A NUMERICAL METHOD FOR SOLVING FREDHOLM INTEGRAL EQUATIONS OF THE SECOND KIND

EVAN S. ALLHANDS

Contents

Date: April 3rd, 2022.

1. Developing The Method

Consider a Fredholm Integral Equation of the Second Kind:

$$\phi(x) = \phi_0(x) + \lambda \int_a^b K(x,y)\phi(y)dy \tag{1}$$

where $\phi_0(x)$ is a given equation, λ is a real-valued scalar, $K(x,y)$ is the kernel of integration, and $x, y \in [a,b]$. In solving for the unknown solution $\phi(x)$, we will need to evaluate the integral appearing in the equation. However, the presence of the unknown solution ϕ in the integrand makes it necessary to approximate the integral, rather than try to evaluate explicitly. This is accomplished by leveraging basic principles of finite-dimensional linear algebra. Before proceeding with the explanation of the method, we will briefly cover an integral approximation method which will be heavily relied upon.

1.1. Newton-Cotes & Composite Newton-Cotes Trapezoidal Method.

The Newton-Cotes Trapezoidal method provides an approximation of an integral by finding the area of an approximating trapezoid. The general formula for one interval is given by:

$$\int_a^b f(x)dx \approx \frac{(b-a)}{2}[f(a) + f(b)].$$

One can obtain a more accurate approximation of the integral by dividing the interval of integration in to subintervals, leading to what is commonly referred to as the *Composite* Newton-Cotes Trapezoidal method.

Composite Newton-Cotes Trapezoidal Method. The Composite Newton-Cotes Trapezoidal method begins by partitioning the interval $[a,b]$ in to subintervals with n quadrature nodes such that $a = x_0 \leq x_1 \leq \cdots \leq x_{n-1} \leq x_n = b$. Observe that we have m subintervals, where $m = n-1$. We will let the quadrature nodes be evenly spaced such that any node can be given by $x_i = a + ih$ where h is the length of a subinterval given by $h = \frac{(b-a)}{m}$ for $i = 0, 1, \ldots, n-1, n$. Then an

approximation of the integral for m subintervals can be obtained by:

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{2}\sum_{i=0}^{m}(f(x_i)+f(x_{i+1}))\\ &\approx \frac{h}{2}\Big[[f(x_0)+f(x_1)]+[f(x_1)+f(x_2)]+\cdots+[f(x_{n-1})+f(x_n)]\Big]\\ &\approx \frac{h}{2}\Big[f(x_0)+2f(x_1)+2f(x_2)+\cdots+f(x_n)\Big]\\ &\approx h\Big[\frac{1}{2}f(x_0)+f(x_1)+f(x_2)+\cdots+\frac{1}{2}f(x_n)\Big].\end{aligned} \tag{2}$$

Observe that in the Composite Newton Cotes formula, the first and last terms are counted only once, where as the interior terms are counted twice.

1.2. **Untangling the Fredholm Integral Equation of the Second Kind.** We now return our attention to the development of a numerical method for solving Fredholm Integral Equations of the Second Kind:

$$\phi(x) = \phi_0(x) + \lambda\int_a^b K(x,y)\phi(y)dy. \tag{3}$$

We begin by choosing quadrature nodes x_i to subdivide the interval of integration such that $a = x_1 \le x_2 \le \cdots \le x_n = b$. Evaluating (3) at each quadrature node x_i yields a system of equations to be solved:

$$\begin{aligned}\phi(x_1) - \lambda\int_a^b K(x_1,y)\phi(y)dy &= \phi_0(x_1)\\ \phi(x_2) - \lambda\int_a^b K(x_2,y)\phi_0(y)dy &= \phi_0(x_2)\\ &\vdots\\ \phi_(x_n) - \lambda\int_a^b K(x_n,y)\phi_0(y)dy &= \phi_0(x_n)\end{aligned} \tag{4}$$

