<u>Project Report</u>

MATH 205 Numerical Analysis

Kapetanović Kerim 220302143

Kruho Vedad 220302139

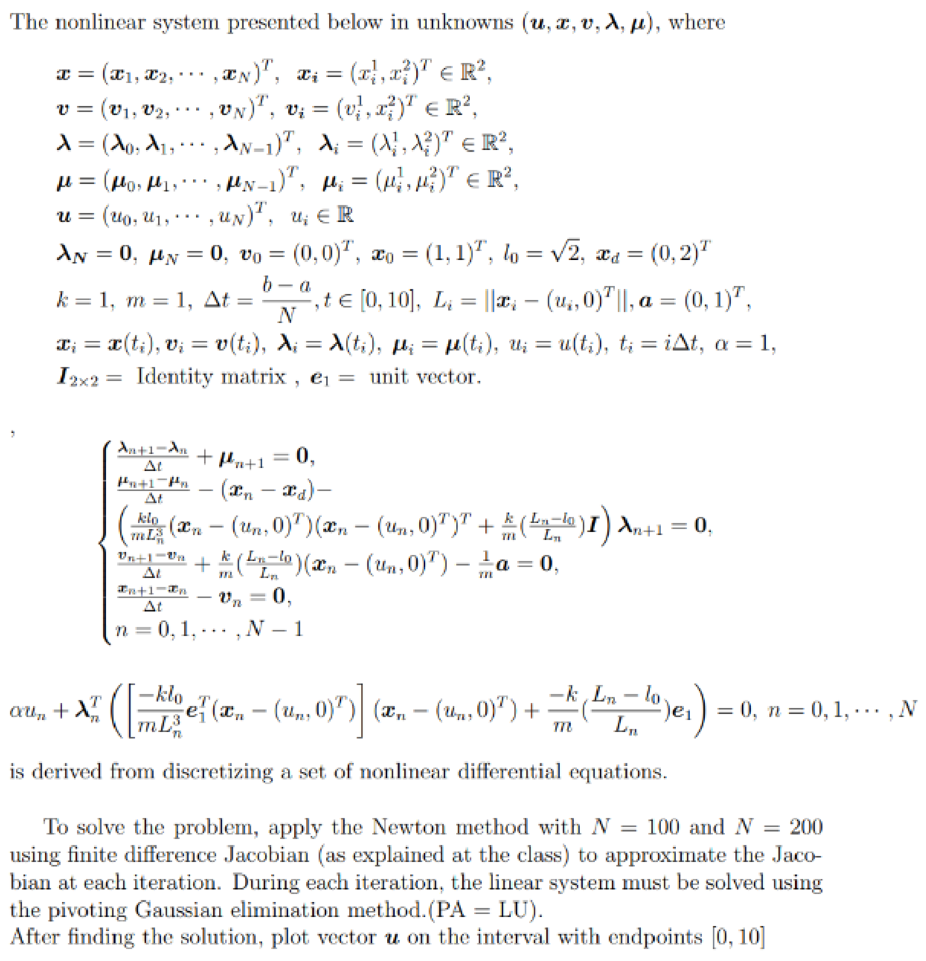
Ibrulj Tarik 220302142

***Professor: dr.* SeyedNima Rabiei**

Sarajevo, Spring 2024.

Table of Contents

Table of Figures



The nonlinear system presented below in unknowns $(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{v}, \boldsymbol{\lambda}, \boldsymbol{\mu})$, where

$$
\begin{aligned}
&\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_N)^T, \quad \boldsymbol{x_i} = (x_i^1, x_i^2)^T \in \mathbb{R}^2,\\
&\boldsymbol{v} = (\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_N)^T, \ \boldsymbol{v_i} = (v_i^1, x_i^2)^T \in \mathbb{R}^2,\\
&\boldsymbol{\lambda} = (\boldsymbol{\lambda}_0, \boldsymbol{\lambda}_1, \cdots, \boldsymbol{\lambda}_{N-1})^T, \quad \boldsymbol{\lambda}_i = (\lambda_i^1, \lambda_i^2)^T \in \mathbb{R}^2,\\
&\boldsymbol{\mu} = (\boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_{N-1})^T, \quad \boldsymbol{\mu}_i = (\mu_i^1, \mu_i^2)^T \in \mathbb{R}^2,\\
&\boldsymbol{u} = (u_0, u_1, \cdots, u_N)^T, \quad u_i \in \mathbb{R}\\
&\boldsymbol{\lambda_N} = \boldsymbol{0}, \ \boldsymbol{\mu}_N = \boldsymbol{0}, \ \boldsymbol{v}_0 = (0,0)^T, \ \boldsymbol{x}_0 = (1,1)^T, \ l_0 = \sqrt{2}, \ \boldsymbol{x}_d = (0,2)^T\\
&k = 1, \ m = 1, \ \Delta t = \frac{b-a}{N}, t \in [0,10], \ L_i = \|\boldsymbol{x}_i - (u_i, 0)^T\|, \boldsymbol{a} = (0,1)^T,\\
&\boldsymbol{x}_i = \boldsymbol{x}(t_i), \boldsymbol{v}_i = \boldsymbol{v}(t_i), \ \boldsymbol{\lambda}_i = \boldsymbol{\lambda}(t_i), \ \boldsymbol{\mu}_i = \boldsymbol{\mu}(t_i), \ u_i = u(t_i), \ t_i = i\Delta t, \ \alpha = 1,\\
&\boldsymbol{I}_{2\times 2} = \text{ Identity matrix }, \ \boldsymbol{e}_1 = \text{ unit vector.}
\end{aligned}
$$

