PHSCS 513R - Linear Algebra Group Assignment

Team:

- Scott Johnstun
- Aiden Harbick
- Mark Anderson

Git Repo
[https://git.physics.byu.edu/computational-physics/linearalgebra2 (https://git.physics.byu.edu/computational-physics/linearalgebra2)](https://git.physics.byu.edu/computational-physics/linearalgebra2)

Overview

Consider the one-dimensional boundary value problem that arises in fluid dynamics:

$$-u''(x) + V(x)u'(x) = f(x), x \in [0,1] \tag{1}$$

$$u(0) = u(1) = 0 \tag{2}$$

where we will take $V(x)$, $f(x)$ to be constants:

$$V(x) = \gamma \tag{3}$$

$$f(x) = 1 \tag{4}$$

Therefore, the total problem is:

$$-u''(x) + \gamma u'(x) = 1, x \in [0,1] \tag{5}$$

$$u(0) = u(1) = 0 \tag{6}$$

1. Problem Formulation

Part A

Write this boundary value problem as a variational problem for some test function ϕ:

$$A(u,\phi) = F(\phi) \tag{1}$$

First we multiply by our test function and then integrate over the domain of the ODE:

$$\int_0^1 -u''(x)\phi(x)dx + \int_0^1 \gamma u'(x)\phi(x)dx = \int_0^1 f(x)\phi(x)dx. \tag{2}$$

Now we integrate the first term by parts to obtain

$$\int_0^1 u'(x)\phi'(x)dx - u'(x)\phi(x)\Big|_0^1 + \int_0^1 \gamma u'(x)\phi(x)dx = \int_0^1 f(x)\phi(x)dx. \tag{3}$$

Assuming ϕ has the same boundary conditions as u, the boundary term goes to zero. We then define

$$A(u, \phi) \equiv \int_0^1 u'(x)\phi'(x)dx + \int_0^1 \gamma u'(x)\phi(x)dx \tag{4}$$

and

$$F(\phi) \equiv \int_0^1 f(x)\phi(x)dx. \tag{5}$$

so we have

$$A(u, \phi) = F(\phi). \tag{6}$$

Part B

Take ϕ_i to be the "hat" functions discussed in class and approximate $u(x)$ as a linear combination of these basis vectors:

$$u(x) = \sum_i u_i \phi_i(x) \tag{1}$$

Show that the variational problem from part (a) becomes a linear algebra problem of the form

$$A\vec{x} = \vec{b}. \tag{2}$$

Derive expressions for the matrix A and vector $\vec{b}$. Show specifically that $A_{ij} = A(\phi_j, \phi_i)$, and that it can be written as the sum of two matrices $A = A_1 + A_2$ where A_1 and A_2 correspond to the first two terms on the left-hand side of the ODE. (Hint: you should find that A_1 is symmetric while A_2 is skew-symmetric).

We now define ϕ_i to be the ith hat function, defined for a grid of n cells as

$$\phi_i(x) = \begin{cases} nx - (i-1) & \text{if } \frac{i-1}{n} < x < \frac{i}{n} \\ -nx + i + 1 & \text{if } \frac{i}{n} < x < \frac{i+1}{n} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

and then approximate $u(x)$ as a linear combination of these hat functions: $u(x) = \sum_j u_j \phi_j(x)$, where each u_j is a real number.

Then, substituting this expression into our weakly-formulated differential equation with test function $\phi_i(x)$, we get

$$\int_0^1 \sum_j u_j \phi_j'(x)\phi_i'(x)dx + \int_0^1 \gamma \sum_j u_j \phi_j'(x)\phi_i(x)dx = \int_0^1 f(x)\phi_i(x)dx \tag{4}$$

$$\implies \sum_j u_j \left(\int_0^1 \phi_j'(x)\phi_i'(x)dx + \int_0^1 \gamma \phi_j'(x)\phi_i(x)dx \right) = \int_0^1 f(x)\phi_i(x)dx, \tag{5}$$

which has the form $\sum_i A_{ij} x_j = b_i$, which is the exact form of a matrix equation. Thus, defining

$$A_{ij} = \int_0^1 \phi_j'(x)\phi_i'(x)dx + \int_0^1 \gamma \phi_j'(x)\phi_i(x)dx \tag{6}$$

and

$$b_i = \int_0^1 f(x)\phi_i(x)dx, \tag{7}$$

we arrive at the linear algebra problem

$$Ax = b \tag{8}$$

whose solution $x_j = u_j$ gives us the weights for the test function from which we can construct the solution $u(x) = \sum_j u_j \phi_j(x)$.

