Problem Set 1
22.S904 Nuclear Reactor Kinetics

Due: 17 September 2011

Bryan Herman

PART A

An 8-group point kinetics code was developed using Fortran. The code uses an xml-based user input where the user can input a reactivity and time profile. Also, each coarse time mesh can have an associated time step. The standard point kinetics equations (constant kinetics parameters, no external source) solved in the code are listed below:

$$\frac{d}{dt}T\left(t\right) = \frac{\rho\left(t\right) - \beta}{\Lambda}T\left(t\right) + \sum_{i}\lambda_{i}C_{i}\left(t\right)$$

$$\frac{d}{dt}C_{i}\left(t\right) = \frac{\beta_{i}}{\Lambda}T\left(t\right) - \lambda_{i}C_{i}\left(t\right).$$

Note that this gives us 9 equations, 1 for the power and 8 for the precursors. The steady state relationship between power and precursor can be determined by setting the time derivative in the precursor equation to zero. This yields

$$C_{i}^{0} = \frac{\beta_{i}}{\Lambda\lambda_{i}}T^{0}.$$

In this code the power always starts from unity (fraction of nominal power) where the reactivity is initially zero. Setting up a linear system of these 9 equations yields

$$\frac{d}{dt}\left[N\left(t\right)\right] = \frac{d}{dt}\begin{bmatrix} T\left(t\right) \\ C_{1}\left(t\right) \\ C_{2}\left(t\right) \\ C_{3}\left(t\right) \\ C_{4}\left(t\right) \\ C_{5}\left(t\right) \\ C_{6}\left(t\right) \\ C_{7}\left(t\right) \\ C_{8}\left(t\right) \end{bmatrix} = \begin{bmatrix} \frac{\rho(t)-\beta}{\Lambda} & \lambda_{1} & \lambda_{2} & \lambda_{3} & \lambda_{4} & \lambda_{5} & \lambda_{6} & \lambda_{7} & \lambda_{8} \\ \frac{\beta_{1}}{\Lambda} & -\lambda_{1} & & & & & & & \\ \frac{\beta_{2}}{\Lambda} & & -\lambda_{2} & & & & & & \\ \frac{\beta_{3}}{\Lambda} & & & -\lambda_{3} & & & & & \\ \frac{\beta_{4}}{\Lambda} & & & & -\lambda_{4} & & & & \\ \frac{\beta_{5}}{\Lambda} & & & & & -\lambda_{5} & & & \\ \frac{\beta_{6}}{\Lambda} & & & & & & -\lambda_{6} & & \\ \frac{\beta_{7}}{\Lambda} & & & & & & & -\lambda_{7} & \\ \frac{\beta_{8}}{\Lambda} & & & & & & & & -\lambda_{8} \end{bmatrix}\left[N\left(t\right)\right].$$

We can write this simply in matrix notation as

$$\frac{d}{dt}\left[N\left(t\right)\right] = \left[A\left(t\right)\right]\left[N\left(t\right)\right].$$

We can integrate this analytically as

$$\left[N\left(t\right)\right] = \left[N\left(0\right)\right]\exp\left\{\int_0^{t'}\left[A\left(t'\right)\right]dt'\right\}.$$

We can make an assumption that we take the matrix inside the integrate at time t if it does not rapidly change. Therefore, the final equation is

$$\left[N\left(t\right)\right] = \left[N\left(0\right)\right]\exp\left\{\left[A\left(t\right)\right]dt\right\}.$$

The exponential term on the right hand side is known as the *matrix exponential*. There are many ways to approximate this. In this Fortran implementation we use a Pade approximation. The solution algorithm is presented in the next section. We can discretize the system for any time step, k, as

$$\boxed{\left[N^k\right] = \left[N^{k-1}\right]\exp\left\{\left[A^k\right]dt^{k-1}\right\}.}$$

The source code to the main point kinetics is listed in the Appendix.