We now approximate the integral in each of the above equations using a quadrature method. Using the Composite Newton-Cotes Trapezoidal method, we choose quadrature nodes x_j such that $a = x_1 \le x_2 \le \cdots \le x_n = b$, and proceed to approximate the integral by:

$$\int_a^b K(x,x_j)\phi(x_j)dx \approx h\Big[\frac{1}{2}K(x,x_1)\phi(x_1)+K(x,x_2)\phi(x_2)+\cdots+\frac{1}{2}K(x_n,x_n)\phi(x_n)\Big].$$

Thus our system of equations can then be represented by:

$$\begin{aligned}
&\phi(x_1) - \frac{\lambda h}{2}K(x_1,x_1)\phi(x_1) + \lambda h K(x_1,x_2)\phi(x_2) + \cdots + \frac{\lambda h}{2}K(x_1,x_n)\phi(x_n) = \phi_0(x_1)\\
&\phi(x_2) - \frac{\lambda h}{2}K(x_2,x_1)\phi(x_1) + \lambda h K(x_2,x_2)\phi(x_2) + \cdots + \frac{\lambda h}{2}K(x_2,x_n)\phi(x_n) = \phi_0(x_2)\\
&\qquad\qquad\qquad\qquad\vdots\\
&\phi(x_n) - \frac{\lambda h}{2}K(x_n,x_1)\phi(x_1) + \lambda h K(x_n,x_2)\phi(x_2) + \cdots + \frac{\lambda h}{2}K(x_n,x_n)\phi(x_n) = \phi_0(x_n).
\end{aligned} \tag{5}$$

Observe that when $i = j$ for $K(x_i, x_j)$, the unknown function $\phi(x_j)$ appears twice in the equation–once as the initial term, but also in the j^{th} term of the integral approximation. Thus, we can factor out $\phi(x_j)$ and–by some manipulation and rearrangement–arrive at the following system of equations:

$$\begin{aligned}
&\left(1 - \frac{\lambda h}{2}K(x_1,x_1)\right)\phi(x_1) + \lambda h K(x_1,x_2)\phi(x_2) + \cdots + \frac{\lambda h}{2}K(x_1,x_n)\phi(x_n) = \phi_0(x_1)\\
&\frac{\lambda h}{2}K(x_2,x_1)\phi(x_1) + \left(1 - \lambda h K(x_2,x_2)\right)\phi(x_2) + \cdots + \frac{\lambda h}{2}K(x_2,x_n)\phi(x_n) = \phi_0(x_2)\\
&\qquad\qquad\qquad\qquad\ddots\\
&\frac{\lambda h}{2}K(x_n,x_1)\phi(x_1) + \lambda h K(x_n,x_2)\phi(x_2) + \cdots + \left(1 - \frac{\lambda h}{2}K(x_n,x_n)\right)\phi(x_n) = \phi_0(x_n).
\end{aligned} \tag{6}$$

This system can be decomposed and written formally as:

$$(\mathbf{I} - h\lambda\mathbf{K}\mathbf{D})\vec{\Phi} = \vec{\Phi_0} \tag{7}$$

where $\mathbf{I}$ is an $n \times n$ identity matrix, h is the length of one subinterval, and λ is a real-valued scalar. The elements of the matrix $\mathbf{K}$ are the evaluations of $K_{i,j} = K(x_i, x_j)$ and the matrix $\mathbf{D}$ is a diagonal matrix containing the weights for the Composite Newton-Cotes Trapezoidal method. The components of $\vec{\Phi}_0$ are evaluations of $\phi_0(x_i)$ and $\vec{\Phi}$ is our solution to solve for. In matrix form, (7) is given by:

$$\left(\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} - h\lambda \begin{bmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ K_{21} & K_{22} & & K_{2n} \\ \vdots & & \ddots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} \end{bmatrix} \begin{bmatrix} \frac{1}{2} & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{2} \end{bmatrix}\right) \begin{bmatrix} \phi(x_1) \\ \phi(x_2) \\ \vdots \\ \phi(x_n) \end{bmatrix} = \begin{bmatrix} \phi_0(x_1) \\ \phi_0(x_2) \\ \vdots \\ \phi_0(x_n) \end{bmatrix}.$$

Upon solving for $\vec{\Phi}$, numerical interpolation methods can be used to obtain an approximate solution to the Fredholm Integral Equation of the Second Kind[1].