We see that A_{ij} can be written as $A^{(1)} + A^{(2)}$. Here we have

$$A_{ij}^{(1)} = \int_0^1 \phi_j'(x)\phi_i'(x)dx = \begin{cases} 2n & \text{if } 1 \leq i \leq n-1 \text{ and } i = j \\ -n & \text{if } |i-j| = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{9}$$

which has the matrix form

$$\begin{bmatrix} 2n & -n & 0 & & 0 \\ -n & 2n & -n & \dots & 0 \\ 0 & -n & 2n & & 0 \\ & \vdots & & & \vdots \\ 0 & 0 & 0 & \dots & 2n \end{bmatrix}. \tag{10}$$

We can clearly see that $A^{(1)}$ is symmetric. We also have

$$A_{ij}^{(2)} = \int_0^1 \gamma\phi_j'(x)\phi_i(x)dx = \begin{cases} -\gamma/2 & \text{if } j = i-1 \\ \gamma/2 & \text{if } j = i+1 \\ 0 & \text{otherwise} \end{cases}, \tag{11}$$

which has the matrix form

$$\begin{bmatrix} 0 & \gamma/2 & 0 & & 0 \\ -\gamma/2 & 0 & \gamma/2 & \dots & 0 \\ 0 & -\gamma/2 & 0 & & \vdots \\ & \vdots & & & \gamma/2 \\ 0 & 0 & \dots & -\gamma/2 & 0 \end{bmatrix}. \tag{12}$$

With $f(x) = 1$, we have for b

$$b_i = \int_0^1 \phi_i(x)dx = \frac{1}{n}. \tag{13}$$

Part C

Implement a driver routine that will return A and $\vec{b}$ given inputs n and γ. The matrix A should be implemented as a sparse representation in your environment.

In [1]:
```
import numpy as np
from scipy.sparse import diags
def generate_system(n, gam):

    # diagonal entries
    d = np.ones(n)*2*n

    # upper diagonal
    u = np.ones(n-1)*(gam/2-n)

    # lower diagonal
    l = np.ones(n-1)*(-gam/2-n)

    # construct sparse matrix
    A = diags([l, d, u], [-1, 0, 1])

    # assemble b
    b = np.ones((n,1))*1/n

    return A, b
```

In [2]:
```
A,b = generate_system(10,1)

print(A,"\n");
print(b);
```

```
  (1, 0)	-10.5
  (2, 1)	-10.5
  (3, 2)	-10.5
  (4, 3)	-10.5
  (5, 4)	-10.5
  (6, 5)	-10.5
  (7, 6)	-10.5
  (8, 7)	-10.5
  (9, 8)	-10.5
  (0, 0)	20.0
  (1, 1)	20.0
  (2, 2)	20.0
  (3, 3)	20.0
  (4, 4)	20.0
  (5, 5)	20.0
  (6, 6)	20.0
  (7, 7)	20.0
  (8, 8)	20.0
  (9, 9)	20.0
  (0, 1)	-9.5
  (1, 2)	-9.5
  (2, 3)	-9.5
  (3, 4)	-9.5
  (4, 5)	-9.5
  (5, 6)	-9.5
  (6, 7)	-9.5
  (7, 8)	-9.5
  (8, 9)	-9.5

[[0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]
 [0.1]]
```

2. Implement the GMRES Algorithm

Your function should have the following signature: *mygmres(l, b, x0, n, M, A)* and should compute I iterations of the GMRES and return the approximate solution of $A\vec{x} = \vec{b}$ with initial iterate x0. Here, n is the dimension of the problem, A is an $n \times n$ matrix, M is an $n \times n$ matrix that defines the inner product used for calculating vector norms (and therefore the error).