Solving the Matrix Exponential - Pade Approximation

The theory for the approximation of the Pade Approximation presented here is summarized from *Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later*. The Pade approximation of an exponential, $\exp\left(A\right)$ is defined as

$$R_{pq}\left(A\right) = \left[D_{pq}\left(A\right)\right]^{-1}E_{pq}\left(A\right).$$

The numerator and denominator are defined by

$$E_{pq}\left(A\right) = \sum_{j=0}^{q}\frac{(p+q-j)!p!}{(p+q)!j!\,(p-j)!}A^j$$

and

$$D_{pq}\left(A\right) = \sum_{j=0}^{q}\frac{(p+q-j)!p!}{(p+q)!j!\,(p-j)!}\left(-A^j\right).$$

Before the Pade approximation is made, the method of *scaling and squaring* is used to avoid numerical roundoff error. To scale the exponential, it can be written as

$$\exp\left(A\right) = \left[\exp\left(A/m\right)\right]^m.$$

The criterion to choose a parameter like m is that

$$\left\|A/2^{e-1}\right\|_\infty \leq 1/2.$$

The integer value of e can be easily calculation and will be used at the end of the approximation to undo the scaling. After the matrix is scaled, the Pade approximation can be made. From above, the Pade approximation coefficients have numerous factorials and can be costly to compute. For example if we take this out to 4 terms and compute the diagonal approximate where, $p = q = 6$, we can see that it is

$$\sum_{j=0}^{q} \frac{(p+q-j)!p!}{(p+q)!j!\,(p-j)!} = 1 + \frac{11!6!}{12!5!} + \frac{10!6!}{12!2!4!} + \frac{9!6!}{12!3!3!} + ...$$

We can simply some of the factorials as

$$1 + \frac{6}{12} + \frac{6 \cdot 5}{12 \cdot 11 \cdot 2} + \frac{6 \cdot 5 \cdot 4}{12 \cdot 11 \cdot 10 \cdot 3 \cdot 2} + ...$$

What we see from this is that each succeeding term has the terms of the previous value when greater than the second term. Thus, we can always factor out all of the preceeding coefficients,

$$1 + \frac{6}{12} + \left(\frac{6}{12}\right)\frac{5}{11 \cdot 2} + \left(\frac{6}{12}\right)\left(\frac{5}{11 \cdot 2}\right)\frac{4}{10 \cdot 3} + ...$$

Therefore for every term, we will now the multiplication of the terms in parathesis from the previous coefficient. We just need to write an algorithm for the terms in fractions. We notice that starting from $j = 1$ term, the value of the numerator has to do with q. The numerator can always be computed as $num = q - j + 1$. The denominator is always made up of two terms. One of them is the term number, j. The other is just $2q - j + 1$. Therefore we can compute any Pade coefficient, c_j with the following algorithm:

$$c_j = c_{j-1}\frac{q - j + 1}{j \cdot (2q - j + 1)}, \quad j = 1, 2, ...$$

Once the numerator, E_{qq} and denominator D_{qq} are computed the Pade approximation of them is just the division of the matrices. The last step is undo the scaling that was done at the beginning of the algorithm. This can be done with

$$\exp(A) \approx R_{qq}^{s}.$$

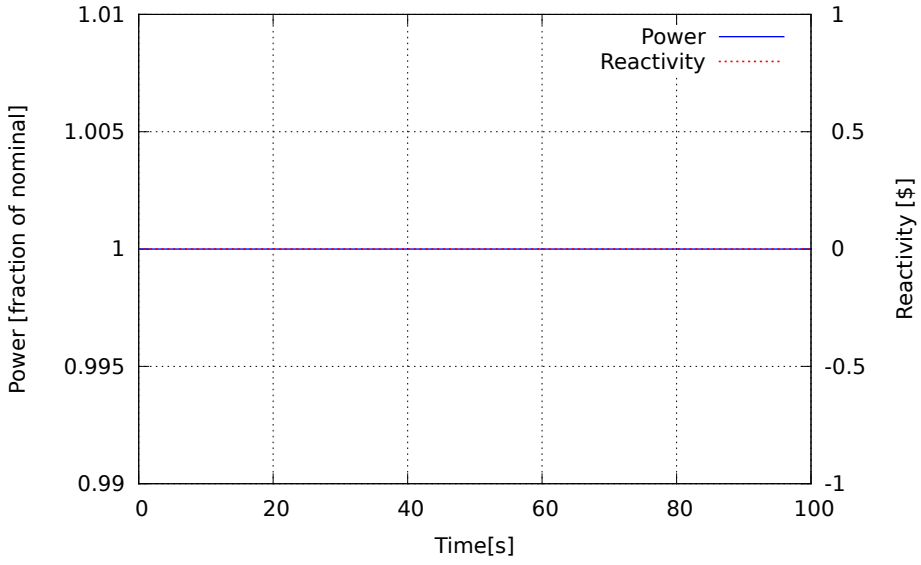
The algorithm of this method is presented below. The acutal Fortran source code written is available in the Appendix. For all complicated matrix operations, LAPACK routines were used.