[1]Details in section 1.4

1.3. **Error Estimation.** For an integral $I = \int_a^b f(x)dx$, the approximation by the trapezoidal rule for one interval is given by

$$A_0 = \frac{(b-a)}{2}[f(a) + f(b)]$$

and the error would be

$$I - A_0 = E_0 \leq -\frac{(b-a)^3}{12} f''(\xi) \text{ for some } \xi \in [a, b]$$

with I representing the exact value of the integral. Since we don't have the exact value of the integral to work with, we will make another approximation by halving the lengths of the subintervals. Our new approximation, for two subintervals, is given by:

$$A_1 = \frac{(b-a)}{4}[f(a) + f(c) + f(c) + f(b)] \text{ where } c = \frac{(a+b)}{2}$$

and, the error of our approximation would be obtained by:

$$I - A_1 = E_1 \leq -\frac{(b-a)^3}{12(2)^2} f''(\eta) \text{ for some } \eta \in [a, b]. \tag{8}$$

The difference between our approximations can be found by:

$$\begin{aligned} \Delta &= A_1 - A_0 \\ &= (I - E_1) - (I - E_0) \\ &= \frac{(b-a)^3}{12} f''(\xi) - \frac{(b-a)^3}{12(2)^2} f''(\eta) \\ &\text{Assume that } f''(\eta) \approx f''(\xi), \\ &= \frac{(b-a)^3}{16} f''(\xi) \end{aligned} \tag{9}$$

We conclude, therefore, that $f''(\xi) \approx \frac{16}{(b-a)^3}\Delta$, which we will substitute in to (8) find the error of our second approximation[2].

$$\begin{aligned} E_1 &\approx \frac{(b-a)^3}{48} f''(\xi) \\ &\approx \frac{(b-a)^3}{48} \frac{16\Delta}{(b-a)^3} \\ &\approx \frac{\Delta}{3}. \end{aligned} \tag{10}$$

[2]Recall that in (9) we assumed that $f''(\eta) \approx f''(\xi)$.

The error can thus be approximated by computing one-third the difference between approximations when halving the length of the subintervals. This method will provide an approximate error when no known solution is available to measure our approximation against, and details of implementation are provided in section 1.4.

1.4. **Implementation in Python.** In order to implement our method in Python, we will first need to import the following modules: NumPy, NumPy Linear Algebra, SciPy Interpolation, and Matplotlib PyPlot. We will also import time so that we can record the time it takes to solve each problem.

```
from time import time
import numpy as np
import numpy.linalg as nla
import scipy.interpolate as intrp
import matplotlib.pyplot as plt
```

build_D(). We will define several smaller functions that are later wrapped in a main solving function. Our first function takes the number of quadrature points (N) as as argument and builds an $N \times N$ matrix containing the weights corresponding to the Newton Cotes Trapezoidal Method. This is accomplished by initializing a vector whose components are all 1 with length N. Next, we will replace the first and last components with $\frac{1}{2}$ before using np.diag() to place the values on the diagonal of an $N \times N$ matrix. The return of this function is our matrix $\mathbf{D}$ as seen in section 1.2.

```
def build_D(N):
    D=np.ones(N) # A 1-D array of length N containing ones
    D[0]=.5 # replacing the first component with .5
    D[N-1]=.5 # replacing the last component with .5
    D=np.diag(D) # build diagonal matrix whose entries are the
   components of the 1-D array D
    return D
```

build_kern(). The next function will build a matrix of kernel evaluations for the given function $K(x, y)$. This function will take the number of quadrature points (N), the kernel function (K), and two vectors of quadrature points (x,y) as arguments. The function will initialize an $N \times N$ matrix containing all zeros, before assigning to each entry an evaluation of the kernel function at the corresponding quadrature points. For example, the entry $M_{3,4}$ is the value of the kernel function evaluated at $K(x_3, x_4)$ as discussed in section 1.2. The return of this function is the kernel matrix $\mathbf{M}$.

```
def build_kern(N, K, x, y):
    M=np.zeros((N,N)) # initializes an NxN matrix of zeroes
    for i in range(N): # loops through each value of x_i in x
        for j in range(N): # loops through each value of y_j in y
   while x_i is held constant
            M[i,j]=(K(x[i],y[j]))  # each entry in the matrix M_ij
    is replaced with K(x_i, y_j)
    return M
```