Coordinate your group work through a git repository on the deparment server. Make your repository publicly visible and share the link as a solution to this problem. The commit history should include commits from all group members.

In [3]:

```
def mygmres(I,b,x0,n,M,A):

    A = np.linalg.inv(M)@A
    b = np.linalg.inv(M)@b

    r0 = b - A@x0

    beta = np.linalg.norm(r0)

    V = np.zeros((n,n+1))
    H = np.zeros((n+1,n))
    W = np.zeros((n,n))
    xs = np.zeros((n,1))

    V[:,0] = np.transpose(r0 / beta)

    for j in range(0,I):

        W[:,j] = np.dot(A,V[:,j])

        for i in range(0,j+1):
            H[i,j] = np.dot(W[:,j],V[:,i])
            W[:,j] = W[:,j] - np.dot(H[i,j],V[:,i])

        H[j+1,j] = np.linalg.norm(W[:,j])
        if H[j+1,j] == 0:
            break

        V[:,j+1] = W[:,j] / H[j+1,j]

    n,m = H.shape

    a = np.zeros((n,1))
    a[0] = beta

    ys = np.linalg.lstsq(H,a,rcond = None)[0]

    for i in range(0,len(ys)):
        xs[i] = x0[i] + V[i,0:len(ys)] @ ys

    Vs = V

    Hs = H


    return xs, ys, Vs, Hs
```

Example from the paper

In [4]:
```
A = np.array([[1,4,7],[2,9,7],[5,8,3]])
b = np.array([[1],[8],[2]])
I = 3
M = np.identity(3)
n = 3
x0 = np.array([[1],[1],[1]])

xs, ys, Vs, Hs = mygmres(I,b,x0,n,M,A)

print(xs)
```

```
[[-2.18103448]
 [ 1.8362069 ]
 [-0.59482759]]
```

Checking solutions against SciPy

In [5]:

```
import scipy.linalg as la

# Defining our values
I = 10
n = 10
A = np.random.rand(n,n)
b = np.random.rand(n,1)
x0 = np.random.rand(n,1)
M = np.identity(n)

# Solving in scipy
x_scipy = la.solve(A,b)
print("SciPy:\n",x_scipy)

# # Solving in KryPy
# q = krypy.linsys.LinearSystem(A,b)
# krypy.linsys.Gmres(q)

# Using our GMRES algorithm
xs, ys, Vs, Hs = mygmres(I,b,x0,n,M,A)
print("\nGMRES:\n",xs)
```

```
SciPy:
 [[-0.37302578]
 [ 0.94743841]
 [ 0.13725691]
 [ 1.03744751]
 [-1.30860706]
 [ 1.51702971]
 [-0.34418406]
 [ 0.85087283]
 [-1.02413067]
 [-0.92436146]]

GMRES:
 [[-0.37302578]
 [ 0.94743841]
 [ 0.13725691]
 [ 1.03744751]
 [-1.30860706]
 [ 1.51702971]
 [-0.34418406]
 [ 0.85087283]
 [-1.02413067]
 [-0.92436146]]
```

3. Solving the FEM Problem

Use your GMRES function to solve the finite-element formulation of the variational problem for the cases

\begin{equation}V(x) = 1\end{equation}

and

\begin{equation}V(x) = n+1\end{equation}

using M as the identity matrix. For each case, run with $n = 16,32,64,128$ and $I = 2,4,8,16,32,64,...$, increasing I until the error (I.e., norm of the residual divided by n) is below 10^{-6}. Plot your most accurate solution (as a function of x) as well as the error versus functions of n and I.

