HOMEWORK 3 SUBMISSION

Jinen Setpal
October 5, 2025

Checklist

1. Cross-checked independent work with Kunal Kapur.
2. No use of AI tools.
3. Code is included!

Problem 1

1. Let $\hat{\mathbf{x}}_k = \texttt{approx\_evolve\_steps}(\cdot)$ after running k steps. If we assume this vector as an eigenvector of $\rho\boldsymbol{A}$, we have:

$$\begin{aligned}
\rho\boldsymbol{A}\hat{\mathbf{x}}_k &= \lambda\hat{\mathbf{x}}_k \\
\hat{\mathbf{x}}_k^T\rho\boldsymbol{A}\hat{\mathbf{x}}_k &= \hat{\mathbf{x}}_k^T\lambda\hat{\mathbf{x}}_k \\
\frac{\hat{\mathbf{x}}_k^T\rho\boldsymbol{A}\hat{\mathbf{x}}_k}{\hat{\mathbf{x}}_k^T\hat{\mathbf{x}}_k} &= \lambda\frac{\hat{\mathbf{x}}_k^T\hat{\mathbf{x}}_k}{\hat{\mathbf{x}}_k^T\hat{\mathbf{x}}_k} \\
\lambda &= \frac{\hat{\mathbf{x}}_k^T\rho\boldsymbol{A}\hat{\mathbf{x}}_k}{\hat{\mathbf{x}}_k^T\hat{\mathbf{x}}_k}
\end{aligned}$$

   As hinted, this is the Rayleigh quotient.

   We can compute with the following Julia code (HW1 part omitted for brevity):

Code

```
using Random
Random.seed!(10) # ensure repeatable results...

p = 0.2
for nodes in [10, 1000]
        println("nodes: ", nodes)

        global A, xy = spatial_network(nodes, 2)
        if nodes == 10
                global x0 = zeros(size(A,1)); x0[1] = 1
        else
                global x0 = zeros(size(A,1)); x0[end] = 1
        end

        true_eig = maximum(eigvals(Matrix(p * A)))

        for steps in [10, 50, 100]
                global final_state = approx_evolve_steps(x0, p, A, steps)
                    [:, end]
                # global final_state = power_method(x0, p, A, steps)

                global eigenvalue = (final_state' * (p * A) * final_state
                    ) / norm(final_state)^2
                println("steps: ", steps, ", eigenvalue: ", eigenvalue, "
                    , approximation err:", true_eig - eigenvalue)
        end
        println("-----")
```

```
end
```

Output

```
nodes: 10
steps: 10, eigenvalue: 1.1671278170061723, approximation err
    :1.774407976107284e-8
steps: 50, eigenvalue: 1.167127834750253, approximation err
    :-8.881784197001252e-16
steps: 100, eigenvalue: 1.1671278347502532, approximation err
    :-1.1102230246251565e-15
-----
nodes: 1000
steps: 10, eigenvalue: 2.5670421693023777, approximation err
    :0.38478351765839625
steps: 50, eigenvalue: 2.941749847772847, approximation err
    :0.01007583918792676
steps: 100, eigenvalue: 2.951758629626842, approximation err
    :6.705733393186364e-5
-----
```

2. For this, we simply replace `final_state` in the previous function with `power_method`:

Code

```
function power_method(x0::Vector, p::Real, A::AbstractMatrix, k::Int)
        res = (p * A)^k * x0
        return res / norm(res)
end
```

Output

```
nodes: 10
steps: 10, eigenvalue: 1.1671278134538405, approximation err
    :2.129641152315287e-8
steps: 50, eigenvalue: 1.1671278347502532, approximation err
    :-1.1102230246251565e-15
steps: 100, eigenvalue: 1.167127834750253, approximation err
    :-8.881784197001252e-16
-----
nodes: 1000
steps: 10, eigenvalue: 2.5746707226625274, approximation err
    :0.3771549642982466
steps: 50, eigenvalue: 2.9423439918214873, approximation err
    :0.00948169513928665
steps: 100, eigenvalue: 2.951759961375165, approximation err
    :6.572558560913322e-5
-----
```

The approximation error is practically equivalent. This makes sense, given `approx_evolve_steps` is effectively just power iteration.

