EE2703 - Week 2

Niranjan A. Kartha <ee21b095@smail.iitm.ac.in>

February 8, 2023

1 Applied Programming Lab: Week 2 Submission

by Niranjan A. Kartha (EE21B095)

---

2 Please use the notebook!

I have used the `ipynb` file to run and test code, as well as add documentation and explanations for it. So please use the Jupyter Notebook file as it provides a more streamlined experience.

---

3 Running the notebook

This notebook uses the `numpy` library which needs to be installed for the cells to run without errors.

[1]:
```
import numpy as np # for working with numbers
import timeit # for timing functions
```

---

4 Performance in Python

Python is an interpreted language, and so every line needs to be parsed by the interpreter for it to be executed. It also does not implement any parallelization by default. It also has extra runtime overhead like a garbage collector. So, compared to a compiled language like C with manual memory management, it will be much slower.

4.1 AOT Compilation

Languages like C and C++ are compiled ahead-of-time (AOT). This gives us a binary with instructions that can be directly executed by a processor. Assembly generated by C can be optimized by the compiler, and the processor can execute multiple assembly instructions at once through pipelining.

4.2 JIT Compilation

Some of the disadvantages of Python performance-wise can be dealt with through just-in-time (JIT) compilation. This is the compilation of parts of a program during the execution of the program.

This adds some initial overhead as we need to compile the code each time we are running it, but as we run the same code again and again, the performance gain makes up for it.

Python interpreters like PyPy support JIT compilation. We can perform JIT compilation in CPython (the "official" Python implementation) using libraries like numba.

For demonstrations relating to JIT, we use the `numba` package.

[ ]:
```
import sys
!{sys.executable} -m pip install numba
```

[2]:
```
from numba import njit
```

---

5 Factorial

Below are various different but equivalent ways to implement a factorial in Python.

[3]:
```
# compute x factorial using recursion
def factorial_recursive(x):
    if x == 0:
        return 1
    return x * factorial_recursive(x - 1)

# compute x factorial using a for-loop
def factorial_for(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i
    return prod

# compute x factorial using a while-loop
def factorial_while(x):
    prod = 1
    while x > 0:
        prod *= x
        x -= 1
    return prod

# use python's builtin factorial function
import math
def factorial_builtin(x):
    return math.factorial(x)
```

```
# use numpy's built-in factorial function
def factorial_numpy(x):
    return np.math.factorial(x)
```

5.1 Timing

[4]:
```
num = 10
```

[5]:
```
print(factorial_recursive(num))
%timeit factorial_recursive(num)
```

```
3628800
991 ns ± 51.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[6]:
```
print(factorial_for(num))
%timeit factorial_for(num)
```

```
3628800
522 ns ± 17.5 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[7]:
```
print(factorial_while(num))
%timeit factorial_while(num)
```

```
3628800
687 ns ± 6.06 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[8]:
```
print(factorial_builtin(num))
%timeit factorial_builtin(num)
```

```
3628800
128 ns ± 1.43 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[9]:
```
print(factorial_numpy(num))
%timeit factorial_numpy(num)
```

```
3628800
154 ns ± 1.62 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

5.2 Observations

5.2.1 Built-in/library functions are faster than the hand-written ones

As we can see above, the `numpy` built-in factorial function is 3 to 6 times faster than our implementations made in Python, even though it provides identical functionality (even for large integers). This is because `numpy` does not compute the factorial in Python, and instead contains bindings to compiled C code.

5.2.2 Looping is faster than recursion

We can also see that the looping version is twice as fast as the recursive one. This is because function calls add a lot of overhead. Each time we multiply a new number, the `factorial_recursive`

function needs to be called, and this involves the creation of a whole new stack frame on the call stack.

5.2.3 The `while`-loop version runs slower than the `for`-loop version

This is because `range` is a built-in Python function, and the incrementing and checks for the loop are done in optimized and compiled C code. In the `while` version, the incrementing and checks are done in Python.

5.3 JIT-compiled tests

The JIT-compiled versions of these functions should run faster:

[10]:
```
# compute x factorial using recursion
@njit
def factorial_recursive_jit(x):
    if x == 0:
        return 1
    return x * factorial_recursive_jit(x - 1)