In [6]:

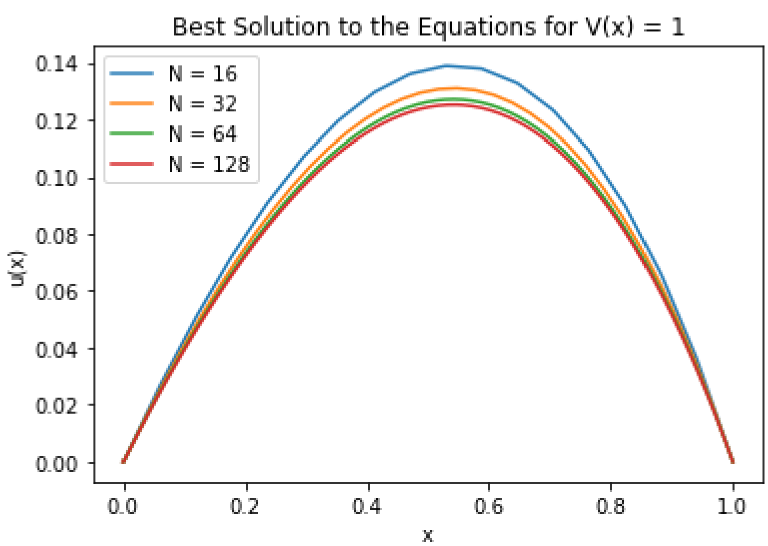
```
import matplotlib.pyplot as plt
```

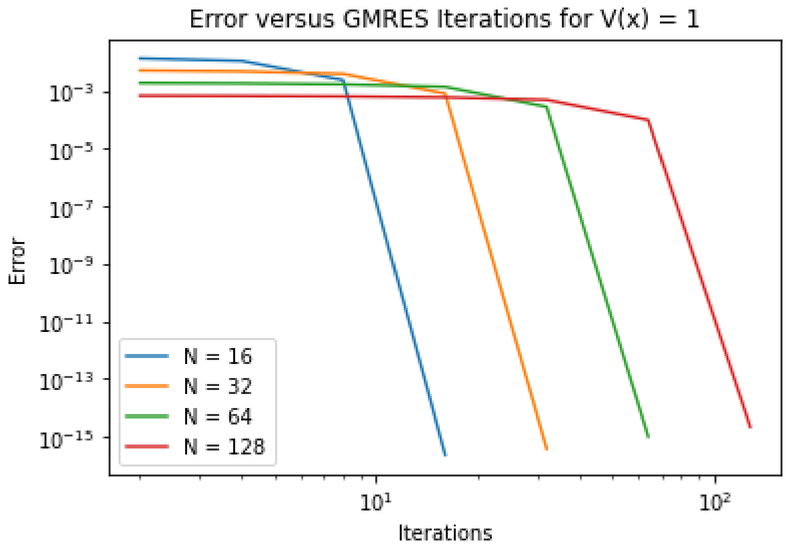
In [8]:

```
def sol_to_func(uvals, x):
    # There should be exactly n values of u_i
    n = len(uvals)

    # Hat functions
    def phi(i, x, n):
        xnew = np.zeros_like(x)
        for (k, val) in enumerate(x):
            if val > (i-1)/n and val <= i/n:
                xnew[k] = n*val - (i - 1)
            if val > i/n and val < (i+1)/n:
                xnew[k] = -n*val + i + 1
        return xnew

    # construct u with the hat functions and ui's
    u = np.zeros_like(x)
    for (i, ui) in enumerate(uvals):
        u += ui*phi(i+1, x, n+1)

    return u

def find_sol(n, Vx, x):
    A, b = generate_system(n, Vx)
    x0 = np.zeros((n,1))
    M = np.identity(n)
    errs = []
    Is = []
    error = 1
    count = 1

    while error > 1E-6:
        I = int(2**count)
        xs, ys, Vs, Hs = mygmres(I,b,x0,n,M,A)
        error = np.linalg.norm((b - A@xs)/n)
        errs.append(error)
        Is.append(I)
        count += 1
        if count > 20:
            break
    u = sol_to_func(xs, x)
    return u, errs, Is
```

