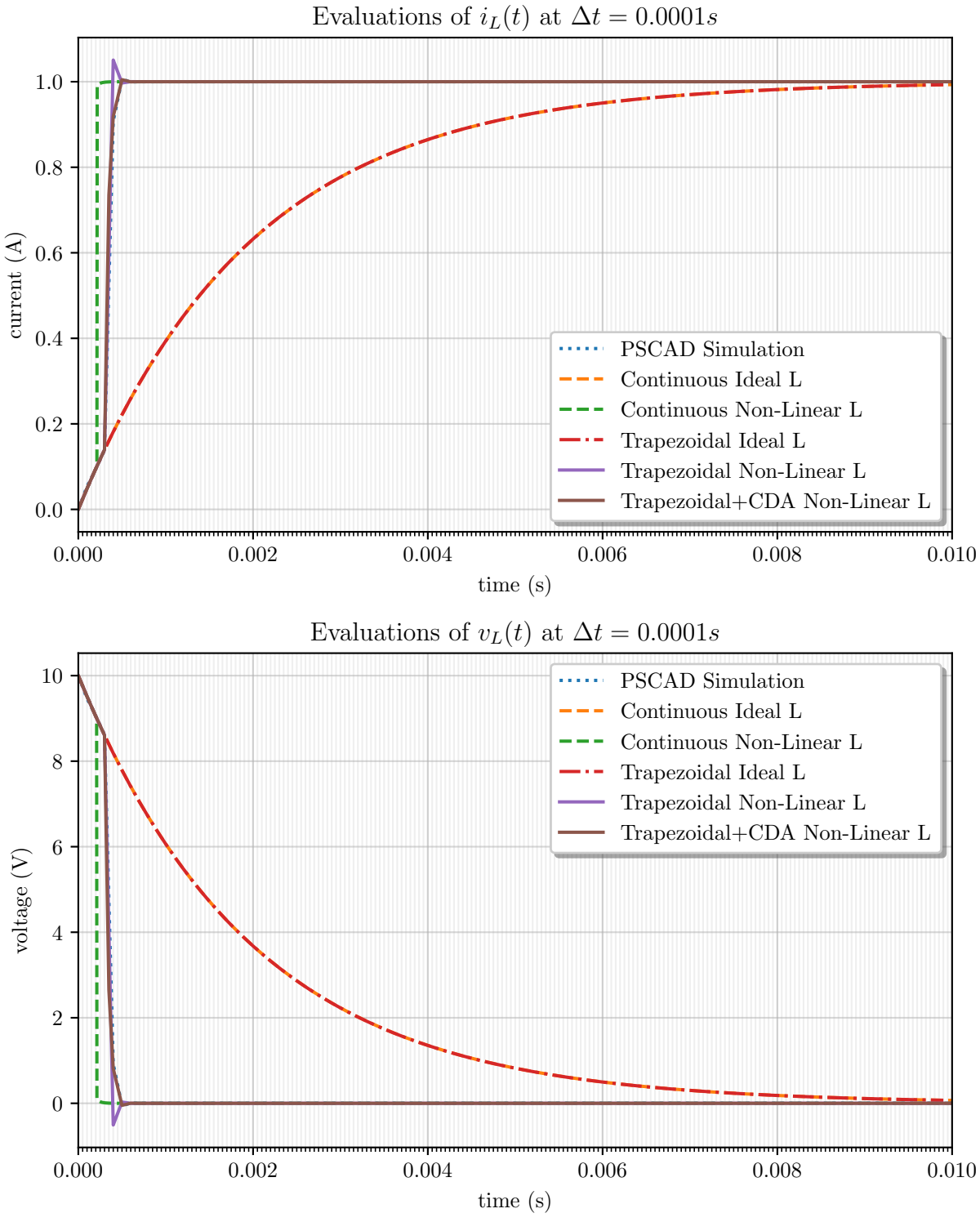

Figure 4: Current and voltage simulation results for $\Delta t = 100\mu s$

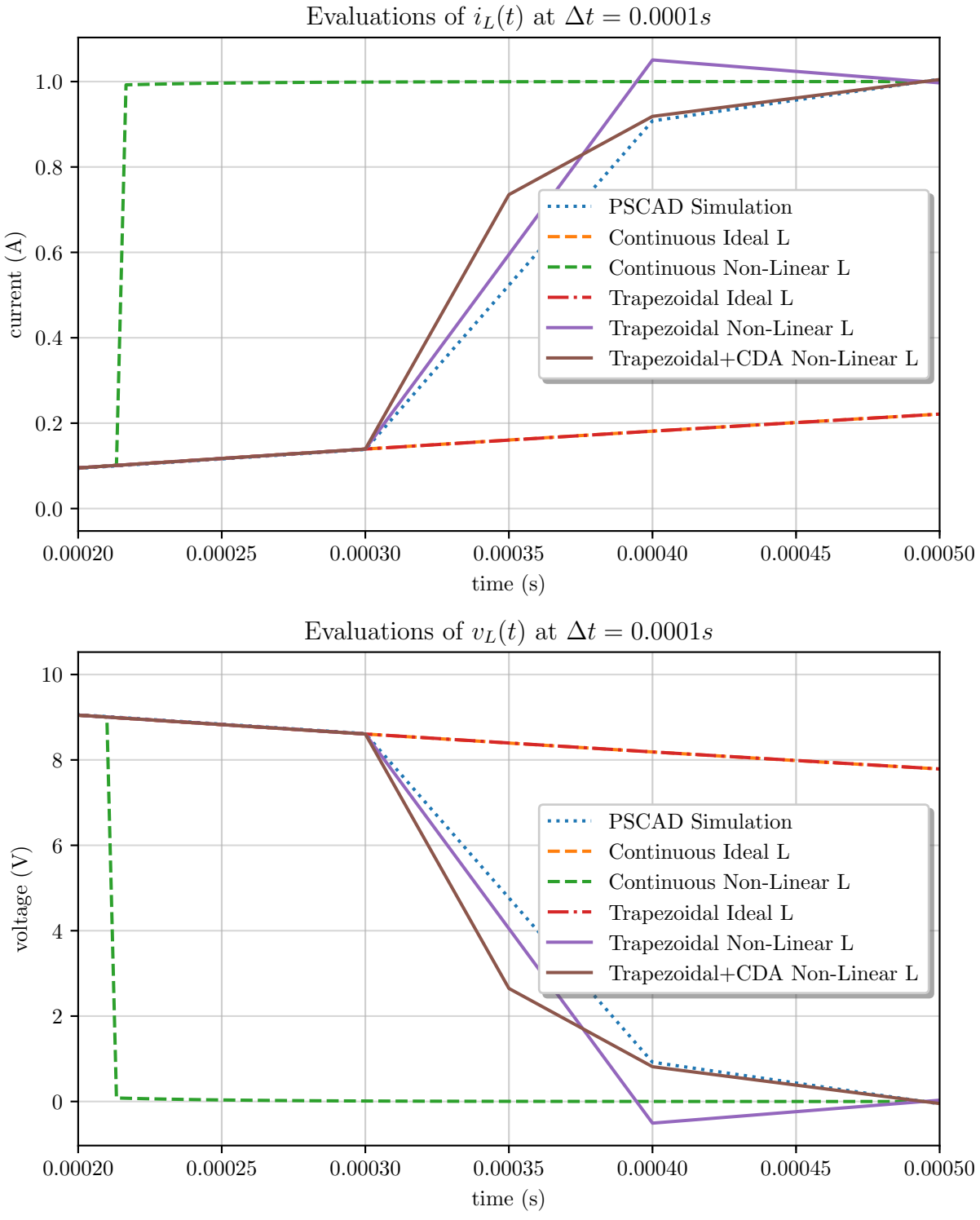

Figure 5: Zoomed in current and voltage simulation results for $\Delta t = 100\mu s$

