Homework 4

Simon Bolding
NUEN 629

November 17, 2015

NUEN 629, Homework 4

Due Date Nov. 19

Solve the following problem and submit a detailed report, including a justification of why a reader should believe your results and a description of your methods and iteration strategies.

1 Vaquer

(150 points + 50 points extra credit) In class we discussed the diamond-difference spatial discretization. Another discretization is the step discretization (this has several other names from other disciplines). It writes the discrete ordinates equations with isotropic scattering as, for $\mu_n > 0$ to

$$\mu_n \frac{\psi_{i,n} - \psi_{i-1,n}}{h_x} + \Sigma_t \psi_{i,n} = \frac{\Sigma_s}{2}\phi_i + \frac{Q}{2}, \tag{1}$$

and for $\mu_n < 0$

$$\mu_n \frac{\psi_{i+1,n} - \psi_{i,n}}{h_x} + \Sigma_t \psi_{i,n} = \frac{\Sigma_s}{2}\phi_i + \frac{Q}{2}. \tag{2}$$

You should be able to modify the codes I have already provided to implement this discretization.

1. (50 points) Your task is to solve a problem with uniform source of $Q = 0.01$, $\Sigma_t = \Sigma_s = 100$ for a slab in vacuum of width 10 using step and diamond difference discretizations. Use 10, 50, and 100 zones ($h_x = 1, 0.02, 0.01$) and your expert choice of angular quadratures. Discuss your results and how the two methods compare at each number of zones.

2. (10 points) Discuss why there is a different form of the discretization for the different signs of μ.

3. (40 points) Plot the error after each iteration using a 0 initial guess for the step discretization with source iteration and GMRES.

4. (50 points) Solve Reed's problem (see finite difference diffusion codes). Present convergence plots for the solution in space and angle to a "refined" solution in space and angle.

5. (50 points extra credit) Solve a time dependent problem for a slab surrounded by vacuum with $\Sigma_t = \Sigma_s = 1$ and initial condition given by $\phi(0) = 1/h_x$. Plot the solution at $t = 1$ s using step and diamond difference. The particles have a speed of 1 cm/s. Which discretization is better with a small time step? What do you see with a small number of ordinates compared to a really large number (100s)?

Solution 1-1:

To modify the provided code to use the step discretization, essentially only the 1DSweep function needs to be modified. For example, for a positive direction of μ_n the flux in the i-th cell is, for the $k-th$ sweep,

$$\psi_{i,n}^{(k+1)} = \frac{\frac{1}{2}\left(\phi_i^{(k)} + Q\right) + \frac{\mu_n}{h_x}\psi_{i-1,n}^{(k+1)}}{\Sigma_t + \frac{\mu_n}{h_x}}, \tag{1}$$

where $\psi_{i-1,n}$ is either defined by the boundary conditon, i.e., $\psi_0 = f(\mu_n)$, or is known from solution of the previous cell in the sweep. The negative direction sweep is defined analogously. A correction was also made to the diamond difference sweep code.

For the given problem parameters, source iteration was too slow to converge because the scattering ratio is $c = 1$, so the GMRES solver was used for both spatial discretizations. For one case GMRES was verified to converge to the same solution as source iteration. A plot of the different solutions for the different resolutions and spatial discretizations is given below, for $N = 16$ angles, in Fig. 1. On the finest mesh, increasing to $N = 24$ had no visible effect on the solution. The step discretization does not get the asymptotic diffusion limit, so the solution produces inaccurate and variable results for the different refinement levels. The DD solution appears to be converging correctly.

Also plotted below is an analytic diffusion solution to the same problem using Mark Boundary conditions, compared to the DD solution and step with a very fine mesh. For this problem diffusion theory should be accurate, particularly several diffusion lengths from the boundary. Although the mesh size for the step discretization is fine to the $O(1/\Sigma_t)$, the solution still disagrees, although it does appear to be converging towards the correct solution. This seems to be due to the fact that it is a pure scattering solution. Keeping the same thickness but with a small amount of absorption showed better agreement.

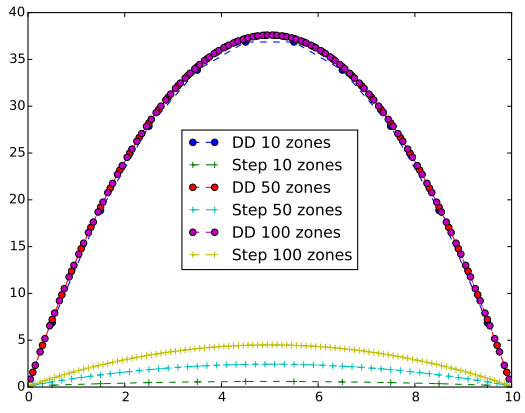

Figure 1: Comparison of step and DD spatial discretizations for different numbers of zones.

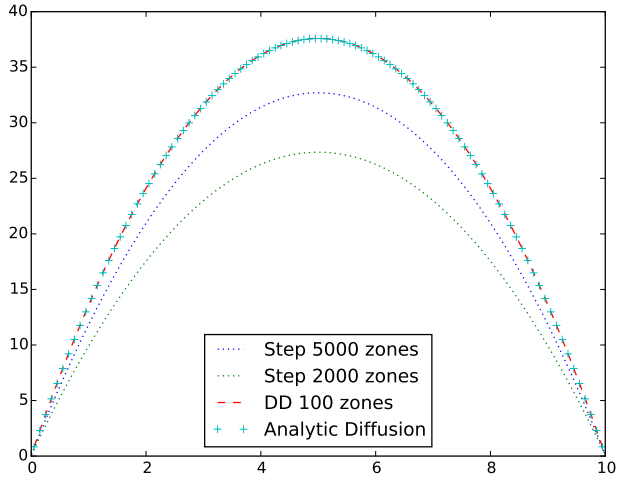

Figure 2: Comparison of analytic diffusion theory, diamond difference, and a highly refined step solution.