In [9]:
```
x = np.linspace(0,1,num=200)
u16, errs16, Is16 = find_sol(16, 1, x)
u32, errs32, Is32 = find_sol(32, 1, x)
u64, errs64, Is64 = find_sol(64, 1, x)
u128, errs128, Is128 = find_sol(128, 1, x)
plt.figure(1)
plt.plot(x,u16, label = 'N = 16')
plt.plot(x,u32, label = 'N = 32')
plt.plot(x,u64, label = 'N = 64')
plt.plot(x,u128, label = 'N = 128')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title(f'Best Solution to the Equations for V(x) = 1')
plt.legend()
plt.figure(2)
plt.loglog(Is16,errs16, label = 'N = 16')
plt.loglog(Is32,errs32, label = 'N = 32')
plt.loglog(Is64,errs64, label = 'N = 64')
plt.loglog(Is128,errs128, label = 'N = 128')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Error versus GMRES Iterations for V(x) = 1')
plt.legend()
```

Out[9]: <matplotlib.legend.Legend at 0x7f92e0431670>

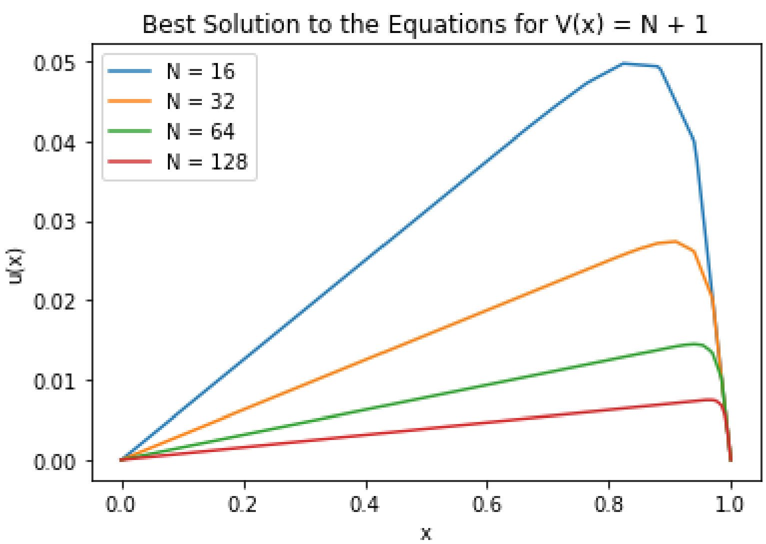


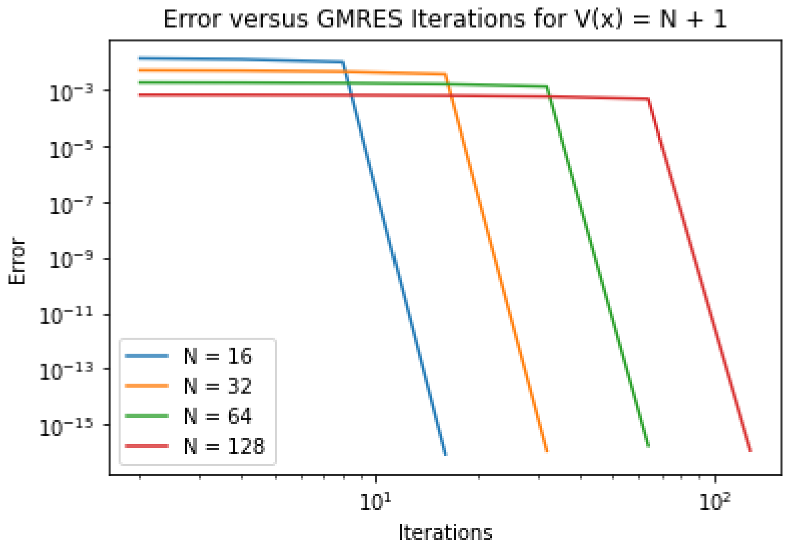

Error versus GMRES Iterations for V(x) = 1
Error
Iterations
N = 16
N = 32
N = 64
N = 128

