HW_PDE

September 30, 2024

1 Homework 2

James Larsen

PHYSICS 514 – Emmanuel Gull

[81]:
```
import numpy as np
from matplotlib import pyplot as plt
from time import time
```

1.1 Finite Difference

Discretizing the Poisson equation in two space dimensions yields the following system of linear equations:

$$\Phi(x_{n+1}, y_m) + \Phi(x_{n-1}, y_m) + \Phi(x_n, y_{m+1}) + \Phi(x_n, y_{m-1}) - 4\Phi(x_n, y_m) = -4\pi\rho(x_n, y_m)\Delta x^2. \quad (1)$$

Manipulating this equation gives us the following iterative method:

$$\Phi_{k+1}(x_n, y_m) \leftarrow \frac{1}{4}\left[\Phi(x_{n+1}, y_m) + \Phi(x_{n-1}, y_m) + \Phi(x_n, y_{m+1}) + \Phi(x_n, y_{m-1}) + 4\pi\rho(x_n, y_m)\Delta x^2\right]. \quad (2)$$

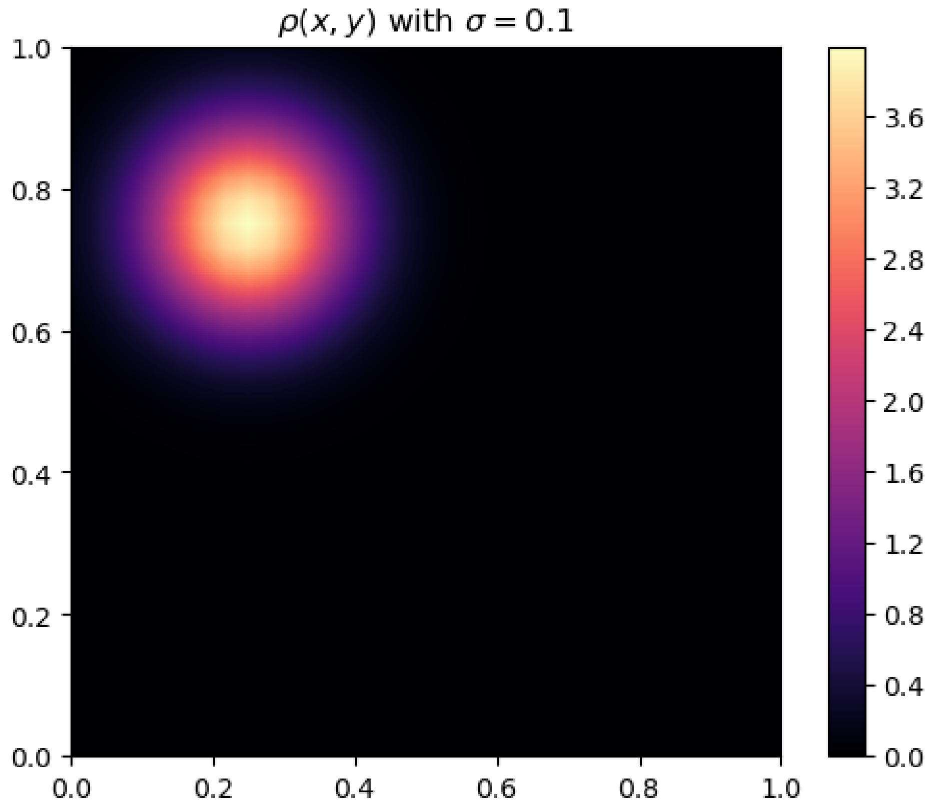
[139]:
```
def finite_diff(dx, Phi_j, tol,  s):
    '''
    Solve Poisson equation using finite difference iteration

    dx:     discretization step size
    Phi_j:  initial guess
    tol:    error tolerance
     s:     charge density

    returns solution (Phi_j) and number of iterations to solution (j)
    '''
    # iterate until converged
      = 1
    j = 0
    while   > tol:
        Phi_next = np.copy(Phi_j)
        for n in range(1, Phi_j.shape[0]-1):
            for m in range(1, Phi_j.shape[1]-1):
```