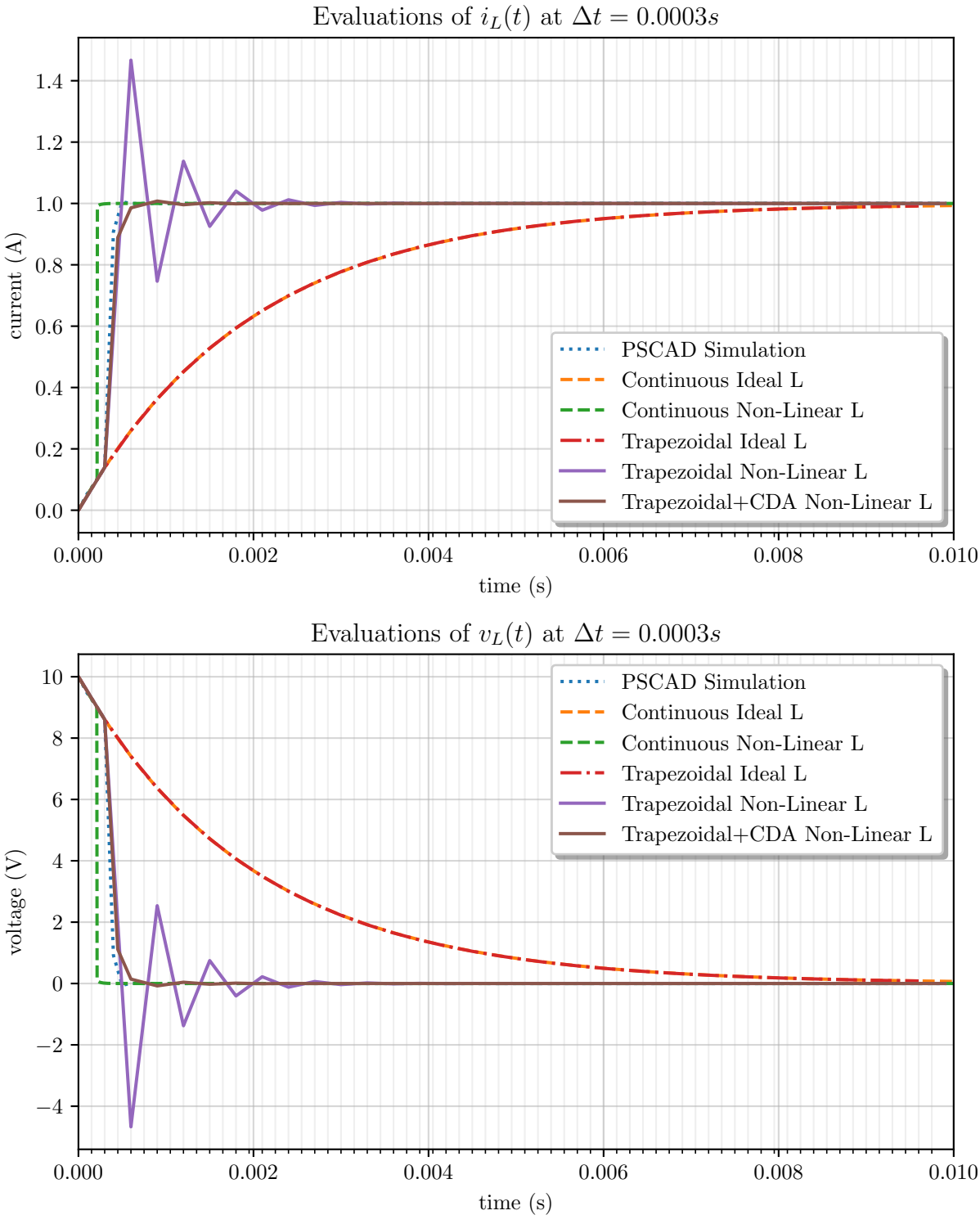

Figure 6: Current and voltage simulation results for $\Delta t = 300\mu s$

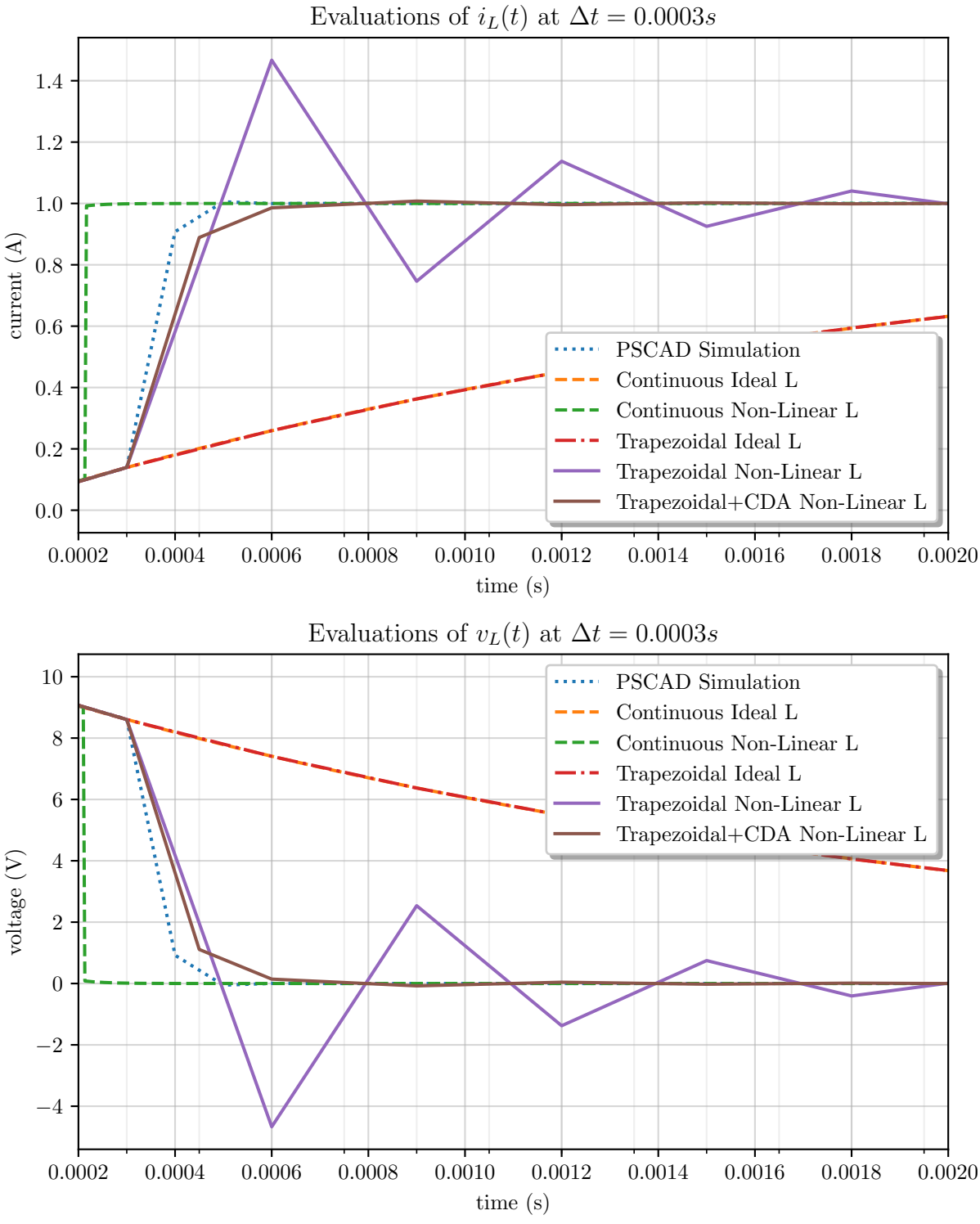

Figure 7: Zoomed in current and voltage simulation results for $\Delta t = 300\mu s$