# compute x factorial using a for-loop
@njit
def factorial_for_jit(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i
    return prod

# compute x factorial using a while-loop
@njit
def factorial_while_jit(x):
    prod = 1
    while x > 0:
        prod *= x
        x -= 1
    return prod
```

[11]:
```
num = 10
```

[12]:
```
print(factorial_recursive_jit(num))
%timeit -n 10000000 factorial_recursive_jit(num)
```

```
3628800
211 ns ± 3.21 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[13]:
```
print(factorial_for_jit(num))
%timeit -n 10000000 factorial_for_jit(num)
```

```
3628800
192 ns ± 2.98 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[14]:
```
print(factorial_while_jit(num))
%timeit -n 10000000 factorial_while_jit(num)
```

```
3628800
184 ns ± 2.93 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

All the code now runs natively, so there is barely any between the `for` and `while` versions. Also, the recursive version is now more optimized, since the JIT compiler also performs compile-time optimizations.

---

6 Gaussian elimination

The below code performs Gaussian elimination with partial pivoting.

[15]:
```
# solves Ax = b and returns x
def solve(A, b):
    A = np.array(A, dtype=np.complex_)
    b = np.array(b, dtype=np.complex_)
    n = A.shape[0] # number of unknowns
    B = np.concatenate((A, np.expand_dims(b, axis=1)), axis=1) # augmented␣
 ↪matrix

    # bring B to row echelon form
    for i in range(0, n): # loop through diagonal elements
        # implement partial pivoting
        # find the maximum absolute value in this column
        max_k = i
        for k in range(i + 1, n):
            if np.abs(B[k][i]) > np.abs(B[max_k][i]):
                max_k = k

        if B[max_k][i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        # swap rows so that the maximum value is our new pivot
        B[[i, max_k]] = B[[max_k, i]]

        # reduce the values below
        for j in range(i + 1, n): # loop through rows below
            B[j,i:n+1] -= (B[j][i] / B[i][i]) * B[i,i:n+1]

    # now find the variables
    for i in range(n - 1, -1, -1): # loop up the matrix
        B[i][n] /= B[i][i]
        B[0:i,n] -= B[i][n] * B[0:i,i]
```

```
    return B[0:n,n]
```

We can use the above code to solve the following equation

$$\begin{bmatrix} 2 & 3 \\ 1 & -1 \end{bmatrix} \cdot X^T = \begin{bmatrix} 6 \\ \frac{1}{2} \end{bmatrix}$$

[16]:
```
np.real(solve([[2, 3], [1, -1]], [6, 1/2]))
```

[16]:
```
array([1.5, 1. ])
```

We need to do `np.real` because the function treats the numbers in the arrays as complex numbers.

The answer returned is:

$$X = \begin{bmatrix} \frac{3}{2} & 1 \end{bmatrix}$$

Now, if we feed in a singular matrix to the solver:

$$\begin{bmatrix} 1 & 2 & 1 \\ 1 & -1 & 0 \\ 2 & 1 & 0 \end{bmatrix} \cdot X^T = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

[17]:
```
np.real(solve([[1, 2, 1], [1, -1, 0], [2, 1, 1]], [1, 2, 3])) # this should␣
↪throw an error
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[17], line 1
----> 1 np.real(solve([[1, 2, 1], [1, -1, 0], [2, 1, 1]], [1, 2, 3])) # this␣
↪should throw an error

Cell In[15], line 18, in solve(A, b)
     15         max_k = k
     17 if B[max_k][i] == 0:
---> 18     raise ZeroDivisionError("unsolvable matrix")
     20 # swap rows so that the maximum value is our new pivot
     21 B[[i, max_k]] = B[[max_k, i]]

ZeroDivisionError: unsolvable matrix
```

6.1 Timing

Generate a random 10x10 matrix - the probability that this is singular is theoretically zero.

