Informal Lab Report 7

Elliptic Example: 2D Laplacian

We want to model the electric potential for an empty 2D box, 10cm x 10cm in size, where the top wall is held at V = 1.0V and the other walls at 0V.

$$0 = \nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2},$$
$$\phi(y = 10) = 1.0V$$
$$\phi(y = 0) = \phi(x = 0) = \phi(x = 10) = 0$$

Exercise 1

Setup up the problem:

- discretize space in x and y, using an MxM grid
- implement the boundary conditions

Then use Jacobi Relaxation to solve it, with target accuracy 1e-04 and M=10. Print the number of iterations required to reach the target accuracy.

You can consult the textbook's `laplace.py` for help.

[19]:
```
#THE WHILE LOOP IS THE JACOBI METHOD ESSENTIALLY USEFUL!

import numpy as np
import matplotlib.pyplot as plt

target = 10**-4
M = 10
Vtop = 1

grid = np.zeros((M + 1, M + 1), float)
grid[0, :] = Vtop
gridprime = np.empty((M + 1, M + 1), float)

delta = 1
iterations = 0
while delta > target :
    iterations += 1
    for i in range(M +1):
        for j in range(M+1):
            if i == 0 or j ==0 or i == M or j ==M:
                gridprime[i,j] = grid[i, j]
            else:
                gridprime[i,j] = (grid[i+1, j] + grid[i-1, j] + grid[i, j+ 1] + grid[i, j-1])/4
    delta = np.max(abs(grid - gridprime))
    grid, gridprime = gridprime, grid

print(iterations)
```

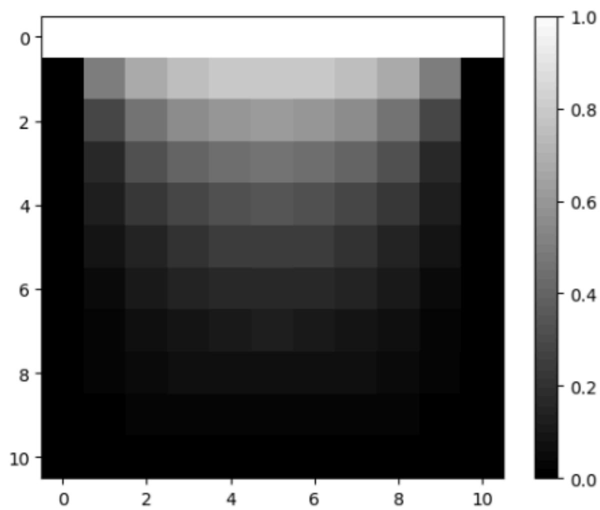
```
106
```

Exercise 2

Plot the solution (you can use matplotlib.pyplot.imshow)

In [20]:
```
plt.imshow(grid)
plt.colorbar()
```

Out[20]: <matplotlib.colorbar.Colorbar at 0x7fd0f1ab4650>

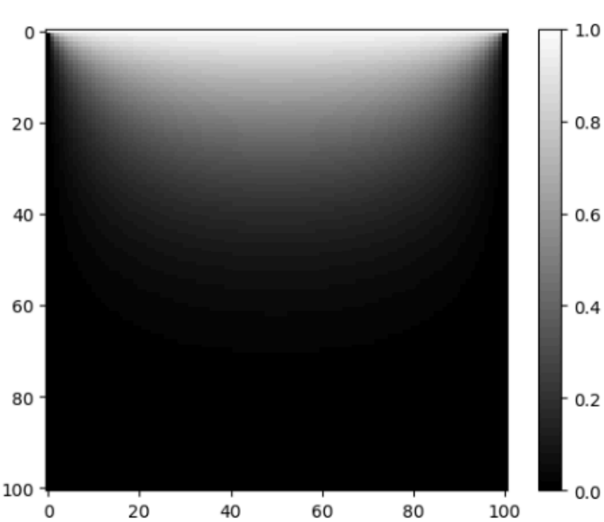


Exercise 3

Now repeat Exercises 1 and 2 with M=100. Do you notice a difference in runtime?