solve_fredholm(). We are now ready to define our main solving function. This function takes the following arguments: a is the lower bound of integration; b is the upper bound of integration; N is the number of quadrature points; K is the kernel function (passed as a lambda function); g is the known function (passed as a lambda function); and, l is the value of the real-valued scalar λ. This function will create the left-hand side of the matrix equation

$$\mathbf{I} - h\lambda\mathbf{KD}$$

by building the diagonal matrix $\mathbf{D}$ of quadrature weights and the kernel matrix $\mathbf{M}$ of kernel evaluations; multiplying these two matrices together and scaling their product by $h\lambda$ before subtracting this from an $N \times N$ identity matrix. It will then build the right-hand side, $\vec{\Phi}_0$, by evaluating the known function $\phi(x_i)$ at each quadrature node x_i. Then, the function will solve the system of linear of equations for the values of the unknown function, $\vec{\Phi}$, at each quadrature node. Finally, it will interpolate the data $\vec{x}$ and $\vec{\Phi}$ by fitting a cubic spline to these points. The return of this function is the interpolant f which is our approximation of the unknown solution $\phi(x)$.

```
def solve_fredholm(a,b,N,K,g,l):
    # Builds linear system, solves it, and then interpolates
    D=build_D(N) # builds diagonal matrix of newton-cotes
   trapezoidal weights
    x=np.linspace(a,b,N) # quadrature nodes
    y=np.linspace(a,b,N) # quadrature nodes
    M=build_kern(N,K,x,y) # Builds Kernel matrix
    h=(b-a)/(N-1) # length of one sub interval
    L=(np.eye(N)-l*h*(np.dot(M,D))) # builds left hand side
    G=g(x) # builds right hand side
    F=nla.solve(L,G) #solves for unknown solution
    f=intrp.InterpolatedUnivariateSpline(x, F, k=3) # interpolates
    return f
```

my_bad(). The error is calculated by the method described in section 1.3. This function takes two interpolants, f1 and f2, as arguments; as well as the upper and lower bounds of integration, a and b. It creates a vector of 40000 quadrature nodes to pass to both interpolants and returns the absolute value of the maximum difference divided by 3 as the error.

```
def my_bad(f1, f2, a, b):
    # calculates the error
    x=np.linspace(a,b,40000) # points to be passed to interpolants
    delta=(f2(x)-f1(x)) # computes the difference between the
   interpolants
    err=np.max(delta/3) # finds the maximum difference between
   them and divides by 3
    return np.abs(err)
```

make_pretty(). An optional function to plot the approximate solution is also defined. This function takes the upper and lower bounds of integration, a and b; the number of quadrature points, N; and the approximation of the unknown solution, f2.

```
def make_pretty(a,b,N,f2):
    # plots the solution
    x=np.linspace(a,b,10) # discrete points to plot
    xx=np.linspace(a,b,N1) # points for graphing the curve
    plt.plot(x,f2(x), 'o', xx, f2(xx)) # setting up the functions
   and points to plot
    #plt.title(r"$f(x)=e^{x}-\int_{-1}^{1}xe^{y(1-x)}f(y)dy$, $n
   =500$") #optional title
    plt.legend(['Discrete Points', 'Approximate Solution']) # plot
    legend information
    plt.xlabel(r'$x$') # label x-axis
    plt.ylabel(r'$y$') # label y-axis
    plt.show() # show plot
    return print("Plotted it!")
```

After creating these functions, solving a Fredholm Integral Equation of the Second Kind is reduced to only the following lines of codes (not including the declarations of the kernel function, bounds of integration, etc.).

```
N1=6 # Number of quadrature points for first approximation
N2=2*N1 # Double the quadrature points for the second
   approximation
f1=solve_fredholm(a,b,N1,K,g,l) # Solve for first approximation
f2=solve_fredholm(a,b,N2,K,g,l) # solve for second approximation
err=my_bad(f1,f2,a,b) # calculate the error
print("The error is",err)
make_pretty(a,b,N1,f2) # Plot the approximate solution and
   discrete data points
```

Full source code is presented in sections (2.1-4) as well as plots of approximate solutions and tables of error estimations.