,

$$
\begin{cases}
\frac{\boldsymbol{\lambda}_{n+1} - \boldsymbol{\lambda}_n}{\Delta t} + \boldsymbol{\mu}_{n+1} = \boldsymbol{0},\\
\frac{\boldsymbol{\mu}_{n+1} - \boldsymbol{\mu}_n}{\Delta t} - (\boldsymbol{x}_n - \boldsymbol{x}_d) -\\
\left(\frac{kl_0}{mL_n^3}(\boldsymbol{x}_n - (u_n,0)^T)(\boldsymbol{x}_n - (u_n,0)^T)^T + \frac{k}{m}(\frac{L_n - l_0}{L_n})\boldsymbol{I}\right)\boldsymbol{\lambda}_{n+1} = \boldsymbol{0},\\
\frac{\boldsymbol{v}_{n+1} - \boldsymbol{v}_n}{\Delta t} + \frac{k}{m}(\frac{L_n - l_0}{L_n})(\boldsymbol{x}_n - (u_n,0)^T) - \frac{1}{m}\boldsymbol{a} = \boldsymbol{0},\\
\frac{\boldsymbol{x}_{n+1} - \boldsymbol{x}_n}{\Delta t} - \boldsymbol{v}_n = \boldsymbol{0},\\
n = 0, 1, \cdots, N-1
\end{cases}
$$

$$
\alpha u_n + \boldsymbol{\lambda}_n^T \left(\left[\frac{-kl_0}{mL_n^3} \boldsymbol{e}_1^T (\boldsymbol{x}_n - (u_n, 0)^T) \right] (\boldsymbol{x}_n - (u_n,0)^T) + \frac{-k}{m}(\frac{L_n - l_0}{L_n})\boldsymbol{e}_1 \right) = 0, \ n = 0, 1, \cdots, N
$$

is derived from discretizing a set of nonlinear differential equations.

To solve the problem, apply the Newton method with $N = 100$ and $N = 200$ using finite difference Jacobian (as explained at the class) to approximate the Jacobian at each iteration. During each iteration, the linear system must be solved using the pivoting Gaussian elimination method.(PA = LU).
After finding the solution, plot vector $\boldsymbol{u}$ on the interval with endpoints $[0, 10]$

Figure 1. Project Problem

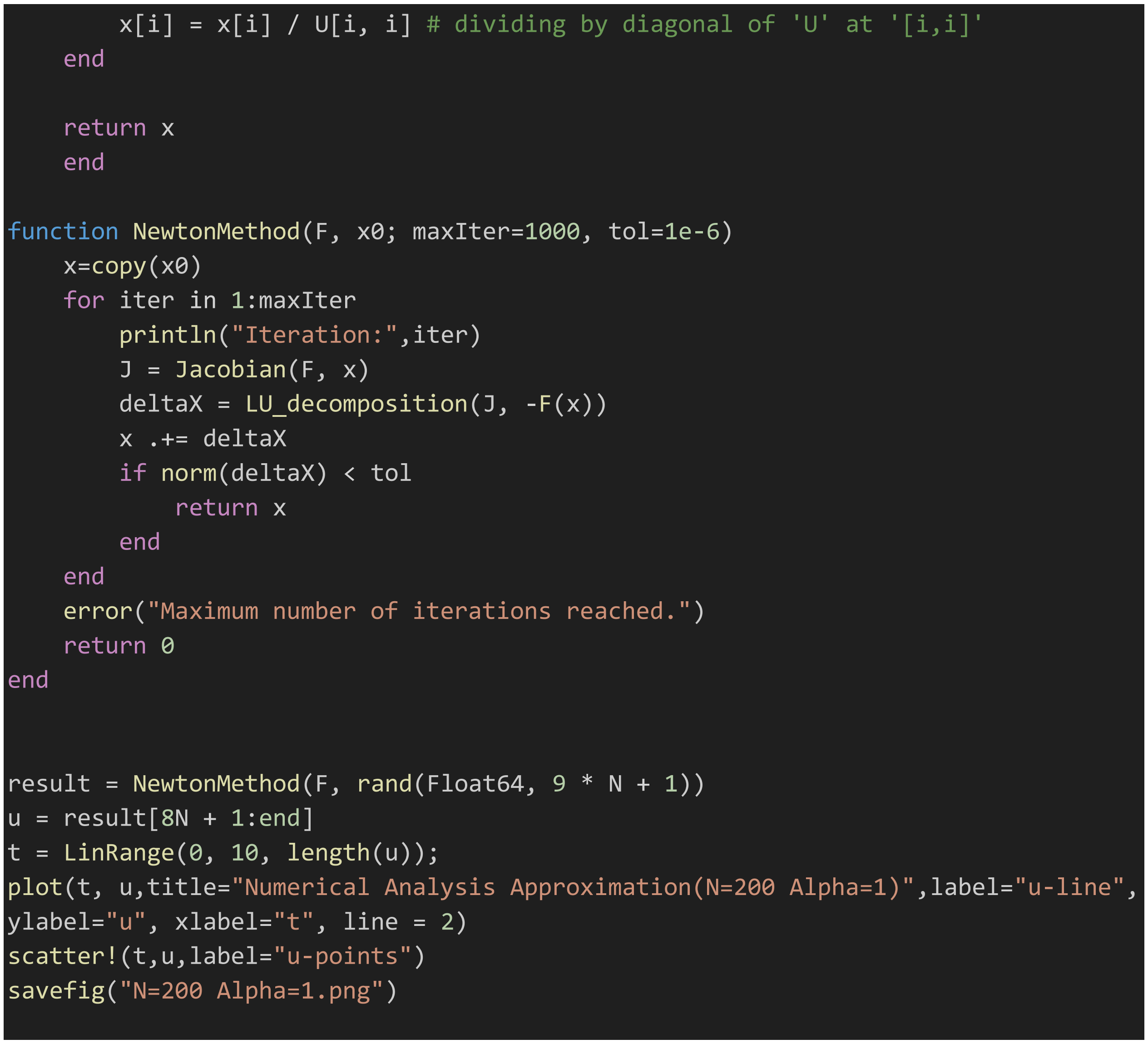
Full Code

```
using LinearAlgebra
using Plots

# variables given
N = 200
alpha = 1
k = m = 1
l0 = sqrt(2)
xd = Float64[0, 2] # floating point numbers, for stability and precise
calculation
```

```
a = Float64[0, 1]
x0 = Float64[1, 1]
v0 = Float64[0, 0]
lamda_n = Float64[0, 0]
mju_n = Float64[0, 0]
delta_t = 10 / N # difference between (b-a) / N


function Li(xi,ui) # making 'Li' as a function for easier handling, given in a
task
    norm(xi - [ui,0])
end

function F(X)
    N = (length(X) - 1) ÷ 9 # 9*N + 1 = X (number of equations), just writing in
form of N
    result = Float64[]

    #= 'reshape' using for organization of input vector 'X' into array;
    variables of system at each step (if we look 'deeper' in task, can see some
position, velocity, damping parameters...)=#
    x = reshape(X[1:2N], 2, N) # '2, N' - 2-dimesional array with 'N' columns
    v = reshape(X[2N + 1:4N], 2, N)
    lamda = reshape(X[4N + 1:6N], 2, N)
    mju = reshape(X[6N + 1:8N], 2, N)
    u = X[8N + 1:9N + 1]

    for i in 1:N-2 # 'N-2' - allows handling boundary conditions seperately

        # first equation
        # let's explain it for the first system, for others, similar logic is
used
        # r = (subtraction of all rows of columns 'i+2' and 'i+1') / delta_t +
all rows in coulmns 'i+2' of mju
        r = (lamda[:, i+2] - lamda[:, i+1]) / delta_t + mju[:, i+2]
        push!(result, r[1]) # adds first element of 'r' into result
        push!(result, r[2])

        # second equation
        d = (x[:, i] - [u[i+1], 0]) # making in portion, reducing complexity and
increasing readability
        Ln = Li(x[:, i], u[i+1])
        r = ((mju[:, i+2] - mju[:, i+1]) / delta_t) - (x[:, i] - xd) - (k * l0 *
d * d' / (m * Ln^3)+k/m * (Ln - l0)/Ln * [1 0; 0 1])*lamda[:, i+2]
```