4 Discussion

- When comparing to PSCAD, we see that the trapezoidal solution with CDA compares favorably. Both show numerical stability at the discontinuity which means that PSCAD must also be implementing something similar to prevent numerical oscillations from occurring.
- The continuous solution of the non-linear inductance simulation changes much more quickly than the discretized solution, due to it seeing current changes immediately. The step size in the discretized solutions also means that region checking can only occur in the steps themselves and not in between, so the delay in inductance change compared to the continuous solution makes sense. In all non-linear inductance simulation plots, we observe that they eventually stabilize to the same values.
- When comparing the results to assignment 1, we see that before we hit 0.1A (where the flux linkage region changes), the plot is followed exactly. If assignment 1 had encountered similar discontinuities, it would have exhibited numerical stability just like this assignment.
- A saturating inductance will render an inductance useless after the saturating current has been reached. When choosing an inductor, it would be prudent to choose one whose saturating current threshold is well above the expected current operating threshold of the inductor in circuit. This would pose a problem in switch-mode power supplies which use inductors as energy storage device. If a load were to pull higher than expected current from the regulator's output, the reduction in inductance would cause it to become more resistive and likely generate wasted power in heat. This could damage the circuit due to the heat being generated.
- In order to generalize the implementation of CDA in a solver program, one could detect numerical instabilities by detecting if the solution's slope is oscillating between positive and negative slopes. The solver could then walk back the solution, apply backward Euler to resolve the numerical instability for some time, and then switch back to trapezoidal. Perhaps this could also be applied to a "live" input by buffering the input and applying both discretization methods simultaneously, and switching to backward Euler when the numerical instability is detected.
- Not a comment on this assignment, but in general for this class. I've learned a lot in doing these assignments and wanted to thank you for teaching such an interesting class. This will help me professionally in more ways than one.