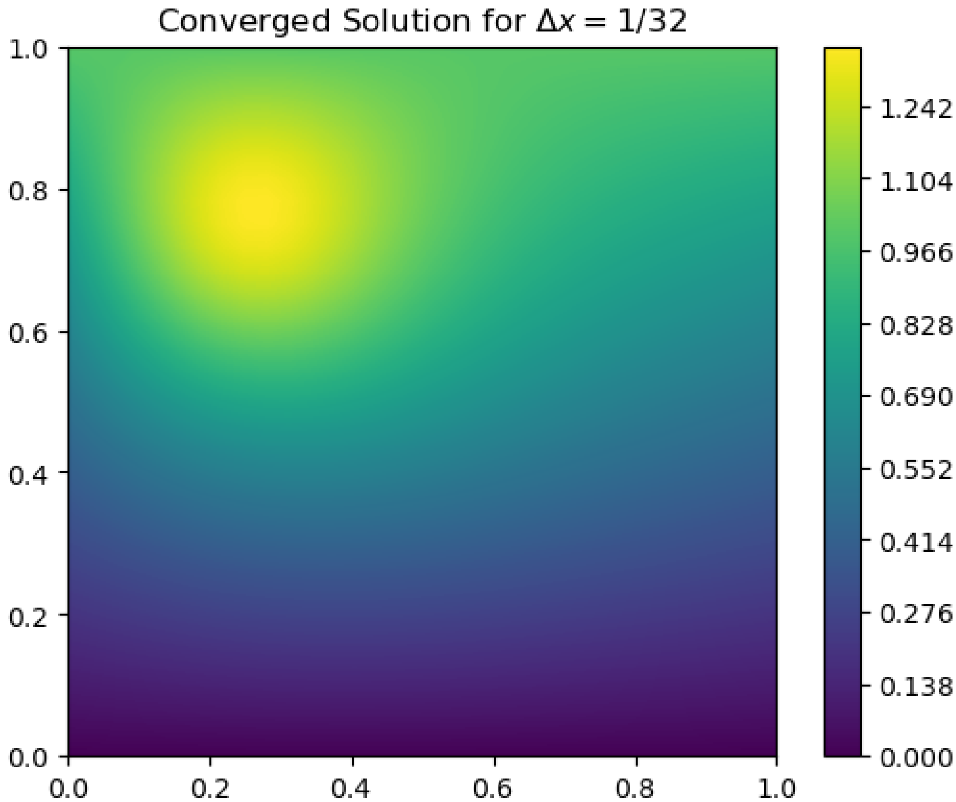
```
                Phi_next[n,m] = (Phi_j[n+1,m] + Phi_j[n-1,m] + Phi_j[n,m+1] +␣
↪Phi_j[n,m-1] + 4*np.pi* s[n,m]*dx**2) / 4
          = np.linalg.norm(Phi_next - Phi_j)
        Phi_j = Phi_next
        j += 1

    # return desired values
    return Phi_j, j
```

[149]:
```
def iteration( ):
    '''
    repeat iterative method for specified value of
    '''
    # initialize global values
    ks = np.arange(2,6)
    dxs = 1 / (2**ks)
    numiters_fd = np.zeros(len(dxs))
    times_fd = np.zeros(len(dxs))

    # loop over discretizations
    for i, dx in enumerate(dxs):
        x = np.arange(0, 1+dx, dx)
        y = np.arange(0, 1+dx, dx)
        xs, ys = np.meshgrid(x, y, sparse=True)
         s = np.exp(-( (xs-0.25)**2 + (ys-0.75)**2 ) / (2* **2)) / (  * np.
↪sqrt(2*np.pi))
        tol = 1e-8

        # initialize Phi to satisfy boundary conditions
        Phi_j = np.zeros( s.shape)
        Phi_j[:,0] = x
        Phi_j[:,-1] = x
        Phi_j[-1] = np.ones(len(x))

        # solve via finite difference iteration
        start = time()
        Phi_j, numiters_fd[i] = finite_diff(dx, Phi_j, tol,  s)
        end = time()
        times_fd[i] = end-start

    # return values
    return  s, Phi_j, numiters_fd, times_fd
```

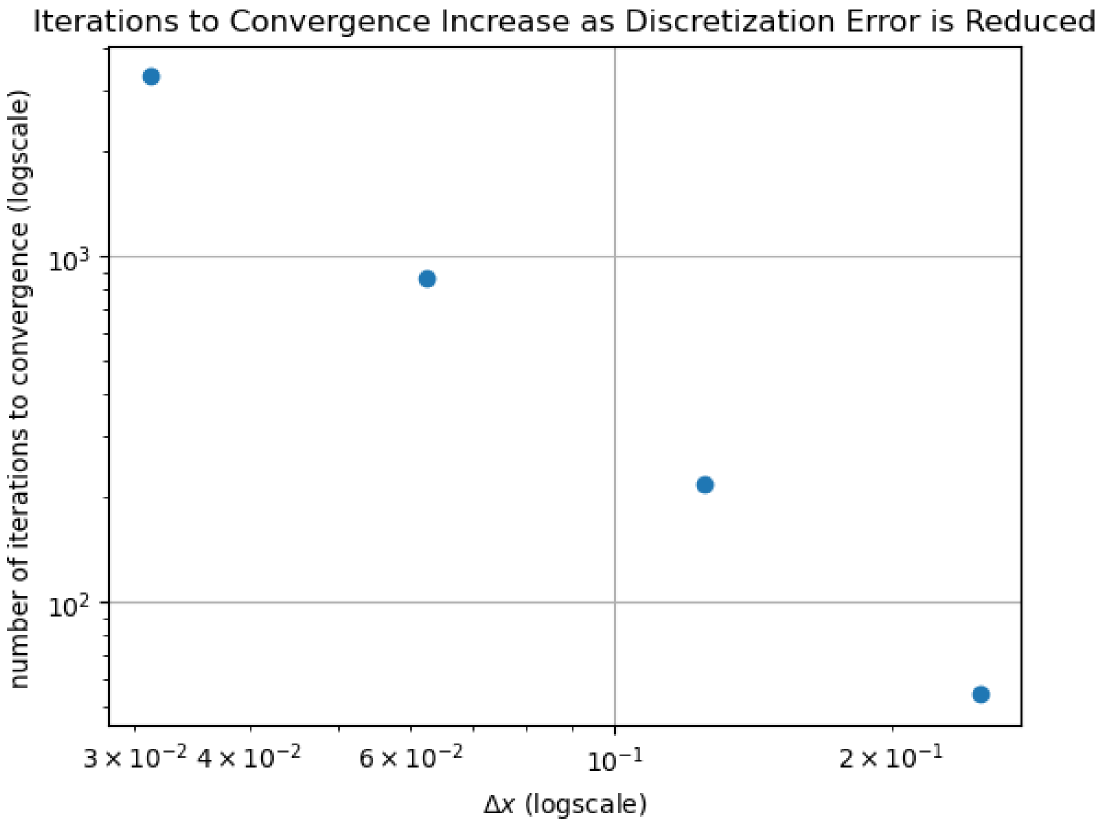
[144]:
```
# get values
 s, Phi_j, numiters_fd, times_fd = iteration(0.1)

# plot result
```

```
plt.contourf(x, y,  s, levels=500, cmap="magma")
plt.axis('scaled')
plt.title(r"$\rho(x,y)$ with $\sigma = 0.1$")
plt.colorbar()
plt.show()

# plot result
plt.contourf(x, y, Phi_j, levels=500)
plt.axis('scaled')
plt.title(r"Converged Solution for $\Delta x = 1/32$")
plt.colorbar()
plt.show()

# plot number of iterations to convergence
plt.scatter(dxs, numiters_fd)
plt.xscale("log")
plt.xlabel(r"$\Delta x$ (logscale)")
plt.ylabel("number of iterations to convergence (logscale)")
plt.yscale("log")
plt.title("Iterations to Convergence Increase as Discretization Error is Reduced")
plt.grid()
plt.show()

# plot time to convergence
plt.scatter(dxs, times_fd)
plt.xscale("log")
plt.xlabel(r"$\Delta x$ (logscale)")
plt.ylabel("time to convergence (secs, logscale)")
plt.yscale("log")
plt.title("Time to Convergence Increases as Discretization Error is Reduced")
plt.grid()
plt.show()
```