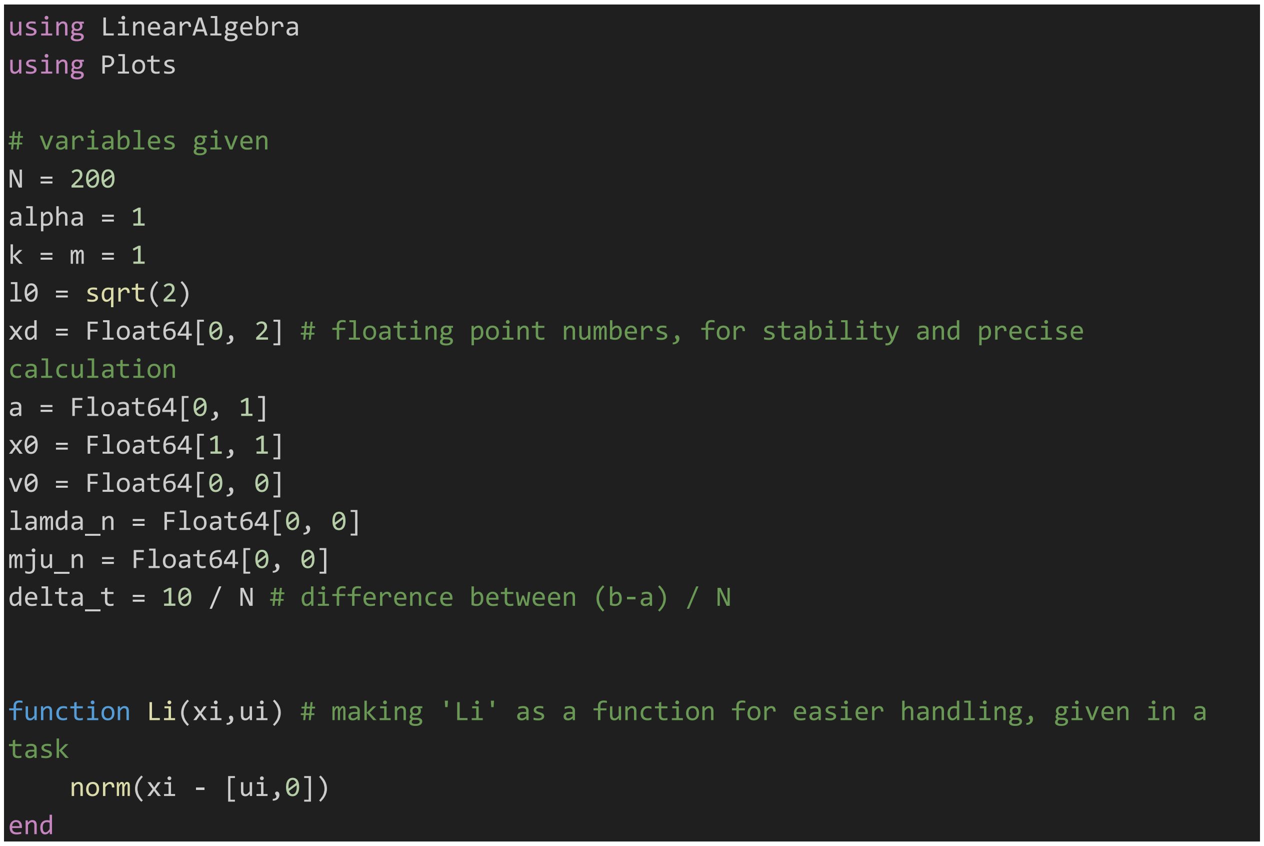
```
        push!(result, r[1])
        push!(result, r[2])

        # third equation
        d = (x[:, i] - [u[i+1], 0])
        Ln = Li(x[:, i], u[i+1])
        r = ((v[:, i+1] - v[:, i]) / delta_t)+( k * (Ln - l0) * d / (m * Ln))-
(a/m)

        push!(result, r[1])
        push!(result, r[2])

        # fourth equation
        r = ((x[:, i+1] - x[:, i]) / delta_t)-v[:, i]

        push!(result, r[1])
        push!(result, r[2])

        # fifth equation
        d = (x[:, i] - [u[i+1], 0])
        Ln = Li(x[:, i], u[i+1])
        r = (alpha*u[i+1])+ (lamda[:, i+1]')*((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d+(-k / m * (Ln - l0) / Ln * [1, 0]))

        push!(result, r)
    end

    # first equation
    rspecial = (lamda[:, 2] - lamda[:, 1]) / delta_t + mju[:, 2] # second and
first time steps
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    rspecial = (lamda_n - lamda[:, N]) / delta_t + mju_n
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # second equation
    d = (x0 - [u[1], 0])
    Ln = Li(x0, u[1])
    rspecial = ((mju[:, 2] - mju[:, 1]) / delta_t)-(x0 - xd)-((k * l0 * d * d' /
(m * Ln^3))+(k/m * (Ln - l0)/Ln * [1 0; 0 1]))* lamda[:, 2]

    push!(result, rspecial[1])
    push!(result, rspecial[2])
    d = (x[:, N-1] - [u[N], 0])
    Ln = Li(x[:, N-1], u[N])
    rspecial = ((mju_n - mju[:, N]) / delta_t)-(x[:, N-1] - xd)-((k * l0 * d * d'
/ (m * Ln^3))+(k/m * (Ln - l0)/Ln * [1 0; 0 1]))* lamda_n

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # third equation
    d = (x0 - [u[1], 0])
    Ln = Li(x0, u[1])
    rspecial = ((v[:, 1] - v0) / delta_t)+(k * (Ln - l0) * d / (m * Ln))-(a / m)

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    d = (x[:, N-1] - [u[N], 0])
    Ln = Li(x[:, N-1], u[N])
    rspecial = ((v[:, N] - v[:, N-1]) / delta_t)+(k * (Ln - l0) * d / (m * Ln))-
(a / m)

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # fourth equation
    rspecial = ((x[:, 1] - x0) / delta_t)-v0
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    rspecial = ((x[:, N] - x[:, N-1]) / delta_t)- v[:, N-1]
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # fifth Equation
    d = (x0 - [u[1], 0])
    Ln = Li(x0, u[1])
    rspecial = (alpha*u[1])+(lamda[:, 1]')*(((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d)+(-k / m * (Ln - l0) / Ln * [1, 0]))

    push!(result, rspecial)

    d = (x[:, N-1] - [u[N], 0])
    Ln = Li(x[:, N-1], u[N])
```

```julia
    rspecial = (alpha*u[N])+(lamda[:, N]')*(((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d)+(-k / m * (Ln - l0) / Ln * [1, 0]))

    push!(result, rspecial)

    d = (x[:, N] - [u[N + 1], 0])
    Ln = Li(x[:, N], u[N + 1])
    rspecial = (alpha*u[N + 1])+(lamda_n')*(((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d)+(-k / m * (Ln - l0) / Ln * [1, 0]))

    push!(result, rspecial)

    return result
end

function Jacobian(F, X)
    N = length(X) # calculating vector's size
    J = zeros(N, N) # currently initialized as 'zeros', Jacobian will be stored
    tol = 1e-6 #tolerance
