# Natalia_Tabja_Lab4

December 6, 2024

## 0.1 Finding Roots of a Non-Linear Equation

Consider the equation: $5e^{-x} + x - 5 = 0$ . Suppose we want to solve it with (absolute) accuracy tolerance $\epsilon = 10^{-6}$.

[1]:
```
import numpy as np

# Define the function g(x)
def g(x):
    return 5*(1 - np.exp(-x))

# Define the function f(x)
def f(x):
    return g(x) - x

epsilon = 1e-6
```

### 0.1.1 Exercise 1

Solve using binary search. How many iterations are required?

[2]:
```
# Method 1: Binary Search
a = 0.1
b = 100.0



if f(a) * f(b) > 0:
    print("Root is not bracketed in the interval [a, b].")

else:
    iterations = 0
    while abs(b - a) > epsilon:
        # Calculate the midpoint
        midpoint = (a + b) / 2

        # Check if midpoint is close enough to the root
        if abs(f(midpoint)) < epsilon:
            break
```

```
        # Update either a or b based on the sign of f(midpoint)
        if f(a) * f(midpoint) < 0:
            b = midpoint
        else:
            a = midpoint

        iterations += 1

print(f"Root found at x = {midpoint} after {iterations} iterations.")
```

```
Root found at x = 4.965114106237888 after 25 iterations.
f(x) at root: 1.211280018509342e-07
```

### 0.1.2 Exercise 2

Solve using relaxation. How many iterations are required?

[3]:
```
a = 0.
b = 1000.
iterations = 0

while abs(b - a) > epsilon:
    a = b
    b = g(a)
    iterations += 1

print(f"Root found at x = {b} after {iterations} iterations.")
```

```
Root found at x = 4.96511423351466 after 6 iterations.
```

### 0.1.3 Exercise 3

Solve using Newton's Method. How many iterations are required?

[4]:
```
def f_prime(x):
    return 5 * np.exp(-x) - 1


def newtons_method(x0):
    x = x0 # Initial guess
    iterations = 0
    while (abs(f(x)) > epsilon):
        x = x - f(x) /f_prime(x)
        iterations += 1

    return x, iterations

# Initial guess
```

```
x0 = 1.0
root, iterations = newtons_method(x0)

print(f"Root found at x = {root} after {iterations} iterations.")
```

```
Root found at x = -3.868258278188412e-13 after 7 iterations.
```

### 0.1.4 Exercise 4

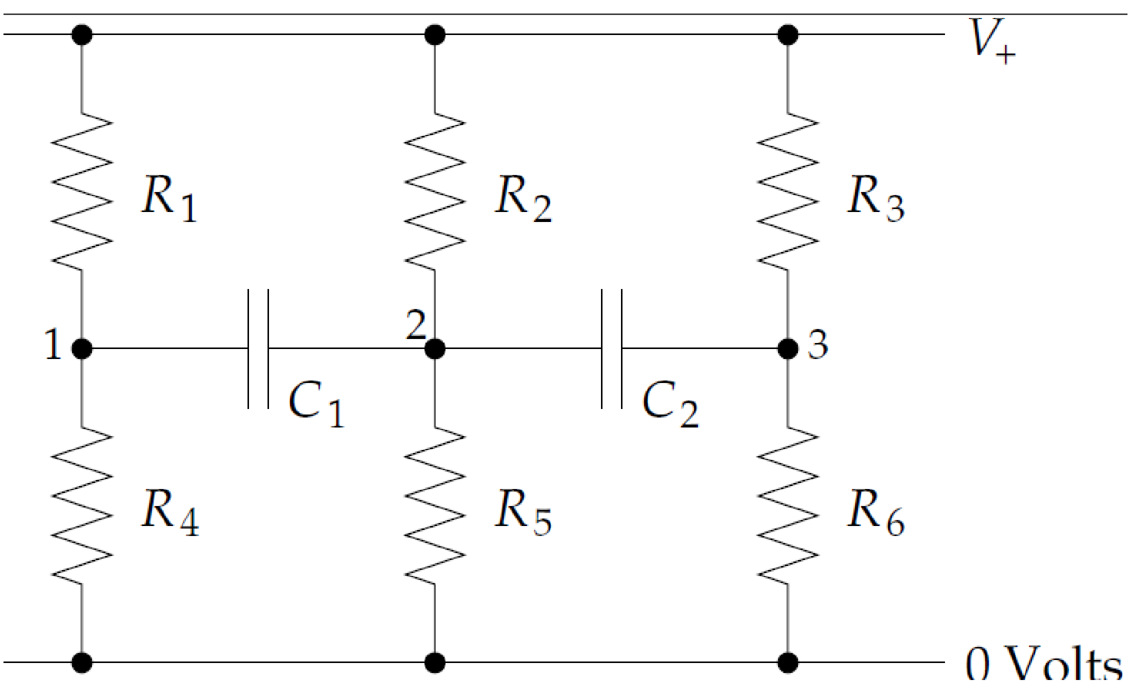
Solve using Secant Method. How many iterations are required?