5 Code Listings and Data

5.1 Python Code Listing

The following is the code written in Python to generate the solutions and plots used in this report.

```
import argparse
import csv
import math
import matplotlib
import matplotlib.pyplot as plt
import numpy

# Matplotlib export settings
matplotlib.use("pgf")
matplotlib.rcParams.update({
    "pgf.texsystem": "pdflatex",
    "font.size": 10 ,
    "font.family": "serif",  # use serif/main font for text elements
    "text.usetex": True,     # use inline math for ticks
    "pgf.rcfonts": False     # don't setup fonts from rc parameters
})

# Constants

# indices for the results tuple
TIME_INDEX = 0
CURRENT_INDEX = 1
VOLTAGE_INDEX = 2

inductance = 0.02 # 20 mH
resistance = 10 # 10 Ohms
voltage = 10 # 10 Volts
simulation_time = 0.01 # 10 ms


GET_FLUX_KNEE = 0
GET_FLUX_INDUCTANCE = 1
GET_FLUX = 2
# Inductor flux anchor and inductance lookup table
def get_flux(current_input):

    # Normal region
    if (current_input >= -0.1) and (current_input <= 0.1):
        vals = (0.0, 0.02, (0.0 + 0.02*current_input))
    # Saturation region
    elif (current_input > 0.1) or (current_input < -0.1):
        vals = (1.956e-3, 4.444e-4, (1.956e-3 + 4.444e-4*current_input))
    else:
        raise ValueError('Invalid current')

    return vals

# Dataset Class
class Dataset:

    def __init__(self, description, linestyle):
        self.description = description
        self.data = [[],[],[]]
        self.linestyle = linestyle

# Plot generation
def generate_plots(name, datasets, xlims, delta_t):

    fig, ax = plt.subplots(2)
    for index, axis in enumerate(ax):
```

```
        # current
        if index == 0:
            for dataset in datasets:
                axis.plot(dataset.data[TIME_INDEX], dataset.data[CURRENT_INDEX], linestyle=
                    dataset.linestyle, label=dataset.description)
            axis.set(xlabel='time (s)', ylabel=r'current (A)', title='Evaluations of $i_L(t)
                $ at $\Delta{}t=%gs$' % delta_t)
        # voltage
        elif index == 1:
            for dataset in datasets:
                axis.plot(dataset.data[TIME_INDEX], dataset.data[VOLTAGE_INDEX], linestyle=
                    dataset.linestyle, label=dataset.description)
            axis.set(xlabel='time (s)', ylabel=r'voltage (V)', title='Evaluations of $v_L(t)
                $ at $\Delta{}t=%gs$' % delta_t)
        axis.legend(loc='best', fancybox=True, shadow=True)
        axis.grid()
        axis.set_xticks(numpy.arange(dataset.data[TIME_INDEX][0], dataset.data[TIME_INDEX
            ][-1], delta_t/2), minor=True)
        axis.set_xlim(xlims)
        axis.grid(which='minor', alpha=0.2)
        axis.grid(which='major', alpha=0.5)

    fig.set_size_inches(6.5,8)
    fig.tight_layout()
    fig.savefig(name + '.pgf')
    fig.savefig(name + '.png')

# Flux/current plot generation
def generate_flux_plot():

    # piecewise flux linkage
    currents = [0.0,0.1,1.0]
    flux = [0.0,0.002,0.0024]
    # knee extrapolation
    knee_currents = [0.0, 0.1]
    knee_flux = [1.956e-3, 0.002]

    fig, axis = plt.subplots(1)
    axis.plot(currents, flux, color='blue')
    axis.plot(knee_currents, knee_flux, color='blue', linestyle='dotted', linewidth='0.5')
    axis.axhline(y=0.002, linestyle='dashed', c='green', linewidth='0.5')
    axis.text(0.5, 0.002-0.00008, '0.0020', fontsize=6, va='center', ha='center')
    axis.axhline(y=0.0024, linestyle='dashed', c='green', linewidth='0.5')
    axis.text(0.5, 0.0024-0.00008, '0.0024', fontsize=6, va='center', ha='center')
    axis.axvline(x=0.1, linestyle='dashed', c='green', linewidth='0.5')
    axis.axvline(x=1.0, linestyle='dashed', c='green', linewidth='0.5')
    axis.set(xlabel=r'current $(A)$', ylabel=r'$\lambda{}(i)$ $Wb-t$')
    axis.set_xlim([0.0,1.0])
    axis.set_xticks(list(axis.get_xticks()) + [0.1])
    yticks = list(axis.get_yticks())
    yticks.remove(0.002)
    axis.set_yticks(yticks + [1.956e-3])
    axis.set_ylim([0.0,0.0025])

    fig.set_size_inches(4,2.5)
    fig.tight_layout()
    fig.savefig('flux_plot.pgf')
    fig.savefig('flux_plot.png')

# Main function
def main(args):

    # Read in PSCAD .CSV data
    pscad_dat_0p0001 = Dataset('PSCAD Simulation', linestyle='dotted') # PSCAD data (non-
        linear inductance, delta_t = 0.1ms, sim_time = 10ms)
    print('*** Opening assignment 8 CSV data file...')
    with open('pscad-dat-0p0001.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
```

```
        line_count = 0
        # Read in row data
        for row in csv_reader:
            if line_count == 0:
                line_count = line_count + 1
                continue
            else:
                pscad_dat_0p0001.data[TIME_INDEX].append(float(row[0]))
                pscad_dat_0p0001.data[VOLTAGE_INDEX].append(float(row[1]))
                pscad_dat_0p0001.data[CURRENT_INDEX].append(float(row[2]))
                line_count = line_count + 1
        # Figure out when break switched
        print('Processed ' + str(line_count) + ' lines.')
    pscad_dat_0p0003 = Dataset('PSCAD Simulation', linestyle='dotted') # PSCAD data (non-
        linear inductance, delta_t = 0.1ms, sim_time = 10ms)
    print('*** Opening assignment 8 CSV data file...')
    with open('pscad-dat-0p0001.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        line_count = 0
        # Read in row data
        for row in csv_reader:
            if line_count == 0:
                line_count = line_count + 1
                continue
            else:
                pscad_dat_0p0003.data[TIME_INDEX].append(float(row[0]))
                pscad_dat_0p0003.data[VOLTAGE_INDEX].append(float(row[1]))
                pscad_dat_0p0003.data[CURRENT_INDEX].append(float(row[2]))
                line_count = line_count + 1
        # Figure out when break switched
        print('Processed ' + str(line_count) + ' lines.')

    delta_ts = [0.0001, 0.00015, 0.0002, 0.0003]

    for delta_t in delta_ts:

        print("*** Running simulation ***")
        print("Simulation time = %g" % simulation_time)
        print("Time delta = %g" % delta_t)

        cont_results = Dataset('Continuous Ideal L', linestyle='dashed') # Continuous
            solution (ideal inductance)
        cont_nl_results = Dataset('Continuous Non-Linear L', linestyle='dashed') #
            Continuous solution (non-linear inductance)
        trap_results = Dataset('Trapezoidal Ideal L', linestyle='dashdot') # Trapezoidal
            discretized solution (ideal inductance)
        nonlin_results = Dataset('Trapezoidal Non-Linear L', linestyle='solid') #
            Trapezoidal discretized solution (non-linear inductance)
        cda_results = Dataset('Trapezoidal+CDA Non-Linear L', linestyle='solid') #
            Trapezoidal discretized solution + CDA (non-linear inductance)

        # PSCAD and continuous only really apply to delta_t == 0.0001
        if delta_t == 0.0001:
            datasets = [pscad_dat_0p0001, cont_results, cont_nl_results, trap_results,
                nonlin_results, cda_results]
        else:
            datasets = [pscad_dat_0p0003, cont_results, cont_nl_results, trap_results,
                nonlin_results, cda_results]

        # Set initial values
        i_0 = 0
        v_0 = 10

        # Calculate ideal inductance results using continuous solution
        # Calculate 3000 points for the continuous solution
        for step in range(3000):
            time = 0.0 + (step * simulation_time/3000)
            # calculate continuous solution
```

```
        # current
        i_cont = (10/resistance) - ((10/resistance)*math.exp(-(resistance/inductance)*
            time))
        # inductor voltage
        v_cont = 10 * math.exp((-(resistance)/(inductance))*time)
        # append the result
        cont_results.data[TIME_INDEX].append(0+(step*simulation_time/3000))
