# Homework 2: Circuit 1

ME492 instruction revised Oct. 3

due extended by midnight on October 6, 2024

## 1 Introduction

Revision: This instruction file contains all the information from the previous version, with additional, edited, and new parts highlighted in blue.

What is OOP? OOP, or Object-Oriented Programming, is a programming paradigm based on the concept of 'objects,' which can contain data in the form of fields (often called attributes or properties) and code in the form of methods. In this homework, you will eventually create a class called **`Circuit`**, which represents a virtual circuit object that can calculate the voltage and current at each node.

Note that this homework will be extended and used in your midterm project. While asking AI for some knowledge might be okay, relying on AI for direct answers could prevent you from fully understanding the concepts needed for the midterm project.

## 2 Brief Explanation

You may remember solving for voltage and current in arbitrary electric circuit from General Physics. In Homework 2, you will construct classes to calculate nodal voltage and current values for any circuit containing voltage source and resistors. For the Midterm project, we will provide the solution code from Homework 2 and you will further extend the Homework 2 programming code to solve for blood flow and pressure in a simplified cardiovascular system.

For practicing OOP and easy grading, the basic structure of the code will be provided as a skeleton code. You are free to add your own methods to the class, but **do not change the names of the predeclared methods and classes**. These classes and methods will be imported directly into the grader, so if you change the names or delete any of them, you might not even receive partial credit for that problem.

There are four Python files in this project: **`circuit.py`**, **`elements.py`**, **`matrix.py`**, and **`node.py`**. Two files, **`node.py`** and **`elements.py`**, will be provided, and you won't need to edit them unless necessary. The other two files, **`matrix.py`** and **`circuit.py`**, contain skeleton code that you should complete. Please read these instruction carefully to understand what you should do.

If you are unfamiliar with or confused by Python syntax and operations, or if you have any questions about this homework, feel free to ask via Classum or email.

## 3 Class Matrix

For solving circuit, you should construct the class Matrix which will be used in class Circuit. Necessary explanation will be commented in the skeleton code too.

Pre-implemented methods are **`__init__()`**, **`__getitem__()`**, **`input_check()`**, **`value_builder()`**, **`value_checker()`**. You don't have to modify those methods, so if your code editor or IDE supports code folding, it is recommended to just fold it. Methods you have to implement are **`matrix_add`**, **`matrix_sub`**, **`matrix_mul`**, **`determinant`**, **`multiply`**, **`solve`**. Details are written below.

```
class Matrix:
# Examples in Appendix A.
  def matrix_add(self, matrix_1, matrix_2=None): # do you know default parameter?
# If matrix_2 is None: return the result of self_matrix + matrix_1
# If matrix_2 is not None: return the result of matrix_1 + matrix_2
# The return type should be an instance of the Matrix class

  def matrix_sub(self, matrix_1, matrix_2=None):
# If matrix_2 is None: return the result of self_matrix - matrix_1
# If matrix_2 is not None: return the result of matrix_1 - matrix_2
# Note that matrix_1 is not always the first operand
# The return type should be an instance of the Matrix class

  def matrix_mul(self, matrix_1, matrix_2=None):
# If matrix_2 is None: return the product of the instance matrix (self) and matrix_1
# If matrix_2 is not None: return the product of matrix_1 and matrix_2
# Note that matrix_1 is not always the first operand
# The return type should be an instance of the Matrix class

  def determinant(self):
# Calculate and return the determinant of the instance matrix (self).
# The return type should be of type float.

  def multiply(self, factor, matrix_1=None):
# If matrix_1 is None: multiply each element of self_matrix by the factor
# If matrix_1 is provided: multiply each element of matrix_1 by the factor
# The factor should be a float
# The return type should be an instance of the Matrix class

  def solve(self, matrix_b):
# Let the self_matrix be A and matrix_b be B. Solve the equation AX = B
# Return the solution matrix X.
# The return type should be an instance of the Matrix class.
# You can use either NumPy module or implement the method by your own

```

Detail on the usage of the **`Matrix`** class, along with examples, can be found in **Appendix A**.
The code you need to fill in for **`matrix.py`** will likely be about 8 to 15 lines per method. It would take a short amount of time if you have basic knowledge of matrix and Python class. Partial credit will be given for each method.