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt

target = 10**-4
M = 100
Vtop = 1

grid = np.zeros((M + 1, M + 1), float)
grid[0, :] = Vtop
gridprime = np.empty((M + 1, M + 1), float)

delta = 1
iterations = 0
while delta > target :
    iterations += 1
    for i in range(M +1):
        for j in range(M+1):
            if i == 0 or j ==0 or i == M or j ==M:
                gridprime[i,j] = grid[i, j]
            else:
                gridprime[i,j] = (grid[i+1, j] + grid[i-1, j] + grid[i, j+ 1] + grid[i, j-1])/4
    delta = np.max(abs(grid - gridprime))
    grid, gridprime = gridprime, grid

plt.imshow(grid)
plt.colorbar()
print(iterations)
```

1909



Hyperbolic Example: Wave Equation

Recall the 1D wave equation:

$$\frac{\partial^2 \phi}{\partial t^2} = v^2 \frac{\partial^2 \phi}{\partial x^2}$$

Consider a piano string of length L, initially at rest. At time $t = 0$ the string is struck by the piano hammer a distance d from the end of from the string. The string vibrates as a result of being struck, except at the ends, $x = 0$, and $x = L$, where it is held fixed.

Consider the case $v = 100\text{ms}^{-1}$, with the initial condition that $\phi(x) = 0$ everywhere but the velocity $\psi(x)$ is nonzero, with profile

$$\psi(x) = C\frac{x(L-x)}{L^2}\exp\left[-\frac{(x-d)^2}{2\sigma^2}\right],$$

where $L = 1$m, $d = 10$cm, $C = 1\text{ms}^{-1}$, and $\sigma = 0.3$m.

Exercise 4

Solve using the FTCS method, with grid spacing (in x) $a = 5$ mm, from times 0 to 0.1s using time--step $h = 10^{-6}$ s. Make a plot of ϕ vs x over the entire length of string, at each of the following times:

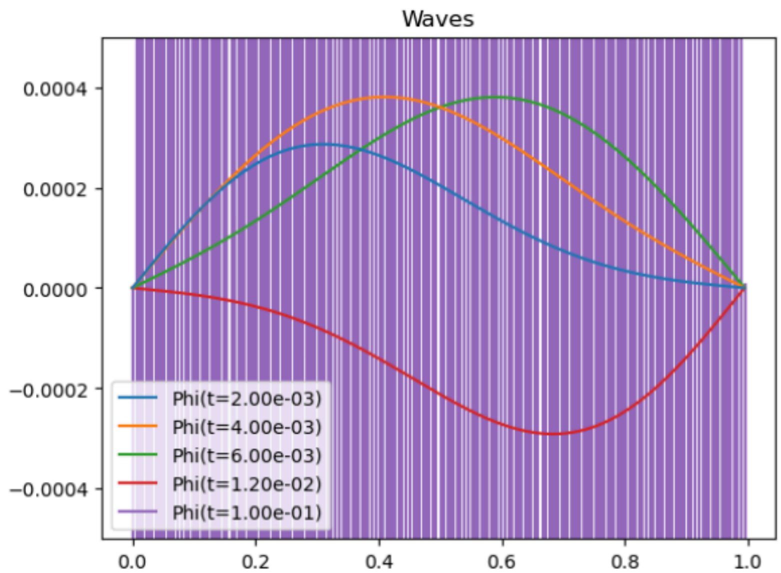
- 0.006 s
- 0.004 s
- 0.002 s
- 0.012 s
- 0.100 s

You'll see your first 4 plots look good, then the instability of the solution shows up!

In [5]:
```
velocity = 100
phi1 = 0
L = 1
d = 0.1
C = 1
sigma = 0.3
h = 10**-6
a = 0.005
M = L/a
SnapshotTimes = np.array([2e-3, 4e-3, 6e-3, 12e-3, 100e-3])

def setup_grid():
    x = np.arange(0, L,a)
    psi = C*x*(L-x)*np.exp(-(x-d)**2/(2*sigma**2))/L**2
    phi = np.zeros_like(x)
    return x, phi, psi


def f(y, alpha):
    N = len(y) - 1
    res = np.empty(N+1, float)
    res[1:N] = (y[0:N-1] +y[2:N+1] - 2*y[1:N]) * alpha
    res[0] = res[N] = 0.0
    return res

def  integrate_wave_ftcs(positions, phi, psi, SnapshotTimes, h = h, v = velocity):
    alpha = v**2/a**2