    for j in 1:N # iterating
        xi = copy(X) # making copy so function is not directly effected at each
iteration
        xi[j] += tol # approximate derivatives at the perturbed points
        J[:,j] = (F(xi) - F(X)) / tol # computing finite difference approximation
    end

    return J
end

function LU_decomposition(A, b)
    n = size(A, 1) # calculation of square matrix
    L = zeros(n, n) # initialization of empty matrix, will store LT part of LU
decomposition
    U = copy(A) # UT part of LU decomposition
    P = Matrix(I, n, n) # permutation matrix to keep track of row interchanges
and for better stability

    for k in 1:n-1
        val, index = findmax(abs.(U[k:n, k])) # finds the index of MAX absolute
value in given submatrix; used for partial pivoting
        index += k - 1 # we need to make it relative to original matrix, not
submatrix

        if val == 0 # if MAX absolute value is zero
            error("Division by 0!")
        end

        # partial pivoting, element with the largest absolute value is placed at
diagonal for stability; swaping rows 'k' and 'index'
        U[[k, index], :] = U[[index, k], :]
        L[[k, index], :] = L[[index, k], :]
        P[[k, index], :] = P[[index, k], :]

        for i in k+1:n
            L[i, k] = U[i, k] / U[k, k] # calculating to eliminate entries below
diagonal
            for j in k+1:n
                U[i, j] -= L[i, k] * U[k, j] # eliminates the entries below
diagonal in column 'k' of 'U'
            end
        end
    end

    for i in 1:n
        L[i, i] = 1.0 # set diagonal entries to 1
    end

    # apply permutation to b
    b = P * b
    n = length(b) # length of vector b
    y = zeros(n)
    x = zeros(n)

    # forward substitution: Ly = b
    for i in 1:n # itterating over each eq in system
        y[i] = b[i] # from each eq y, to correspond b
        for j in 1:i-1
            y[i] -= L[i, j] * y[j] # updating 'y[i]', subtracting product of
correspond element from 'L' and previously calculated value 'y[j]'
        end
        y[i] = y[i] / L[i, i]
    end

    # backward substitution: Ux = y
    for i in n:-1:1 # in reverse order from last eq
        x[i] = y[i]
        for j in i+1:n # iterates over elements of 'x'
            x[i] -= U[i, j] * x[j] # updaitng 'x[i]', similarly like in forward
substitution
        end
```



```julia
        x[i] = x[i] / U[i, i] # dividing by diagonal of 'U' at '[i,i]'
    end

    return x
    end

function NewtonMethod(F, x0; maxIter=1000, tol=1e-6)
    x=copy(x0)
    for iter in 1:maxIter
        println("Iteration:",iter)
        J = Jacobian(F, x)
        deltaX = LU_decomposition(J, -F(x))
        x .+= deltaX
        if norm(deltaX) < tol
            return x
        end
    end
    error("Maximum number of iterations reached.")
    return 0
end


result = NewtonMethod(F, rand(Float64, 9 * N + 1))
u = result[8N + 1:end]
t = LinRange(0, 10, length(u));
plot(t, u,title="Numerical Analysis Approximation(N=200 Alpha=1)",label="u-line",
ylabel="u", xlabel="t", line = 2)
scatter!(t,u,label="u-points")
savefig("N=200 Alpha=1.png")
```