Solution 1-2:

The different forms of the discretization are result of the solution being generally undefined at the faces of cells, due to the discontinuity of the solution at cell edges. A closure of some kind must be defined because even in the weak-form we need a value for ψ on the face. The given equations define ψ on the face using the upwind closure, which attempts to numerically propagate information in the physical direction of flow, based on the characteristic flow of information in each direction. This, in theory, resolves strong spatial gradients with higher physical accuracy. This closure also provides stability to the equations, which could demonstrate oscillations with a poor choice of closure.

Solution 1-3:

For convenience, slightly different measures of convergence error are compared between source iteration and GMRES. The GMRES solver returns the relative residual which demonstrates how well the solution is satisfied, whereas for source iteration the relative change between iterations of the scalar flux $(\phi^k - \phi^{k-1})/\phi^k$ is used. GMRES with no restart and a restart of 20 are compared. Results are shown for the case of 100 cells and 16 angles. As seen in the figure below, GMRES without restart quickly converges after an initial slow start. Because the problem is slowly converging, most of the krylov space needs to be formed before the error is rapidly converged upon. Restarting limits the space that is projected onto, leading to a steady, slower convergence. Source iteration is very slow to converge. Initially source iteration looks better than GMRES. This is probably partially due to the difference in errors. In slowly converging systems, the difference in solution is not an accurate estimate of the error, and should really take into account the dominance ratio ρ as $1/(1-\rho)(\phi^k - \phi^{k-1})$.

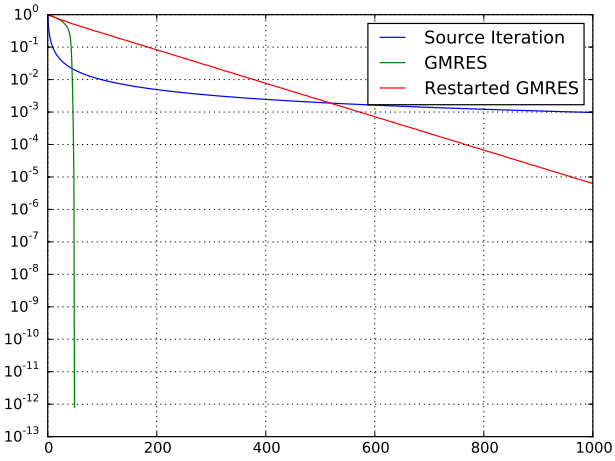

Figure 3: Convergence rates of scattering iterations for source iteration and GMRES for Problem 1-1.

Solution 1-4:

Plots of the solution to Reed's problem for DD and step discretization are given in Fig. **??**. Step has trouble resolving the high spatial variation, particularly in the scattering and source dominated region, and spreads out the answer. However, in the optically thick, pure absorber regions, DD demonstrates spurious oscillations. This is because diamond difference is not "L-stable", similar to crank nicolson. For the case of a pure absorber you can write the amplitude between cells as

$$\psi_{i+1/2} = \left(\frac{1 - \frac{\Sigma \Delta x}{2\mu}}{1 + \frac{\Sigma \Delta x}{2\mu}} \right) \psi_{i-1/2}$$

which is always stable because the quantity in parenthesis has magnitude less than one, but the quantity does not to go 0 as $\Sigma \Delta x$ goes to infinity. If $\frac{\Sigma \Delta x}{2\mu} > 1$, then the amplification will be negative, which would result in oscillations from cell to cell.

For error convergence an approximate $L2$ error was computed as

$$e^{(k)} = \sqrt{\sum_{i=0}^{N_{cells}} (\phi_i^{(k)} - \phi^{\text{ref}}(x_i))^2 \Delta x_i} \tag{2}$$

where ϕ_i is the cell average and $\phi^{\text{ref}}(x_i)$ is evaluated at the cell center of i using a cubic interpolation based on the finest mesh solution. The fine solution is taken to be 600 x cells and 120 directions. Plots of the error convergence can be seen below. The convergence in space is not very consistent, although it does appear to be converging towards the refined solution. This is partially due to the discontinuities in material properties and that the number of cells with different properties varies as the mesh sizes change. The convergence in angle shows a consistent and convergent behavior for both spatial discretizations.

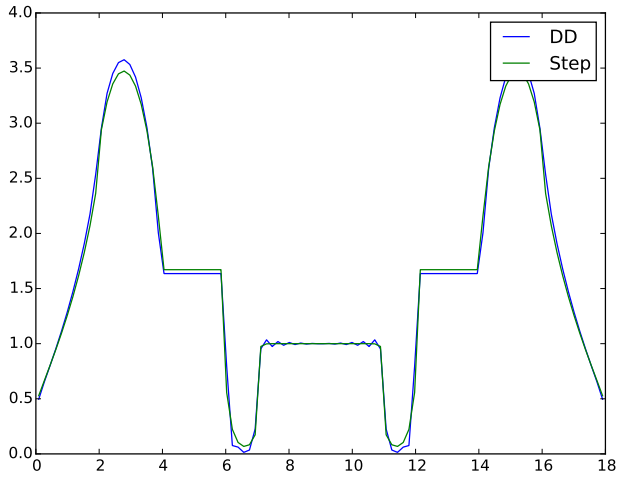

Figure 4: Solutions for Reed's problem.