Algorithm 1 Matrix Exponential - Pade Approximation with Squaring and Scaling

1: compute norm-inf of matrix, $\|A\|_\infty$
2: get scaling parameter (s): determine e such that $norm = f \cdot 2^e$, $s = \max(0, e+1)$
3: initialize Pade numerator (E) and denominator (D) with a diagonal of ones (takes care of zeroth term)
4: initialize Pade coefficient, $c = 1$
5: **for** $j = 1, 2, 3, ...q$ **do**
6: Compute Pade coefficient, c=c*(q-j+1)/(j*(2(q-j+1))
7: Perform matrix multiplication for A^j
8: Compute numerator, $E = E + c * A^j$
9: Compute denominator, $D = D(+/-)c * A^j$
10: **end for**
11: Compute approximate, $R = D^{-1}E$
12: Undo scaling, $R = R^s$

Steady State Results

With any transient code, it is important to check that if the system starts in equilibrium and not perturbation is made to the system, the values do not change. This was simulated in the point kinetics code where a reactivity curve was input of all zeros. Therefore, we expect that the fractional power stays constant at zero. Below is a plot that shows the code can hold steady state.

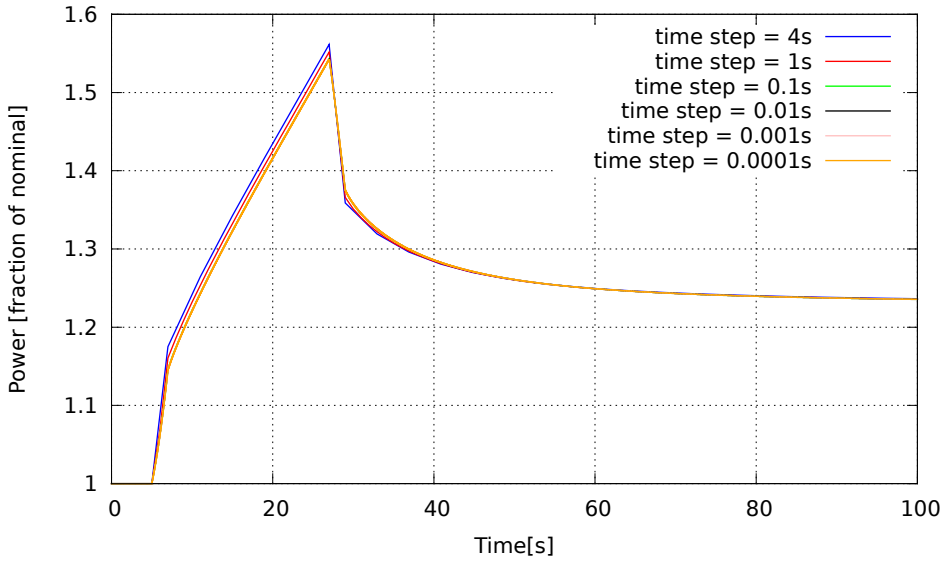


PART B

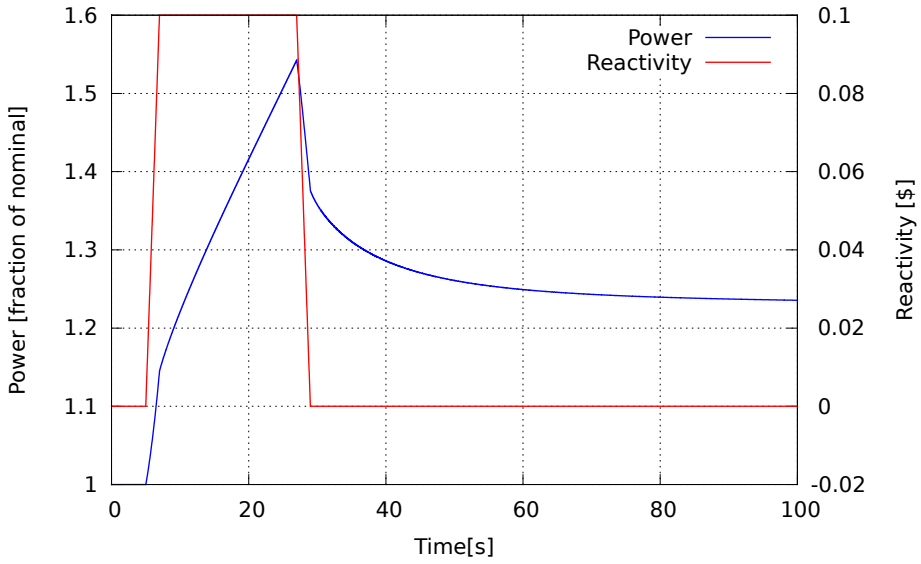
From the output of the code, the total value of beta and pnl is:

$$\boxed{\beta = 0.006648 \qquad \Lambda = 1.866 \times 10^{-5}\,\text{s}.}$$

The output of a point kinetics run can be viewed in the Appendix. The plot below shows fraction power vs. time during the transient for different time steps.



The plot above shows that once the time step becomes 0.1 seconds it very hard to visually see any differences. To be conservative we use a time step of 0.01 seconds. This time step is shown below with the corresponding reactivity input curve.



PART C

In this part we are tasked with writing an inverse point kinetics tool. In inverse kinetics, a power amplitude shape is used to determine a corresponding reactivity curve. We can start the derivation looking at the precursor balance equation for group i,

$$\frac{d}{dt}C_i\left(t\right) = \frac{\beta_i}{\Lambda}T\left(t\right) - \lambda C_i\left(t\right).$$

Since we know the power dependence the equations are no longer coupled. We can solve this differential equation analytically as

$$\frac{d}{dt}C_i\left(t\right)+\lambda C_i\left(t\right)=\frac{\beta_i}{\Lambda}T\left(t\right).$$

The integrating factor is $\exp\left(\lambda_i t\right)$. Going through the analysis we can arrive at:

$$\exp\left(\lambda_i t\right)\frac{d}{dt}C_i\left(t\right)+\exp\left(\lambda_i t\right)\lambda C_i\left(t\right)=\exp\left(\lambda_i t\right)\frac{\beta_i}{\Lambda}T\left(t\right),$$

$$\frac{d}{dt}\left[\exp\left(\lambda_i t\right)C_i\left(t\right)\right]=\exp\left(\lambda_i t\right)\frac{\beta_i}{\Lambda}T\left(t\right),$$

$$\int\frac{d}{dt}\left[\exp\left(\lambda_i t\right)C_i\left(t\right)\right]dt=\int\exp\left(\lambda_i t\right)\frac{\beta_i}{\Lambda}T\left(t\right)dt,$$

$$\exp\left(\lambda_i t\right)C_i\left(t\right)=\frac{\beta_i}{\Lambda}\int\exp\left(\lambda_i t\right)T\left(t\right)dt.$$

In order to perform that integral on the right hand side the analytic shape of the power amplitude must be known. However, if the time step is small enough we can just pull it out of the integral as a constant and assume it is the average power of the time step. This is shown as

$$\exp\left(\lambda_i t\right)C_i\left(t\right)=\frac{\beta_i}{\Lambda}\bar{T}\int\exp\left(\lambda_i t\right)dt.$$

The general form of the equation is

$$\exp\left(\lambda_i t\right)C_i\left(t\right)=\frac{\beta_i}{\lambda_i\Lambda}\bar{T}\exp\left(\lambda_i t\right)+\tilde{C},$$

$$C_i\left(t\right)=\frac{\beta_i}{\lambda_i\Lambda}\bar{T}+\tilde{C}\exp\left(\lambda_i t\right).$$

The integration constant can be determine since we know from the initial power what the initial precursor concentration is. This constant can be evaluated as

$$C_i^0=\frac{\beta_i}{\lambda_i\Lambda}\bar{T}+\tilde{C},$$

$$\tilde{C}=C_i^0-\frac{\beta_i}{\lambda_i\Lambda}\bar{T}.$$

Substituting this back in to the general equation yields the final for of the precursor equation,

$$C_i\left(t\right)=\frac{\beta_i}{\lambda_i\Lambda}\bar{T}+\left(C_i^0-\frac{\beta_i}{\lambda_i\Lambda}\bar{T}\right)\exp\left(\lambda_i t\right),$$

$$C_i\left(t\right)=C_i^0\exp\left(\lambda_i t\right)+\frac{\beta_i}{\lambda_i\Lambda}\bar{T}\left[1-\exp\left(\lambda_i t\right)\right].$$