Figure 2. Full Code



```
using LinearAlgebra
using Plots

# variables given
N = 200
alpha = 1
k = m = 1
l0 = sqrt(2)
xd = Float64[0, 2] # floating point numbers, for stability and precise
calculation
a = Float64[0, 1]
x0 = Float64[1, 1]
v0 = Float64[0, 0]
lamda_n = Float64[0, 0]
mju_n = Float64[0, 0]
delta_t = 10 / N # difference between (b-a) / N


function Li(xi,ui) # making 'Li' as a function for easier handling, given in a
task
    norm(xi - [ui,0])
end
```

Figure 3. Initial Values

```
function F(X)
    N = (length(X) - 1) ÷ 9 # 9*N + 1 = X (number of equations), just writing in
form of N
    result = Float64[]

    #= 'reshape' using for organization of input vector 'X' into array;
    variables of system at each step (if we look 'deeper' in task, can see some
position, velocity, damping parameters...)=#
    x = reshape(X[1:2N], 2, N) # '2, N' - 2-dimesional array with 'N' columns
    v = reshape(X[2N + 1:4N], 2, N)
    lamda = reshape(X[4N + 1:6N], 2, N)
    mju = reshape(X[6N + 1:8N], 2, N)
    u = X[8N + 1:9N + 1]

    for i in 1:N-2 # 'N-2' - allows handling boundary conditions seperately

        # first equation
        # let's explain it for the first system, for others, similar logic is
used
        # r = (subtraction of all rows of columns 'i+2' and 'i+1') / delta_t +
all rows in coulmns 'i+2' of mju
        r = (lamda[:, i+2] - lamda[:, i+1]) / delta_t + mju[:, i+2]
        push!(result, r[1]) # adds first element of 'r' into result
        push!(result, r[2])

        # second equation
        d = (x[:, i] - [u[i+1], 0]) # making in portion, reducing complexity and
increasing readability
        Ln = Li(x[:, i], u[i+1])
        r = ((mju[:, i+2] - mju[:, i+1]) / delta_t) - (x[:, i] - xd) - (k * l0 *
d * d' / (m * Ln^3)+k/m * (Ln - l0)/Ln * [1 0; 0 1])*lamda[:, i+2]

        push!(result, r[1])
        push!(result, r[2])

        # third equation
        d = (x[:, i] - [u[i+1], 0])
        Ln = Li(x[:, i], u[i+1])
        r = ((v[:, i+1] - v[:, i]) / delta_t)+( k * (Ln - l0) * d / (m * Ln))-
(a/m)

        push!(result, r[1])
        push!(result, r[2])
        # fourth equation
        r = ((x[:, i+1] - x[:, i]) / delta_t)-v[:, i]

        push!(result, r[1])
        push!(result, r[2])

        # fifth equation
        d = (x[:, i] - [u[i+1], 0])
        Ln = Li(x[:, i], u[i+1])
        r = (alpha*u[i+1])+ (lamda[:, i+1]')*((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d+(-k / m * (Ln - l0) / Ln * [1, 0]))

        push!(result, r)
    end

    # first equation
    rspecial = (lamda[:, 2] - lamda[:, 1]) / delta_t + mju[:, 2] # second and
first time steps
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    rspecial = (lamda_n - lamda[:, N]) / delta_t + mju_n
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # second equation
    d = (x0 - [u[1], 0])
    Ln = Li(x0, u[1])
    rspecial = ((mju[:, 2] - mju[:, 1]) / delta_t)-(x0 - xd)-((k * l0 * d * d' /
(m * Ln^3))+(k/m * (Ln - l0)/Ln * [1 0; 0 1]))* lamda[:, 2]

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    d = (x[:, N-1] - [u[N], 0])
    Ln = Li(x[:, N-1], u[N])
    rspecial = ((mju_n - mju[:, N]) / delta_t)-(x[:, N-1] - xd)-((k * l0 * d * d'
/ (m * Ln^3))+(k/m * (Ln - l0)/Ln * [1 0; 0 1]))* lamda_n

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # third equation
    d = (x0 - [u[1], 0])
    Ln = Li(x0, u[1])
    rspecial = ((v[:, 1] - v0) / delta_t)+(k * (Ln - l0) * d / (m * Ln))-(a / m)

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    d = (x[:, N-1] - [u[N], 0])
    Ln = Li(x[:, N-1], u[N])
    rspecial = ((v[:, N] - v[:, N-1]) / delta_t)+(k * (Ln - l0) * d / (m * Ln))-
(a / m)

    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # fourth equation
    rspecial = ((x[:, 1] - x0) / delta_t)-v0
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    rspecial = ((x[:, N] - x[:, N-1]) / delta_t)- v[:, N-1]
    push!(result, rspecial[1])
    push!(result, rspecial[2])

    # fifth Equation
    d = (x0 - [u[1], 0])
    Ln = Li(x0, u[1])
    rspecial = (alpha*u[1])+(lamda[:, 1]')*(((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d)+(-k / m * (Ln - l0) / Ln * [1, 0]))

    push!(result, rspecial)

    d = (x[:, N-1] - [u[N], 0])
    Ln = Li(x[:, N-1], u[N])
    rspecial = (alpha*u[N])+(lamda[:, N]')*(((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d)+(-k / m * (Ln - l0) / Ln * [1, 0]))

    push!(result, rspecial)

    d = (x[:, N] - [u[N + 1], 0])
    Ln = Li(x[:, N], u[N + 1])
    rspecial = (alpha*u[N + 1])+(lamda_n')*(((-k * l0 * [1 0] * d / (m *
Ln^3))[1] * d)+(-k / m * (Ln - l0) / Ln * [1, 0]))

    push!(result, rspecial)
    return result
end
```