[18]:
```
A = np.random.rand(10, 10)
b = np.random.rand(10)
```

[19]:
```
print(np.real(solve(A, b)))
%timeit solve(A, b)
```

```
[ 0.99112942  1.71737079 -0.94346733  0.08045528  1.28104585 -1.9403032
  0.5330979  -0.16598857  0.28021515  0.53090258]
522 µs ± 15.9 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[20]:
```
print(np.linalg.solve(A, b))
%timeit np.linalg.solve(A, b)
```

```
[ 0.99112942  1.71737079 -0.94346733  0.08045528  1.28104585 -1.9403032
  0.5330979  -0.16598857  0.28021515  0.53090258]
7.25 µs ± 160 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As we can see above, the more complex the code we execute within Python, the larger the disparity between using library and hand-written functions. The function we have created runs 50-100 times slower than the numpy one written in C.

---

7 SPICE

The below code blocks set up a `SpiceSolver` class which can be used to load and solve SPICE netlists.

We use matrices of conductance, capacitance and 1/inductance values, as follows

Conductance matrix

$$\mathbf{G} = \begin{bmatrix} -(G_{11}+G_{12}+\cdots+G_{1n}) & G_{12} & \cdots & G_{1n} \\ G_{21} & -(G_{21}+G_{22}+\cdots+G_{2n}) & \cdots & G_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ G_{n1} & G_{n2} & \cdots & -(G_{n1}+G_{n2}+\cdots+G_{nn}) \end{bmatrix}$$

Capacitance matrix

$$\mathbf{C} = \begin{bmatrix} -(C_{11}+C_{12}+\cdots+C_{1n}) & C_{12} & \cdots & C_{1n} \\ C_{21} & -(C_{21}+C_{22}+\cdots+C_{2n}) & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & -(C_{n1}+C_{n2}+\cdots+C_{nn}) \end{bmatrix}$$

Inverse inductance matrix

$$\mathbf{L}_{\text{inv}} = \begin{bmatrix} -({L^{-1}}_{11}+{L^{-1}}_{12}+\cdots+{L^{-1}}_{1n}) & {L^{-1}}_{12} & \cdots & {L^{-1}}_{1n} \\ {L^{-1}}_{21} & -({L^{-1}}_{21}+{L^{-1}}_{22}+\cdots+{L^{-1}}_{2n}) & \cdots & {L^{-1}}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ {L^{-1}}_{n1} & {L^{-1}}_{n2} & \cdots & -({L^{-1}}_{n1}+{L^{-1}}_{n2}+\cdots+{L^{-1}}_{nn}) \end{bmatrix}$$

At a given frequency ω, the net complex conductance of the circuit is given by

$$\mathbf{Y} = \mathbf{G} + j\omega\mathbf{C} + \frac{1}{j\omega}\mathbf{L}_{\text{inv}}$$

7.1 DC

For DC, we replace all inductors with 0-voltage sources, and set all capacitors to zero-conductances.

7.2 Modified nodal analysis

We follow the sign convention where current going out of a node is considered positive.

For every voltage source, we take the current through it as a variable to solve for, and we apply KCL at each node. Assuming there is a voltage source V_{12} between nodes 1 and 2, and a current source I_{23} between nodes 2 and 3, we can represent the equation as follows:

$$\begin{bmatrix} -(Y_{11}+Y_{12}+Y_{13}) & Y_{12} & Y_{13} & -1 \\ Y_{21} & -(Y_{21}+Y_{22}+Y_{23}) & Y_{33} & 1 \\ Y_{31} & Y_{32} & -(Y_{31}+Y_{32}+Y_{33}) & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \cdot X = \begin{bmatrix} 0 \\ I_{23} \\ -I_{23} \\ V_{12} \end{bmatrix}$$

7.3 General class definitions

The classes below are representations of different components that we will use in the `SpiceSolver` class.