3. Computing powers of sparser matrices is much quicker (in part because we're using sparse matrix format, but also just fewer values to worry about), so we'll start from the sparsest graph and progressively add edges until we have a maximum eigenvalue that exceeeds one:

Code

```
A, xy = spatial_network(1000, 2)
step = 0.0001
prev_eigenvalue = 0
for f in 0.9999:-step:0
        global final_state = power_method(x0, p, social_distance(A, xy, f
            ), 100)
        global eigenvalue = (final_state' * (p * A) * final_state) / norm
            (final_state)^2
        if eigenvalue > 1
                println("minimum social distancing needed: ", f+step, "
                    with eigenvalue: ", prev_eigenvalue)
                break
        end
        global prev_eigenvalue = eigenvalue
end
```

Output

```
minimum social distancing needed: 0.8612 with eigenvalue:
    0.9501369235820016
```

So we need to remove 86.12% of connections for an epidemic to not occur.

Let's try achieving the same result instead by nerfing the virus. I use a 'big step' / 'small step' strategy to help guide the search, where I set a minimum threshold using a big step size. I then progressively reduce the search space as well as the step size by setting the minimum value to be the solution of the previous iteration. I manually tuned the threshold each time getting one extra significant figure.

Code

```
step = 0.0001
prev_eigenvalue = 0
for p in 0.071:step:0.2  # we know 0.2 has eigenvalue > 1
        global final_state = power_method(x0, p, A, 100)
        global eigenvalue = (final_state' * (p * A) * final_state) / norm
            (final_state)^2
        if eigenvalue > 1
                println("most powerful failing virus has p: ", p+step, "
                    with eigenvalue: ", prev_eigenvalue)
                break
        end
        global prev_eigenvalue = eigenvalue
end
```

Output

```
most powerful failing virus has p: 0.0721 with eigenvalue:
    0.9993946006708243
```

So, we have to significantly nerf the virus to not require social distancing, or commit to a lot of social distancing given a potent virus to prevent an epidemic.

4. The answer to the first question (do we need more or less social distancing) is easy, we obviously will need less social distancing, but we should show this mathematically. We can simply pop each row and column that corresponds to a node, and our new adjacency matrix will represent the graph with those nodes removed. We randomly select 72% of the nodes:

Code

```
using StatsBase
function vaccinate(A::SparseMatrixCSC, f::Real)
        A = 1 * A   # copy by value
        for rc in StatsBase.sample(1:size(A)[1]-1, Int(f * size(A)[1]),
           replace=false)   # can't vaccinate patient zero
                A[:, rc] .= 0
                A[rc, :] .= 0
        end

        return dropzeros(A)
end

eigenvals = []
for trial in 1:1000
        global A_vac = vaccinate(A, .72)
        global final_state = power_method(x0, p, A_vac, 10)
        if final_state != zeros(length(final_state))  # in this case, all
            the neighbors of patient zero are vaccinated
                global eigenvalue = (final_state' * (p * A_vac) *
                   final_state) / norm(final_state)^2
                push!(eigenvals, eigenvalue)
        end
end

println("Mean: ", mean(eigenvals), ", Std. Dev: ", std(eigenvals))
```