In [10]:

```
x = np.linspace(0,1,num=200)
u16, errs16, Is16 = find_sol(16, 17, x)
u32, errs32, Is32 = find_sol(32, 33, x)
u64, errs64, Is64 = find_sol(64, 65, x)
u128, errs128, Is128 = find_sol(128, 129, x)
plt.figure(1)
plt.plot(x,u16, label = 'N = 16')
plt.plot(x,u32, label = 'N = 32')
plt.plot(x,u64, label = 'N = 64')
plt.plot(x,u128, label = 'N = 128')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title(f'Best Solution to the Equations for V(x) = N + 1')
plt.legend()
plt.figure(2)
plt.loglog(Is16,errs16, label = 'N = 16')
plt.loglog(Is32,errs32, label = 'N = 32')
plt.loglog(Is64,errs64, label = 'N = 64')
plt.loglog(Is128,errs128, label = 'N = 128')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Error versus GMRES Iterations for V(x) = N + 1')
plt.legend()
```

Out[10]: <matplotlib.legend.Legend at 0x7f92e08eac70>





4. Preconditioning GMRES

Now consider a preconditioned version of the problem: $\tilde{A}\vec{x} = \tilde{b}$ where

- $\tilde{A} = A_1^{-1} A$
- $\tilde{b} = A_1^{-1} \vec{b}$
- $M = A_1$

Here, A_1 is known as a preconditioning matrix and is used to speed up the convergence or improve the accuracy of solution methods.

Part A

Show that this problem is formally equivalent to the one you considered in probelm 3 (i.e., show that any candidtate solution $\vec{x}$ will have the same residual for both problems).

Lets say that we have a candidate solution $\vec{x}$ to the equation $A\vec{x} = \vec{b}$ with residual $\vec{r} = \vec{b} - A\vec{x}$. We also let $\tilde{A} = A_1^{-1} A$ and $\tilde{b} = A_1^{-1} \vec{b}$. This preconditioned problem is formally equivalent for the same $\vec{x}$ we can see this by considering: \begin{align} \tilde{r} &= \tilde{b} - \tilde{A}\vec{x} \\ \tilde{r} &= A_1^{-1} \vec{b} - A_1^{-1} A\vec{x} \\ \tilde{r} &= A_1^{-1}(\vec{b} - A\vec{x}) \\ \tilde{r} &= A_1^{-1}\vec{r} \end{align} So clearly these residuals approach one another as $\vec{r} \to 0$ (or as $\vec{x}$ approaches the exact solution).

Part B

Argue that $\tilde{A}$ and $\tilde{b}$ can be calculated efficiently, even though they formally involve a matrix inverse (This is a requirement for a preconditioning matrix to be useful).

For this particular case, our preconditioning matrix (A_1) is a tridiagonal matrix, which means that using LU decomposition to compute the inverse is an $O(N)$ process (as stated in Numerical Recipes Section 2.4), so $\tilde{A}$ and $\tilde{b}$ can be calculated efficently, and as long as the preconditioning saves a significant amount of time, this is well worth it.

Part C

Repeat problem 3 for the preconditioned version of the problem.

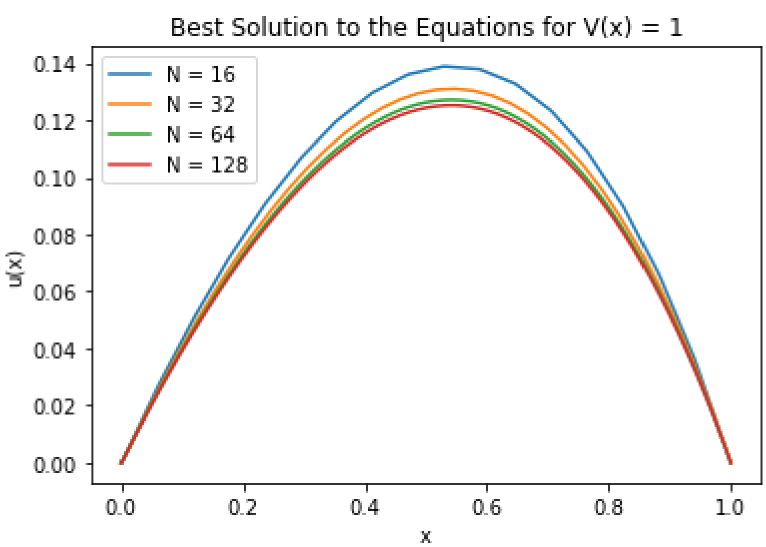
In [11]:
```python
def generate_A1(n):
    A1 = np.zeros((n,n))
    for i in range(n):
        A1[i][i] = 2*n
        if i > 0:
            A1[i][i-1] = -n
        if i < n-1:
            A1[i][i+1] = -n

    return A1
```