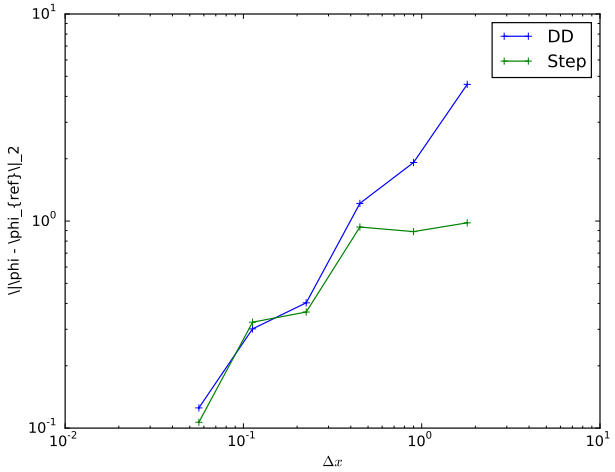

Figure 5: Convergence in space for Reed's problem.

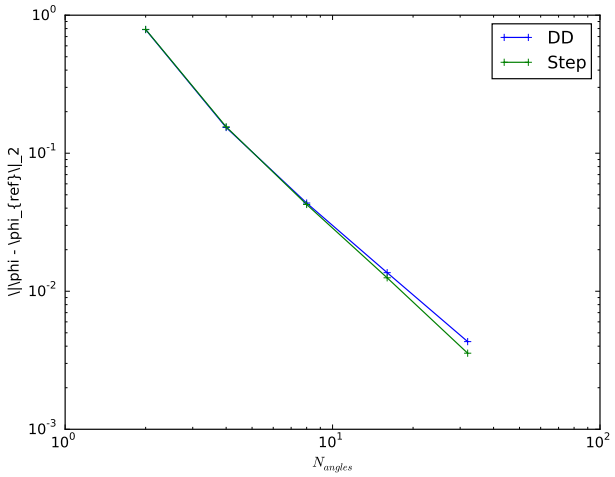

Figure 6: Convergence in angle for Reed's problem.

Solution 1-5:

The code was modified to account for a backward Euler time discretization. The changes to the equations yield the simple changes

$$\Sigma_t \rightarrow \Sigma_t + \frac{1}{v\Delta t} \tag{3}$$

$$q_n \rightarrow q_n + \phi_i^{m+1} 2 + \frac{1}{v\Delta t}\psi_{n,i}^m \tag{4}$$

for the m-th time step. The angular flux is stored between solves. The old angular flux is used as the guess for the scattering source iteration, which significantly improves convergence in later time steps. The angular flux in the cell at the center of the slab is initialized to $1/(2h_x)$ for all μ_n.

A plot of the solution for both discretizations with 200 spatial cells is shown below for two different time steps. The DD solution is inaccurate in both cases, and for the short time-step DD demonstrates severe oscillations. These oscillations result once the effective Σ_t becomes too large due to the $1/(v\Delta t)$ term. Even though step is generally a more diffusive discretization, it is the more accurate choice in this case as DD oscillates and goes negative, although it appears DD is getting a more accurate location of the wave. Both discretizations agree on the location of the wave-front, although the step solution for $\Delta t \leq 0.01$ has a slightly faster wave at the edge due to the diffusive nature of the discretization. As the time step goes from 0.01 s to 0.002 s, the wave-front does not get sharper for step, where as it does for DD (the DD seems more accurate in this case). This is because the spatial discretization error is dominating, and more mesh cells are needed. It is noted that unless Σ_t is very large or Δt is very small, the Backward Euler has significant artificial diffusion in the time variable, which is part of the cause of the difference in shape of the solutions on the very coarse time-step. For this problem 200 directions were needed to eliminate any spikes in the solution that occured from angular discretization error rather than spatial resolution issues.

Plotted below is the solution for various numbers of angles for the step discretization. For smaller number of angles there is noticable ray effects as the particles advect only in specific directions at the same speed, producing an artificially fast wavefront with spikes present, particularly noticeable for the case of 2 angles. As more angles are added, the wave front approach a more physically accurate solution.

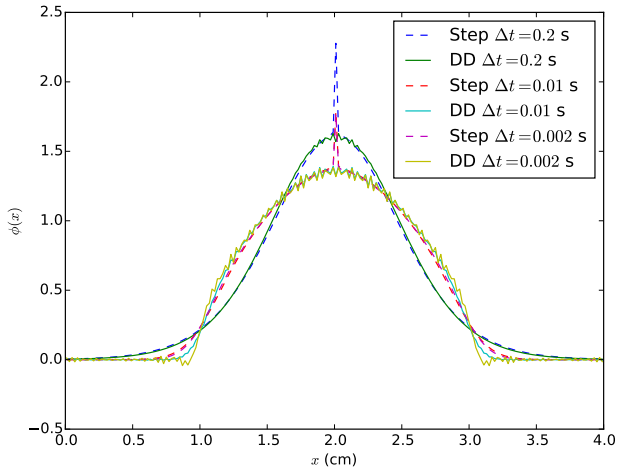

Figure 7: Comparison of spatial discretizations after 1 s, for different time step sizes.

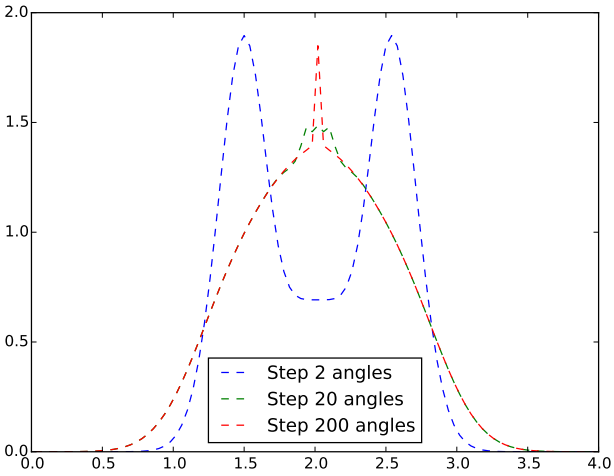

Figure 8: Comparison of step solutions for various numbers of quadrature directions.