2. The Problems

We will now apply our method to 4 problems for which we do not know the exact analytical solution. Source code, plots of the approximate solution and estimates of the errors are presented for each of the four following Fredholm Integral Equations of the Second Kind

$$\Psi(x) = \Psi_0(x) + \lambda \int_a^b K(x,y)\Psi(y)dy$$

for $a \leq x \leq b$, where $\Psi(x)$ is the unknown solution, $\Psi_0(x)$ is a given function, $K(x,y)$ is the given function called the kernel, and a, b, and λ are constants.

	(a,b)	λ	$\Psi_0(x)$	$K(x,y)$
Problem #1	$(-1,1)$	-1	e^x	$xe^{y(1-x)}$
Problem #2	$(0,\pi)$	-1	$\sin(10x)$	$\sin(x+y)$
Problem #3	$(0,\pi)$	-1	$1+\sin(\pi x)$	$x\cos(xy)$
Problem #4	$(-1,1)$	45/8	$((x+2)(2x-1))/2$	xy^2

2.1. **Problem 1.**

$$\Psi(x) = e^x - \int_{-1}^{1} x e^{y(1-x)} \Psi(y) dy$$

.

```
        #--Problem 1 --#
#-----------------------------------#
# Given functions, parameters, etc. #
#-----------------------------------#
K=lambda x,y: x*np.exp(y*(1-x)) #kernel function
g=lambda x: np.exp(x) # forcing function
l = -1 # lambda
a, b = -1,1 # Upper and lower bounds of integration


print('Solving Problem #1...')
start_time = time()

#-------------------------------------------#
# Solve Fredholm Equation of the Second Kind #
#-------------------------------------------#
N1=500
N2=2*N1
f1=solve_fredholm(a,b,N1,K,g,l)
f2=solve_fredholm(a,b,N2,K,g,l)
time_elapsed = time() - start_time
print(("-->Solved in {:1.2f} seconds!").format(time_elapsed))
print("Calculating the error when n=", N1, " and ", N2,"...")
err=my_bad(f1,f2,a,b)
print("The error is",err)

make_pretty(a,b,N1,f2)
```