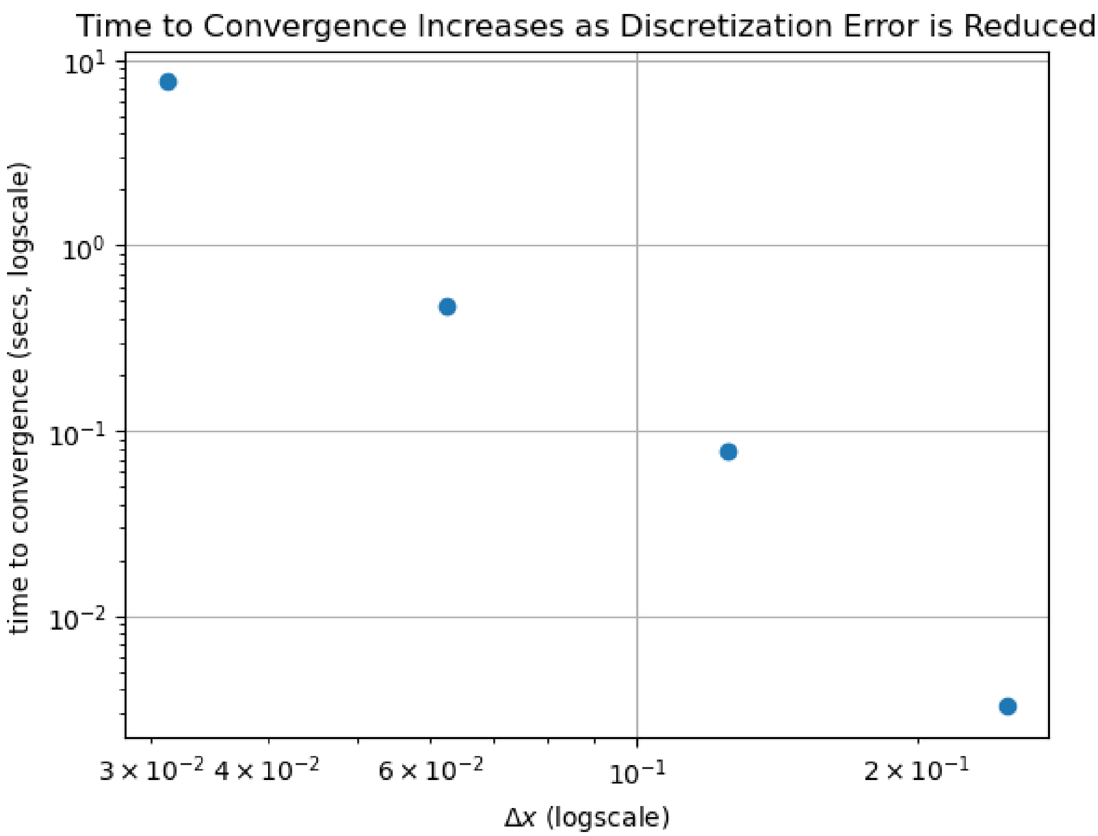

$\rho(x, y)$ with $\sigma = 0.1$
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
3.6
3.2
2.8
2.4
2.0
1.6
1.2
0.8
0.4
0.0



Converged Solution for $\Delta x = 1/32$
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.242
1.104
0.966
0.828
0.690
0.552
0.414
0.276
0.138
0.000



Iterations to Convergence Increase as Discretization Error is Reduced
number of iterations to convergence (logscale)
10^3
10^2
3×10^{-2}
4×10^{-2}
6×10^{-2}
10^{-1}
2×10^{-1}
Δx (logscale)