Output

```
Mean: 0.6512950995772196, Std. Dev: 0.326976444201098
```

Problem 2

1. **Code**

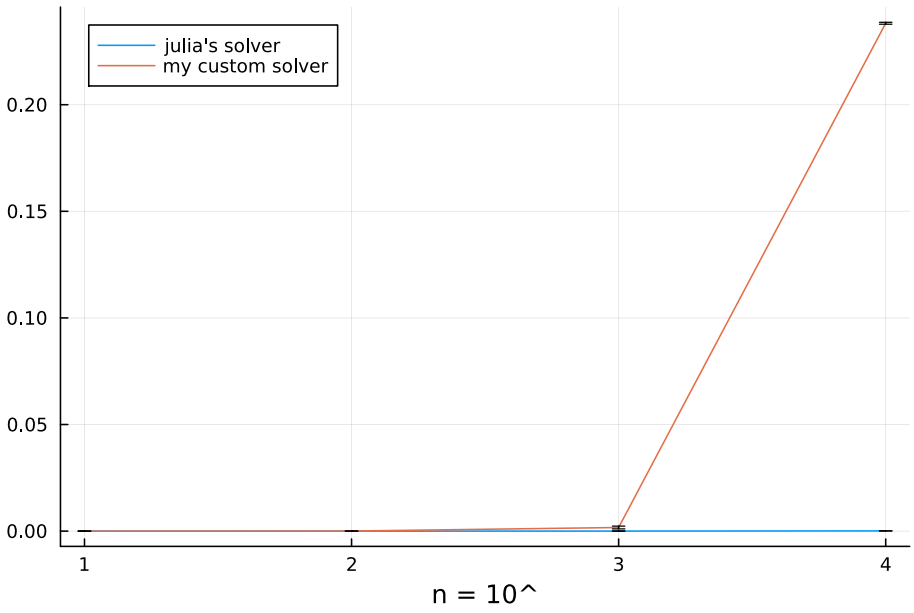
```
using LinearAlgebra, BenchmarkTools, Plots

function forwardsolve(L, y)  # Lower triangular L
        x = zeros(length(y))  # set to unset values to zero, allows us to
            directly call dot and compute a partial sum

        for i in 1:length(y)
                x[i] = (y[i] - dot(L[i, :], x))/L[i, i]  # iterate and
                    solve sequentially
        end
        return x
end

function backsolve(U, y)  # Upper Triangular U
        x = zeros(length(y))

        for i in length(y):-1:1  # reverse iteration order
                x[i] = (y[i] - dot(U[i, :], x))/U[i, i]
        end
        return x
end
```

2. **Code**

```
function lt_solve(A, b)
        return forwardsolve(A, b)
end


tol = 1e-5
n_trials = 25

lt_sol_mean = []
lt_sol_std = []
lt_opt_mean = []
lt_opt_std = []
for n_pow in 1:4
        local n = 10^n_pow
        local lt_sol_time = []
        local lt_opt_time = []
        for n_trial in 1:n_trials
                global A = sparse(2:n, 1:n-1, -1.0, n, n) + I
                global b = ones(n)

                @assert norm(A\b - lt_solve(A, b)) < tol
                push!(lt_opt_time, @belapsed A\b)
                push!(lt_sol_time, @belapsed lt_solve(A, b))
        end
        push!(lt_opt_mean, mean(lt_opt_time))
        push!(lt_opt_std, std(lt_opt_time))
        push!(lt_sol_mean, mean(lt_sol_time))
        push!(lt_sol_std, std(lt_sol_time))
end

plot(lt_opt_mean, err=lt_opt_std, label="julia's solver")
```

```
p = plot!(lt_sol_mean, err=lt_sol_std, label="my custom solver", xlabel="
    n = 10^")
```