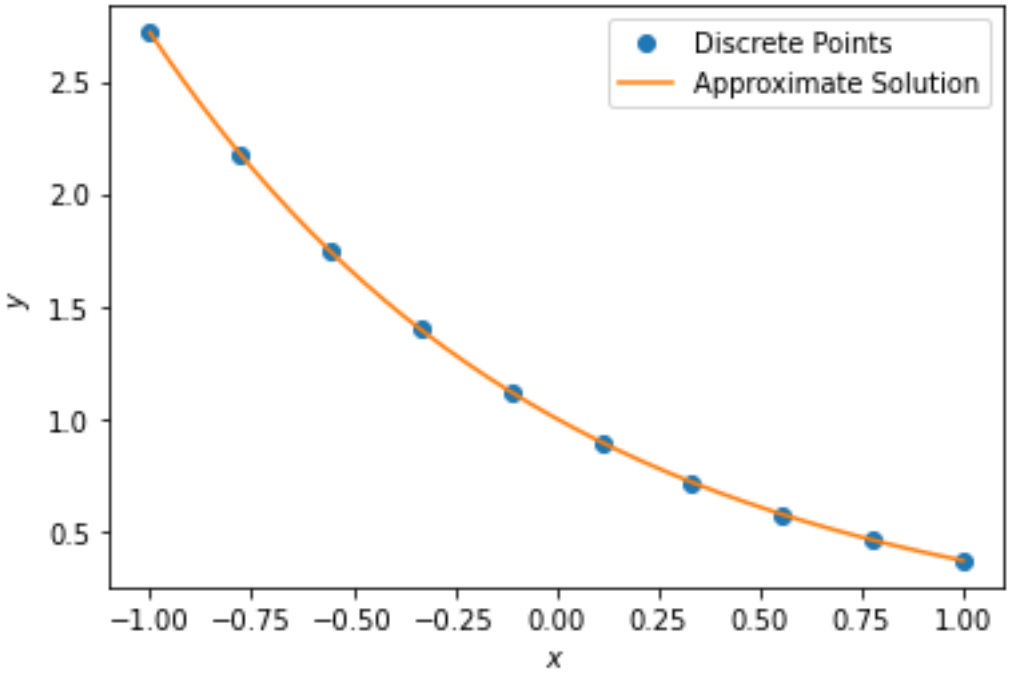

FIGURE 1. $n = 1000$,error=6.335152e-07

n	Error Approximation
20	0.001929
100	6.600111e-05
1000	6.335152e-07
3000	7.017255e-08
7000	1.287739e-08

2.2. **Problem 2.**

$$\Psi(x) = \sin(10x) - \int_0^{\pi} \sin(x+y)\Psi(y)dy$$

.

```
        #--Problem 2 --#
#-----------------------------------#
# Given functions, parameters, etc. #
#-----------------------------------#
K=lambda x,y: np.sin(x+y) #kernel function
g=lambda x: np.sin(10*x) # forcing function
l = -1 # lambda
a, b = 0,np.pi # Upper and lower bounds of integration


print('Solving Problem #2...')
start_time = time()

#-------------------------------------------#
# Solve Fredholm Equation of the Second Kind #
#-------------------------------------------#
N1=2500
N2=2*N1
f1=solve_fredholm(a,b,N1,K,g,l)
f2=solve_fredholm(a,b,N2,K,g,l)
time_elapsed = time() - start_time
print(("-->Solved in {:1.2f} seconds!").format(time_elapsed))
print("Calculating the error when n=", N1, " and ", N2,"...")
err=my_bad(f1,f2,a,b)
print("The error is",err)

make_pretty(a,b,N1,f2)
```

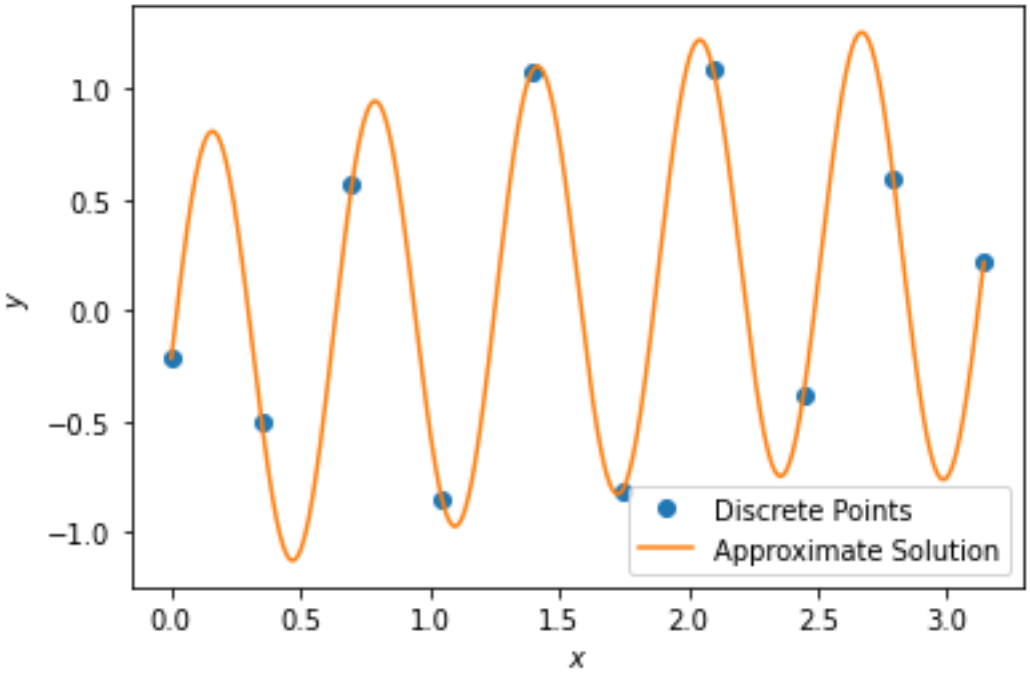

FIGURE 2. $n = 5000$, error=8.357471e-07

n	Error Approximation
20	0.543004
100	0.002852
1000	2.098128e-05
3000	2.323040e-06
7000	4.262841e-07