1.2 Over-relaxation

To speed up convergence, we can use multi-grid methods. The general method is given by the following steps:

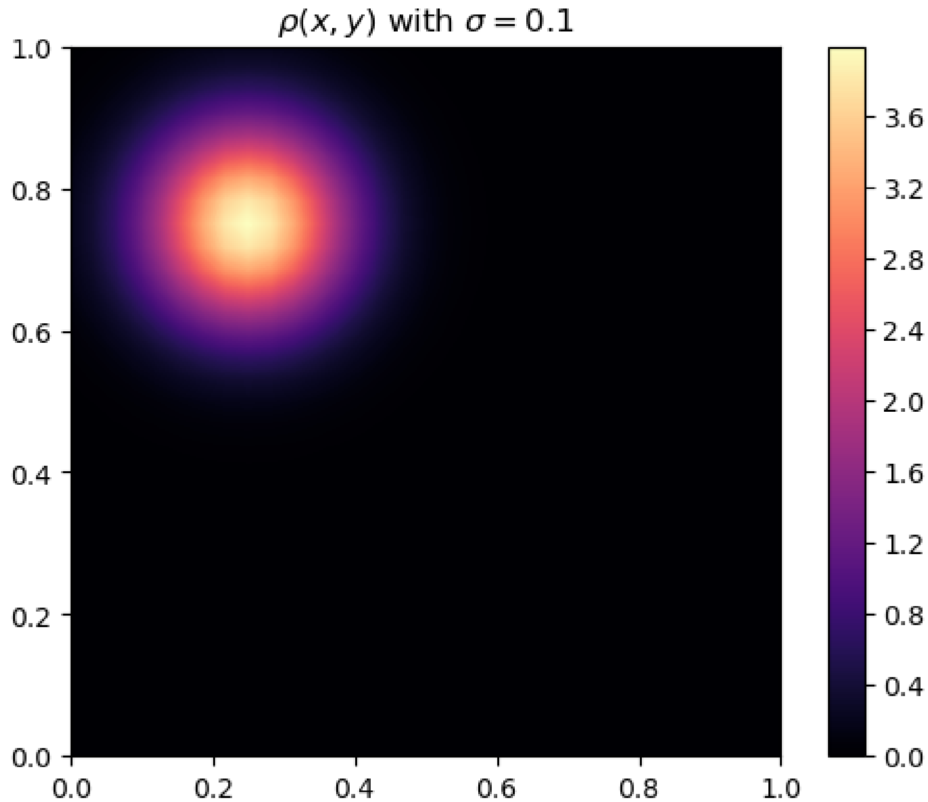
1. Solve the Poisson equation on a grid with spacing Δx.
2. Refine the grid $\Delta x \leftarrow \Delta x/2$.
3. Interpolate the potential at the new grid points.
4. Repeat until the desired final fine grid spacing is reached.

[147]:
```
def multigrid( ):
    '''
    repeat multi-grid method for specified value of
    '''
    # initialize global values
    ks = np.arange(2,6)
    dxs = 1 / (2**ks)
    numiters_mg = np.zeros(len(dxs))
    times_mg = np.zeros(len(dxs))
    tol = 1e-8
```