## 4 Class node & Class Element

Two files, **`node.py`**, **`element.py`** will be fully provided, so only a brief explanation will be given in this section. The Node class contains variables: index, voltage, and adjacent. The index is a number used for convenient calculation and grading, while the voltage represents the actual voltage level of the node (which will remain **`None`** before calculation in **`circuit.py`**). However, node with index 0 is a special node that denotes ground (where the voltage is zero). The variable adjacent is a list that contains adjacent elements directly connected to that node. The **`input_check`** method is for efficient error detection, so you don't need to fully understand it.

Element is a parent class that contains the information on the connected nodes from both sides (head and tail) which is inherited by the VoltageSource and Resistor classes. For the VoltageSource, the head_node is connected to the negative side, and the tail_node is connected to the positive side.(Refer to the image in **Appendix B**) For resistors, the head_node is considered the node with relatively higher voltage compared to the tail_node. Although we don't yet know the nodal voltage, we need to determine the direction of the unknown current for calculation purposes, so we will make an assumption. If the result turns out to be opposite, we can simply consider the current as negative, with the direction being from head to tail. You will be able to understand these two files **`node.py, element.py`** by examining the **Appendix B**.

## 5 Class Circuit

For your convenience, the initialization and circuit construction method will be provided. Details are written below:

```
Class Circuit:
  def __init__(self, nodes, elements, relations): # already provided
    # nodes: a list that contains nodes
    # Ex: [Node(0), Node(1), Node(2), Node(3)]
    # elements: a list that contains element.
    # Ex: [VoltageSource(0, 1), Resistor(1, 1), Resistor(2, 2),
    # Resistor(3, 3), Resistor(4, 4)]
    # The first parameter of element is its index (unique number),
    # the second parameter is either the resistance or the voltage.
    # relations: a list that contains lists of relation of
    # [element_index, head_node_index, tail_node_index]
    # Ex: [[0, 0, 1], [1, 1, 2], [2, 2, 3], [3, 3, 0], [4, 3, 0]]
  def construct_circuit(self): # already provided
    # Since the head_node and tail_node of element and adjacent of nodes are empty,
    # This method fill in those.
  def solve_node_voltage(self): # you should implement!
    # This is the key method for Homework 2.
    # You should construct system of equation based on Kirchhoff's Current Law and
    # Update node voltages.
    # You should return the node voltage list in the order of node indices.
```

If you are unsure about how the virtual circuit is constructed with this information, refer to **Appendix C** for examples. You may also ask the TA for more details. The Code you need to write could be quite long.

# 6 Grading

Partial points will be awarded for each class method based on several test cases. Some test cases will be provided, while others will remain hidden. There will be **no test cases designed to produce error on purpose**. It is guaranteed that every test case will produce a normal result, so you don't need to write some try-except structures or raise exceptions.

The Matrix class will be worth a total of **60 points**, and the Circuit class will be worth **40 points**.

The six methods in Matrix: **`matrix_add, matrix_sub, matrix_mul, determinant, multiply, solve`** will each be worth 10 points. If you implement the **`solve`** method without using any external modules (such as NumPy), bonus point will be awarded. Partial points will be given for corrected test cases, even if not all cases are correct. During grading, small errors due to float-point calculations will be allowed flexibly.

In Circuit class, **`solve_node_voltage`** will be worth a total of 40 points, and multiple test cases will be used to evaluate this method. **Each test case will contain one voltage source, one ground, and one or more resistors.** There will be no test cases without a voltage source, ground, or resistor.

A sample grading file **`grader.ipynb`** is now provided. One or two test cases are included for each method. You can run the **`.ipynb`** file locally with your own Python environment. Since the same file (with additional different cases) will be used for grading, ensure that your code works without errors with sample grader.

# 7 Executing

In this homework, there are multiple files: Python files you need to complete, a grader file that checks your code, and example test cases. Ensure that all the files are located in the same folder, as execution may not work correctly otherwise. Unlike Homework 1, the file format is not **`.ipynb`**, so it may be difficult to use Google Colab or Jupyter Notebook.

The file format is **`.py`** because learning Object-Oriented Programming using the **`.ipynb`** format file is difficult and inefficient. Since this is the first homework that requires you to run Python on your local device, some of you may be unfamiliar with running Python locally or may not know how to install it. If this is the case, please feel free to ask the TA through Classum.

It is highly recommended to use PyCharm or VSCode to execute Python on your local device. If you need instructions on installing Python or using PyCharm or VSCode, please ask TA or the professor for the guidance.