[21]:
```python
# an edge in the network that goes between two nodes
class Edge:
    def __init__(self, name, node_left, node_right):
        self.name = name
        self.node_left = node_left
        self.node_right = node_right

# a general passive component
class Passive(Edge):
    def __init__(self, name, node_left, node_right, value, cond=0):
        Edge.__init__(self, name, node_left, node_right)
        self.value = value
        self.cond = cond # condition, like voltage of a capacitor

# a general source (voltage or current)
class Source(Edge):
    def __init__(self, name, node_left, node_right, value, phase):
        Edge.__init__(self, name, node_left, node_right)
        self.value = value
        self.phase = phase
```

7.4 Spice Solver class

[22]:
```python
# class that parses a spice file and solves it
class SpiceSolver:
    # throw an error specifying the line where the error occurred
    @staticmethod
```

```
    def __spice_err(line_index, line, message):
        raise Exception(
            f"SPICE ERROR on line {line_index + 1}:\n" +
            f"{line}\n" +
            message
        )

    # parse a float, and throw an error if it's not valid
    @staticmethod
    def __parse_float(x, line_index, line):
        try:
            x = float(x)
        except ValueError:
            SpiceSolver.__spice_err(line_index, line, f"couldn't parse as␣
↪number: `{command[3]}`")
        return x

    # validate the number of arguments to a command
    @staticmethod
    def __assert_arg_count(command, expected, line_index, line):
        if len(command) not in expected:
            SpiceSolver.__spice_err(line_index, line, f"invalid number of␣
↪arguments: expected {expected}, got {len(command)}")

    def __init__(self):
        # we will replace all nodes with numbers in our matrix
        # this map will remember the mapping between node names and the␣
↪numbering we assign
        self.node_map = { "GND": 0 }

        # number of nodes (excluding ground)
        self.node_count = 0

        # lists of sources by frequency
        self.voltages = { 0: [] }
        self.currents = { 0: [] }

        # store a list of components
        self.resistors = []
        self.capacitors = []
        self.inductors = []

        # we represent the passive elements in the circuit as 3 weighted graphs␣
↪where
        # components form edges:
        # one has conductances as its weights
        # one has capacitance as its weights
```

```
        # one has 1/inductance as its weights

        # we use the diagonal elements as the negative of the sum of the rest␣
↪of the
        # values in the corresponding row (plus ground), since this is what␣
↪appears
        # in the MNA matrix

        # adjacency matrix of conductances between nodes (excluding GND)
        self.conductance_matrix = None

        # adjacency matrix of capacitances between nodes (excluding GND)
        self.capacitance_matrix = None

        # adjacency matrix of 1/inductances between nodes (excluding GND)
        self.inv_inductance_matrix = None

    # read a file and create a representation of the circuit in memory
    def read_file(self, filename):
        # open file
        with open(filename) as f:
            l = f.readlines()

            if len(l) == 0: # make sure the file is not empty
                raise Exception("SPICE ERROR: empty file")

            # seek to start of circuit
            start = 0
            while len(l[start]) > 0 and l[start].split("#")[0].strip() != ".
↪circuit":
                start += 1
                if start >= len(l): # we need to have a ".circuit" somewhere
                    raise Exception("SPICE ERROR: couldn't find start of␣
↪circuit")

            # find end of circuit
            end = start
            while len(l[end]) > 0 and l[end].split("#")[0].strip() != ".end":
                end += 1
                if end >= len(l): # we need to have a ".end" somewhere
                    raise Exception("SPICE ERROR: couldn't find end of circuit")

            # generate a dictionary of source names to frequencies based on the␣
↪directives given after ".end"
            frequencies = {}

            # loop through lines after the ".end" directive
```

```
            for i in range(end + 1, len(l)):
                command = l[i].split('#')[0].split() # get command

                if command[0] == "": # if there's no command, move on
                    continue

                # look for and parse ".ac" command
                if command[0].lower() == ".ac":
                    self.__assert_arg_count(command, [3], i, l[i])

                    if command[1].upper() in frequencies: # each source should␣
↪have its frequency defined only once
                        self.__spice_error(i, l[i], "redefinition of freqency")

                    # add frequency to dictionary
                    freq = self.__parse_float(command[2], i, l[i])
                    frequencies[command[1].upper()] = freq