Figure 4. Function Definition

Function F(X) : Defines the system of nonlinear equations. It creates a vector of 9N+1 equations.

```
function Jacobian(F, X)
    N = length(X) # calculating vector's size
    J = zeros(N, N) # currently initialized as 'zeros', Jacobian will be stored
    tol = 1e-6 #tolerance
    for j in 1:N # iterating
        xi = copy(X) # making copy so function is not directly effected at each
iteration
        xi[j] += tol # approximate derivatives at the perturbed points
        J[:,j] = (F(xi) - F(X)) / tol # computing finite difference approximation
    end

    return J
end
```

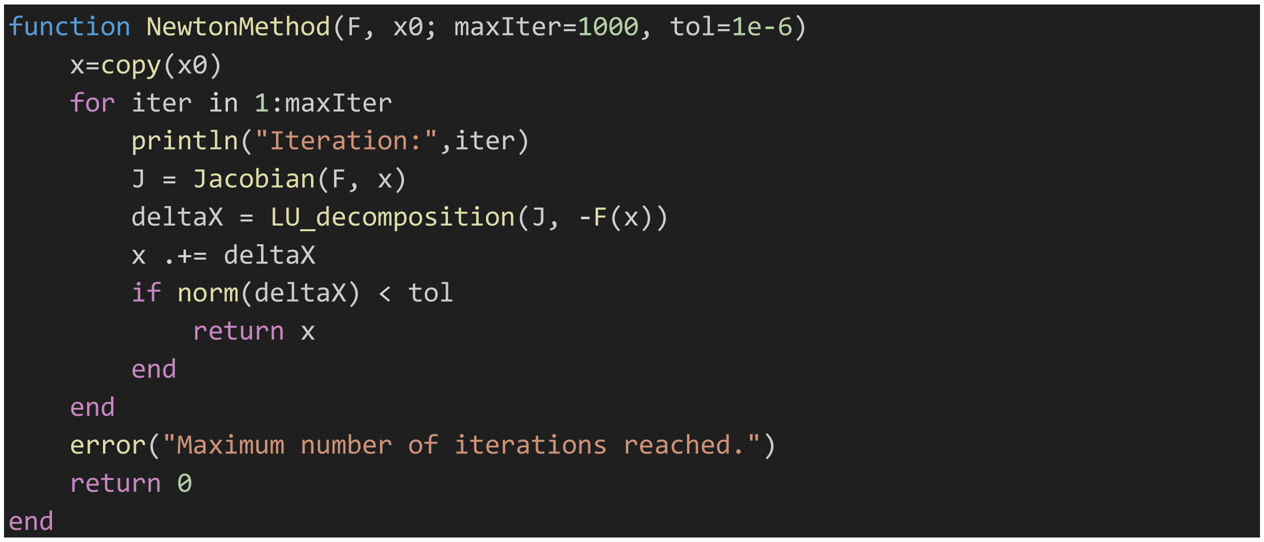
Figure 5. Jacobian Function

Function Jacobian(F, X): Determines the system of equations Jacobian matrix at a given point. It computes the appropriate modification in F and estimates elements of Jacobian matrix using finite differences.

```
function LU_decomposition(A, b)
    n = size(A, 1) # calculation of square matrix
    L = zeros(n, n) # initialization of empty matrix, will store LT part of LU
decomposition
    U = copy(A) # UT part of LU decomposition
    P = Matrix(I, n, n) # permutation matrix to keep track of row interchanges
and for better stability

    for k in 1:n-1
        val, index = findmax(abs.(U[k:n, k])) # finds the index of MAX absolute
value in given submatrix; used for partial pivoting
        index += k - 1 # we need to make it relative to original matrix, not
submatrix

        if val == 0 # if MAX absolute value is zero
            error("Division by 0!")
        end

        # partial pivoting, element with the largest absolute value is placed at
diagonal for stability; swaping rows 'k' and 'index'
        U[[k, index], :] = U[[index, k], :]
        L[[k, index], :] = L[[index, k], :]
        P[[k, index], :] = P[[index, k], :]

        for i in k+1:n
            L[i, k] = U[i, k] / U[k, k] # calculating to eliminate entries below
diagonal
            for j in k+1:n
                U[i, j] -= L[i, k] * U[k, j] # eliminates the entries below
diagonal in column 'k' of 'U'
            end
        end
    end

    for i in 1:n
        L[i, i] = 1.0 # set diagonal entries to 1
    end

    # apply permutation to b
    b = P * b
    n = length(b) # length of vector b
    y = zeros(n)
    x = zeros(n)

    # forward substitution: Ly = b
    for i in 1:n # itterating over each eq in system
        y[i] = b[i] # from each eq y, to correspond b
        for j in 1:i-1
            y[i] -= L[i, j] * y[j] # updating 'y[i]', subtracting product of
correspond element from 'L' and previously calculated value 'y[j]'
        end
        y[i] = y[i] / L[i, i]
    end

    # backward substitution: Ux = y
    for i in n:-1:1 # in reverse order from last eq
        x[i] = y[i]
        for j in i+1:n # iterates over elements of 'x'
            x[i] -= U[i, j] * x[j] # updaitng 'x[i]', similarly like in forward
substitution
        end
        x[i] = x[i] / U[i, i] # dividing by diagonal of 'U' at '[i,i]'
    end

    return x
    end
```

