PROBLEM SET 2

22.S904 Nuclear Reactor Kinetics

Due: 24 September 2012

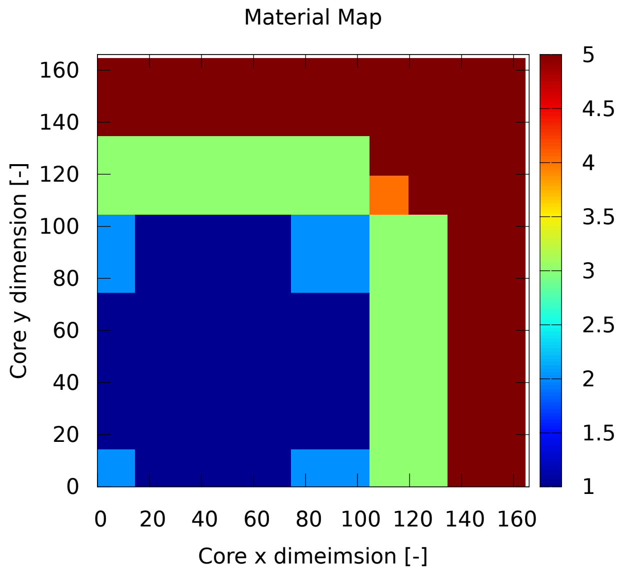
Bryan Herman

Diffusion Code

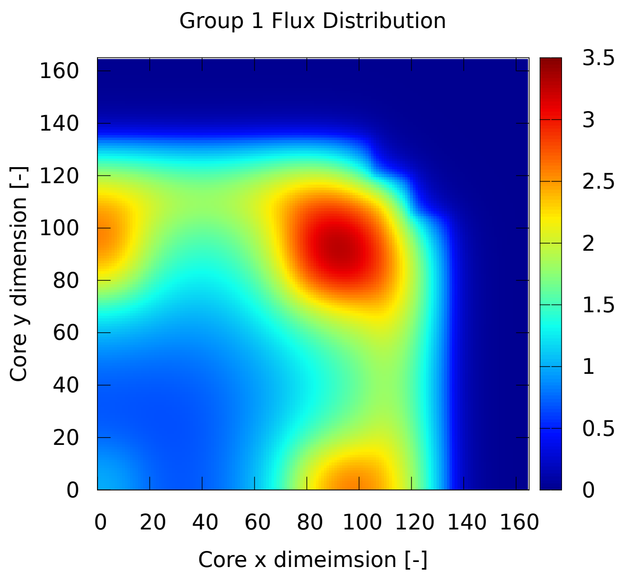
A general three-dimensional second order finite volume code was written to solve the neutron diffusion equation. **The source code can be reviewed at:**

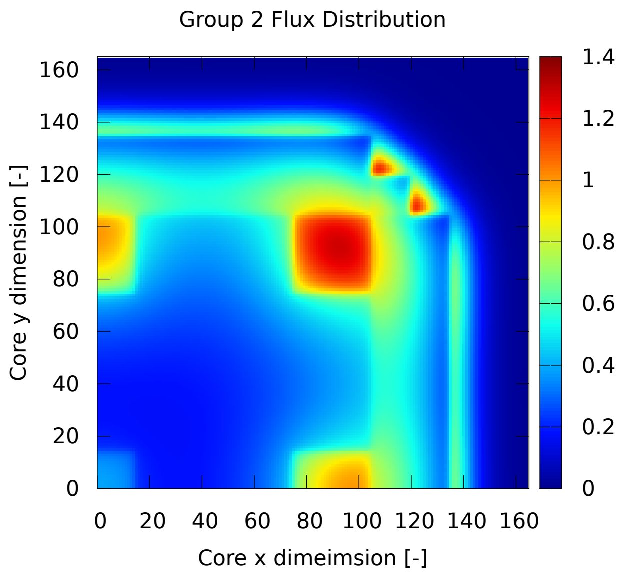
http://github.com/bhermanmit/Kinetics/tree/master/HW2/src.

The code uses an xml-based input and outputs results in HDF5 format for postprocessing in languages such as python. All of the plots in this report were made with GNUPLOT. This code was verified in 2-D by running the LRA BWR benchmark. The input file used for this comparison is listed in the Appendix. The material map arrangement for the LRA core is shown below.



Material 1 is a low enriched bundle, material 2 is medium enriched while material 3 is the highest enriched bundle. Material region 4 is exactly the same as 3 except a control rod has been ejected such that the thermal absorption cross section is lower. Finally material region 5 is the water reflector. The resulting group 1 and group 2 flux distributions are shown below.





Finally the difference in eigenvalue as compared to the benchmark reference is listed below. It shows very good agreement with the benchmark reference and gives me confidence in the diffusion solver.

My Code k_{eff}	Reference k_{eff}	Δk (pcm)
0.99633	0.99636	3

Part A - Difference Equations

Derive the expression for the first-order finite-difference net curent at a nodal interface for the case of variable mesh spacing/material properties.