Code

```
import scipy.sparse.linalg as spla
from copy import deepcopy
from scipy import sparse
import matplotlib.pyplot as plt
# coding: utf-8

# We begin by defining a function that will perform a transport sweep in 1-D slabs.

# In[78]:

import numpy as np
def sweepDD1D(I,hx,q,sigma_t,mu,boundary):
    """Compute a transport sweep for a given
    Inputs:
        I:              number of zones
        hx:             size of each zone
        q:              source array
        sigma_t:        array of total cross-sections
        mu:             direction to sweep
        boundary:       value of angular flux on the boundary
    Outputs:
        psi:            value of angular flux in each zone
    """
    assert(np.abs(mu) > 1e-10)
    psi = np.zeros(I)
    ihx = 1/hx
    if (mu > 0):
        psi_left = boundary
        for i in range(I):
            psi_right = (q[i]*0.5 + psi_left*(mu*ihx - 0.5*sigma_t[i]))/(0.5*sigma_t[i] + mu*ihx)
            psi[i] = 0.5*(psi_right + psi_left)
            psi_left = psi_right
    else:
        psi_right = boundary
        for i in reversed(range(I)):
            psi_left = (q[i]*0.5 - psi_right*(mu*ihx + 0.5*sigma_t[i]))/(0.5*sigma_t[i] - mu*ihx)
            psi[i] = 0.5*(psi_right + psi_left)
            psi_right = psi_left
    return psi

def sweepStep1D(I,hx,q,sigma_t,mu,boundary):
    """Compute a transport sweep using a step discretization
    Inputs:
        I:              number of zones
        hx:             size of each zone
        q:              source array
        sigma_t:        array of total cross-sections
        mu:             direction to sweep
        boundary:       value of angular flux on the boundary
    Outputs:
        psi:            value of angular flux in each zone
    """
    assert(np.abs(mu) > 1e-10)
    psi = np.zeros(I)
    ihx = 1/hx
    if (mu > 0):
        psi_in = boundary
        for i in range(0,I):
            psi[i] = (q[i]*0.5 + mu*psi_in*ihx)/(sigma_t[i] + mu*ihx)
            psi_in = 1.*psi[i]
    else:
        psi_in = boundary
        for i in reversed(range(I)):
            psi[i] = (q[i]*0.5 - mu*psi_in*ihx)/(sigma_t[i] - mu*ihx)
            psi_in = 1.*psi[i]
```

```
    return psi



def time_dependent(I,hx,t_end,T,v,q,sigma_t,sigma_s,N,BCs, tolerance = 1.0e-8,maxits = 500, LOUD=False, sw
    """Solve time dependent problem
    Inputs:
        I:              number of zones
        hx:             size of each zone
        t_end:          end of simulation time
        T:              number of time steps
        v:              speed of particles
        sigma_t:        array of total cross-sections format [i]
        sigma_s:        array of scattering cross-sections format [i]
        N:              number of angles
        tolerance:      the relative convergence tolerance for the iterations
        maxits:         the maximum number of iterations
        LOUD:           boolean to print out iteration stats
    Outputs:
        x:              value of center of each zone
        phi(I):         value of scalar flux in each zone
    """
    #Hard coded initial condition
    phi_init = np.zeros(I)
    phi_init[int(I/2)] = (1/hx)
    psi_old  = np.zeros((N,I))
    psi_old[:,I/2] = 1/hx
    psi = np.zeros((N,I))
    phi_old = phi_init #phi from last time step is first guess
    MU, W = np.polynomial.legendre.leggauss(N)
    dt = t_end/T

    for it in range(T):

        if (LOUD > 0) or (it==T-1 and LOUD < 0):
            print("\nTime is: ", (it+1)*dt, "step ", it+1, " of ", T)

        #Compute artificial source
        sigma_t_eff = sigma_t + np.ones(I)*(1./(v*dt))
        src_old     = psi_old*(1./(v*dt))
        phi =  phi_old.copy()
        converged = False
        iteration = 1
        while not(converged):
            phi = np.zeros(I)
            #sweep over each direction
            for n in range(N):
                tmp_psi = sweep1D(I,hx,q + phi_old*sigma_s + 2.*src_old[n,:],sigma_t_eff,MU[n],BCs[n])
                psi[n,:] = tmp_psi
                phi += tmp_psi*W[n]
            #check convergence
            change = np.linalg.norm(phi-phi_old)/np.linalg.norm(phi)
            converged = (change < tolerance) or (iteration > maxits)
            if (LOUD>0) or (converged and LOUD<0):
                print("Iteration",iteration,": Relative Change =",change)
            if (iteration > maxits):
                print("Warning: Source Iteration did not converge")
            iteration += 1
            phi_old = phi.copy() #We dont every actually use phi_old, so it gets overwritten every time

        #Update old psi
        psi_old = psi.copy()

    x = np.linspace(hx/2,I*hx-hx/2,I)
    return x, phi

def gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, tolerance = 1.0e-8,maxits = 100, LOUD=False, restart = 100, s
```

```
    """Solve, via GMRES, a single-group steady state problem
    Inputs:
        I:                  number of zones
        hx:                 size of each zone