        cont_results.data[CURRENT_INDEX].append(i_cont)
        cont_results.data[VOLTAGE_INDEX].append(v_cont)

    # Calculate non-linear inductance results using continuous solution
    # Calculate 3000 points for the continuous solution
    i_cont = 0.0
    for step in range(3000):
        time = 0.0 + (step * simulation_time/3000)
        # calculate continuous solution
        # current
        i_cont = (10/resistance) - ((10/resistance)*math.exp(-(resistance/get_flux(
            i_cont)[GET_FLUX_INDUCTANCE])*time))
        # inductor voltage
        v_cont = 10 * math.exp((-(resistance)/(get_flux(i_cont)[GET_FLUX_INDUCTANCE]))*
            time)
        # append the result
        cont_nl_results.data[TIME_INDEX].append(0+(step*simulation_time/3000))
        cont_nl_results.data[CURRENT_INDEX].append(i_cont)
        cont_nl_results.data[VOLTAGE_INDEX].append(v_cont)

    # Calculate ideal inductance results using trapezoidal discretization
    sim_steps = int(simulation_time/delta_t)
    for step in range(sim_steps):
        if step == 0:
            # Use initial conditions
            i_prev = i_0
            v_prev = v_0
        else:
            # Use previous value
            i_prev = i_approx_next
            v_prev = v_approx_next

        # append the result
        trap_results.data[TIME_INDEX].append(0+(step*delta_t))
        trap_results.data[CURRENT_INDEX].append(i_prev)
        trap_results.data[VOLTAGE_INDEX].append(v_prev)

        # perform trapezoidal discretization step
        # current
        i_approx_next = ((i_prev*((2*inductance) - (resistance*delta_t))) + (20 *
            delta_t))/((resistance*delta_t)+(2*inductance))
        # voltage
        v_approx_next = (((2*inductance)/(delta_t))*(i_approx_next - i_prev)) - v_prev

    # Calculate non-linear inductance results using trapezoidal discretization
    # Nodal analysis
    switch_open_flag = False
    # Start at t = 0
    t = 0
    # Starting conditions
    v1 = 10.0
    v2 = 0.0
    v3 = 10.0
    i30 = 0.0
    r = resistance

    while t < simulation_time:

        # Get values for l, flux, and flux_knee
        l = get_flux(i30)[GET_FLUX_INDUCTANCE]
        flux = get_flux(i30)[GET_FLUX]
```

```
        flux_knee = get_flux(i30)[GET_FLUX_KNEE]

        # Check if breaker is opened or closed and set breaker resistance accordingly
        if switch_open_flag == False:
            # Breaker has very small resistance (closed)
            r_sw = 1e-9
        else:
            # Break has very large resistance (open)
            r_sw = 1e9

        # Voltage calculations
        # These voltage calculations are generated in MATLAB
        # by v_next[] = inv(GAA)*hA - inv(GAA)*GAB*vs

        v1_next = 10

        # trapezoidal
        v2_next_trap = (10*(2*l + delta_t*r))/(2*l + delta_t*r + delta_t*r_sw) - (
            delta_t*r_sw*(v3 + (2*flux)/delta_t - (2*flux_knee)/delta_t))/(2*l + delta_t
            *r + delta_t*r_sw)
        v3_next_trap = (20*l)/(2*l + delta_t*r + delta_t*r_sw) - (delta_t*(r + r_sw)*(v3
             + (2*flux)/delta_t - (2*flux_knee)/delta_t))/(2*l + delta_t*r + delta_t*
            r_sw)

        # Next currents
        i30_trap_next = (delta_t*v3_next_trap)/(2*l) + (delta_t*(v3 + (2*flux)/delta_t -
             (2*flux_knee)/delta_t))/(2*l)

        # Append results
        nonlin_results.data[TIME_INDEX].append(t)
        nonlin_results.data[CURRENT_INDEX].append(i30)
        nonlin_results.data[VOLTAGE_INDEX].append(v3)

        # Next iteration
        i30 = i30_trap_next
        v1 = v1_next
        v2 = v2_next_trap
        v3 = v3_next_trap

        # Go to next time step
        t = t + delta_t

    # Calculate non-linear inductance results using trapezoidal discretization + CDA
    # Nodal analysis
    switch_open_flag = False
    # Start at t = 0
    t = 0
    # Starting conditions
    v1 = 10.0
    v2 = 0.0
    v3 = 10.0
    i30 = 0.0
    r = resistance
    # Get an initial inductance
    l_prev = get_flux(i30)[GET_FLUX_INDUCTANCE]
    # CDA initially inactive
    cda_active = False
    cda_counter = 0
    while t < simulation_time:

        # Get values for l, flux, and flux_knee
        l = get_flux(i30)[GET_FLUX_INDUCTANCE]
        # Check if we need to perform CDA
        if (l != l_prev):
            # We've entered a new region, for the next two half time_deltas
            # Perform backward Euler instead
            cda_active = True
            cda_counter = 2
```

```
        flux = get_flux(i30)[GET_FLUX]
        flux_knee = get_flux(i30)[GET_FLUX_KNEE]

        # Check if breaker is opened or closed and set breaker resistance accordingly
        if switch_open_flag == False:
            # Breaker has very small resistance (closed)
            r_sw = 1e-9
        else:
            # Break has very large resistance (open)
            r_sw = 1e9

        # Voltage calculations
        # These voltage calculations are generated in MATLAB
        # by v_next[] = inv(GAA)*hA - inv(GAA)*GAB*vs

        v1_next = 10

        if (cda_active):
            # backward euler
            # voltages
            v2_next = (10*(l + delta_t*r))/(l + delta_t*r + delta_t*r_sw) - (delta_t*
                r_sw*(flux/delta_t - flux_knee/delta_t))/(l + delta_t*r + delta_t*r_sw)
            v3_next = (10*l)/(l + delta_t*r + delta_t*r_sw) - (delta_t*(r + r_sw)*(flux/
                delta_t - flux_knee/delta_t))/(l + delta_t*r + delta_t*r_sw)
            # current
            i30_next = (delta_t*(flux/delta_t - flux_knee/delta_t))/l + (delta_t*v3_next
                )/l
        else:
            # trapezoidal
            # voltages
            v2_next = (10*(2*l + delta_t*r))/(2*l + delta_t*r + delta_t*r_sw) - (delta_t
                *r_sw*(v3 + (2*flux)/delta_t - (2*flux_knee)/delta_t))/(2*l + delta_t*r
                + delta_t*r_sw)
            v3_next = (20*l)/(2*l + delta_t*r + delta_t*r_sw) - (delta_t*(r + r_sw)*(v3
                + (2*flux)/delta_t - (2*flux_knee)/delta_t))/(2*l + delta_t*r + delta_t*
                r_sw)
            # current
            i30_next = (delta_t*v3_next)/(2*l) + (delta_t*(v3 + (2*flux)/delta_t - (2*
                flux_knee)/delta_t))/(2*l)

        # Append results
        cda_results.data[TIME_INDEX].append(t)
        cda_results.data[CURRENT_INDEX].append(i30)
        cda_results.data[VOLTAGE_INDEX].append(v3)

        # Next iteration
        i30 = i30_next
        v1 = v1_next
        v2 = v2_next
        v3 = v3_next
        l_prev = l

        # Go to next time step
        if (cda_active):
            cda_counter = cda_counter - 1
            # Check if we're done with CDA
            if cda_counter == 0:
                cda_active = False
            t = t + delta_t/2.0
        else:
            t = t + delta_t

    # Plots for publication
    # Comparisons between all techniques
    generate_flux_plot()
    generate_plots('compare_plots_' + str(delta_t).replace('.','p'), datasets, (0, 0.01)
        , delta_t)
```

```
        # Change zoom level depending on time step
        if delta_t == 0.0001:
            x_zoom_min = 0.0002
            x_zoom_max = 0.0005
        elif delta_t == 0.00015:
            x_zoom_min = 0.0002
            x_zoom_max = 0.001
        elif delta_t == 0.0002:
            x_zoom_min = 0.0002
            x_zoom_max = 0.001
        elif delta_t == 0.0003:
            x_zoom_min = 0.0002
            x_zoom_max = 0.002
        else:
            # default zoom
            x_zoom_min = 0.0002
            x_zoom_max = 0.0005
        generate_plots('compare_plots_zoom_' + str(delta_t).replace('.','p'), datasets, (
            x_zoom_min, x_zoom_max), delta_t)


if __name__ == '__main__':
    # the following sets up the argument parser for the program
    parser = argparse.ArgumentParser(description='Assignment 8 solution generator')

    args = parser.parse_args()

    main(args)
```