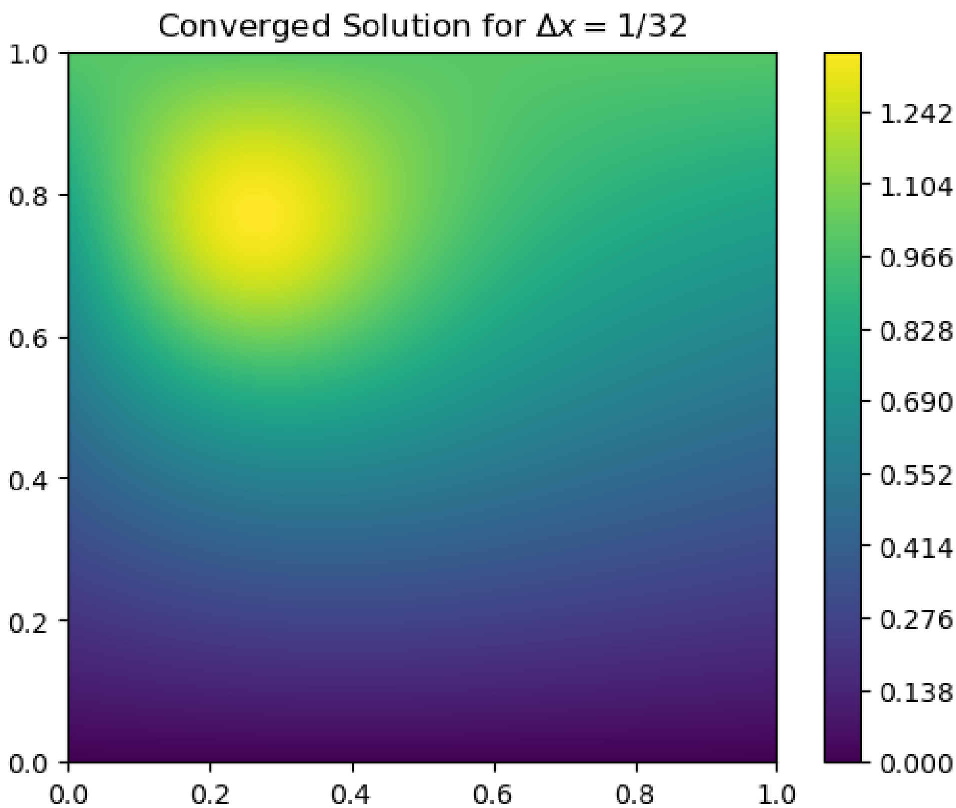
```
    # loop over discretizations
    for i, dx in enumerate(dxs):

        # initialize very coarse mesh
        dx_curr = 1 / 2

        # loop until desired grid spacing reached
        start = time()
        while dx_curr != dx:
            dx_curr /= 2

            # initialize grid and charge density
            x = np.arange(0, 1+dx_curr, dx_curr)
            y = np.arange(0, 1+dx_curr, dx_curr)
            xs, ys = np.meshgrid(x, y, sparse=True)
             s = np.exp(-( (xs-0.25)**2 + (ys-0.75)**2 ) / (2* **2)) / (  * np.
↪sqrt(2*np.pi))

            # initialize Phi to satisfy boundary conditions
            Phi_new = np.zeros( s.shape)
            Phi_new[:,0] = x
            Phi_new[:,-1] = x
            Phi_new[-1] = np.ones(len(x))

            # interpolate potential if not first iteration
            if dx_curr != 1/4:
                Phi_new[::2,::2] = Phi_j
                Phi_new[::2,1::2] = (Phi_j[:,:-1:] + Phi_j[:,1::]) / 2
                Phi_new[1::2,::2] = (Phi_j[:-1:,:] + Phi_j[1::,:]) / 2
                Phi_new[1::2, 1::2] = (Phi_new[1::2,:-1:2] + Phi_new[1::2,2::
↪2]) / 2

            # solve via finite difference iteration
            Phi_j, numiters_fd_curr = finite_diff(dx, Phi_new, tol,  s)
            numiters_mg[i] += numiters_fd_curr

        # add times
        end = time()
        times_mg[i] = end-start

    # return values
    return  s, Phi_j, numiters_mg, times_mg
```

[148]:
```
# get values
s, Phi_j, numiters_mg, times_mg = multigrid(0.1)

# plot result
```

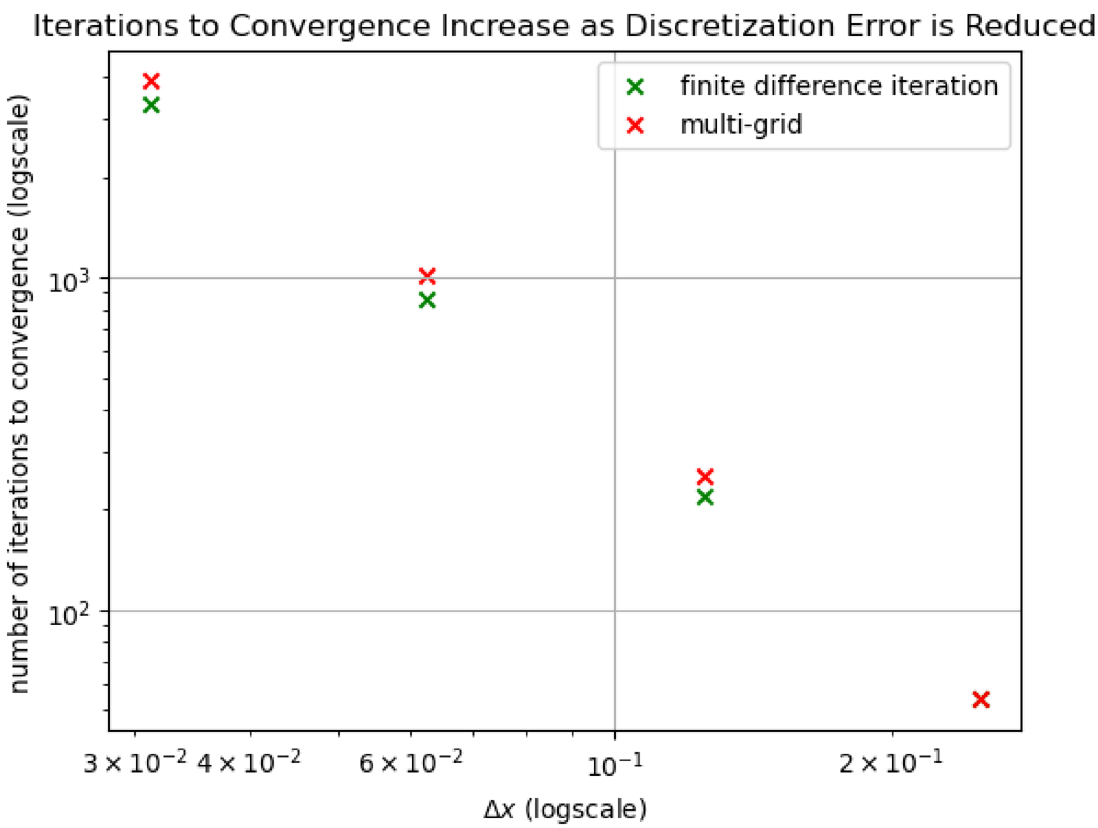
```python
plt.contourf(x, y,  s, levels=500, cmap="magma")
plt.axis('scaled')
plt.title(r"$\rho(x,y)$ with $\sigma = 0.1$")
plt.colorbar()
plt.show()

# plot result
plt.contourf(x, y, Phi_j, levels=500)
plt.axis('scaled')
plt.title(r"Converged Solution for $\Delta x = 1/32$")
plt.colorbar()
plt.show()

# plot number of iterations to convergence
plt.scatter(dxs, numiters_fd, marker="x", color="green", label="finite␣
 ↪difference iteration")
plt.scatter(dxs, numiters_mg, marker="x", color="red", label="multi-grid")
plt.xscale("log")
plt.xlabel(r"$\Delta x$ (logscale)")
plt.ylabel("number of iterations to convergence (logscale)")
plt.yscale("log")
plt.title("Iterations to Convergence Increase as Discretization Error is␣
 ↪Reduced")
plt.grid()
plt.legend(loc="best")
plt.show()

# plot time to convergence
plt.scatter(dxs, times_fd, marker="x", color="green", label="finite difference␣
 ↪iteration")
plt.scatter(dxs, times_mg, marker="x", color="red", label="multi-grid")
plt.xscale("log")
plt.xlabel(r"$\Delta x$ (logscale)")
plt.ylabel("time to convergence (secs, logscale)")
plt.yscale("log")
plt.title("Time to Convergence Increases as Discretization Error is Reduced")
plt.grid()
plt.legend(loc="best")
plt.show()
```