We will begin the derivation by writing expressions for the net current at a suface between two arbitrary nodes. We assume that cross sections and diffusion coefficients are constants in each cell. The surface currents are

$$\overline{J}_u^g\Big|_{l+1/2,m,n}^{-} = -D_{l,m,n}^g \frac{d}{du}\overline{\phi}_u^g\Big|_{l+1/2,m,n}^{-}$$

$$\overline{J}_u^g\Big|_{l+1/2,m,n}^{+} = -D_{l+1,m,n}^g \frac{d}{du}\overline{\phi}_u^g\Big|_{l+1/2,m,n}^{+}.$$

In this notation,

- $\overline{J}_u^g\Big|_{l+1/2,m,n}^{-}$ is the group g surface-averaged net current at arbitrary location $l{+}1/2, m, n$, approaching the surface from the negative sense
- $\overline{J}_u^g\Big|_{l+1/2,m,n}^{+}$ is the group g surface-averaged net current at arbitrary location $l{+}1/2, m, n$, approaching the surface from the positive sense
- $D_{l,m,n}^g$ and $D_{l+1,m,n}^g$ are the cell-averaged diffusion coefficients in their respective cells
- $\frac{d}{du}\overline{\phi}_u^g\Big|_{l+1/2,m,n}^{-}$ is the gradient with respective to arbitrary direction u of the group g surface-averaged flux at location $l+1/2, m, n$ approaching from the left
- $\frac{d}{du}\overline{\phi}_u^g\Big|_{l+1/2,m,n}^{+}$ is the gradient with respective to arbitrary direction u of the group g surface-averaged flux at location $l+1/2, m, n$ approaching from the right

We can approximate each of these spatial derivatives by taking either a forward or backward difference between the surface-averaged flux and cell-averaged flux, which we approximate to be the flux at the center of the cell. Therefore each equation becomes

$$\overline{J}_u^g\Big|_{l+1/2,m,n}^{-} = -D_{l,m,n}^g \frac{\overline{\phi}_u^g\Big|_{l+1/2,m,n}^{-} - \overline{\overline{\phi}}_{l,m,n}^g}{h_l^u/2}$$

$$\overline{J}_u^g\Big|_{l+1/2,m,n}^{+} = -D_{l+1,m,n}^g \frac{\overline{\overline{\phi}}_{l+1,m,n}^g - \overline{\phi}_u^g\Big|_{l+1/2,m,n}^{+}}{h_{l+1}^u/2},$$

where h_l^u is the width of a cell in the u direction for any cell with arbitrary index l. The first constraint place on these equations is that we have continuity of the surface flux,

$$\overline{\phi}_u^g\Big|_{l+1/2,m,n}^{-} = \overline{\phi}_u^g\Big|_{l+1/2,m,n}^{+} = \overline{\phi}_{u_{l+1/2,m,n}}^g.$$

The current relations can be rewritten as

$$\overline{J}^g_u\Big|^-_{l+1/2,m,n} = -D^g_{l,m,n}\frac{\overline{\phi}^g_{u_{l+1/2,m,n}} - \overline{\overline{\phi}}^g_{l,m,n}}{h^u_l/2}$$

$$\overline{J}^g_u\Big|^+_{l+1/2,m,n} = -D^g_{l+1,m,n}\frac{\overline{\overline{\phi}}^g_{l+1,m,n} - \overline{\phi}^g_{u_{l+1/2,m,n}}}{h^u_{l+1}/2}.$$

The next constraint is that the surface current is continuous,

$$\overline{J}^g_u\Big|^-_{l+1/2,m,n} = \overline{J}^g_u\Big|^+_{l+1/2,m,n} = \overline{J}^g_{u_{l+1/2,m,n}}.$$

The current relations now become

$$\overline{J}^g_{u_{l+1/2,m,n}} = -D^g_{l,m,n}\frac{\overline{\phi}^g_{u_{l+1/2,m,n}} - \overline{\overline{\phi}}^g_{l,m,n}}{h^u_l/2}$$

$$\overline{J}^g_{u_{l+1/2,m,n}} = -D^g_{l+1,m,n}\frac{\overline{\overline{\phi}}^g_{l+1,m,n} - \overline{\phi}^g_{u_{l+1/2,m,n}}}{h^u_{l+1}/2}.$$

Now, we are left with two equations and two unknowns. We can set both equations equation to each and solve for the surface averaged flux:

$$-D^g_{l,m,n}\frac{\overline{\phi}^g_{u_{l+1/2,m,n}} - \overline{\overline{\phi}}^g_{l,m,n}}{h^u_l/2} = -D^g_{l+1,m,n}\frac{\overline{\overline{\phi}}^g_{l+1,m,n} - \overline{\phi}^g_{u_{l+1/2,m,n}}}{h^u_{l+1}/2}$$

$$h^u_{l+1}D^g_{l,m,n}\left(\overline{\phi}^g_{u_{l+1/2,m,n}} - \overline{\overline{\phi}}^g_{l,m,n}\right) = h^u_l D^g_{l+1,m,n}\left(\overline{\overline{\phi}}^g_{l+1,m,n} - \overline{\phi}^g_{u_{l+1/2,m,n}}\right)$$

$$\overline{\phi}^g_{u_{l+1/2,m,n}} = \frac{h^u_l D^g_{l+1,m,n}\overline{\overline{\phi}}^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}\overline{\overline{\phi}}^g_{l,m,n}}{h^u_l D^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}}.$$