2.3. **Problem 3.**

$$\Psi(x) = 1 + \sin(\pi x) - \int_0^{\pi} x\cos(xy)\Psi(y)dy$$

.

```
        #--Problem 3 --#
#-----------------------------------#
# Given functions, parameters, etc. #
#-----------------------------------#
K=lambda x,y: x*np.cos(x*y) #kernel function
g=lambda x: 1+np.sin(np.pi*x) # forcing function
l = -1 # lambda
a, b = 0,np.pi # Upper and lower bounds of integration


print('Solving Problem #3...')
start_time = time()

#-------------------------------------------#
# Solve Fredholm Equation of the Second Kind #
#-------------------------------------------#
N1=1500
N2=2*N1
f1=solve_fredholm(a,b,N1,K,g,l)
f2=solve_fredholm(a,b,N2,K,g,l)
time_elapsed = time() - start_time
print(("-->Solved in {:1.2f} seconds!").format(time_elapsed))
print("Calculating the error when n=", N1, " and ", N2,"...")
err=my_bad(f1,f2,a,b)
print("The error is",err)

make_pretty(a,b,N1,f2)
```

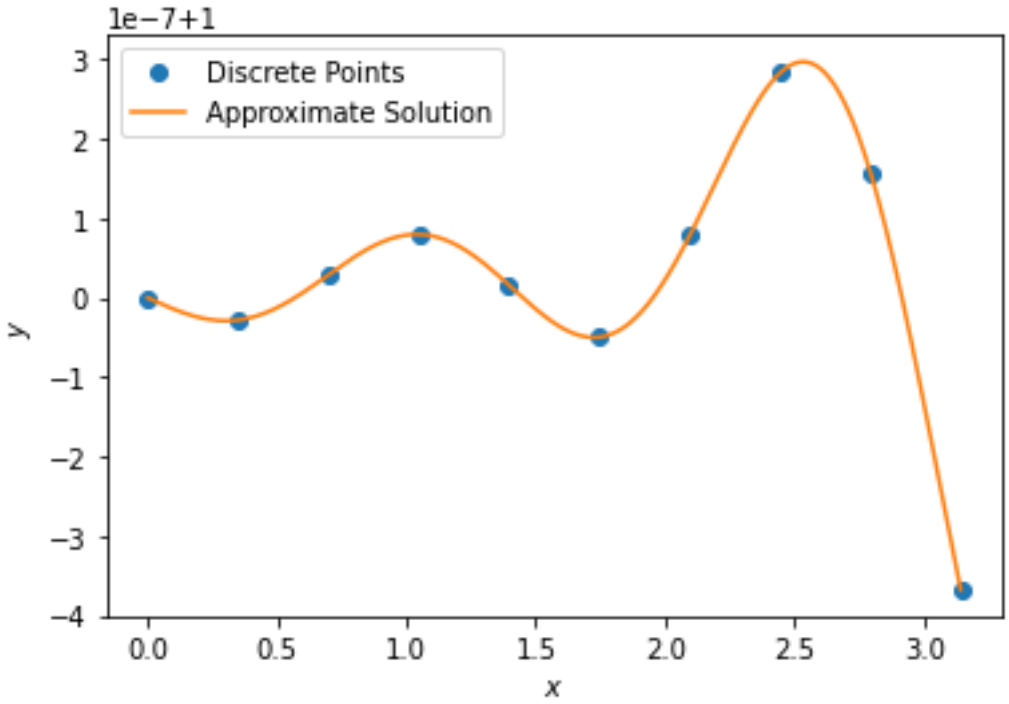

FIGURE 3. $n = 3000$, error=3.679669e-07

n	Error Approximation
20	0.005673
100	0.000341
1000	3.321581e-06
3000	3.679669e-07
7000	6.752662e-08