5.2 PSCAD CSV File ($\Delta t = 0.0001s$)

The following is the data generated by PSCAD for plotting against the hand-implemented solution.

```
Domain, voltage, current, inductance
0.0,10.0,5.55111512313e-17,0.02
0.0001,9.46155744048,0.0538442559523,0.02
0.0002,9.05950645684,0.0940493543164,0.02
0.0003,8.61757931691,0.138242068309,0.0004444
0.0004,0.920371956066,0.907962804393,0.0004444
0.0005,-0.0541853837636,1.00541853838,0.0004444
0.0006,0.00319007526713,0.999680992473,0.0004444
0.0007,-0.000187810429734,1.00001878104,0.0004444
0.0008,1.10570298704e-05,0.999998894297,0.0004444
0.0009,-6.50964431692e-07,1.0000000651,0.0004444
0.001,3.83244600499e-08,0.999999996168,0.0004444
0.0011,-2.25629026573e-09,1.00000000023,0.0004444
0.0012,1.3283596445e-10,0.999999999987,0.0004444
0.0013,-7.82129916388e-12,1.0,0.0004444
0.0014,4.60076421405e-13,1.0,0.0004444
0.0015,-2.6645352591e-14,1.0,0.0004444
0.0016,2.6645352591e-15,1.0,0.0004444
0.0017,8.881784197e-16,1.0,0.0004444
0.0018,8.881784197e-16,1.0,0.0004444
0.0019,8.881784197e-16,1.0,0.0004444
0.002,8.881784197e-16,1.0,0.0004444
0.0021,8.881784197e-16,1.0,0.0004444
0.0022,8.881784197e-16,1.0,0.0004444
0.0023,8.881784197e-16,1.0,0.0004444
0.0024,8.881784197e-16,1.0,0.0004444
0.0025,8.881784197e-16,1.0,0.0004444
0.0026,8.881784197e-16,1.0,0.0004444
0.0027,8.881784197e-16,1.0,0.0004444
0.0028,8.881784197e-16,1.0,0.0004444
0.0029,8.881784197e-16,1.0,0.0004444
0.003,8.881784197e-16,1.0,0.0004444
```

```
0.0031,8.881784197e-16,1.0,0.0004444
0.0032,8.881784197e-16,1.0,0.0004444
0.0033,8.881784197e-16,1.0,0.0004444
0.0034,8.881784197e-16,1.0,0.0004444
0.0035,8.881784197e-16,1.0,0.0004444
0.0036,8.881784197e-16,1.0,0.0004444
0.0037,8.881784197e-16,1.0,0.0004444
0.0038,8.881784197e-16,1.0,0.0004444
0.0039,8.881784197e-16,1.0,0.0004444
0.004,8.881784197e-16,1.0,0.0004444
0.0041,8.881784197e-16,1.0,0.0004444
0.0042,8.881784197e-16,1.0,0.0004444
0.0043,8.881784197e-16,1.0,0.0004444
0.0044,8.881784197e-16,1.0,0.0004444
0.0045,8.881784197e-16,1.0,0.0004444
0.0046,8.881784197e-16,1.0,0.0004444
0.0047,8.881784197e-16,1.0,0.0004444
0.0048,8.881784197e-16,1.0,0.0004444
0.0049,8.881784197e-16,1.0,0.0004444
0.005,8.881784197e-16,1.0,0.0004444
0.0051,8.881784197e-16,1.0,0.0004444
0.0052,8.881784197e-16,1.0,0.0004444
0.0053,8.881784197e-16,1.0,0.0004444
0.0054,8.881784197e-16,1.0,0.0004444
0.0055,8.881784197e-16,1.0,0.0004444
0.0056,8.881784197e-16,1.0,0.0004444
0.0057,8.881784197e-16,1.0,0.0004444
0.0058,8.881784197e-16,1.0,0.0004444
0.0059,8.881784197e-16,1.0,0.0004444
0.006,8.881784197e-16,1.0,0.0004444
0.0061,8.881784197e-16,1.0,0.0004444
0.0062,8.881784197e-16,1.0,0.0004444
0.0063,8.881784197e-16,1.0,0.0004444
0.0064,8.881784197e-16,1.0,0.0004444
0.0065,8.881784197e-16,1.0,0.0004444
0.0066,8.881784197e-16,1.0,0.0004444
0.0067,8.881784197e-16,1.0,0.0004444
0.0068,8.881784197e-16,1.0,0.0004444
0.0069,8.881784197e-16,1.0,0.0004444
0.007,8.881784197e-16,1.0,0.0004444
0.0071,8.881784197e-16,1.0,0.0004444
0.0072,8.881784197e-16,1.0,0.0004444
0.0073,8.881784197e-16,1.0,0.0004444
0.0074,8.881784197e-16,1.0,0.0004444
0.0075,8.881784197e-16,1.0,0.0004444
0.0076,8.881784197e-16,1.0,0.0004444
0.0077,8.881784197e-16,1.0,0.0004444
0.0078,8.881784197e-16,1.0,0.0004444
0.0079,8.881784197e-16,1.0,0.0004444
0.008,8.881784197e-16,1.0,0.0004444
0.0081,8.881784197e-16,1.0,0.0004444
0.0082,8.881784197e-16,1.0,0.0004444
0.0083,8.881784197e-16,1.0,0.0004444
0.0084,8.881784197e-16,1.0,0.0004444
0.0085,8.881784197e-16,1.0,0.0004444
0.0086,8.881784197e-16,1.0,0.0004444
0.0087,8.881784197e-16,1.0,0.0004444
0.0088,8.881784197e-16,1.0,0.0004444
0.0089,8.881784197e-16,1.0,0.0004444
0.009,8.881784197e-16,1.0,0.0004444
0.0091,8.881784197e-16,1.0,0.0004444
0.0092,8.881784197e-16,1.0,0.0004444
0.0093,8.881784197e-16,1.0,0.0004444
0.0094,8.881784197e-16,1.0,0.0004444
0.0095,8.881784197e-16,1.0,0.0004444
0.0096,8.881784197e-16,1.0,0.0004444
0.0097,8.881784197e-16,1.0,0.0004444
0.0098,8.881784197e-16,1.0,0.0004444
```

```
0.0099,8.881784197e-16,1.0,0.0004444
```