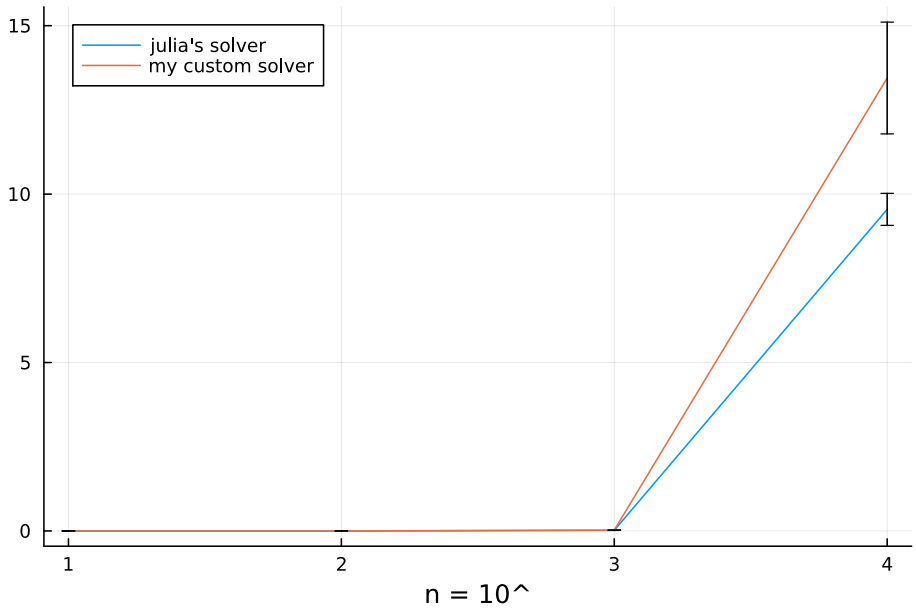
Output



3. **Code**

```
function solve(A, b)
        L, U, p = lu(A)
        return backsolve(U, forwardsolve(L, b[p]))
end

sol_mean = []
sol_std = []
opt_mean = []
opt_std = []
for n_pow in 1:4
        local n = 10^n_pow
        local sol_time = []
        local opt_time = []
        for n_trial in 1:n_trials
                global A = rand(n, n)
                global b = rand(n)

                @assert norm(A\b - solve(A, b)) < tol
                push!(opt_time, @belapsed A\b)
                push!(sol_time, @belapsed solve(A, b))
        end
        push!(opt_mean, mean(opt_time))
        push!(opt_std, std(opt_time))
        push!(sol_mean, mean(sol_time))
        push!(sol_std, std(sol_time))
end

plot(opt_mean, err=opt_std, label="julia's solver")
p = plot!(sol_mean, err=sol_std, label="my custom solver", xlabel="n =
    10^")
```

Output



Regarding accuracy, we check within the code that the solution obtained by our solver is within tolerance of the optimal solution. Although our naive solution leverages sparse computation, it doesn't beat Julia's optimizations.

Problem 3

Let $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. We have:

$$\boldsymbol{M}\mathbf{x} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} = \mathbf{b}$$

With $\mathbf{x}_1 \in \mathbb{R}^d$ known for some $d \leq n$.

1. Let's merge matrices:

   $$\text{Let } \boldsymbol{Y} := \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{C} \end{bmatrix} \quad \text{and} \quad \boldsymbol{Z} := \begin{bmatrix} \boldsymbol{B} \\ \boldsymbol{D} \end{bmatrix}$$

   We now have:

   $$\begin{aligned} \boldsymbol{M}\mathbf{x} = \begin{bmatrix} \boldsymbol{Y} & \boldsymbol{Z} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} &= \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} \\ \boldsymbol{Y}\mathbf{x}_1 + \boldsymbol{Z}\mathbf{x}_2 &= \mathbf{b} \\ \boldsymbol{Z}\mathbf{x}_2 &= \mathbf{b} - \boldsymbol{Y}\mathbf{x}_1 \end{aligned}$$

   Here, we apply the pseudoinverse on both sides:

   $$\begin{aligned} \boldsymbol{Z}\boldsymbol{Z}^\dagger \boldsymbol{Z}\mathbf{x}_2 &= \boldsymbol{Z}\boldsymbol{Z}^\dagger(\mathbf{b} - \boldsymbol{Y}\mathbf{x}_1) \\ \mathbf{x}_2 &= \boldsymbol{Z}\boldsymbol{Z}^\dagger(\mathbf{b} - \boldsymbol{Y}\mathbf{x}_1) \end{aligned}$$

   The key assumption here is the existence of the pseudoinverse $\boldsymbol{Z}^\dagger$.

2. We can perform a similar operation to obtain $\mathbf{x}_1$ given $\mathbf{x}_2$:

   $$\begin{aligned} \boldsymbol{M}\mathbf{x} = \begin{bmatrix} \boldsymbol{Y} & \boldsymbol{Z} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} &= \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} \\ \boldsymbol{Y}\mathbf{x}_1 + \boldsymbol{Z}\mathbf{x}_2 &= \mathbf{b} \\ \boldsymbol{Y}\mathbf{x}_1 &= \mathbf{b} - \boldsymbol{Z}\mathbf{x}_2 \\ \boldsymbol{Y}\boldsymbol{Y}^\dagger \boldsymbol{Y}\mathbf{x}_1 &= \boldsymbol{Y}\boldsymbol{Y}^\dagger(\mathbf{b} - \boldsymbol{Z}\mathbf{x}_2) \\ \mathbf{x}_1 &= \boldsymbol{Y}\boldsymbol{Y}^\dagger(\mathbf{b} - \boldsymbol{Z}\mathbf{x}_2) \end{aligned}$$

   Like earlier, we assume the existence of $\boldsymbol{Y}^\dagger$.