Figure 6. LU Decomposition Function

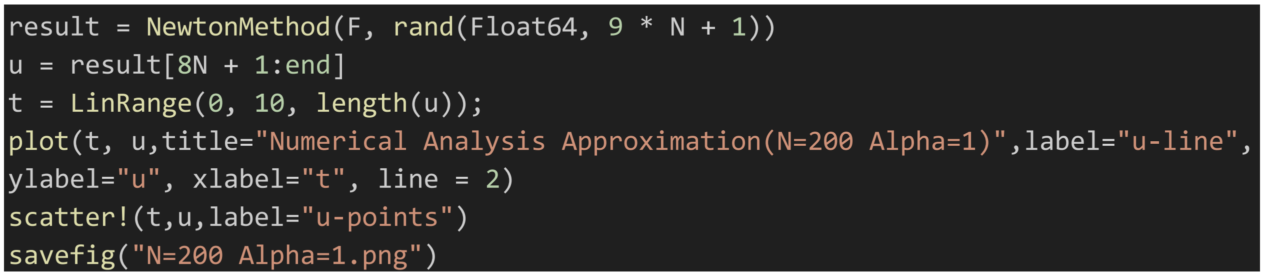
Function LU_decomposition(A, b): Performes the LU decomposition. It decomposes the given matrix A into an upper triangular matrix U and lower triangular matrix L, ***A = LU.*** And after that solves the LU decomposition using forward and backward substitution.



```julia
function NewtonMethod(F, x0; maxIter=1000, tol=1e-6)
    x=copy(x0)
    for iter in 1:maxIter
        println("Iteration:",iter)
        J = Jacobian(F, x)
        deltaX = LU_decomposition(J, -F(x))
        x .+= deltaX
        if norm(deltaX) < tol
            return x
        end
    end
    error("Maximum number of iterations reached.")
    return 0
end
```

Figure 7. Newton Method

Funtction NewtonMethod(F, x0): The Newton-Raphson technique for solving nonlinear equations. The solution is updated using LU decomposition at each iteration.



```julia
result = NewtonMethod(F, rand(Float64, 9 * N + 1))
u = result[8N + 1:end]
t = LinRange(0, 10, length(u));
plot(t, u,title="Numerical Analysis Approximation(N=200 Alpha=1)",label="u-line",
ylabel="u", xlabel="t", line = 2)
scatter!(t,u,label="u-points")
savefig("N=200 Alpha=1.png")
```

Figure 8. Plotting

Plotting: The result is plotted and scattered on the graph.

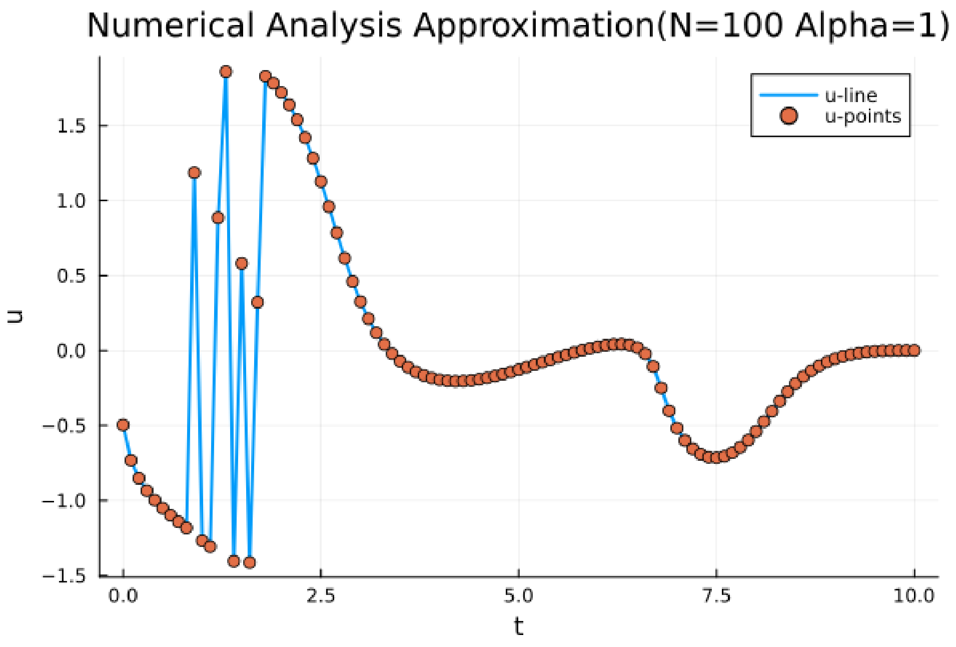


Figure 9. Graph of the Nonlinear EQ (N=100 Alpha=1)