2.4. **Problem 4.**

$$\Psi(x) = \frac{(x+2)(2x-1)}{2} + \frac{45}{8}\int_{-1}^{1} xy^2\Psi(y)dy$$

.

```
        #--Problem 4 --#
#-----------------------------------#
# Given functions, parameters, etc. #
#-----------------------------------#
K=lambda x,y: x*(y**2) #kernel function
g=lambda x: ((x+2)*((2*x)-1))/2 # forcing function
l = 45/8 # lambda
a, b = -1,1 # Upper and lower bounds of integration


print('Solving Problem #4...')
start_time = time()

#-------------------------------------------#
# Solve Fredholm Equation of the Second Kind #
#-------------------------------------------#
N1=1500
N2=2*N1
f1=solve_fredholm(a,b,N1,K,g,l)
f2=solve_fredholm(a,b,N2,K,g,l)
time_elapsed = time() - start_time
print(("-->Solved in {:1.2f} seconds!").format(time_elapsed))
print("Calculating the error when n=", N1, " and ", N2,"...")
err=my_bad(f1,f2,a,b)
print("The error is",err)

make_pretty(a,b,N1,f2)
```

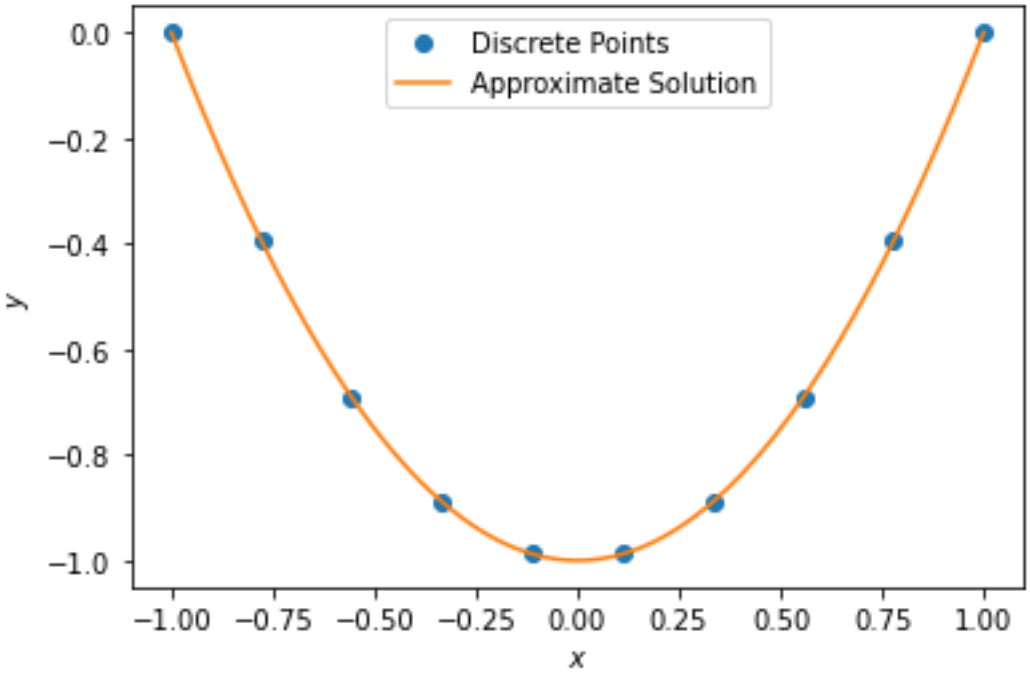

FIGURE 4. $n = 3000$, error=8.346306e-07

n	Error Approximation
20	0.023650
00	0.000786
1000	7.535083e-06
3000	8.346306e-07
7000	1.531633e-07

3. Conclusion

The numerical method developed in this paper solves a Fredholm Integral Equation of the Second Kind by leveraging principles of finite-dimensional linear algebra in order to solve a system of equations. Solving this system yields a vector which has as its components the values of an unknown function at each quadrature node. Interpolation methods can then be used to find the approximate solution to 6 decimal places of accuracy. More work needs to be done to assess whether there are limitations to the types of kernels this method is effective for–as well as whether or not this method can be effectively applied to Volterra integral equation.