        q:                  source array
        sigma_t:            array of total cross-sections
        sigma_s:            array of scattering cross-sections
        N:                  number of angles
        tolerance:          the relative convergence tolerance for the iterations
        maxits:             the maximum number of iterations
        LOUD:               boolean to print out iteration stats
    Outputs:
        x:                  value of center of each zone
        phi:                value of scalar flux in each zone
    """
    #compute left-hand side
    LHS = np.zeros(I)

    MU, W = np.polynomial.legendre.leggauss(N)
    for n in range(N):
            tmp_psi = sweep1D(I,hx,q,sigma_t,MU[n],BCs[n])
            LHS += tmp_psi*W[n]
    #define linear operator for gmres
    def linop(phi):
        tmp = phi*0
        #sweep over each direction
        for n in range(N):
            tmp_psi = sweep1D(I,hx,phi*sigma_s,sigma_t,MU[n],BCs[n])
            tmp += tmp_psi*W[n]
        return phi-tmp
    A = spla.LinearOperator((I,I), matvec = linop, dtype='d')
    #define a little function to call when the iteration is called
    iteration = np.zeros(1)
    err_list = []
    def callback(rk, iteration=iteration, err_list=err_list):
        iteration += 1
        err_list.append(np.linalg.norm(rk))
        if (LOUD>0):
            print("Iteration",iteration[0],"norm of residual",np.linalg.norm(rk))
    #now call GMRES
    phi,info = spla.gmres(A,LHS,x0=LHS,restart=restart, maxiter=maxits,tol=tolerance, callback=callback)
    if (LOUD):
        print("Finished in",iteration[0],"iterations.")
    if (info >0):
        print("Warning, convergence not achieved")
    x = np.linspace(hx*.5,I*hx-hx*.5,I)
    return x, phi, err_list

# We create the source iteration algorithm to that will call the sweep next. Luckily, NumPy has the Gauss-

# In[348]:

def source_iteration(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D=sweepDD1D, tolerance = 1.0e-8,maxits = 100, LOU
    """Perform source iteration for single-group steady state problem
    Inputs:
        I:                  number of zones
        hx:                 size of each zone
        q:                  source array
        sweep1D:            sweeping function
        sigma_t:            array of total cross-sections
        sigma_s:            array of scattering cross-sections
        N:                  number of angles
        tolerance:          the relative convergence tolerance for the iterations
        maxits:             the maximum number of iterations
        LOUD:               boolean to print out iteration stats
    Outputs:
        x:                  value of center of each zone
        phi:                value of scalar flux in each zone
```

```
    """
    phi = np.zeros(I)
    phi_old = phi.copy()
    converged = False
    MU, W = np.polynomial.legendre.leggauss(N)
    iteration = 1
    err_list = []
    while not(converged):
        phi = np.zeros(I)
        #sweep over each direction
        for n in range(N):
            tmp_psi = sweep1D(I,hx,q + phi_old*sigma_s,sigma_t,MU[n],BCs[n])
            phi += tmp_psi*W[n]
        #check convergence
        change = np.linalg.norm(phi-phi_old)/np.linalg.norm(phi)
        err_list.append(change)
        converged = (change < tolerance) or (iteration > maxits)
        if (LOUD>0) or (converged and LOUD<0):
            print("Iteration",iteration,": Relative Change =",change)
        if (iteration > maxits):
            print("Warning: Source Iteration did not converge")
        iteration += 1
        phi_old = phi.copy()
    x = np.linspace(hx/2,I*hx-hx/2,I)
    return x, phi, err_list


def coordLookup_l(i, j, k, I, J):
    """get the position in a 1-D vector
    for the (i,j,k) index
    """
    return i + j*I + k*J*I

def coordLookup_ijk(l, I, J):
    """get the position in a (i,j,k)  coordinates
    for the index l in a 1-D vector
    """
    k = (l // (I*J)) + 1
    j = (l - k*J*I) // I + 1
    i = l - (j*I + k*J*I)-1
    return i,j,k

def diffusion_steady_fixed_source(Dims,Lengths,BCs,D,Sigma,Q, tolerance=1.0e-12, LOUD=False):
    """Solve a steady state, single group diffusion problem with a fixed source
    Inputs:
        Dims:           number of zones (I,J,K)
        Lengths:        size in each dimension (Nx,Ny,Nz)
        BCs:            A, B, and C for each boundary, there are 8 of these
        D,Sigma,Q:      Each is an array of size (I,J,K) containing the quantity
    Outputs:
        x,y,z:          Vectors containing the cell centers in each dimension
        phi:            A vector containing the solution
    """
    I = Dims[0]
    J = Dims[1]
    K = Dims[2]
    L = I*J*K
    Nx = Lengths[0]
    Ny = Lengths[1]
    Nz = Lengths[2]

    hx,hy,hz = np.array(Lengths)/np.array(Dims)
    ihx2,ihy2,ihz2 = (1.0/hx**2,1.0/hy**2,1.0/hz**2)

    #allocate the A matrix, and b vector
    A = sparse.lil_matrix((L,L))
    b = np.zeros(L)