    scounter=0
    results=[]
    times = np.arange(0, 100e-3+h, h)

    for t in times:
        if t >= SnapshotTimes[scounter]:
            results.append((t, phi, psi))
            scounter+= 1
        phi, psi = phi+h*psi, psi+h*f(phi, alpha)
    return positions, results
```

In [6]:
```python
def plot_results(positions, values, titles):
    LegendLabel = []
    for c,t in enumerate(values):
        time, phi , psi = t
        plt.plot(positions,phi,zorder=-c)
        plt.ylim((-5e-4, 5e-4))
        LegendLabel.append('Phi(t=%3.2e)'%time)
    plt.legend(LegendLabel)
    plt.title('Waves')
```

In [7]:
```python
init_conditions = setup_grid()
ftcs = integrate_wave_ftcs(*init_conditions, SnapshotTimes)
plot_results(*ftcs, 'Integration with FTCS')
```



Exercise 5

Repeat the previous exercise using the Crank--Nicolson method. Use a larger time--step, $h = 10^{-4}$ s.

You'll see the solution is stable. It dies out to 0 at about 0.1 s, but this is how the physical system is supposed to behave!

the CN method involves a set of simultaneous equations, one for each grid point. We can solve using the methods for linear systems in Chapter 6 -- in particular, banded matrix. The following snippets of code will help you define the matrix, and the vector to use on the right-hand side, of the CK equations.

In [8]:

```
def matrix(N,alpha):
    """Banded matrix for the Crank-Nicolson
    Args:
        N : number of elements
        alpha = 2*h*v**2/a**2 , h: timestep length, a: spatial grid spacing, v: wave speed"""
    bands = np.zeros((3,N+2))
    bands[0,:-2] = -alpha
    bands[2,1:-1] = -alpha
    bands[1,:] = 1+2*alpha
    return bands

def banded(Aa,va,up=1,down=1):
# from textbook online resources, to solve Ax = v
# Aa is banded matrix A, va is vector v, up and down give band positions in matrix

    # Copy the inputs and determine the size of the system
    A = np.copy(Aa)
    v = np.copy(va)
    N = len(v)

    # Gaussian elimination
    for m in range(N):

        # Normalization factor
        div = A[up,m]

        # Update the vector first
        v[m] /= div
        for k in range(1,down+1):
            if m+k<N:
                v[m+k] -= A[up+k,m]*v[m]

        # Now normalize the pivot row of A and subtract from lower ones
        for i in range(up):
            j = m + up - i
            if j<N:
                A[i,j] /= div
                for k in range(1,down+1):
                    A[i+k,j] -= A[up+k,m]*A[i,j]

    # Backsubstitution
    for m in range(N-2,-1,-1):
        for i in range(up):
            j = m + up - i
            if j<N:
                v[m] -= A[i,j]*v[j]

    return v
```

In [9]:

```
def rhs(phi, psi, alpha,h):
    """Solve the Right hand side of the Crank-Nicolson algorithm.
    Args:
        phi, psi : position and velocity
        alpha = 2*h*v**2/a**2,
        h=dt, timestep

    Returns:
    the column vector for the right hand side.
    """
    r = np.zeros_like(phi)
    r[1:-1] = (h*psi[1:-1] +
               alpha * phi[:-2] +
               (1-2*alpha)*phi[1:-1] +
               alpha * phi[2:])
    return r
```

In [10]:
```
def integrate_waves_cn(positions, phi, psi, SnapshotTimes, h=1e-4):
    T = 0.1
    times = np.arange(0, T+h,h)
    alpha = h**2*velocity**2/(4*a**2)
    A_mat = matrix(len(phi), alpha)

    scounter =0
    results = []
    for t in (times):
        if t>= SnapshotTimes[scounter]:
            results.append((t,phi,psi))
            scounter +=1
        r = rhs(phi,psi,alpha,h)
        phiN=banded(A_mat,r)
        psiN = (2/h)*(phiN-phi)-psi
        psi,phi = psiN.copy(), phiN.copy()
    return positions, results
```

In [11]:
```
init_conditions = setup_grid()
cn = integrate_waves_cn(*init_conditions, SnapshotTimes)
plot_results(*cn, 'Integrations with CN')
plt.grid()
```