Now that we have an expression for the surface flux we can substitute it into any of the current relations to get an expression for the net current. It is

$$\overline{J}^g_{u_{l+1/2,m,n}} = -D^g_{l,m,n}\frac{\left(\frac{h^u_l D^g_{l+1,m,n}\overline{\overline{\phi}}^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}\overline{\overline{\phi}}^g_{l,m,n}}{h^u_l D^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}}\right) - \overline{\overline{\phi}}^g_{l,m,n}}{h^u_l/2}$$

$$\overline{J}^g_{u_{l+1/2,m,n}} = -\frac{2D^g_{l,m,n}}{h^u_l}\left(\frac{h^u_l D^g_{l+1,m,n}\overline{\overline{\phi}}^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}\overline{\overline{\phi}}^g_{l,m,n}}{h^u_l D^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}} - \frac{h^u_l D^g_{l+1,m,n}\overline{\overline{\phi}}^g_{l,m,n} + h^u_{l+1}D^g_{l,m,n}\overline{\overline{\phi}}^g_{l,m,n}}{h^u_l D^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}}\right)$$

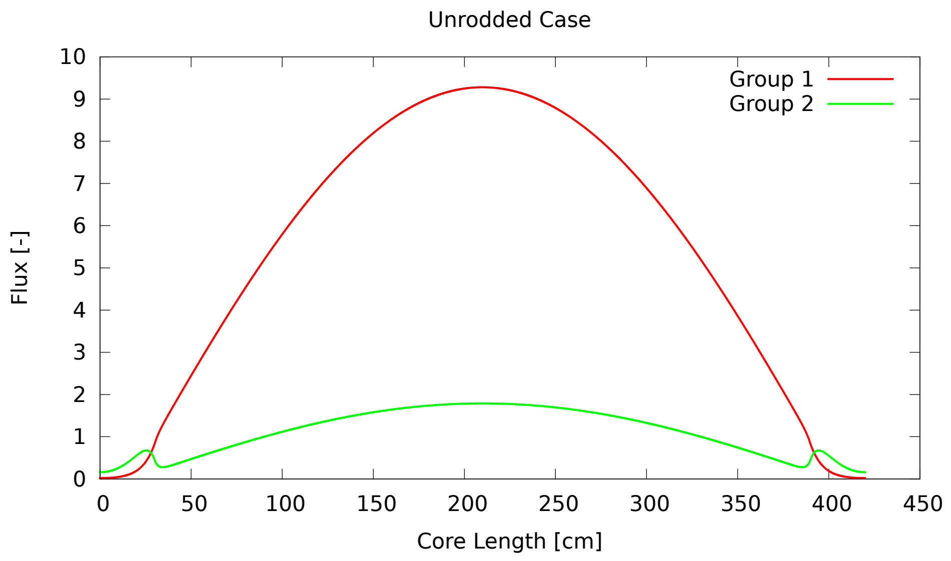
$$\overline{J}^g_{u_{l+1/2,m,n}} = -\frac{2D^g_{l,m,n}}{h^u_l}\left(\frac{h^u_l D^g_{l+1,m,n}\overline{\overline{\phi}}^g_{l+1,m,n} - h^u_l D^g_{l+1,m,n}\overline{\overline{\phi}}^g_{l,m,n}}{h^u_l D^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}}\right)$$

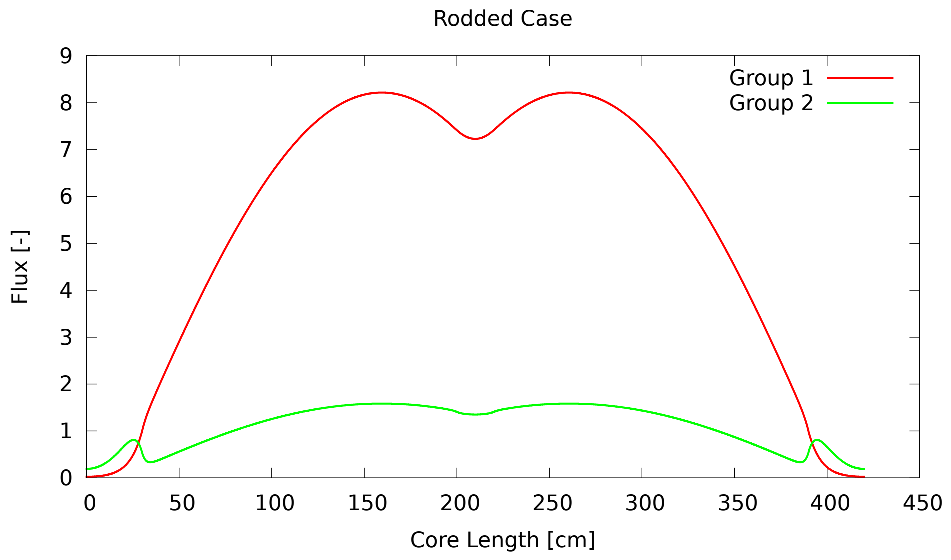
$$\boxed{\overline{J}^g_{u_{l+1/2,m,n}} = -\frac{2D^g_{l,m,n}D^g_{l+1,m,n}}{h^u_l D^g_{l+1,m,n} + h^u_{l+1}D^g_{l,m,n}}\left(\overline{\overline{\phi}}^g_{l+1,m,n} - \overline{\overline{\phi}}^g_{l,m,n}\right).}$$