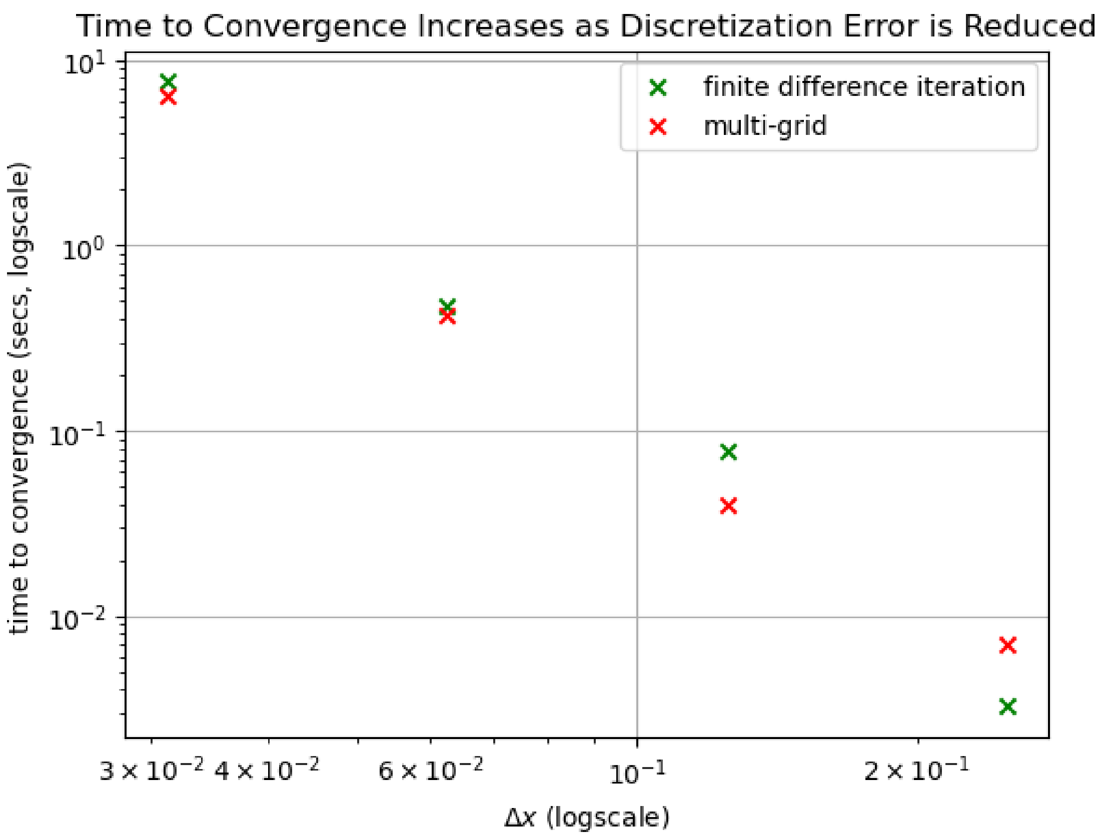


$\rho(x, y)$ with $\sigma = 0.1$
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
3.6
3.2
2.8
2.4
2.0
1.6
1.2
0.8
0.4
0.0



Converged Solution for $\Delta x = 1/32$
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.242
1.104
0.966
0.828
0.690
0.552
0.414
0.276
0.138
0.000


Iterations to Convergence Increase as Discretization Error is Reduced
finite difference iteration
multi-grid
number of iterations to convergence (logscale)
10^3
10^2
3×10^{-2}
4×10^{-2}
6×10^{-2}
10^{-1}
2×10^{-1}
Δx (logscale)