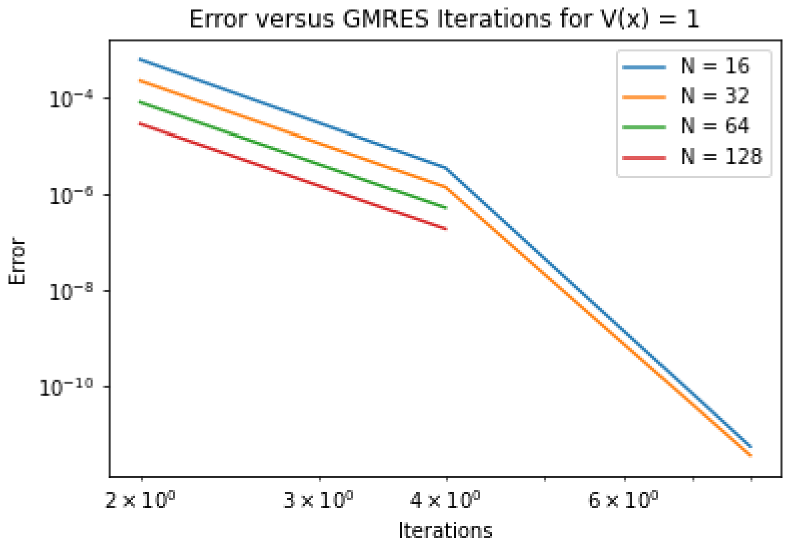
In [12]:
```python
def find_sol_preconditioned(n, Vx, x):
    A, b = generate_system(n, Vx)
    x0 = np.zeros((n,1))
    M = generate_A1(n)
    errs = []
    Is = []
    error = 1
    count = 1

    while error > 1E-6:
        I = int(2**count)
        xs, ys, Vs, Hs = mygmres(I,b,x0,n,M,A)
        error = np.linalg.norm((b - A@xs)/n)
        errs.append(error)
        Is.append(I)
        count += 1
        if count > 20:
            break
    u = sol_to_func(xs, x)
    return u, errs, Is

```

In [13]:

```
x = np.linspace(0,1,num=200)
u16, errs16, Is16 = find_sol_preconditioned(16, 1, x)
u32, errs32, Is32 = find_sol_preconditioned(32, 1, x)
u64, errs64, Is64 = find_sol_preconditioned(64, 1, x)
u128, errs128, Is128 = find_sol_preconditioned(128, 1, x)
plt.figure(1)
plt.plot(x,u16, label = 'N = 16')
plt.plot(x,u32, label = 'N = 32')
plt.plot(x,u64, label = 'N = 64')
plt.plot(x,u128, label = 'N = 128')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title(f'Best Solution to the Equations for V(x) = 1')
plt.legend()
plt.figure(2)
plt.loglog(Is16,errs16, label = 'N = 16')
plt.loglog(Is32,errs32, label = 'N = 32')
plt.loglog(Is64,errs64, label = 'N = 64')
plt.loglog(Is128,errs128, label = 'N = 128')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Error versus GMRES Iterations for V(x) = 1')
plt.legend()
```