Part B - Spatial Convergence

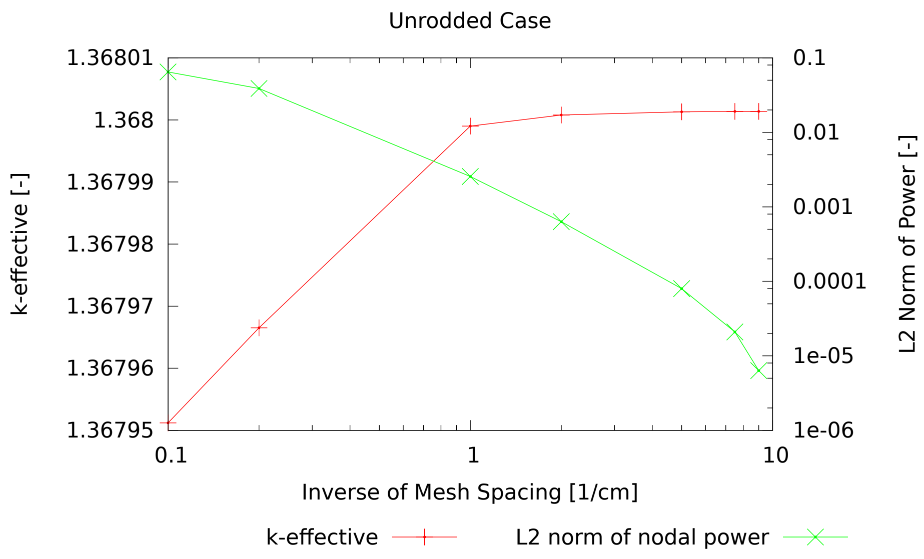
Plot the iteratively-converged eigenvalue and L2 norm of nodal power error (using 10 cm nodes) vs. mesh spacing until the L2 norm of error is converged to $<$ 1e-6 for the rodded and unrodded cores.

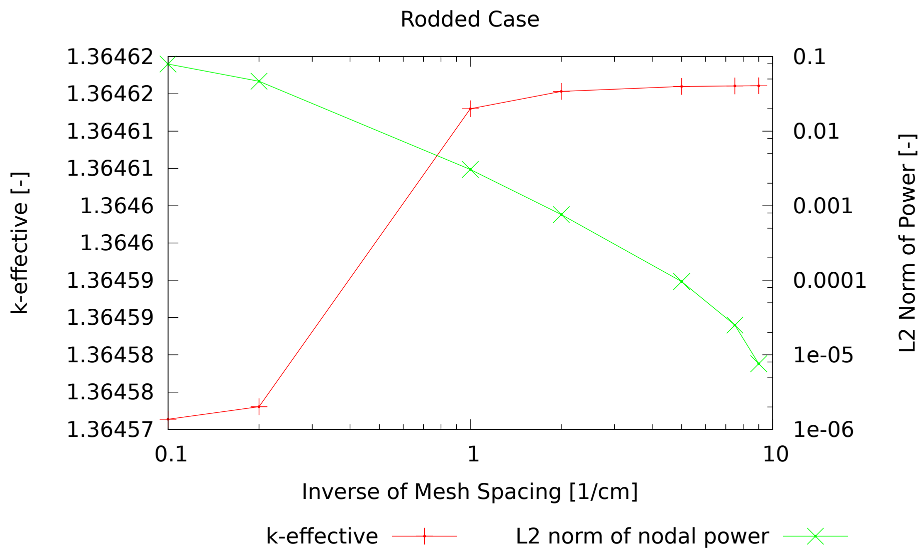
The reference spatial distribution chosen was 0.1 cm mesh. This was chosen due to computational requirements (the code took over an hour to run!). The eigenvalue tolerance was set at 1e-8, the coarse mesh power convergence at 1e-9 and the G-S inner iteration tolerance was set at 1e-10. The rodded and unrodded reference spatial flux distributions are shown below.





Spatial convergence plots were generated against this reference. Unfortunately, the target L2 norm of 1e-6 could not be met with the computer resources at hand. However, the way this norm is defined is arbitrary. I decided to look at the L2 norm for 10 cm coarse mesh powers that were normalized to a core average of unity. Depending on this normalization, a different error can be achieved. The unrodded and rodded plots are shown below.