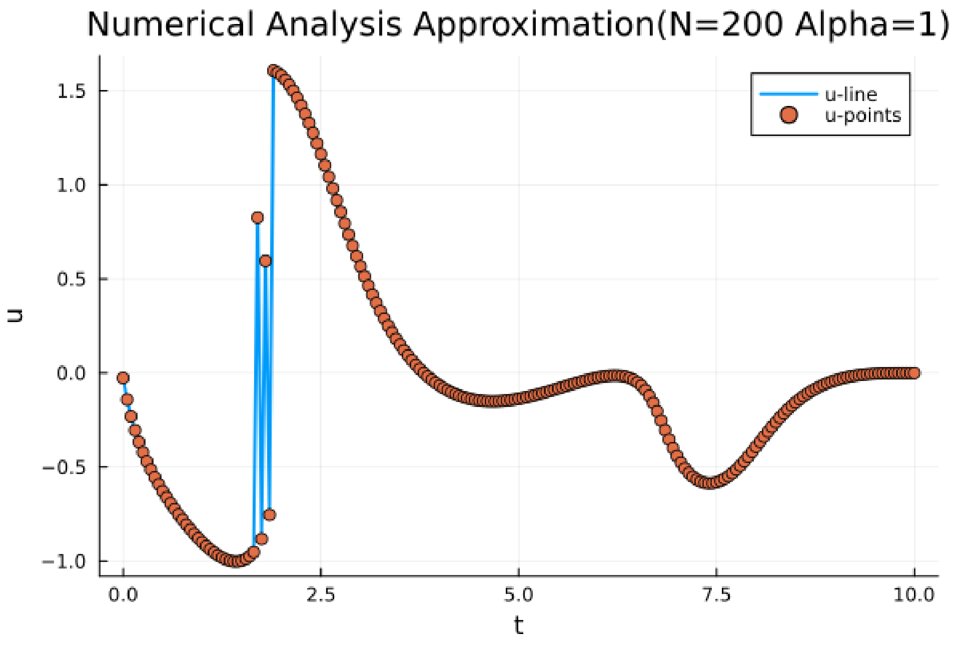


Figure 10. Graph of the Nonlinear EQ (N=200 Alpha=1)

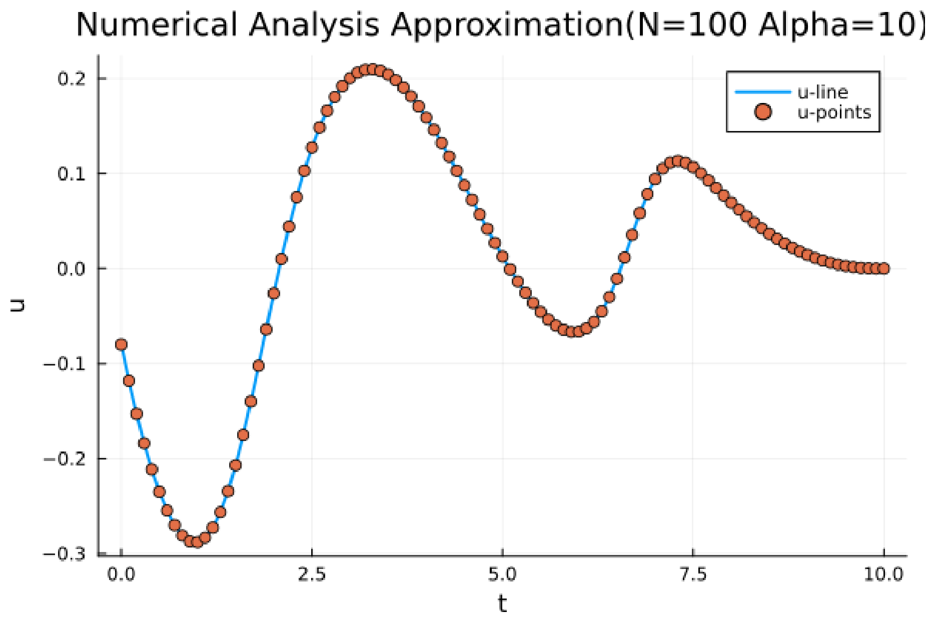


Figure 11. Graph of the Nonlinear EQ (N=100 Alpha=10)

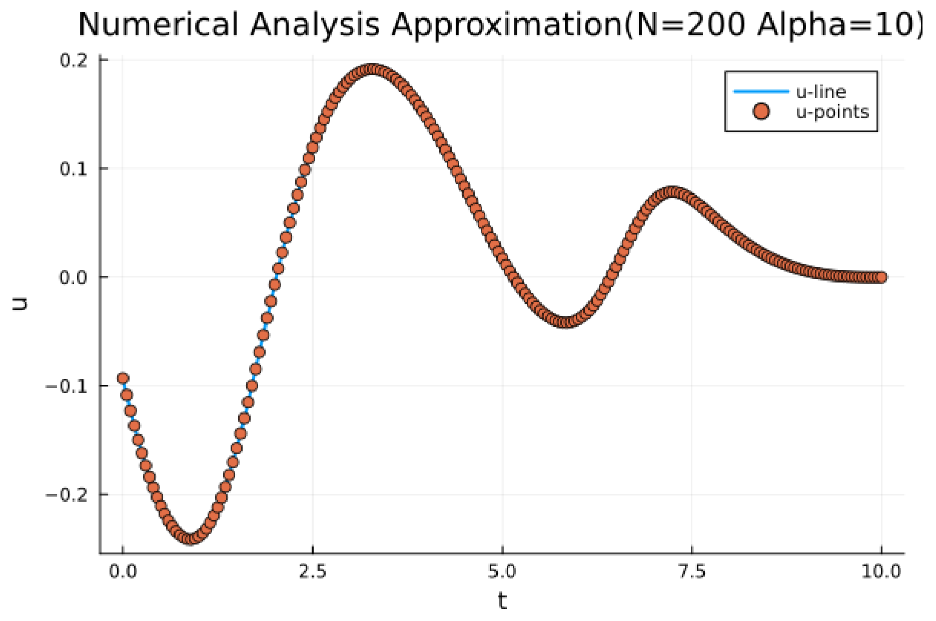


Figure 12. Graph of the Nonlinear EQ (N=200 Alpha=10)

Results: Figures 9, 10, 11, 12 show the graph of the given nonlinear equation on the given interval with endpoints t = [0, 10]