[5]:
```
x1 = 0.
x2 = 10.
x3 = x2 - f(x2)*(x2 - x1)/(f(x2) - f(x1))

iterations = 1
while abs(x3 - x2) > epsilon:
    iterations += 1
    x1 = x2
    x2 = x3
    x3 = x2 - f(x2)*(x2 - x1)/(f(x2) - f(x1))

print(f"Root found at x = {x2} after {iterations} iterations.")
```

```
Root found at x = 0.0 after 2 iterations.
```

## 0.2 Fun with Circuits

### 0.2.1 Physics background



Consider the above circuit. Suppose the voltage $V_+$ is time-varying and sinusoidal of the form $V_+ = x_+ \exp(i\omega t)$ with $x_+$ a constant.

The resistors in the circuit can be treated using Ohm's law. For the capacitors the charge $Q$ and voltage $V$ across them are related by the capacitor law $Q = CV$, where $C$ is the capacitance. Differentiating both sides of this expression gives the current $I$ flowing in on one side of the capacitor and out on the other:

$$I = \frac{dQ}{dt} = C\frac{dV}{dt}. \tag{1}$$

Now assume the voltages at the points labeled 1, 2, and 3 are of the form $V_1 = x_1 \exp(i\omega t)$, $V_2 = x_2 \exp(i\omega t)$, and $V_3 = x_3 \exp(i\omega t)$. If you add up the currents using Kirchoff's law that at a junction the sum of the currents in equals the sum of the currents out, you can find that the constants $x_1$, $x_2$, and $x_3$ satisfy the equations

$$\begin{aligned} \left(\frac{1}{R_1} + \frac{1}{R_4} + i\omega C_1\right) x_1 - i\omega C_1 x_2 &= \frac{x_+}{R_1}, \\ -i\omega C_1 x_1 + \left(\frac{1}{R_2} + \frac{1}{R_5} + i\omega C_1 + i\omega C_2\right) x_2 - i\omega C_2 x_3 &= \frac{x_+}{R_2}, \\ -i\omega C_2 x_2 + \left(\frac{1}{R_3} + \frac{1}{R_6} + i\omega C_2\right) x_3 &= \frac{x_+}{R_3}. \end{aligned}$$

This is a linear system of equations for three complex numbers, $x_1$, $x_2$, and $x_3$.

We will be solving the above linear system of equations in the form $Ax = b$, where $x$ is the vector $(x_1 x_2 x_3$ and $b$ is the vector composed of the right-hand sides of the equations above.

The following function takes as input the list of resistance values ($R_1$ to $R_6$) and the list of capacitances ($C_1$ and $C_2$), and returns (as numpy.array) the matrix $A$.

[6]:
```
def CircuitMatrix(R_, C_):
    """ I define the matrix A as a function of the one element we turn from a
    resistor to an inductor
    IN: element [complex]: the resistor or inductor
    R_ [float]: list of resistors R1 to R5
    C_ [complex]: list of capacitances C1 and C2
    """
    A = np.empty((3, 3), complex)

    # 1st line of matrix
    A[0, 0] = 1./R_[1] + 1./R_[4] + C_[1]
    A[0, 1] = -C_[1]
    A[0, 2] = 0.

    # 2nd line of matrix
    A[1, 0] = -C_[1]
    A[1, 1] = 1./R_[2] + 1./R_[5] + C_[1] + C_[2]
    A[1, 2] = -C_[2]

    # 3rd line of matrix
    A[2, 0] = 0.
```

```
    A[2, 1] = -C_[2]
    A[2, 2] = 1./R_[3] + 1./R_[6] + C_[2]

    return A
```

And the following function takes as input the list of resistance values and the value of $x+$, and returns (as numpy.array) the vector $b$.