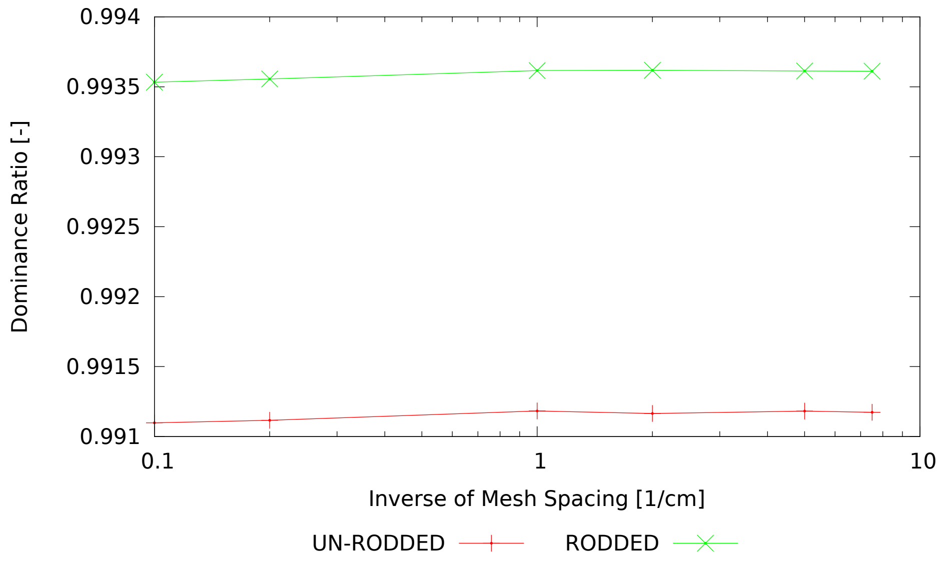


From the plots, the convergence rate is not quite second order. Also, because we get close to the reference spatial distribution a very steep slope of the convergence exists at the end. For the purposes of this homework and time, a 0.2 cm mesh will be used in subsequent analyses.

Part C - Dominance Ratio

Plot the asymptotic dominance ratio vs. mesh spacing for the rodded and unrodded cores.

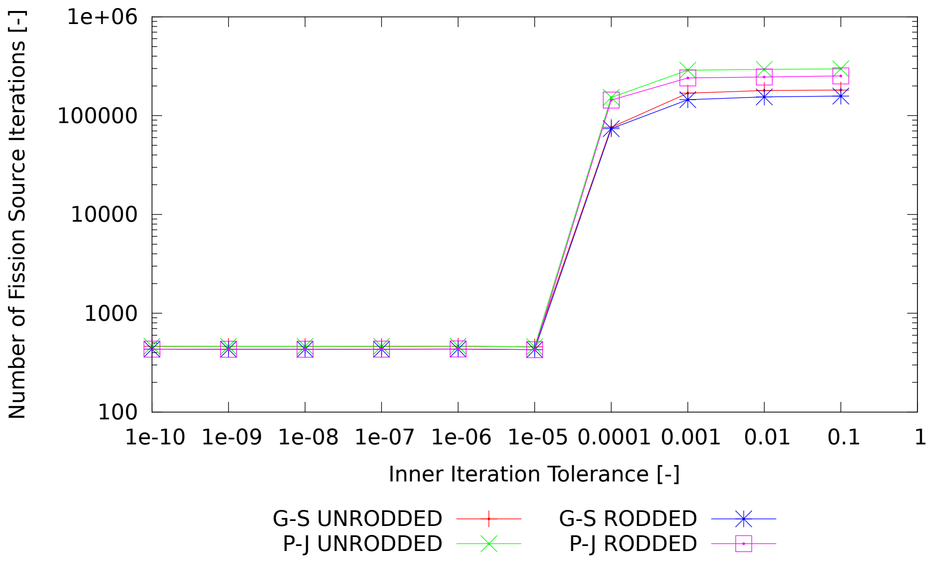
The dominance ratio is mathematically defined as the ratio of the first harmonic eigenvalue to the fundamental mode eigenvalue. Physically, the dominance ratio is impacted by geometry and material composition. Therefore, since these components remain constant when refining the mesh, we expect that the dominance ratio should not change significantly. We computed the dominance ratio as the ratio of the last L2-norm of the nodal power to the previous L2-norm of the nodal power. It agreed well with Matlab's estimate from its Arnoldi solver (`eigs`) which gives multiple eigenvalues. **Note that we started from a random source guess to excite the first harmonic.** It was also evident that the number of fission source iterations increased. Whereas before, we started from a flat symmetric guess so the power iteration converged at the asymptotic rate of the ratio of the second harmonic to the fundamental eigenvalue. The plot is shown below. From the results, the dominance ratio is not very sensitive with spatial resolution.



Part D and Part E - Iterative Convergence of Point Jacobi and Gauss Seided

Plot the number of fission source iterations needed to achieve L2 norm of changes of nodal powers for successive fission source iterations $< 1.e\text{-}6$ vs. flux iteration point-wise L2 norm for flux convergence criteria of 1.e-1, 1.e-2, 1.e-3, 1.e-4, and 1.e-5 for the rodded and unrodded cores.

For this section, we are studying the effect of the tolerance on the inner iterations. The Fortran source code for all of the sparse matrix routines are listed in the Appendix including the Point Jacobi and Gauss Seidel solvers. In general, the higher the tolerance in the inner iterations, the more work the fission source iterations will have to do. The plot below sums up the results for the unrodded and rodded cases with point jacobi and gauss seidel.



As we can see, the lower the tolerance in the inner iterations, less fission source iterations occur. We also see that for loose inner convergence criteria point jacobi is always greater than gauss seidel. This could be due to the fact that there is only one inner iteration occurring during this time. For one iteration, gauss seidel converges faster and thus lead to fewer fission source iterations.