```

```
    temp_term = 0
    for k in range(K):
        for j in range(J):
            for i in range(I):
                temp_term = Sigma[i,j,k]
                row = coordLookup_l(i,j,k,I,J)
                b[row] = Q[i,j,k]
                #do x-term left
                if (i>0):
                    Dhat = 2* D[i,j,k]*D[i-1,j,k] / (D[i,j,k] + D[i-1,j,k])
                    temp_term += Dhat*ihx2
                    A[row, coordLookup_l(i-1,j,k,I,J)]  = -Dhat*ihx2
                else:
                    bA,bB,bC = BCs[0,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (i<I-1):
                            temp_term += -1.5*D[i,j,k]*bA/bB/hx
                            b[row] += -D[i,j,k]/bB*bC/hx
                            A[row,  coordLookup_l(i+1,j,k,I,J)]  += 0.5*D[i,j,k]*bA/bB/hx
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hx
                            b[row] += -D[i,j,k]/bB*bC/hx
                    else:
                        temp_term += D[i,j,k]*ihx2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihx2*2.0
                #do x-term right
                if (i < I-1):
                    Dhat = 2* D[i,j,k]*D[i+1,j,k] / (D[i,j,k] + D[i+1,j,k])
                    temp_term += Dhat*ihx2
                    A[row, coordLookup_l(i+1,j,k,I,J)]  += -Dhat*ihx2
                else:
                    bA,bB,bC = BCs[1,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (i>0):
                            temp_term += 1.5*D[i,j,k]*bA/bB/hx
                            b[row] += D[i,j,k]/bB*bC/hx
                            A[row,  coordLookup_l(i-1,j,k,I,J)]  += -0.5*D[i,j,k]*bA/bB/hx
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hx
                            b[row] += -D[i,j,k]/bB*bC/hx

                    else:
                        temp_term += D[i,j,k]*ihx2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihx2*2.0
                #do y-term
                if (j>0):
                    Dhat = 2* D[i,j,k]*D[i,j-1,k] / (D[i,j,k] + D[i,j-1,k])
                    temp_term += Dhat*ihy2
                    A[row, coordLookup_l(i,j-1,k,I,J)]  += -Dhat*ihy2
                else:
                    bA,bB,bC = BCs[2,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (j<J-1):
                            temp_term += -1.5*D[i,j,k]*bA/bB/hy
                            b[row] += -D[i,j,k]/bB*bC/hy
                            A[row,  coordLookup_l(i,j+1,k,I,J)]  += 0.5*D[i,j,k]*bA/bB/hy
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hy
                            b[row] += -D[i,j,k]/bB*bC/hy
                    else:
                        temp_term += D[i,j,k]*ihy2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihy2*2.0
                if (j < J-1):
                    Dhat = 2* D[i,j,k]*D[i,j+1,k] / (D[i,j,k] + D[i,j+1,k])
                    temp_term += Dhat*ihy2
                    A[row, coordLookup_l(i,j+1,k,I,J)]  += -Dhat*ihy2
                else:
                    bA,bB,bC = BCs[3,:]
```

```
                    if (np.abs(bB) > 1.0e-8):
                        if (j>0):
                            temp_term += 1.5*D[i,j,k]*bA/bB/hy
                            b[row] += D[i,j,k]/bB*bC/hy
                            A[row,  coordLookup_l(i,j-1,k,I,J)]  += -0.5*D[i,j,k]*bA/bB/hy
                        else:
                            temp_term += 0.5*D[i,j,k]*bA/bB/hy
                            b[row] += D[i,j,k]/bB*bC/hy

                    else:
                        temp_term += D[i,j,k]*ihy2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihy2*2.0
                #do z-term
                if (k>0):
                    Dhat = 2* D[i,j,k]*D[i,j,k-1] / (D[i,j,k] + D[i,j,k-1])
                    temp_term += Dhat*ihz2
                    A[row, coordLookup_l(i,j,k-1,I,J)]  += -Dhat*ihz2
                else:
                    bA,bB,bC = BCs[4,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (k<K-1):
                            temp_term += -1.5*D[i,j,k]*bA/bB/hz
                            b[row] += -D[i,j,k]/bB*bC/hz
                            A[row,  coordLookup_l(i,j,k+1,I,J)]  += 0.5*D[i,j,k]*bA/bB/hz
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hz
                            b[row] += -D[i,j,k]/bB*bC/hz
                    else:
                        temp_term += D[i,j,k]*ihz2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihz2*2.0
                if (k < K-1):
                    Dhat = 2* D[i,j,k]*D[i,j,k+1] / (D[i,j,k] + D[i,j,k+1])
                    temp_term += Dhat*ihz2
                    A[row, coordLookup_l(i,j,k+1,I,J)]  += -Dhat*ihz2
                else:
                    bA,bB,bC = BCs[5,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (k>0):