3. The goal is to reduce the system of equations. Unravelling block notation renders two linear systems:

   $$\boldsymbol{A}\mathbf{x}_1 + \boldsymbol{B}\mathbf{x}_2 = \mathbf{b}_1 \quad \text{and} \quad \boldsymbol{C}\mathbf{x}_1 + \boldsymbol{D}\mathbf{x}_2 = \mathbf{b}_2$$

   The main idea is to rewrite $\mathbf{x}_2$ as a function of $\mathbf{x}_1$ (which is known). We now want just one linear system rather than two. The first equation can be rewritten:

   $$\boldsymbol{A}\mathbf{x}_1 + \boldsymbol{B}\mathbf{x}_2 = \mathbf{b}_1 \iff \boldsymbol{A}\mathbf{x}_1 = \mathbf{b}_1 - \boldsymbol{B}\mathbf{x}_2 \iff \mathbf{x}_1 = \boldsymbol{A}^{-1}(\mathbf{b}_1 - \boldsymbol{B}\mathbf{x}_2)$$

   We can plug $\mathbf{x}_1$ into the second equation:

   $$\begin{aligned} \boldsymbol{C}\mathbf{x}_1 + \boldsymbol{D}\mathbf{x}_2 = \mathbf{b}_2 &\iff \boldsymbol{C}\boldsymbol{A}^{-1}(\mathbf{b}_1 - \boldsymbol{B}\mathbf{x}_2) + \boldsymbol{D}\mathbf{x}_2 = \mathbf{b}_2 \\ &\iff \boldsymbol{C}\boldsymbol{A}^{-1}\mathbf{b}_1 - \boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{B}\mathbf{x}_2 + \boldsymbol{D}\mathbf{x}_2 = \mathbf{b}_2 \\ \boldsymbol{C}\mathbf{x}_1 + \boldsymbol{D}\mathbf{x}_2 = \mathbf{b}_2 &\iff \underbrace{(\boldsymbol{C}\boldsymbol{A}^{-1}\boldsymbol{B} - \boldsymbol{D})}_{\in \mathbb{R}^{(n-d)\times(n-d)}}\mathbf{x}_2 = \boldsymbol{C}\boldsymbol{A}^{-1}\mathbf{b}_1 - \mathbf{b}_2 \end{aligned}$$

   Thus, assuming $\boldsymbol{A}^{-1}$ exists, we can reduce our system of equations to one in $(n-d) \times (n-d)$, and we can ignore $\mathbf{x}_1$ as that is already solved. We can borrow from our solver from the previous question to solve for $\mathbf{x}_2$ efficiently. We can also go in the other direction, and use $\boldsymbol{C}^{-1}$ instead. So if one inverse exists but not the other, it should still be possible to simplify computation.

Problem 4

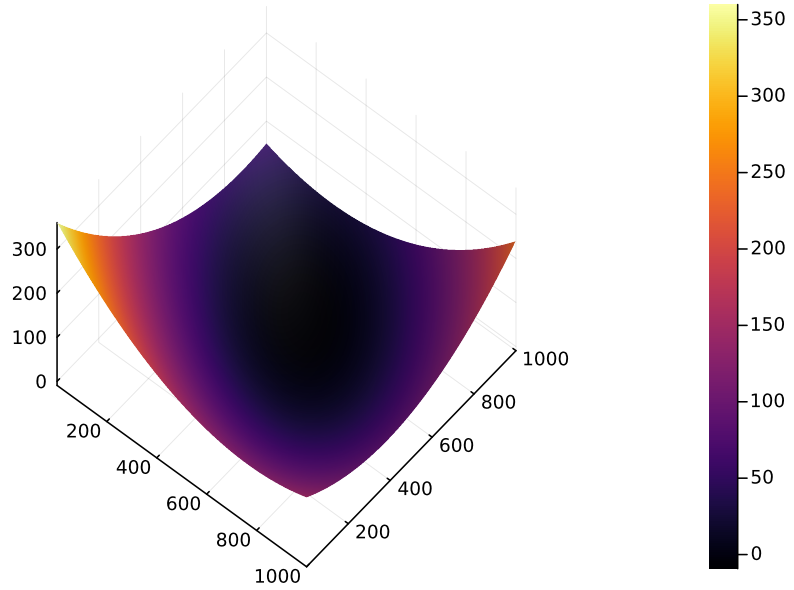
1. Assuming partial pivots are provided in the format of sparse matrices. **Code**

```
function check_lu(A, L, U, p, tol)
        @assert all(diag(L) .== 1)  #  check if L has a diagonal of 1
        @assert tril(L) == L  # check if L is lower triangular
        @assert triu(U) == U  # check if U is upper triangular
        @assert sort(p) == 1:length(p)   # check if it's actually a
           permutation

        @assert norm(A[p, :] - L * U) < tol  # check reconstruction
end
```