[11]:
```
def RHS(R, xplus):
    return xplus*np.array([1./R[1], 1./R[2], 1./R[3]], complex)
```

### 0.2.2 Exercise 5

Use Gaussian Elimination with partial pivoting (see the code fragment below) to solve for $x_1$, $x_2$, and $x_3$. Assume the following:

$$
\begin{aligned}
&R_1 = R_3 = R_5 = 1\,\mathrm{k}\Omega,\\
&R_2 = R_4 = R_6 = 2\,\mathrm{k}\Omega,\\
&C_1 = 1\,\mu\mathrm{F}, \qquad C_2 = 0.5\,\mu\mathrm{F},\\
&x_+ = 3\,\mathrm{V}, \qquad \omega = 1000\ \mathrm{rad/s}.
\end{aligned}
$$

Have your program calculate and print, at $t = 0$, the amplitudes of the three voltages $|V_1|$, $|V_2|$, and $|V_3|$ and their phases (i.e. the phases of the coefficients $x_1, x_2, x_3$) in degrees.

*Notice that the matrix for this problem has complex elements. You will need to define a complex array to hold it, but your routine should be able to work with real or complex arguments.*

*Hint: the built-in abs() will compute the magnitude, and numpy.angle() will compute the phase of a complex number. You could also use polar and phase from the cmath package.*

[8]:
```
import numpy as np
def GaussElim(A_in, v_in, pivot=False):
    """Implement Gaussian Elimination. This should be non-destructive for input
    arrays, so we will copy A and v to temporary variables
    IN:
    A_in [np.array], the matrix to pivot and triangularize
    v_in [np.array], the RHS vector
    pivot [bool, default-False]: user decides if we pivot or not.
    OUT:
    x, the vector solution of A_in x = v_in """
    # copy A and v to temporary variables using copy command
    A = np.copy(A_in)
    v = np.copy(v_in)
    N = len(v)

    for m in range(N):
        if pivot:  # This is where I modify GaussElim
            # compare the mth element to all other mth elements below
            ZeRow = m
```

```
            for mm in range(m+1, N):
                if abs(A[mm, m]) > abs(A[ZeRow, m]):
                    ZeRow = mm  # I could swap everytime I find a hit, but that
                    # would be a lot of operations. Instead, I just take note
                    # of which row emerges as the winner

            if ZeRow != m:  # now we know if and what to swap
                A[ZeRow, :], A[m, :] = np.copy(A[m, :]), np.copy(A[ZeRow, :])
                v[ZeRow], v[m] = np.copy(v[m]), np.copy(v[ZeRow])

        # Divide by the diagonal element
        div = A[m, m]
        A[m, :] /= div
        v[m] /= div

        # Now subtract from the lower rows
        for i in range(m+1, N):
            mult = A[i, m]
            A[i, :] -= mult*A[m, :]
            v[i] -= mult*v[m]

    # Backsubstitution
    # create an array of the same type as the input array
    x = np.empty(N, dtype=v.dtype)
    for m in range(N-1, -1, -1):
        x[m] = v[m]
        for i in range(m+1, N):
            x[m] -= A[m, i]*x[i]
    return x

def PartialPivot(A_in, v_in):
    """ see textbook p. 222) """
    return GaussElim(A_in, v_in, True)
```

[9]:
```
omega = 1e3 # in rad/s
R_vals = ['', 1000., 2000., 1000., 2000., 1000., 2000.] # in Ohms
C_vals = ['', 1e-6 * omega * 1j, 5e-7 * omega * 1j] # in Farads
x_plus = 3 # in Volts
```

[13]:
```
X = PartialPivot(CircuitMatrix(R_vals, C_vals), RHS(R_vals, x_plus))
print ("Amplitudes and phases of voltages:")
for i in range (3):
    print(" |V{0}| = {1: 2e} V, phi{0} (t=0) = {2} degrees". format(i+1,␣
 ↪abs(X[i]), int(np. angle(X[i], deg=True))))
```

```
Amplitudes and phases of voltages:
 |V1| =  1.701439e+00 V, phi1 (t=0) = -5 degrees
```

```
|V2| =  1.480605e+00 V, phi2 (t=0) = 11 degrees
|V3| =  1.860769e+00 V, phi3 (t=0) = -4 degrees
```