Part F: Real vs. Fluxes

For this problem, a 0.2 cm spatial mesh was used with an eigenvalue tolerance of 1e-8, nodal power tolerance of 1e-9 and inner iteration tolerance of 1e-10. The mathematical adjoint is reported here where the transpose of the forward diffusion equation operators were used.

What are the spatially and iteratively converged real and adjoint eigenvalues for the rodded and unrodded problems?

Case	Forward k_{eff}	Adjoint k_{eff}	Δk (pcm)
Unrodded	1.368001	1.368001	0
Rodded	1.364616	1.364616	0

As expect, since we took the mathematical adjoint, we achieve the same eigenvalue to the tolerance reported above.

What is the static rod worth in pcm? Since we have both the unrodded and rodded eigenvalues the static rod worth can be calculated as:

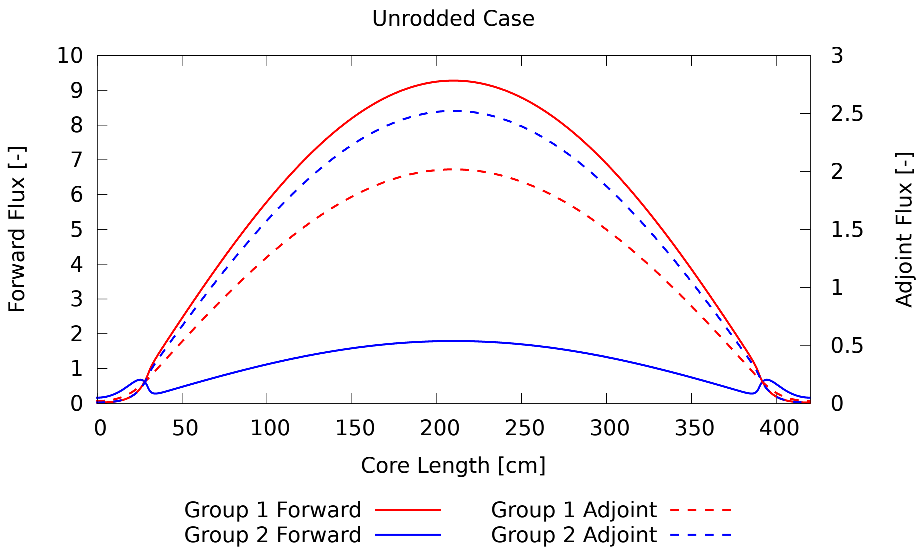
$$\rho_{unrod} = 1 - \frac{1}{k_{unrod}}$$

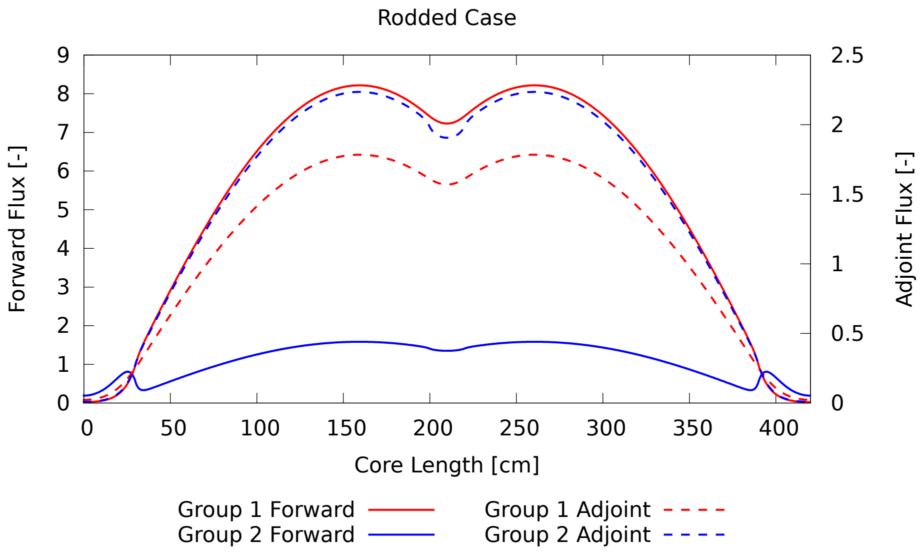
$$\rho_{rod} = 1 - \frac{1}{k_{rod}}$$

$$\Delta\rho = \rho_{unrod} - \rho_{rod} = \frac{1}{k_{unrod}} - \frac{1}{k_{rod}} \approx -180\,\text{pcm}$$

The reactivity worth of the rod in delta reactivity is -180 pcm.

Plot the spatially and iteratively converged real and adjoint fluxes for the rodded and unrodded problems The plots for the rodded and unrodded cases are shown below.





As expected, in the fuel the group 1 adjoint flux is less than the group 2 adjoint flux since fast neutrons are not as important to the overall fission rate when compared to thermal neutrons. However, in the moderator, fast neutrons slow down to the thermal range where they have much higher probability of eventually causing a fission. We see this effect on both graphs where the group 1 adjoint flux becomes greater than than group 2 adjoint flux. For the rodded case, we see the same dip in the adjoint fluxes since neutrons in the rodded region have less of a chance of causing fission.