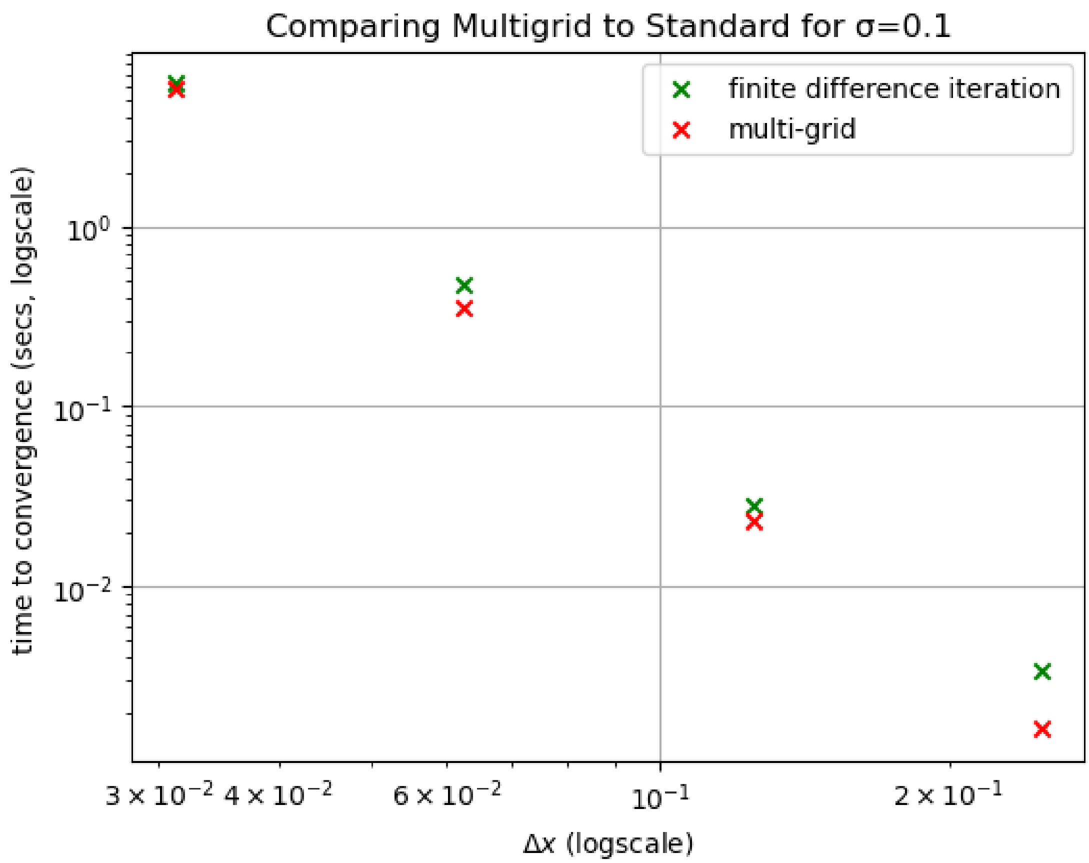


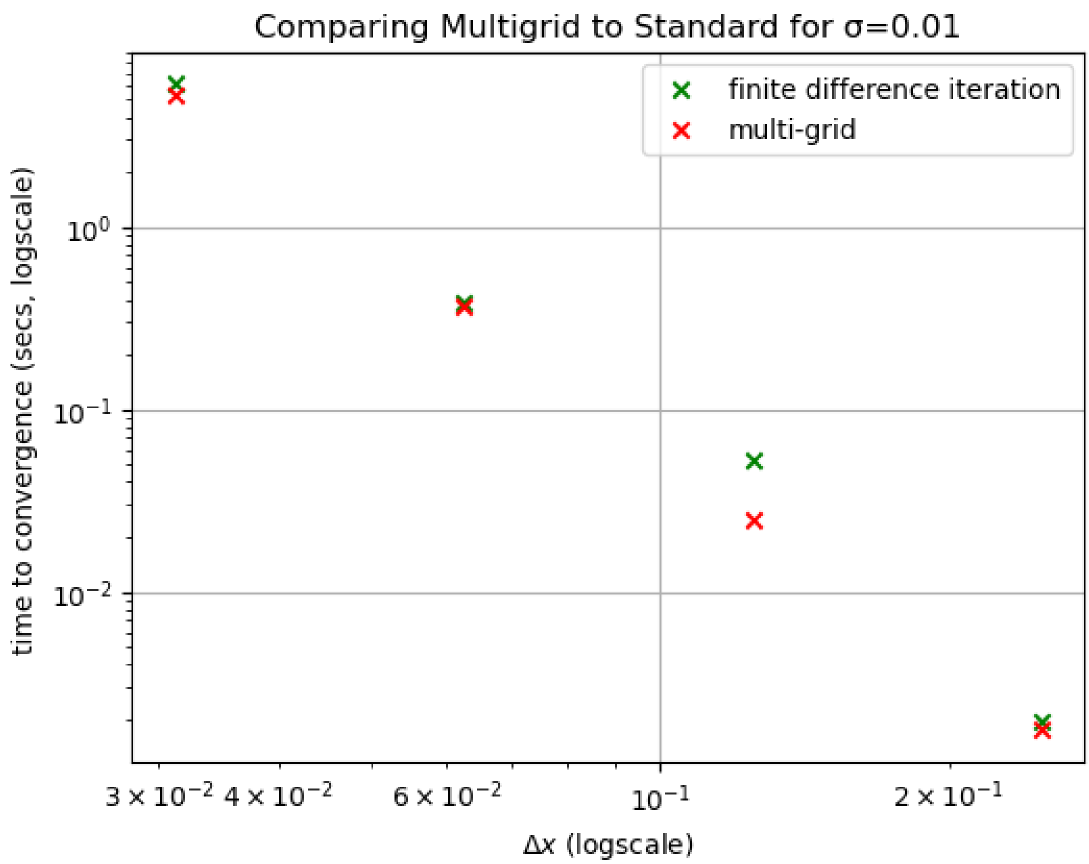
Note that while the overall number of iterations increases for the multi-grid methods in comparison to the standard finite difference with iteration, the wall clock time for smaller Δx decreases. This result reflects the fact that more iterations are performed on easier coarse grids with fewer points. The convergence criterion for this iterative method is chosen to be $||\Phi_{k-1} - \Phi_k|| < 10^{-8}$, thus the method does not require a known solution a priori.