5.3 PSCAD CSV File ($\Delta t = 0.0003s$)

The following is the data generated by PSCAD for plotting against the hand-implemented solution.

```
Domain, voltage, current, inductance
0.0,10.0,5.55111512313e-17,0.02
0.0003,8.51371578652,0.148628421348,0.0004444
0.0006,0.109267985316,0.989073201468,0.0004444
0.0009,-0.0593207593859,1.00593207594,0.0004444
0.0012,0.0322047897557,0.996779521024,0.0004444
0.0015,-0.0174837357773,1.00174837358,0.0004444
0.0018,0.00949178737225,0.999050821263,0.0004444
0.0021,-0.00515301927847,1.00051530193,0.0004444
0.0024,0.00279753503138,0.999720246497,0.0004444
0.0027,0.000146134384934,0.999985386562,0.0004444
0.003,-7.93352477548e-05,1.00000793352,0.0004444
0.0033,4.307050349e-05,0.99999569295,0.0004444
0.0036,-2.33826492431e-05,1.00000233826,0.0004444
0.0039,1.2694262693e-05,0.999998730574,0.0004444
0.0042,-6.89161880851e-06,1.00000068916,0.0004444
0.0045,3.74140751225e-06,0.999999625859,0.0004444
0.0048,-2.03118172415e-06,1.00000020312,0.0004444
0.0051,1.10271313192e-06,0.999999889729,0.0004444
0.0054,-5.98654584572e-07,1.00000005987,0.0004444
0.0057,3.25005027335e-07,0.999999967499,0.0004444
0.006,-1.76442760846e-07,1.00000001764,0.0004444
0.0063,9.57894346065e-08,0.999999990421,0.0004444
0.0066,-5.20033562879e-08,1.0000000052,0.0004444
0.0069,2.82322272227e-08,0.999999997177,0.0004444
0.0072,-1.53270613978e-08,1.00000000153,0.0004444
0.0075,8.32094526615e-09,0.999999999168,0.0004444
0.0078,-4.51737758311e-09,1.00000000045,0.0004444
0.0081,2.4524497988e-09,0.999999999755,0.0004444
0.0084,-1.33141586645e-09,1.00000000013,0.0004444
0.0087,7.22816029253e-10,0.999999999928,0.0004444
0.009,-3.92411880767e-10,1.00000000004,0.0004444
0.0093,2.13037587571e-10,0.999999999979,0.0004444
0.0096,-1.15656373367e-10,1.00000000001,0.0004444
0.0099,6.27888852023e-11,0.999999999994,0.0004444
```

5.4 MATLAB Listing

The following is the MATLAB listing that formed the matrices and generated the next voltage equations used in the Python script.

```
% Source voltage

syms vs delta_t t

vs = 10

% Circuit values

syms r_sw r l

% Discretized Resistances:
r30_trap = 2*l/delta_t
r30_back = l/delta_t
```

```
% History Equations:

syms eh30_trap eh30_back
syms h30_trap h30_back
syms v3
syms v3_next_trap
syms v3_next_back
syms flux flux_knee
syms i30
syms i30_trap_next i30_back_next

eh30_trap = -v3 - (2/delta_t)*flux + (2/delta_t)*flux_knee
eh30_back = -(1/delta_t)*flux + (1/delta_t)*flux_knee

h30_trap = eh30_trap/r30_trap
h30_back = eh30_back/r30_back

i30_trap_next = v3_next_trap/r30_trap - h30_trap
i30_back_next = v3_next_back/r30_back - h30_back

syms g11 g12 g13
syms g21 g22 g23
syms g31 g32 g33
syms h1 h2 h3

% Trapezoidal solution

h1 = 0
h2 = 0
h3 = h30_trap

hA = [h2 ; h3]
hB = h1
H = [hA ; hB]

g11 = 1/r_sw
g12 = -1/r_sw
g13 = 0

g21 = -1/r_sw
g22 = 1/r_sw + 1/r
g23 = -1/r

g31 = 0
g32 = -1/r
g33 = 1/r + 1/r30_trap

GAA = [g22 g23 ; g32 g33]
GAB = [g21 ; g31]
GBA = [g12 g13]
GBB = [g11]
G = [GAA GAB ; GBA GBB]

% Nodal voltage solution vector:

v_next_trap = inv(GAA)*hA - inv(GAA)*GAB*vs

% Backward Euler solution

h1 = 0
h2 = 0
h3 = h30_back

hA = [h2 ; h3]
hB = h1
H = [hA ; hB]

g11 = 1/r_sw
```

```
g12 = -1/r_sw
g13 = 0

g21 = -1/r_sw
g22 = 1/r_sw + 1/r
g23 = -1/r

g31 = 0
g32 = -1/r
g33 = 1/r + 1/r30_back

GAA = [g22 g23 ; g32 g33]
GAB = [g21 ; g31]
GBA = [g12 g13]
GBB = [g11]
G = [GAA GAB ; GBA GBB]

% Nodal voltage solution vector:

v_next_back = inv(GAA)*hA - inv(GAA)*GAB*vs

v_next_trap
v_next_back
i30_trap_next
i30_back_next
```