Example Input File

```
<?xml version="1.0" encoding="UTF-8"?>
<input>

<!-- Definition of Geometry -->
  <geometry>
    <nx>6</nx>
    <ny>6</ny>
    <nz>1</nz>
    <ng>2</ng>
    <mat>
         2 1 2 3 3 5
         1 1 1 3 3 5
         2 1 2 3 3 5
         3 3 3 4 5 5
         3 3 3 5 5 5
         5 5 5 5 5 5
    </mat>
    <reg>
         1  2  3  4  5  6
         7  8  9  10 11 12
         13 14 15 16 17 18
         19 20 21 22 23 24
         25 26 27 28 29 30
         31 32 33 34 35 36
    </reg>
    <xgrid>15 60 30 15 15 30</xgrid>
    <ygrid>15 60 30 15 15 30</ygrid>
    <zgrid>1</zgrid>
    <nnx>15 60 30 15 15 30</nnx>
    <nny>15 60 30 15 15 30</nny>
    <nnz>1</nnz>
    <bc> 1.0 0.0 1.0 0.0 1.0 1.0  </bc>
  </geometry>

<!-- Defition of Materials -->
  <material uid="1">
```

```
    <absxs> 0.008252 0.1003  </absxs>
    <scattxs> 0.0 0.02533 0.0 0.0 </scattxs>
    <nfissxs> 0.004602 0.1091 </nfissxs>
    <chi> 1.0 0.0 </chi>
    <diffcoef> 1.255 0.211 </diffcoef>
    <buckling> 1e-4 </buckling>
  </material>

  <material uid="2">
    <absxs> 0.007181 0.07047 </absxs>
    <scattxs> 0.0 0.02767 0.0 0.0 </scattxs>
    <nfissxs> 0.004609 0.08675 </nfissxs>
    <chi> 1.0 0.0 </chi>
    <diffcoef> 1.268 0.1902 </diffcoef>
    <buckling> 1e-4 </buckling>
  </material>

  <material uid="3">
    <absxs> 0.008002 0.08344 </absxs>
    <scattxs> 0.0 0.02617 0.0 0.0 </scattxs>
    <nfissxs> 0.004663 0.1021 </nfissxs>
    <chi> 1.0 0.0 </chi>
    <diffcoef> 1.259 0.2091 </diffcoef>
    <buckling> 1e-4 </buckling>
  </material>

  <material uid="4">
    <absxs> 0.008002 0.073324 </absxs>
    <scattxs> 0.0 0.02617 0.0 0.0 </scattxs>
    <nfissxs> 0.004663 0.1021 </nfissxs>
    <chi> 1.0 0.0 </chi>
    <diffcoef> 1.259 0.2091 </diffcoef>
    <buckling> 1e-4 </buckling>
  </material>

  <material uid="5">
    <absxs> 0.0006034 0.01911 </absxs>
    <scattxs> 0.0 0.04754 0.0 0.0 </scattxs>
    <nfissxs> 0.0 0.0 </nfissxs>
    <chi> 1.0 0.0 </chi>
    <diffcoef> 1.257 0.1592 </diffcoef>
    <buckling> 1e-4 </buckling>
  </material>

</input>
```