2. **Code**

```
using Plots

A = [2 1 ; 1 2]
b = [5.5 ; 0.5]

function f(A, b, x)
        return x' * A * x / 2 .- x' * b
end

n = 1000
max = 10
min = -10
U = zeros(n, n)

for (idx_i, val_i) in enumerate(range(min, max, length=n))
        for (idx_j, val_j) in enumerate(range(min, max, length=n))
                U[idx_i, idx_j] = f(A, b, [val_i ; val_j])
        end
end

p = surface(U, camera=(40, 60))
```

Output



Next, we can eliminate a variable.

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} 5.5 \\ 0.5 \end{bmatrix}$$

$$2\mathbf{x}_1 + \mathbf{x}_2 = 5.5 \iff \mathbf{x}_1 = \frac{5.5 - \mathbf{x}_2}{2}$$

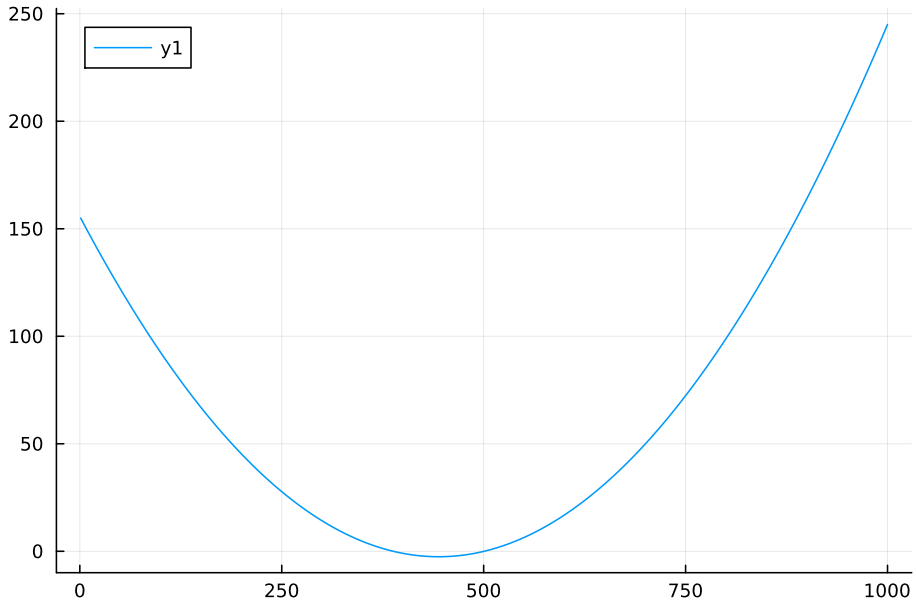
$$\therefore \mathbf{x}_1 + 2\mathbf{x}_2 = 0.5 \iff \frac{5.5 - \mathbf{x}_2}{2} + 2\mathbf{x}_2 = 0.5 \iff 5.5 - \mathbf{x}_2 + 4\mathbf{x}_2 = 1 \iff 4\mathbf{x}_2 = -4.5$$

Thus $\boldsymbol{A}' = 4$, $\mathbf{b} = -4.5$ eliminates $\mathbf{x}_1$. We can plot this:

Code

```
A = [4]
b = [-4.5]

u = zeros(n)
for (idx_i, val_i) in enumerate(range(min, max, length=n))
        u[idx_i] = f(A, b, [val_i])[1]
end

p = plot(u)
```

Output



3. We have:

$$\boldsymbol{A}_{ij} = \begin{cases} 1 & i = j \\ -1 & i = j + 1 \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j \in \{1, 2, \ldots, n\}$$

$$\text{Now, let } \boldsymbol{X}_{ij} := \begin{cases} 1 & j \geq i \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j \in \{1, 2, \ldots, n\}$$