# Appendix A

Usage of **`Matrix`** class methods.

```python
# mtx1: initialized with row and column
mtx1 = Matrix(2, 2) # 2 by 2 matrix with value initialized with 0.
print(mtx1.value) # [[0, 0], [0, 0]]

# mtx2 = Matrix() : Incorrect usage
# mtx3 = Matrix(4) : Incorrect usage

# mtx5: initialized with row, column, value
mtx5 = Matrix(2, 2, [[1, 2], [3, 4]])
# mtx6: initialized with only the value
mtx6 = Matrix(value=[[3, 4], [5, 6]])
# mtx7 = Matrix(2, 3, [[1, 2], [3, 4]]) : Incorrect usage
# mtx8 = Matrix(2, 2, [[1, 2, 3], [4, 5]]) : Incorrect usage

#================ above cases are already implemented ==================
#======== below cases will work well if you implement correctly =========


mtx9 = mtx5.matrix_add(mtx6) # matrix_add
print(mtx9.value) # [[4, 6], [8, 10]]
mtx10 = mtx5.matrix_sub(mtx6) # matrix_sub
print(mtx10.value) # [[-2, -2], [-2, -2]]
mtx10_2 = Matrix.matrix_sub(mtx1, mtx6) # same method with different parameters
print(mtx10_2.value) # [[-3, -4], [-5, -6]]
mtx11 = mtx5.matrix_mul(mtx6) # matrix_mul
print(mtx11.value) # [[13, 16], [29, 36]]
mtx11_det = mtx11.determinant() # determinant
print(mtx11_det) # 4
mtx12 = mtx6.multiply(3) # multiply
print(mtx12.value) # [[9, 12], [15, 18]]

mtx13 = Matrix(2, 1)
mtx13.value = [[10], [1]] # be careful this shouldn't be [10, 1] or [[10, 1]]!!!
print(mtx5.solve(mtx13).value) # [[-19.0], [14.5]]
```

Although some incorrect usage is shown, these will not be in the test cases. Several types of incorrect usage will be filtered out by the **`input_check()`** method, in case there are errors in your own test case. However, be aware that it does not catch every possible incorrect usage.

# Appendix B

Usage of **`Node`**, **`VoltageSource`**, and **`Resistor`** classes.

```
n = [Node(0), Node(1), Node(2)] # list of nodes with initialized index
print(n[0].voltage) # because it is ground, 0
print(n[1].voltage) # not solved yet, None

e = [VoltageSource(0,2), VoltageSource(1,3), Resistor(2,8)]
# list of elements initialized with index and property(voltage or resistance)
print(e[0].voltage) # 2
print(e[1].voltage) # 3
print(e[1].index) # 1
print(e[2].resistance) # 8
# print(e[0].resistance) : AttributeError

print(e[0].head_node) # None
e[0].head_node = n[0]
e[0].tail_node = n[1]
n[0].adjacent.append(e[0])
n[1].adjacent.append(e[0])
print(e[0].head_node.index) # 0
print(e[0].head_node.adjacent[0].head_node.adjacent[0].tail_node.adjacent[0].
    tail_node.index) # 1
# Does this look wrong? If so, search about Python referencing
# You should understand this to implement circuit.py
```