                    self.voltages[freq] = []
                    self.currents[freq] = []
                else:
                    pass # we don't need to throw errors if there is garbage␣
↪after the ".end"

            # generate a list of nodes and components, and check syntax
            for i in range(start + 1, end): # loop through the ".circuit"␣
↪section
                command = l[i].split('#')[0].split()

                if command[0] == "":
                    continue

                if len(command) < 4: # all commands have at least 4 arguments
                    self.__spice_error(i, l[i], "invalid number of arguments")

                # the second and third arguments will contain node names
                # add the nodes mentioned to our node map if they aren't␣
↪already in it
                for j in [1, 2]:
                    node_name = command[j]
                    if node_name.upper() not in self.node_map:
                        self.node_count += 1
                        self.node_map[node_name.upper()] = self.node_count

                # type of component
                ctype = l[i][0].upper()
```

```
                # parse passive components
                if ctype in ["R", "L", "C"]:
                    self.__assert_arg_count(command, [4, 5], i, l[i])

                    edge = Passive(
                        command[0],
                        self.node_map[command[1].upper()],
                        self.node_map[command[2].upper()],
                        self.__parse_float(command[3], i, l[i])
                    )

                    if len(command) == 5: # if an initial condition is␣
↪specified (5th argument), set it
                        edge.cond = self.__parse_float(command[4], i, l[i])

                    # add value to list
                    if ctype == "R":
                        self.resistors.append(edge)
                    elif ctype == "L":
                        self.inductors.append(edge)
                    else:
                        self.capacitors.append(edge)

                # parse sources
                elif ctype in ["V", "I"]:
                    freq = 0
                    phase = 0

                    # check type of source
                    if command[3].lower() == "ac":
                        self.__assert_arg_count(command, [6], i, l[i])

                        # look up our frequencies dictionary for this source
                        if command[0].upper() in frequencies:
                            freq = frequencies[command[0].upper()]
                            phase = self.__parse_float(command[5], i, l[i])
                        else: # throw an error if the frequency for this source␣
↪is not defined
                            self.__spice_error(i, l[i], f"could not find␣
↪frequency for AC source: `{command[0]}`")

                    elif command[3].lower() == "dc":
                        self.__assert_arg_count(command, [5], i, l[i])

                    else:
                        self.__spice_error(i, l[i], f"invalid source type:␣
↪`{command[3]}`")
```

```
                    edge = Source(
                        command[0],
                        self.node_map[command[1].upper()],
                        self.node_map[command[2].upper()],
                        self.__parse_float(command[4], i, l[i]),
                        phase,
                    )

                    if ctype == "V":
                        self.voltages[freq].append(edge)
                    else:
                        self.currents[freq].append(edge)

                else:
                    self.__spice_error(i, l[i], "unidentified command inside␣
↪circuit")

        # delete GND
        del self.node_map["GND"]

        # now use the data we have parsed to generate matrices
        self.__gen_matrices()

    # generate conductance, capacitance and inverse-inductance matrices
    def __gen_matrices(self):
        # initialize to zeroes
        self.conductance_matrix = np.zeros((self.node_count, self.node_count))
        self.capacitance_matrix = np.zeros((self.node_count, self.node_count))
        self.inv_inductance_matrix = np.zeros((self.node_count, self.
↪node_count))

        # conductance matrix
        for r in self.resistors:
            low = min(r.node_left, r.node_right) - 1
            high = max(r.node_left, r.node_right) - 1
            if low == -1:
                # if the resistor is between ground and a node, add it to the␣
↪diagonal
                self.conductance_matrix[high][high] += 1 / r.value
            else:
                self.conductance_matrix[low][high] += 1 / r.value

        # capacitance matrix
        for c in self.capacitors:
            low = min(c.node_left, c.node_right) - 1
            high = max(c.node_left, c.node_right) - 1
```

```
            if low == -1:
                self.capacitance_matrix[high][high] += c.value
            else:
                self.capacitance_matrix[low][high] += c.value