This will be solved over a time step so that the above approximations are valid. The discretized equation then becomes

$$\boxed{C_i^{k+1} = C_i^k \exp\left(\lambda_i dt^k\right) + \frac{\beta_i}{\lambda_i \Lambda} \bar{T}^k \left[1 - \exp\left(\lambda_i dt^k\right)\right].}$$

The power equation solved for reactivity is

$$\rho\left(t\right) = \frac{\Lambda}{T\left(t\right)} \frac{d}{dt} T\left(t\right) + \beta - \frac{\Lambda}{T\left(t\right)} \sum_i \lambda_i C_i\left(t\right).$$

This can be discretized using a simple backward finite difference for the derivative so that

$$\boxed{\rho^{k+1} = \frac{\Lambda}{T^{k+1}} \frac{T^{k+1} - T^k}{dt} + \beta - \frac{\Lambda}{T^{k+1}} \sum_i \lambda_i C_i^k.}$$

These physics equation were implemented into a different Fortran code. The physics routines of the code can be viewed in the Appendix.

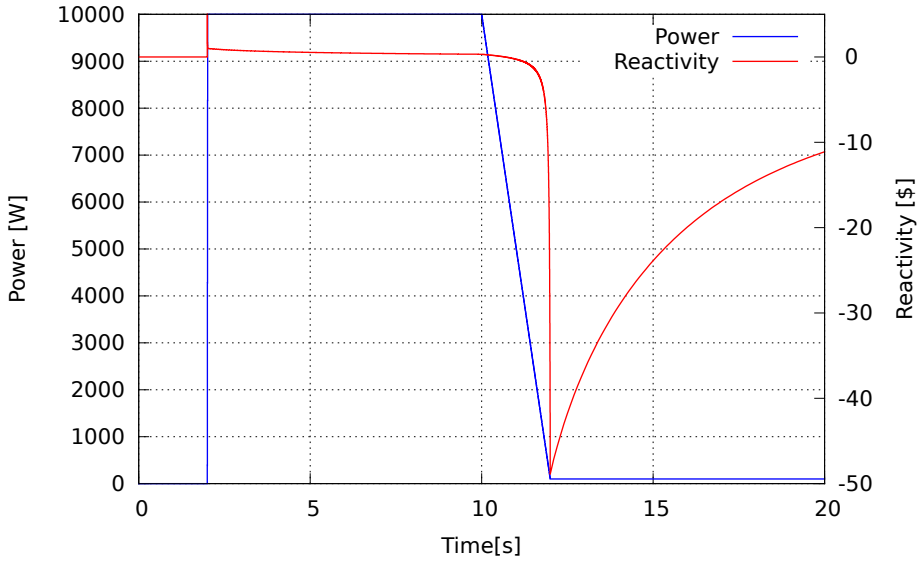
The given problem specifies a fixed power amplitude as a function of time as follows:

$$\begin{bmatrix} Time\,[\mathrm{s}] & Power\,[\mathrm{W}] \\ 0 & 1 \\ 2 & 1 \\ 2.01 & 10000 \\ 10 & 10000 \\ 12 & 100 \\ 20 & 100 \end{bmatrix}$$

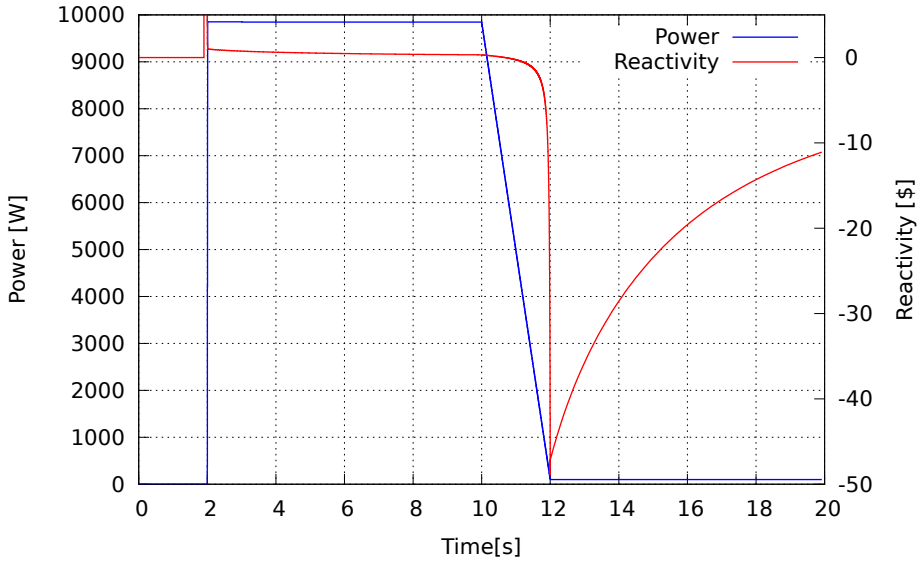
From running the code several times, we added the time step for each of these regions. To get the right shape after the reactivity insertion an extra time was put in to finer resolve the discretization:

$$\begin{bmatrix} Time\,[\mathrm{s}] & Power\,[\mathrm{W}] & Time\,step\,[\mathrm{s}] \\ 0 & 1 & 0.1 \\ 2 & 1 & 3 \times 10^{-8} \\ 2.01 & 10000 & 3 \times 10^{-5} \\ 3.0 & 10000 & 0.001 \\ 10 & 10000 & 0.0001 \\ 12 & 100 & 0.1 \\ 20 & 100 & -- \end{bmatrix}$$

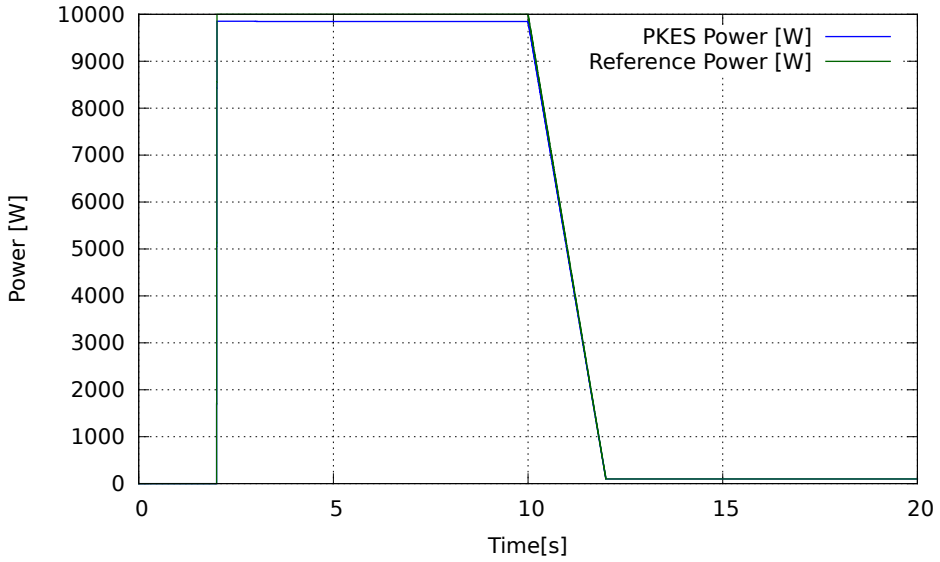
The input and output of the inverse point kinetics run is available in the Appendix. The resulting power and predicted reactivity curves are shown in the plot below.



Although the full scale of the reactivity can't be seen, the reactivity peaks at about **2700 dollars**! This is obviously ridiculous for an actual reactor system. We can then take this reactivity curve and run it in the point kinetics solver to see if the original power amplitude shape can be recovered. The input and output of this run can be viewed in the Appendix. Since the data can get large to port between the codes, a binary output file is written and can be read in by point kinetics code. The same power and reactivity plot is shown below except from the point kinetics solver.



We can see from this plot that the power doesnt quite get up to 10000 W. Other than that it matches the plot from inverse kinetics well. For comparison we plot it against the reference power from the problem statement below.



We miss the power prediction by about 1.5%. This is due to the approximations of the time derivative of the power. As we make the time steps infinitesimally small, the power should be in better agreement. This was the best prediction with the amount of time needed to run the code.