Inner Solver Sparse Matrix Routines

```fortran
module math

!-module options

  implicit none
  private
  public :: sort_csr, csr_matvec_mult, csr_jacobi, csr_gauss_seidel

contains

!===============================================================================
! SORT
!===============================================================================

  recursive subroutine sort_csr(row, col, val, first, last)

!---arguments

    integer :: row(:)
    integer :: col(:)
    integer :: first
    integer :: last
    real(8) :: val(:)

!---local variables

    integer :: mid

!---begin execution

    if (first < last) then
      call split(row, col, val, first, last, mid)     ! split it
      call sort_csr(row, col, val, first, mid-1)      ! sort left half
      call sort_csr(row, col, val, mid+1, last)       ! sort right half
    end if

  end subroutine sort_csr

!===============================================================================
! SPLIT
!===============================================================================

  subroutine split(row, col, val, low, high, mid)

!---arguments
```

```
    integer :: row(:)
    integer :: col(:)
    integer :: low
    integer :: high
    integer :: mid
    real(8) :: val(:)

!g---local variables

    integer :: left
    integer :: right
    integer :: iswap
    integer :: pivot
    integer :: row0
    real(8) :: rswap
    real(8) :: val0

!---begin execution

    left = low
    right = high
    pivot = col(low)
    row0 = row(low)
    val0 = val(low)

    ! repeat the following while left and right havent met
    do while (left < right)

      ! scan right to left to find element < pivot
      do while (left < right .and. col(right) >= pivot)
        right = right - 1
      end do

      ! scan left to right to find element > pivot
      do while (left < right .and. col(left) <= pivot)
        left = left + 1
      end do

      ! if left and right havent met, exchange the items
      if (left < right) then
        iswap = col(left)
        col(left) = col(right)
        col(right) = iswap
        iswap = row(left)
        row(left) = row(right)
        row(right) = iswap
        rswap = val(left)
        val(left) = val(right)
        val(right) = rswap
      end if

    end do

    ! swith the element in split position with pivot
    col(low) = col(right)
    col(right) = pivot
    mid = right
    row(low) = row(right)
    row(right) = row0
    val(low) = val(right)
    val(right) = val0

  end subroutine split

!===============================================================================
! CSR_MATVEC_MULT
!===============================================================================

  function csr_matvec_mult(row,col,val,x,n) result(y)

!---external references

    use constants,  only: ZERO

!---arguments

    integer :: n
    integer :: row(:)
    integer :: col(:)
    real(8) :: val(:)
    real(8) :: x(n)
    real(8) :: y(n)

!---local variables

    integer :: i
    integer :: j

!---begin execution

    ! begin loop around rows
    ROWS: do i = 1, n
```

```
      ! initialize target location in vector
      y(i) = ZERO

      ! begin loop around columns
      COLS: do j = row(i), row(i+1) - 1

        y(i) = y(i) + val(j)*x(col(j))

      end do COLS

    end do ROWS

  end function csr_matvec_mult

!===============================================================================
! CSR_JACOBI
!===============================================================================

  subroutine csr_jacobi(row,col,val,diag,x,b,n,nz,tol,iter)

!---external arguments

    use constants,  only: ZERO, ONE
    use global,     only: geometry

!---arguments

    integer, intent(in)     :: n
    integer, intent(in)     :: nz
    integer, intent(inout)  :: iter
    integer, intent(in)     :: row(n+1)
    integer, intent(in)     :: col(nz)
    integer, intent(in)     :: diag(n)
    real(8), intent(in)     :: val(nz)
    real(8), intent(inout)  :: x(n)
    real(8), intent(in)     :: b(n)
    real(8), intent(in)     :: tol

!---local variables

    integer :: i,j,k,g
    integer :: irow, icol
    real(8) :: sum2
    real(8) :: norm
    real(8) :: vol = ONE
    real(8), allocatable :: tmp(:)
```

```fortran
!---begin execution

    ! allocate temp
    allocate(tmp(n))

    ! start counter
    iter = 1

    ! loop until converged
    do while(iter <= 1000000)

      ! init norm sum
      sum2 = ZERO

      ! begin loop over rows
      do irow = 1, n

        ! initialize y
        tmp(irow) = ZERO

        ! loop over columns in that row but skip diagonal
        do j = row(irow), row(irow+1) - 1

          ! continue if this diagonal element
          if (j == diag(irow)) then
            cycle
          end if

          tmp(irow) = tmp(irow) + val(j)*x(col(j))

        end do

        ! subtract RHS value
        tmp(irow) = b(irow) - tmp(irow)

        ! divide by diagonal
        tmp(irow) = tmp(irow)/val(diag(irow))

        ! get region number
        vol = geometry % fvol_map(ceiling(real(irow)/real(geometry%nfg)))

        ! sum the difference
        sum2 = sum2 + vol*(tmp(irow) - x(irow))**2

      end do

      ! compute point-wise L2 norm
      norm = sqrt(sum2)

      ! set all temp x to x
      x = tmp

      ! check convergence
      if (norm < tol) exit

      ! increase counter
      iter = iter + 1

    end do

    deallocate(tmp)

  end subroutine csr_jacobi

!===============================================================================
! CSR_GAUSS_SEIDEL
!===============================================================================

  subroutine csr_gauss_seidel(row,col,val,diag,x,b,n,nz,tol,iter)

!---external arguments

    use constants,  only: ZERO, ONE
    use global,     only: geometry

!---arguments

    integer, intent(in)      :: n
    integer, intent(in)      :: nz
    integer, intent(inout)   :: iter
    integer, intent(in)      :: row(n+1)
    integer, intent(in)      :: col(nz)
    integer, intent(in)      :: diag(n)
    real(8), intent(in)      :: val(nz)
    real(8), intent(inout)   :: x(n)
    real(8), intent(in)      :: b(n)
    real(8), intent(in)      :: tol

!---local variables

    integer :: irow, icol
    integer :: i, j, k, g
    integer :: idx
    real(8) :: sum2
    real(8) :: norm
    real(8) :: vol=ONE
    real(8), allocatable :: tmp(:)
    real(8), allocatable :: tmp1(:)

!---begin execution

    ! allocate temp
    allocate(tmp(n))

    ! start counter
    iter = 1

    ! loop until converged
    do while(iter <= 1000000)

      ! set norm sum to zero
      sum2 = ZERO

      ! begin loop over rows
      do irow = 1, n

        ! initialize y
        tmp(irow) = ZERO

        ! loop over columns in that row but skip diagonal
        do icol = row(irow), row(irow+1) - 1

          ! continue if this diagonal element
          if (icol == diag(irow)) then
            cycle
          end if

          tmp(irow) = tmp(irow) + val(icol)*x(col(icol))

        end do

        ! subtract RHS value
        tmp(irow) = b(irow) - tmp(irow)

        ! divide by diagonal
        tmp(irow) = tmp(irow)/val(diag(irow))

        ! get region number
        idx = ceiling(real(irow)/real(geometry%nfg))
        vol = geometry % fvol_map(idx)
        ! sum for norm
        sum2 = sum2 + vol*(tmp(irow) - x(irow))**2

        ! set this value in x
        x(irow) = tmp(irow)

      end do

      ! compute point-wise L2 norm
      norm = sqrt(sum2)

      ! check convergence
      if (norm < tol) exit

      ! increment counter
      iter = iter + 1

    end do

    deallocate(tmp)

  end subroutine csr_gauss_seidel

end module math
```