                            temp_term += 1.5*D[i,j,k]*bA/bB/hz
                            b[row] += D[i,j,k]/bB*bC/hz
                            A[row,  coordLookup_l(i,j,k-1,I,J)]  += -0.5*D[i,j,k]*bA/bB/hz
                        else:
                            temp_term += 0.5*D[i,j,k]*bA/bB/hz
                            b[row] += D[i,j,k]/bB*bC/hz

                    else:
                        temp_term += D[i,j,k]*ihz2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihz2*2.0
                A[row,row] += temp_term
    phi,code = spla.cg(A,b, tol=tolerance)
    if (LOUD):
        print("The CG solve exited with code",code)
    phi_block = np.zeros((I,J,K))
    for k in range(K):
        for j in range(J):
            for i in range(I):
                phi_block[i,j,k] = phi[coordLookup_l(i,j,k,I,J)]
    x = np.linspace(hx*.5,Nx-hx*.5,I)
    y = np.linspace(hy*.5,Ny-hy*.5,J)
    z = np.linspace(hz*.5,Nz-hz*.5,K)
    if (I*J*K <= 10):
        print(A.toarray())
    return x,y,z,phi_block
# In[368]:

##simple test problem
#I = 30
#hx = 1/I
```

```
#q = np.zeros(I)
#sigma_t = np.ones(I)
#sigma_s = 0*sigma_t
#N = 2
#BCs = np.zeros(N)
#BCs[(N/2):N] = 1.0
#
#x,phi_sol = source_iteration(I,hx,q,sigma_t,sigma_s,N,BCs, tolerance = 1.0e-8,maxits = 100, LOUD=True, swe
#x,phi_dd = source_iteration(I,hx,q,sigma_t,sigma_s,N,BCs, tolerance = 1.0e-8,maxits = 100, LOUD=True)
#
#
## In[369]:
#
#import matplotlib.pyplot as plt
#plt.figure()
#print(phi_sol)
#plt.plot(x,phi_sol,'+-',label="Step")
#plt.plot(x,phi_dd,'o',label="DD")
#X = np.linspace(0,1,100)
#plt.plot(X,np.exp(-X*np.sqrt(3.0)),label="Exact Solution")
#plt.legend()
#plt.savefig('exact.pdf')

##Now a more complicated test
#W = 10
#n_zones = [10,50,100]
#plt.figure()
#sig_t = 100
#for I in n_zones:
#
#    hx = W/I
#    q = np.ones(I)*0.01
#    sigma_t = sig_t*np.ones(I)
#    sigma_s = deepcopy(sigma_t)
#    sigma_a = sigma_t[0] - sigma_s[0]
#    N = 16
#    BCs = np.zeros(N)
#
#    x,phi_sol,err_vect  = gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepDD1D, tolerance = 1.0e-
#    x,phi_step,err_vect = gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, tolerance = 1.0
#    plt.plot(x,phi_sol,'o--',label="DD "+str(I)+" zones")
#    plt.plot(x,phi_step,'+--',label="Step "+str(I)+" zones")
#
#plt.legend(loc='best')
#plt.savefig('prob1.pdf')
#
##Solve a diffusion problem to get estimate of answer
#I = 100
#J = 1
#K = 1
#Nx = W
#Ny = 1
#Sigma = np.ones((I,J,K))*sigma_a
#D = 1./(3.*sig_t)*np.ones((I,J,K))
#Q = q[0]*np.ones((I,J,K))
#BCs = np.ones((6,3))
#BCs[:,0] = 0 #Reflective in Y and Z
#BCs[:,2] = 0
#BCs[0,:] = [0.25,-np.sqrt(3)/2,0] #Mark vacuum in other variables
#BCs[1,:] = [0.25,np.sqrt(3)/2,0]
#
#plt.figure()
#xd,y,z, phi_diff = diffusion_steady_fixed_source((I,J,K),(Nx,1,1),BCs,D,Sigma,Q,tolerance=1.0e-12, LOUD=T
#
##Analytic diffusion solution
#phid = lambda x: q[0]*(W*W/(8*D[0,0,0]) + W*np.sqrt(3)/2 - x*x/(2*D[0,0,0]))
#phi_diff1 = [phid(i-0.5*W) for i in xd]
#
```

```
#
#I = 5000
#N = 16
#hx = W/I
#q = np.ones(I)*0.01
#sigma_t = sig_t*np.ones(I)
#sigma_s = deepcopy(sigma_t)
#sigma_a = sigma_t[0] - sigma_s[0]
#N = 16
#BCs = np.zeros(N)
#
#x,phi_sol,_ = gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, tolerance = 1.0e-8,maxits
#
#I = 2000
#N = 16
#hx = W/I
#q = np.ones(I)*0.01
#sigma_t = sig_t*np.ones(I)
#sigma_s = deepcopy(sigma_t)
#sigma_a = sigma_t[0] - sigma_s[0]
#N = 16
#BCs = np.zeros(N)
#
#x1,phi_sol1,_ = gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, tolerance = 1.0e-8,maxi
#
#I = 100
#N = 16
#hx = W/I
#q = np.ones(I)*0.01
#sigma_t = sig_t*np.ones(I)
#sigma_s = deepcopy(sigma_t)
#sigma_a = sigma_t[0] - sigma_s[0]
#N = 16
#BCs = np.zeros(N)
#xdd,phi_DD,_ = gmres_solve(I, hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepDD1D, tolerance = 1.0e-8,maxi
#
#print(phi_sol)
#
#plt.plot(x,phi_sol,':',label="Step 5000 zones")
#plt.plot(x1,phi_sol1,':',label="Step 2000 zones")
#plt.plot(xdd,phi_DD,'--',label="DD 100 zones")
##plt.plot(xd,phi_diff[:,0,0],'-',label="Diffusion")
#plt.plot(xd,phi_diff1,'+',label="Analytic Diffusion")
#plt.legend(loc='best')
#plt.savefig('diff.pdf',bbox_inches='tight')
#
#exit()

#Error convergence

#Now a more complicated test
#W = 10
#I = 100
#plt.figure()
#sig_t = 100
#
#hx = W/I
#q = np.ones(I)*0.01
#sigma_t = sig_t*np.ones(I)
#sigma_s = deepcopy(sigma_t)
#sigma_a = sigma_t[0] - sigma_s[0]
#N = 16
#BCs = np.zeros(N)
#
#x,phi_sol,err_si  = source_iteration(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, tolerance = 1.0e
#x,phi_step,err_gmres = gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, tolerance = 1.0e-