5.5 Fortran Code Listing

The following is the code generated by PSCAD to simulate the circuit for this assignment.

```
!=========================================================================
! Generated by  : PSCAD v4.6.3.0
!
! Warning:  The content of this file is automatically generated.
!           Do not modify, as any changes made here will be lost!
!-------------------------------------------------------------------------
! Component     : Main
! Description   :
!-------------------------------------------------------------------------


!=========================================================================

      SUBROUTINE MainDyn()

!---------------------------------------
! Standard includes
!---------------------------------------

      INCLUDE 'nd.h'
      INCLUDE 'emtconst.h'
      INCLUDE 'emtstor.h'
      INCLUDE 's0.h'
      INCLUDE 's1.h'
      INCLUDE 's2.h'
      INCLUDE 's4.h'
      INCLUDE 'branches.h'
      INCLUDE 'pscadv3.h'
      INCLUDE 'fnames.h'
      INCLUDE 'radiolinks.h'
      INCLUDE 'matlab.h'
      INCLUDE 'rtconfig.h'

!---------------------------------------
! Function/Subroutine Declarations
```

```
!---------------------------------------


!---------------------------------------
! Variable Declarations
!---------------------------------------


! Subroutine Arguments

! Electrical Node Indices

! Control Signals
      INTEGER  sw, v_init
      REAL     var_L, i_L, v_L

! Internal Variables
      INTEGER  IVD1_1, IVD1_2
      REAL     RVD1_1, RVD1_2, RVD1_3, RVD1_4

! Indexing variables
      INTEGER ICALL_NO                          ! Module call num
      INTEGER ISTOI, ISTOF, IT_0                ! Storage Indices
      INTEGER SS, INODE, IBRCH                  ! SS/Node/Branch/Xfmr


!---------------------------------------
! Local Indices
!---------------------------------------

! Dsdyn <-> Dsout transfer index storage

      NTXFR = NTXFR + 1

      TXFR(NTXFR,1) = NSTOL
      TXFR(NTXFR,2) = NSTOI
      TXFR(NTXFR,3) = NSTOF
      TXFR(NTXFR,4) = NSTOC

! Define electric network subsystem number

      SS        = NODE(NNODE+1)

! Increment and assign runtime configuration call indices

      ICALL_NO  = NCALL_NO
      NCALL_NO  = NCALL_NO + 1

! Increment global storage indices

      ISTOI     = NSTOI
      NSTOI     = NSTOI + 2
      ISTOF     = NSTOF
      NSTOF     = NSTOF + 3
      NPGB      = NPGB + 3
      INODE     = NNODE + 2
      NNODE     = NNODE + 6
      IBRCH     = NBRCH(SS)
      NBRCH(SS) = NBRCH(SS) + 6
      NCSCS     = NCSCS + 0
      NCSCR     = NCSCR + 0

!---------------------------------------
! Transfers from storage arrays
!---------------------------------------

      var_L     = STOF(ISTOF + 1)
      i_L       = STOF(ISTOF + 2)
```

```
      v_L       = STOF(ISTOF + 3)
      sw        = STOI(ISTOI + 1)
      v_init    = STOI(ISTOI + 2)


!---------------------------------------
! Electrical Node Lookup
!---------------------------------------


!---------------------------------------
! Configuration of Models
!---------------------------------------

      IF ( TIMEZERO ) THEN
         FILENAME = 'Main.dta'
         CALL EMTDC_OPENFILE
         SECTION = 'DATADSD:'
         CALL EMTDC_GOTOSECTION
      ENDIF
!---------------------------------------
! Generated code from module definition
!---------------------------------------


! 10:[tbreakn] Timed Breaker Logic
! Timed breaker logic
      IF ( TIMEZERO ) THEN
         sw = 1
      ELSE
         sw = 1
         IF ( TIME .GE. 0.0 ) sw = (1-1)
      ENDIF

! 20:[breaker1] Single Phase Breaker 'sw'
      IVD1_2 = NSTORI
      NSTORI = NSTORI + 1
      CALL E1PBRKR1_EXE(SS, (IBRCH+5),1.0e-09,1000000000.0,1,NINT(1.0-RE&
     &AL(sw)))
      IVD1_1 = 2*E_BtoI(OPENBR( (IBRCH+5),SS))
      IF (FIRSTSTEP .OR. (STORI(IVD1_2) .NE. IVD1_1)) THEN
         CALL PSCAD_AGI2(ICALL_NO,1210198031,IVD1_1,"BOpen")
      ENDIF
      STORI(IVD1_2) = 2*E_BtoI(OPENBR( (IBRCH+5),SS))

! 40:[tbreakn] Timed Breaker Logic
! Timed breaker logic
      IF ( TIMEZERO ) THEN
         v_init = 0
      ELSE
         v_init = 0
         IF ( TIME .GE. 0.0 ) v_init = (1-0)
      ENDIF

! 90:[varrlc] Variable R, L or C
      CALL COMPONENT_ID(ICALL_NO,1113893919)
      CALL E_VARRLC1_EXE(1 ,SS ,  (IBRCH+1), 0, var_L, 0.0)

! 100:[breaker1] Single Phase Breaker 'v_init'
      IVD1_2 = NSTORI
      NSTORI = NSTORI + 1
      CALL E1PBRKR1_EXE(SS, (IBRCH+2),1.0e-09,1000000000.0,1,NINT(1.0-RE&
     &AL(v_init)))
      IVD1_1 = 2*E_BtoI(OPENBR( (IBRCH+2),SS))
      IF (FIRSTSTEP .OR. (STORI(IVD1_2) .NE. IVD1_1)) THEN
         CALL PSCAD_AGI2(ICALL_NO,1679398847,IVD1_1,"BOpen")
      ENDIF
      STORI(IVD1_2) = 2*E_BtoI(OPENBR( (IBRCH+2),SS))
```

```
! 1:[source_1] Single Phase Voltage Source Model 2 'Source1'
! DC source with specified terminal conditions: Type: Ideal
      RVD1_1 = RTCF(NRTCF)
      RVD1_2 = RTCF(NRTCF+1)
      RVD1_3 = RTCF(NRTCF+2)
      RVD1_4 = RTCF(NRTCF+3)
      NRTCF = NRTCF + 4
      CALL EMTDC_1PVSRC(SS, (IBRCH+4),RVD1_4,0,RVD1_1,RVD1_2,RVD1_3)

!---------------------------------------
! Feedbacks and transfers to storage
!---------------------------------------

      STOF(ISTOF + 1) = var_L
      STOF(ISTOF + 2) = i_L
      STOF(ISTOF + 3) = v_L
      STOI(ISTOI + 1) = sw
      STOI(ISTOI + 2) = v_init


!---------------------------------------
! Transfer to Exports
!---------------------------------------

!---------------------------------------
! Close Model Data read
!---------------------------------------

      IF ( TIMEZERO ) CALL EMTDC_CLOSEFILE
      RETURN
      END

!=======================================================================

      SUBROUTINE MainOut()

!---------------------------------------
! Standard includes
!---------------------------------------

      INCLUDE 'nd.h'
      INCLUDE 'emtconst.h'
      INCLUDE 'emtstor.h'
      INCLUDE 's0.h'
      INCLUDE 's1.h'
      INCLUDE 's2.h'
      INCLUDE 's4.h'
      INCLUDE 'branches.h'
      INCLUDE 'pscadv3.h'
      INCLUDE 'fnames.h'
      INCLUDE 'radiolinks.h'
      INCLUDE 'matlab.h'
      INCLUDE 'rtconfig.h'

!---------------------------------------
! Function/Subroutine Declarations
!---------------------------------------

      REAL    VBRANCH       !
      REAL    EMTDC_VVDC    !

!---------------------------------------
! Variable Declarations
!---------------------------------------


! Electrical Node Indices
```

```
      INTEGER  NT_3

! Control Signals
      REAL     var_L, i_L, v_L

! Internal Variables
      INTEGER  IVD1_1

! Indexing variables
      INTEGER ICALL_NO                              ! Module call num
      INTEGER ISTOL, ISTOI, ISTOF, ISTOC, IT_0      ! Storage Indices
      INTEGER IPGB                                  ! Control/Monitoring
      INTEGER SS, INODE, IBRCH                      ! SS/Node/Branch/Xfmr


!---------------------------------------
! Local Indices
!---------------------------------------

! Dsdyn <-> Dsout transfer index storage

      NTXFR = NTXFR + 1

      ISTOL = TXFR(NTXFR,1)
      ISTOI = TXFR(NTXFR,2)
      ISTOF = TXFR(NTXFR,3)
      ISTOC = TXFR(NTXFR,4)

! Define electric network subsystem number

      SS        = NODE(NNODE+1)

! Increment and assign runtime configuration call indices

      ICALL_NO  = NCALL_NO
      NCALL_NO  = NCALL_NO + 1

! Increment global storage indices

      IPGB      = NPGB
      NPGB      = NPGB + 3
      INODE     = NNODE + 2
      NNODE     = NNODE + 6
      IBRCH     = NBRCH(SS)
      NBRCH(SS) = NBRCH(SS) + 6
      NCSCS     = NCSCS + 0
      NCSCR     = NCSCR + 0

!---------------------------------------
! Transfers from storage arrays
!---------------------------------------

      var_L    = STOF(ISTOF + 1)
      i_L      = STOF(ISTOF + 2)
      v_L      = STOF(ISTOF + 3)


!---------------------------------------
! Electrical Node Lookup
!---------------------------------------

      NT_3  = NODE(INODE + 3)

!---------------------------------------
! Configuration of Models
!---------------------------------------

      IF ( TIMEZERO ) THEN
```

```
         FILENAME = 'Main.dta'
         CALL EMTDC_OPENFILE
         SECTION = 'DATADSO:'
         CALL EMTDC_GOTOSECTION
      ENDIF
!---------------------------------------
! Generated code from module definition
!---------------------------------------


! 20:[breaker1] Single Phase Breaker 'sw'
! Single phase breaker current
!

! 30:[multimeter] Multimeter
      IVD1_1 = NRTCF
      NRTCF  = NRTCF + 5
      i_L = ( CBR((IBRCH+3), SS))
      v_L = EMTDC_VVDC(SS, NT_3, 0)

! 50:[xy_transfer_function] X-Y transfer function
      CALL COMPONENT_ID(ICALL_NO,1026196919)
      CALL XYFUNC1_EXE(7,0,0,0.0,0.0,1.0,1.0,i_L,var_L)

! 60:[pgb] Output Channel 'inductance'

      PGB(IPGB+1) = var_L

! 70:[pgb] Output Channel 'current'

      PGB(IPGB+2) = i_L

! 80:[pgb] Output Channel 'voltage'

      PGB(IPGB+3) = v_L

! 100:[breaker1] Single Phase Breaker 'v_init'
! Single phase breaker current
!

!---------------------------------------
! Feedbacks and transfers to storage
!---------------------------------------

      STOF(ISTOF + 1) = var_L
      STOF(ISTOF + 2) = i_L
      STOF(ISTOF + 3) = v_L


!---------------------------------------
! Close Model Data read
!---------------------------------------

      IF ( TIMEZERO ) CALL EMTDC_CLOSEFILE
      RETURN
      END

!=======================================================================

      SUBROUTINE MainDyn_Begin()

!---------------------------------------
! Standard includes
!---------------------------------------

      INCLUDE 'nd.h'
      INCLUDE 'emtconst.h'
      INCLUDE 's0.h'
```

```
      INCLUDE 's1.h'
      INCLUDE 's4.h'
      INCLUDE 'branches.h'
      INCLUDE 'pscadv3.h'
      INCLUDE 'radiolinks.h'
      INCLUDE 'rtconfig.h'

!---------------------------------------
! Function/Subroutine Declarations
!---------------------------------------


!---------------------------------------
! Variable Declarations
!---------------------------------------


! Subroutine Arguments

! Electrical Node Indices

! Control Signals

! Internal Variables

! Indexing variables
      INTEGER ICALL_NO                          ! Module call num
      INTEGER IT_0                              ! Storage Indices
      INTEGER SS, INODE, IBRCH                  ! SS/Node/Branch/Xfmr


!---------------------------------------
! Local Indices
!---------------------------------------


! Define electric network subsystem number

      SS        = NODE(NNODE+1)

! Increment and assign runtime configuration call indices

      ICALL_NO  = NCALL_NO
      NCALL_NO  = NCALL_NO + 1

! Increment global storage indices

      INODE     = NNODE + 2
      NNODE     = NNODE + 6
      IBRCH     = NBRCH(SS)
      NBRCH(SS) = NBRCH(SS) + 6
      NCSCS     = NCSCS + 0
      NCSCR     = NCSCR + 0

!---------------------------------------
! Electrical Node Lookup
!---------------------------------------


!---------------------------------------
! Generated code from module definition
!---------------------------------------


! 20:[breaker1] Single Phase Breaker 'sw'
      CALL COMPONENT_ID(ICALL_NO,1210198031)
      CALL E1PBRKR1_CFG(1.0e-09,1000000000.0,0.0)
```

```
! 90:[varrlc] Variable R, L or C
      CALL E_VARRLC1_CFG(1 ,SS ,  (IBRCH+1), 0)

! 100:[breaker1] Single Phase Breaker 'v_init'
      CALL COMPONENT_ID(ICALL_NO,1679398847)
      CALL E1PBRKR1_CFG(1.0e-09,1000000000.0,0.0)

! 1:[source_1] Single Phase Voltage Source Model 2 'Source1'
      CALL E_1PVSRC_CFG(0,1,6,10.0,60.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0)

      RETURN
      END

!=======================================================================

      SUBROUTINE MainOut_Begin()

!---------------------------------------
! Standard includes
!---------------------------------------

      INCLUDE 'nd.h'
      INCLUDE 'emtconst.h'
      INCLUDE 's0.h'
      INCLUDE 's1.h'
      INCLUDE 's4.h'
      INCLUDE 'branches.h'
      INCLUDE 'pscadv3.h'
      INCLUDE 'radiolinks.h'
      INCLUDE 'rtconfig.h'

!---------------------------------------
! Function/Subroutine Declarations
!---------------------------------------


!---------------------------------------
! Variable Declarations
!---------------------------------------


! Subroutine Arguments

! Electrical Node Indices
      INTEGER  NT_3

! Control Signals

! Internal Variables
      INTEGER  IVD1_1

! Indexing variables
      INTEGER ICALL_NO                          ! Module call num
      INTEGER IT_0                              ! Storage Indices
      INTEGER SS, INODE, IBRCH                  ! SS/Node/Branch/Xfmr


!---------------------------------------
! Local Indices
!---------------------------------------


! Define electric network subsystem number

      SS        = NODE(NNODE+1)

! Increment and assign runtime configuration call indices
```

```
      ICALL_NO  = NCALL_NO
      NCALL_NO  = NCALL_NO + 1

! Increment global storage indices

      INODE     = NNODE + 2
      NNODE     = NNODE + 6
      IBRCH     = NBRCH(SS)
      NBRCH(SS) = NBRCH(SS) + 6
      NCSCS     = NCSCS + 0
      NCSCR     = NCSCR + 0

!---------------------------------------
! Electrical Node Lookup
!---------------------------------------

      NT_3  = NODE(INODE + 3)

!---------------------------------------
! Generated code from module definition
!---------------------------------------


! 30:[multimeter] Multimeter
      IVD1_1 = NRTCF
      NRTCF  = NRTCF + 5

! 50:[xy_transfer_function] X-Y transfer function
      RTCF(NRTCF)    = -2.0
      RTCF(NRTCF+1)  = 0.0004444
      RTCF(NRTCF+2)  = -0.1
      RTCF(NRTCF+3)  = 0.0004444
      RTCF(NRTCF+4)  = -0.1
      RTCF(NRTCF+5)  = 0.02
      RTCF(NRTCF+6)  = 0.0
      RTCF(NRTCF+7)  = 0.02
      RTCF(NRTCF+8)  = 0.1
      RTCF(NRTCF+9)  = 0.02
      RTCF(NRTCF+10) = 0.1
      RTCF(NRTCF+11) = 0.0004444
      RTCF(NRTCF+12) = 2.0
      RTCF(NRTCF+13) = 0.0004444
      NRTCF = NRTCF + 14

! 60:[pgb] Output Channel 'inductance'

! 70:[pgb] Output Channel 'current'

! 80:[pgb] Output Channel 'voltage'

      RETURN
      END
```