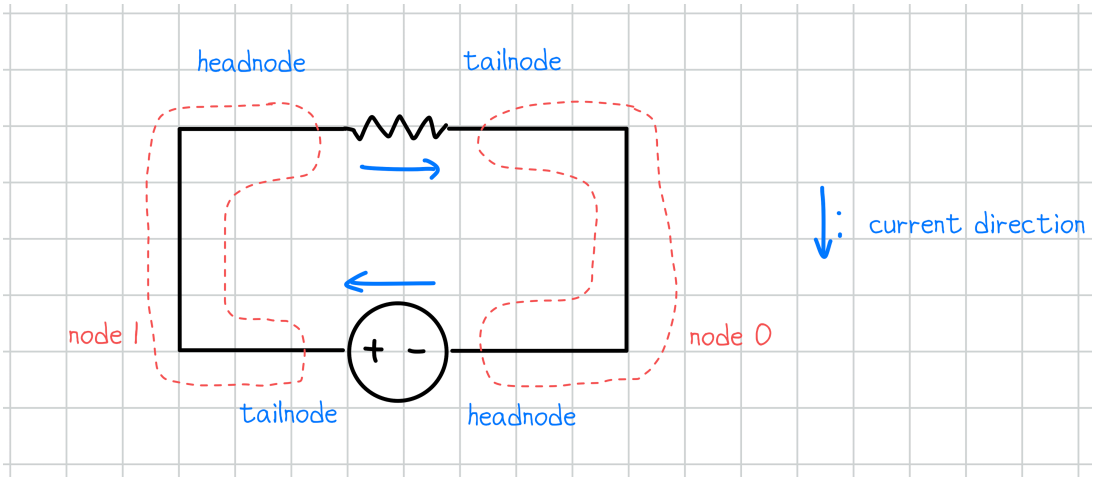


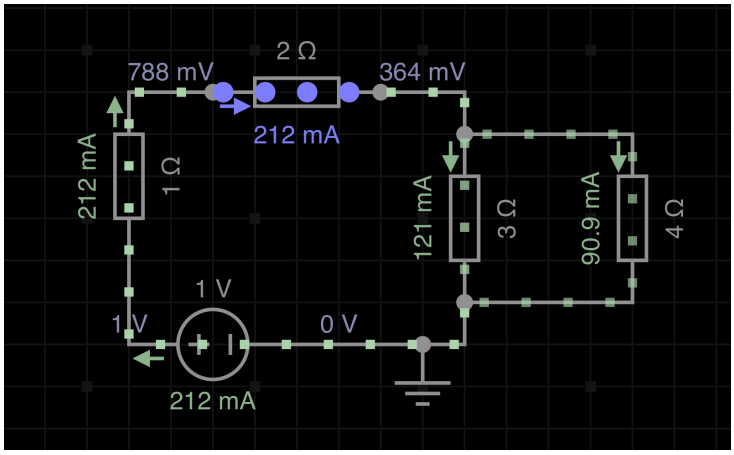

Figure and explanation added: Since we consider the current direction to be from head to tail, in the above case, the head and tail nodes of the voltage source are node 0 and node 1, respectively, while for the resistor, they are node 1 and node 0, respectively.

It is okay to choose your own current direction as long as you maintain consistency; you will still arrive at the correct answer. However, it is better to follow the chosen direction to ensure a better understanding of the test cases.

# Appendix C

Example of Circuit:

```
ccc = Circuit([Node(0), Node(1), Node(2), Node(3)],[VoltageSource(0,1), Resistor
    (1,1), Resistor(2,2), Resistor(3,3), Resistor(4,4)],[[0, 0, 1], [1, 1, 2], [2,
    2, 3], [3, 3, 0], [4, 3, 0]]).solve_node_voltage()
for  i in range(len(ccc)):
  print(i , ":", ccc[i], "V")
# result:
# 0 : 0.0 V
# 1 : 1.0 V
# 2 : 0.78788 V
# 3 : 0.36364 V
```




It is recommended to use Ohm's law and Kirchhoff's Current Law rather than Kirchhoff's Voltage Law because it is easier to implement. How should the solve method construct the system of equations? Let's denote $V_i$ as voltage at node $i$, $R_i$ as the resistance of the resistor element with index $i = 1$ to 3, and $I_i$ is the current passing the element with index $i$, in the head-to-tail direction. We know that $V_0$ is $0V$ because it is grounded. Now we have 8 unknowns, $V_1, V_2, V_3, I_0, I_1, I_2, I_3, I_4$. For completeness, let's also consider $V_0$ as an unknown. (Below equation modified! The previous version contained an error.)

$$V_0 = 0 \tag{1}$$

$$V_1 = V_0 + 1 \tag{2}$$

$$V_2 = V_1 - R_1 \cdot I_1 \tag{3}$$

$$V_3 = V_2 - R_2 \cdot I_2 \tag{4}$$

$$V_0 = V_3 - R_3 \cdot I_3 \tag{5}$$

$$V_0 = V_3 - R_4 \cdot I_4 \tag{6}$$

$$I_1 = I_0 \tag{7}$$

$$I_2 = I_1 \tag{8}$$

$$I_2 = I_3 + I_4 \tag{9}$$

Equations (7), (8) and (9) is from KCL: $\sum i_{\text{entering node}} - \sum i_{\text{leaving node}} = 0$. We know that the equation $I_0 = I_3 + I_4$ is also true, but it can be derived from other equations. Including it might lead to a singular matrix error when solving the system. Apologies for the previous version!

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & -R_1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & -R_2 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & -R_3 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -R_4 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \\ I_0 \\ I_1 \\ I_2 \\ I_3 \\ I_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{10}$$

Now can you see how it works? You now know how to construct system of equation and solve it!