Out[13]: <matplotlib.legend.Legend at 0x7f92dff38070>





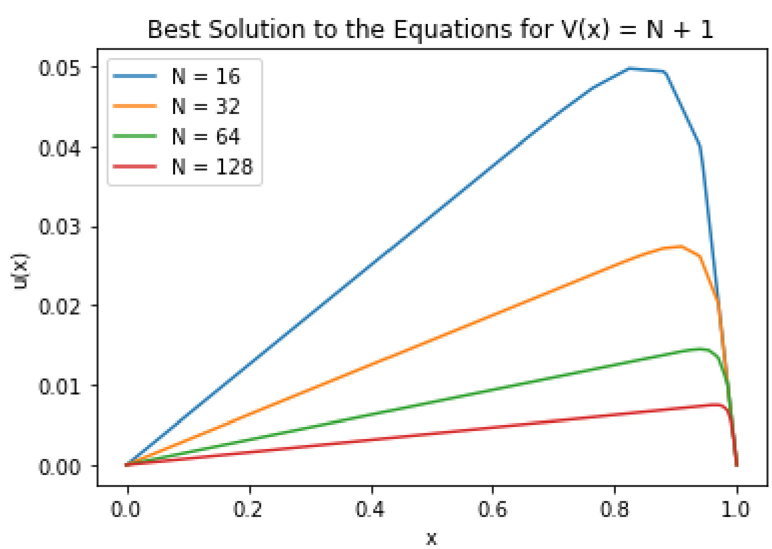
Error versus GMRES Iterations for V(x) = 1
N = 16
N = 32
N = 64
N = 128
Error
10^{-4}
10^{-6}
10^{-8}
10^{-10}
2×10^{0}
3×10^{0}
4×10^{0}
6×10^{0}
Iterations

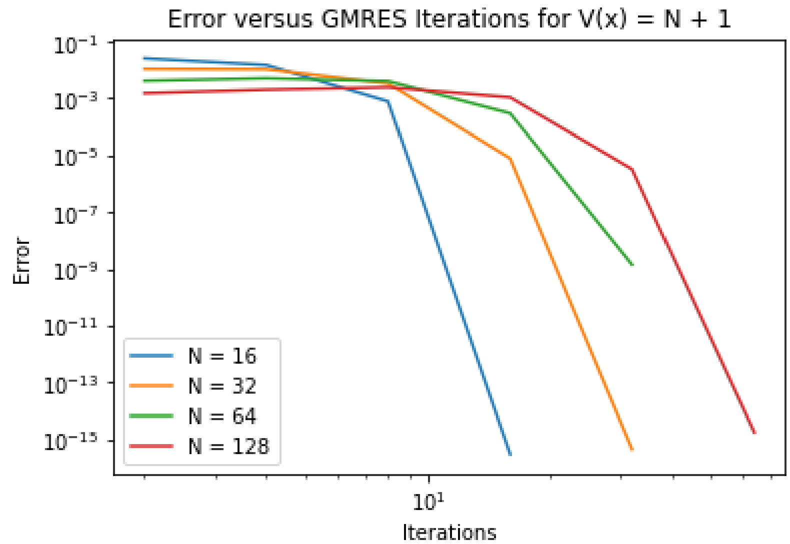
In [14]:

```python
x = np.linspace(0,1,num=200)
u16, errs16, Is16 = find_sol_preconditioned(16, 17, x)
u32, errs32, Is32 = find_sol_preconditioned(32, 33, x)
u64, errs64, Is64 = find_sol_preconditioned(64, 65, x)
u128, errs128, Is128 = find_sol_preconditioned(128, 129, x)
plt.figure(1)
plt.plot(x,u16, label = 'N = 16')
plt.plot(x,u32, label = 'N = 32')
plt.plot(x,u64, label = 'N = 64')
plt.plot(x,u128, label = 'N = 128')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title(f'Best Solution to the Equations for V(x) = N + 1')
plt.legend()
plt.figure(2)
plt.loglog(Is16,errs16, label = 'N = 16')
plt.loglog(Is32,errs32, label = 'N = 32')
plt.loglog(Is64,errs64, label = 'N = 64')
plt.loglog(Is128,errs128, label = 'N = 128')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Error versus GMRES Iterations for V(x) = N + 1')
plt.legend()
```

Out[14]:

```
<matplotlib.legend.Legend at 0x7f92e120d640>
```





Part D

How quickly does the convergence rate for your GMRES algorithm compare with that in problem 3? Why? (Hint: Consider the condition number of hte two problems)

The solutions for the preconditioned system converged much faster in every case, with solutions with error below 10^{-6} occuring for 8 iterations or less, whereas with no preconditioning it took up to 128 iterations for N = 128. This is because the condition number of the preconditioned problem is smaller, and therefore the solutions converge much faster