        # inverse-inductance matrix
        for l in self.inductors:
            low = min(l.node_left, l.node_right) - 1
            high = max(l.node_left, l.node_right) - 1
            if low == -1:
                self.inv_inductance_matrix[high][high] += 1 / l.value
            else:
                self.inv_inductance_matrix[low][high] += 1 / l.value

        # mirror our matrices and fix the diagonal elements
        mats = [self.conductance_matrix, self.inv_inductance_matrix, self.
↪capacitance_matrix]
        for mat in mats:
            for i in range(self.node_count):
                # set diagonal elements
                for j in range(0, i):
                    mat[i][i] -= mat[j][i]

                # mirror
                for j in range(i + 1, self.node_count):
                    mat[i][i] -= mat[i][j]
                    mat[j][i] = mat[i][j]

    # create a linear matrix equation A x = b for MNA (returns A and b)
    def __gen_mna_pair(self, modified_voltages, freq):
        w = 2j * np.pi * freq

        dim = self.node_count + len(modified_voltages) # number of dimensions␣
↪in matrix

        mat = np.zeros((dim, dim), dtype=np.complex_)
        x = np.zeros(dim, dtype=np.complex_)


        if w != 0:
            mat[0:self.node_count, 0:self.node_count] = self.conductance_matrix␣
↪+ \
                                                        1j * w * self.
↪capacitance_matrix + \
                                                        1 / (1j * w) * self.
↪inv_inductance_matrix
        else:
```

```
            mat[0:self.node_count, 0:self.node_count] = self.conductance_matrix

        # outward currents are taken as positive, active sign convention is␣
↪followed for voltage sources

        k = self.node_count

        # create MNA matrix

        # add equations of the form V_left - V_right = V_source
        # and also currents on the RHS of KCL equations
        for v in modified_voltages:
            if v.node_left != 0:
                mat[v.node_left - 1][k] = -1 # current
                mat[k][v.node_left - 1] = 1 # voltage

            if v.node_right != 0:
                mat[v.node_right - 1][k] = 1 # current
                mat[k][v.node_right - 1] = -1 # voltage

            x[k] = v.value * np.exp(1j * v.phase)

            k += 1

        for i in self.currents[freq]:
            if i.node_left != 0:
                x[i.node_left - 1] = i.value * (np.exp(1j * i.phase))

            if i.node_right != 0:
                x[i.node_right - 1] = -i.value * (np.exp(1j * i.phase))

        return (mat, x)

    # generate modified voltage sources according to frequency and solve,
    # returning a dictionary of nodes to voltages/currents
    def __solve_freq(self, freq):
        modified_voltages = []

        # map of extra nodes that we have added
        extra_nodes = {}

        # convert inductors to 0-voltage sources for DC
        if freq == 0:
            for l in self.inductors:
                modified_voltages.append(
                    Source(l.name, l.node_left, l.node_right, 0, 0)
                )
```

```
        for f in self.voltages:
            # if the source is at a different frequency, short it
            if freq != f:
                for v in self.voltages[f]:
                    modified_voltages.append(
                        Source(v.name, v.node_left, v.node_right, 0, 0)
                    )
            else:
                for v in self.voltages[f]:
                    modified_voltages.append(v)

        # keep track of where each voltage is
        new_node_count = self.node_count
        for v in modified_voltages:
            new_node_count += 1
            extra_nodes[f"I_{v.name}"] = new_node_count

        solution = solve(*self.__gen_mna_pair(modified_voltages, freq))

        if freq == 0:
            solution = np.real(solution)

        result = {}

        for key in self.node_map:
            result[f"V_{key}"] = solution[self.node_map[key] - 1]

        for key in extra_nodes:
            result[key] = solution[extra_nodes[key] - 1]

        return result

    # print the steady state solution of the system
    def solve_steady(self):
        for freq in self.voltages:
            print(f"frequency {freq}:")
            print()
            try:
                sol = self.__solve_freq(freq)
                for key in sol:
                    print(f"{key:<10}{sol[key]}")
            except ZeroDivisionError:
                print("no steady state at this frequency")

            print()
            print("=" * 20)
```

```
            print()
```

We can invoke the solver as follows:

It prints the different responses of the circuit to the different frequencies of sources that are present.

[23]:
```
solver = SpiceSolver()
solver.read_file("spice/ckt1.netlist")
solver.solve_steady()
```

```
frequency 0:

V_1       0.0
V_2       -0.0
V_3       -0.0
V_4       -5.0
I_V1      0.0005

====================
```