We can then show that $\boldsymbol{AX} = \boldsymbol{I}$ and $\boldsymbol{XA} = \boldsymbol{I}$. The overall strategy is simple, we exclude all the zero-valued indices for each case and compute the resulting value. In particular, we will show that $\boldsymbol{Y}_{i,j} = 0$ if $i \neq j$ and 1 if $i = j$ for $\boldsymbol{Y} = \boldsymbol{XA}$ and $\boldsymbol{Y} = \boldsymbol{AX}$, and $\boldsymbol{Y}$ is the identity matrix $\boldsymbol{I}$.

Case 1. $\boldsymbol{AX} = \boldsymbol{I}$. Consider indices $i, j \in \{1, 2, \ldots, n\}$:

a. **Case a:** $i = j$

$$(\boldsymbol{AX})_{ii} = \langle \boldsymbol{A}_{i,:}, \boldsymbol{X}_{:,i} \rangle = \sum_{k=1}^{n} \boldsymbol{A}_{ik}\boldsymbol{X}_{ki} = \boldsymbol{A}_{ii}\boldsymbol{X}_{ii} + \boldsymbol{A}_{i,i-1}\boldsymbol{X}_{i-1,i} + 0 = 1 \cdot 1 + -1 \cdot 0 + 0 = 1$$

b. **Case b:** $i < j$

$$(\boldsymbol{AX})_{ij} = \langle \boldsymbol{A}_{i,:}, \boldsymbol{X}_{:,j} \rangle = \sum_{k=1}^{n} \boldsymbol{A}_{ik}\boldsymbol{X}_{kj} = \boldsymbol{A}_{ii}\boldsymbol{X}_{ij} + \boldsymbol{A}_{i,i-1}\boldsymbol{X}_{i-1,j} = 0 + 0 = 0$$

c. **Case c:** $i > j$

$$(\boldsymbol{AX})_{ij} = \langle \boldsymbol{A}_{i,:}, \boldsymbol{X}_{:,j} \rangle = \sum_{k=1}^{n} \boldsymbol{A}_{ik}\boldsymbol{X}_{kj} = \boldsymbol{A}_{ii}\boldsymbol{X}_{ij} + \boldsymbol{A}_{i,i-1}\boldsymbol{X}_{i-1,j} = 1 - 1 = 0$$

Case 2. $\boldsymbol{XA} = \boldsymbol{I}$

a. **Case a:** $i = j$

$$(\boldsymbol{XA})_{ii} = \langle \boldsymbol{X}_{i,:}, \boldsymbol{A}_{:,i} \rangle = \sum_{k=1}^{n} \boldsymbol{X}_{ik}\boldsymbol{A}_{ki} = \sum_{k=1}^{i} \boldsymbol{A}_{ki} = \sum_{k=1}^{i-1} 0 + \boldsymbol{A}_{ii} = 0 + 1 = 1$$

b. **Case b:** $i < j$

$$(\boldsymbol{X}\boldsymbol{A})_{ij} = \langle \boldsymbol{X}_{i,:}, \boldsymbol{A}_{:,j} \rangle = \sum_{k=1}^{n} \boldsymbol{X}_{ik}\boldsymbol{A}_{kj} = \sum_{k=1}^{i} \boldsymbol{A}_{kj} = 0$$

c. **Case c:** $i > j$

$$(\boldsymbol{X}\boldsymbol{A})_{ij} = \langle \boldsymbol{X}_{i,:}, \boldsymbol{A}_{:,j} \rangle = \sum_{k=1}^{n} \boldsymbol{X}_{ik}\boldsymbol{A}_{kj} = \sum_{k=1}^{i} \boldsymbol{A}_{kj} = \boldsymbol{A}_{j-1,j} + \boldsymbol{A}_{jj} + 0 = -1 + 1 = 0$$

Since we have that $\boldsymbol{A}\boldsymbol{X} = \boldsymbol{X}\boldsymbol{A} = \boldsymbol{I}$, we have found a generalized form of the inverse as $\boldsymbol{A}^{-1} = \boldsymbol{X}$ for arbitrary $n \in \mathbb{N}$.