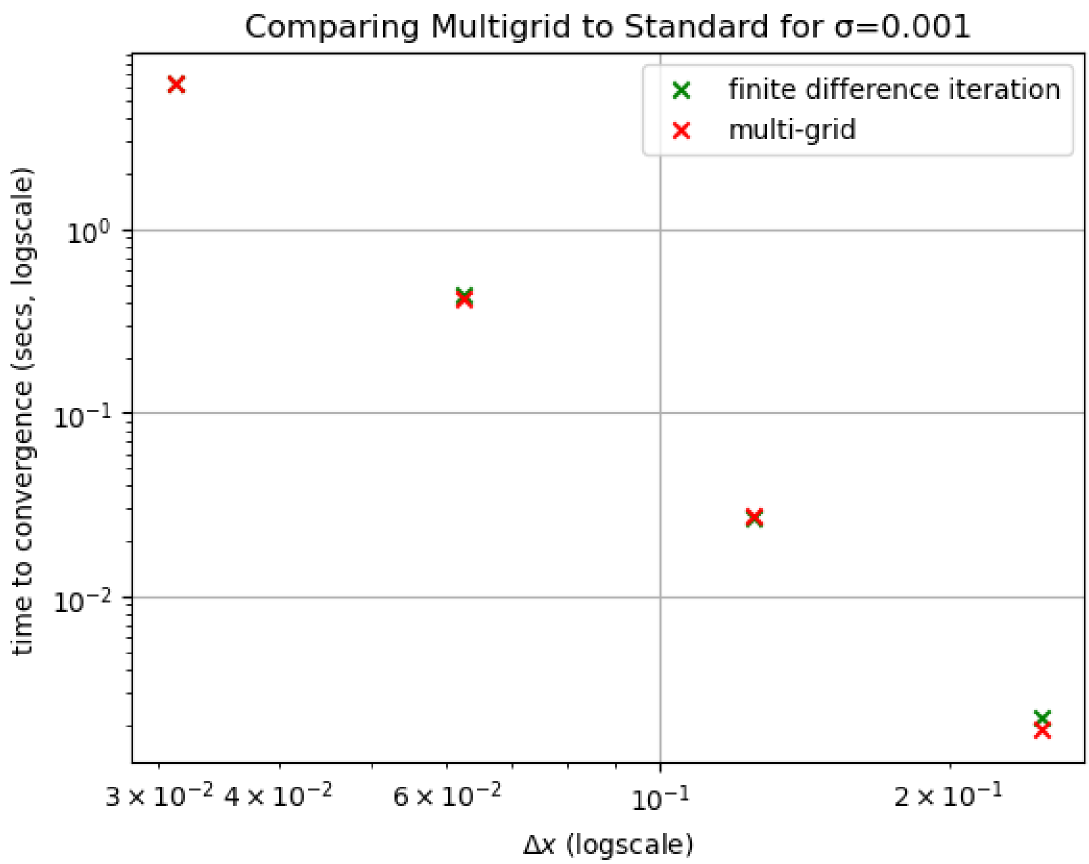
1.3 Comparative Results

[152]:
```
s = [0.1, 0.01, 0.001]
for i,  in enumerate( s):
    _, _, _, times_fd = iteration( )
    _, _, _, times_mg = multigrid( )
    plt.scatter(dxs, times_fd, marker="x", color="green", label=fr"finite␣
 ↪difference iteration")
    plt.scatter(dxs, times_mg, marker="x", color="red", label=fr"multi-grid")

    # format and show
    plt.xscale("log")
    plt.xlabel(r"$\Delta x$ (logscale)")
    plt.ylabel("time to convergence (secs, logscale)")
```

```
plt.yscale("log")
plt.title(f"Comparing Multigrid to Standard for  ={ }")
plt.grid()
plt.legend(loc="best")
plt.show()
```




Comparing Multigrid to Standard for σ=0.01
finite difference iteration
multi-grid
time to convergence (secs, logscale)
10^0
10^{-1}
10^{-2}
3×10^{-2}
4×10^{-2}
6×10^{-2}
10^{-1}
2×10^{-1}
Δx (logscale)



In the above plots, we see that the advantage of multi-grid is more evident for larger choices of σ. This fact makes sense: smaller σ requires a finer grid to refine the tightly concentrated potential, hence the coarse grid steps in the multi-grid method will not alleviate as much computational work for the later finer iterations.