#x,phi_step,err_gmresr = gmres_solve(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, restart=20,tolera
#
```

```
#iters_si = [i for i in range(len(err_si))]
#iters_gmres = [i for i in range(len(err_gmres))]
#iters_gmresr = [i for i in range(len(err_gmresr))]
#
#plt.semilogy(iters_si, err_si, label="Source Iteration")
#plt.semilogy(iters_gmres, err_gmres, label="GMRES")
#plt.semilogy(iters_gmresr, err_gmresr, label="Restarted GMRES")
#print(err_gmres)
#plt.grid()
#
#
#plt.legend(loc='best')
#plt.savefig('err.pdf', bbox_inches='tight')
#exit()


# Reed's Problem

# In[309]:

#in this case all three are constant
def Sigma_t(r):
    value = 0 + ((1.0*(r>=14) + 1.0*(r<=4)) +
                 5.0 *((np.abs(r-11.5)<0.5) or (np.abs(r-6.5)<0.5)) +
                 50.0 * (np.abs(r-9)<=2) )
    return value;
def Sigma_a(r):
    value = 0 + (0.1*(r>=14) + 0.1*(r<=4) +
                 5.0 *((np.abs(r-11.5)<0.5) or (np.abs(r-6.5)<0.5)) +
                 50.0 * (np.abs(r-9)<=2) )
    return value;
def Q(r):
    value = 0 + 1.0*((r<16) * (r>14))+ 1.0*((r>2) * (r<4)) + 50.0*(np.abs(r-9)<=2)
    return value;

from scipy import interpolate
def computeL2Error(phi_ref, x_ref, phi, x, dx):

    phi_ref = interpolate.interp1d(x_ref,phi_ref)
    phi     = interpolate.interp1d(x,phi)
    errsq = sum((phi_ref(xi) - phi(xi))**2*dx for xi in x)
    return np.sqrt(errsq)

# In[310]:


#n_dir = []
#xpts = []
#dx = []
#phi_step = []
#phi_dd = []
##for I in [10,20,40,80,160,320,600]:
#I = 600
#for N in [2,4,8,16,32,120]:
#
#    L = 18
#    hx = L/I
#    xpos = hx/2;
#    q = np.zeros(I)
#    sigma_t = np.zeros(I)
#    sigma_s = np.zeros(I)
#    for i in range(I):
#        sigma_t[i] = Sigma_t(xpos)
#        sigma_s[i] = sigma_t[i]-Sigma_a(xpos)
#        q[i] = Q(xpos)
#        xpos += hx
#
#    BCs = np.zeros(N)
```

```
#
#     x,phi_sol,_ = source_iteration(I,hx,q,sigma_t,sigma_s,N,BCs, tolerance = 1.0e-8,maxits = 1000, LOUD=-
#     x,phi_step1,_ = source_iteration(I,hx,q,sigma_t,sigma_s,N,BCs, sweep1D = sweepStep1D, tolerance = 1.0
#
#     xpts.append(x)
#     n_dir.append(N)
#     dx.append(hx)
#     phi_step.append(phi_step1)
#     phi_dd.append(phi_sol)
#
##Compute L2 error
#errs_dd = []
#errs_step = []
#for i in range(len(phi_step)-1):
#
#     errs_dd.append(computeL2Error(phi_dd[-1],xpts[-1],phi_dd[i],xpts[i],dx[i]))
#     errs_step.append(computeL2Error(phi_step[-1],xpts[-1],phi_step[i],xpts[i],dx[i]))
#
#plt.figure()
#del dx[-1]
#del n_dir[-1]
#plt.loglog(n_dir,errs_dd,'-+',label="DD")
#plt.loglog(n_dir,errs_step,'-+',label="Step")
#plt.xlabel("$N_{angles}$")
#plt.ylabel("\|\phi - \phi_{ref}\|_2")
#plt.legend()
#plt.savefig('err_reedmu.pdf',bbox_inches='tight')

#Time dependent problem
I = 200
hx = 4/I
q = np.zeros(I)
t_end = 1
T     = 5
sigma_t = np.ones(I)*1
sigma_s = sigma_t.copy()
N = 200
BCs = np.zeros(N)
#q   = sigma_t - sigma_s
vel = 1.

x,phi_dd   = time_dependent(I,hx,t_end,T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits = 100, LOUD=
x,phi_step = time_dependent(I,hx,t_end,T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits = 100, LOUD=
x,phi_dd_fine   = time_dependent(I,hx,t_end, 20*T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits =
x,phi_step_fine = time_dependent(I,hx,t_end, 20*T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits =
x,phi_dd_sfine   = time_dependent(I,hx,t_end, 100*T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits =
x,phi_step_sfine = time_dependent(I,hx,t_end,100*T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits =
plt.figure()
plt.plot(x,phi_step,"--",label="Step $\Delta t = 0.2$ s")
plt.plot(x,phi_dd  ,"-",label="DD $\Delta t=0.2$ s")
plt.plot(x,phi_step_fine,"--",label="Step $\Delta t=0.01$ s")
plt.plot(x,phi_dd_fine  ,"-",label="DD $\Delta t=0.01$ s")
plt.plot(x,phi_step_sfine,"--",label="Step $\Delta t=0.002$ s")
plt.plot(x,phi_dd_sfine  ,"-",label="DD $\Delta t=0.002$ s")
plt.xlabel("$x$ (cm)")
plt.ylabel("$\phi(x)$")
plt.legend(loc='best')
plt.savefig('time_dep_steps.pdf',bbox_inches='tight')

plt.figure()
for N in [2,20,200]:

    # Time dependent problem
    I = 100
    hx = 4/I
    q = np.zeros(I)
    t_end = 1
    T     = 100
```

```
678         sigma_t = np.ones(I)*1
679         sigma_s = sigma_t.copy()
680         BCs = np.zeros(N)
681         #q   = sigma_t - sigma_s
682         vel = 1.
683 
684         x,phi_dd   = time_dependent(I,hx,t_end,T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits = 100, I
685         x,phi_step = time_dependent(I,hx,t_end,T,vel,q,sigma_t,sigma_s,N,BCs,tolerance = 1.0e-8,maxits = 100, I
686         plt.plot(x,phi_step,"--",label="Step "+str(N)+" angles")
687 
688     plt.legend(loc='best')
689     plt.savefig('time_dep_angles.pdf',bbox_inches='tight')
```