APPENDIX

Source Code

Point Kinetics Main Physics Solver

```
module physics

!-module options

  implicit none
  private
  public :: run_kinetics

contains

!=======================================================================
! RUN_KINETICS
!=======================================================================

  subroutine run_kinetics()

!---external references

    use constants,  only: NUM_PRECS
```

```
!    use expokit,     only: dense_pade
     use global,      only: pke, restart
     use output,      only: header
     use solvers,     only: expm_pade

!---local variables

     integer :: i  ! loop counter
     real(8) :: dt ! local time step

!---begin execution

     ! print header for run
     call header("POINT KINETICS SIMULATION", level=1)

     ! set up coefficient matrix
     call setup_coefmat()

     ! set up initial conditions
     call set_init()

     ! begin loop through time steps
     do i = 1, sum(pke % nt)

       ! print progress
       if (sum(pke%nt) > 100000 .and. restart) then
         if (mod(i,sum(pke%nt)/10000) == 0) write(*,*) 'On Time Step: ',i
       end if

       ! compute exponential matrix
       call set_reactivity(i,dt)

       ! solve matrix exponential
!      call dense_pade(pke % coef, NUM_PRECS + 1, dt, pke % expm) ! expokit
       call expm_pade(pke % coef, NUM_PRECS + 1, dt, pke % expm)  ! my code

       ! get new vector
       pke % N(:,i+1) = matmul(pke % expm, pke % N(:,i))

     end do

   end subroutine run_kinetics

!==========================================================================
! SETUP_COEFMAT
!==========================================================================
```

```
  subroutine setup_coefmat()

!---external references

    use constants,  only: beta, NUM_PRECS, lambda, pnl
    use global,     only: pke

!---local variables

    integer :: i ! loop counter

!---begin execution

    ! set up coeffient matrix manually for time 0
    pke % coef(1,1) = (pke % rho(1)*sum(beta) - sum(beta))/pnl

    ! begin loop around rest of matrix
    do i = 2, NUM_PRECS + 1

      ! set row 1
      pke % coef(1,i) = lambda(i - 1)

      ! set diagonal
      pke % coef(i,i) = -lambda(i - 1)

      ! set column 1
      pke % coef(i,1) = beta(i-1) / pnl

    end do

  end subroutine setup_coefmat

!=============================================================================
! SET_INIT
!=============================================================================

  subroutine set_init()

!---external references

    use constants,  only: ONE, beta, lambda, pnl, NUM_PRECS
    use global,     only: pke

!---local variables

    integer :: i ! loop counter
```

```
!---begin execution

    ! set power at 1.0
    pke % N(1,1) = ONE

    ! loop through precursors
    do i = 1, NUM_PRECS

      ! set initial value
      pke % N(i+1,1) = beta(i)/(pnl*lambda(i)) * pke % N(1,1)

    end do

  end subroutine set_init

!=======================================================================
! SET_REACTIVITY
!=======================================================================

  subroutine set_reactivity(i,dt)

!---external references

    use constants,  only: beta, pnl
    use global,     only: pke

!---arguments

    integer :: i   ! current time step
    real(8) :: dt  ! current dt

!---local variables

    integer :: idx ! interpolation index

!---begin execution

    ! check if index should be moved in input vectors
    if (i > sum(pke % nt(1:pke % idx))) pke % idx = pke % idx + 1
    idx = pke % idx

    ! compute current time
    dt = pke % dt(idx)
    pke % time(i+1) = pke % time(i) + dt

    ! interpolate on reactivity
    pke % react(i) = pke % rho(idx) + ((pke % rho(idx+1) - pke % rho(idx))  /  &
```

```
                                        (pke % t(idx + 1) - pke % t(idx)))    *  &
                                        (pke % time(i+1) - pke % t(idx))

    ! set values in coefficient matrix
    pke % coef(1,1) = (pke % react(i)*sum(beta) - sum(beta))/pnl

  end subroutine set_reactivity

end module physics
```

Inverse Point Kinetics Main Physics Solver

```
module physics

!-module options

  implicit none
  private
  public :: run_invkinetics

contains

!=======================================================================
! RUN_INVKINETICS
!=======================================================================

  subroutine run_invkinetics()

!---external references

    use constants,  only: NUM_PRECS, pnl, beta, lambda
    use global,     only: ipke, total_time
    use output,     only: header

!---local variables

    integer :: i         ! loop counter
    real(8) :: dt        ! local time step
    real(8) :: avgpower ! avg. power between timesteps

!---begin execution

    ! print header for run
    call header("INVERSE POINT KINETICS SIMULATION", level=1)

    ! set initial precursors
    ipke % N(1,1) = ipke % power(1)
```

```
    ipke % N(2:NUM_PRECS+1,1) = beta/(pnl*lambda) * ipke % N(1,1)

    ! begin loop through time steps
    do i = 1, sum(ipke % nt)

      ! compute exponential matrix
      call set_power(i,dt)

      ! compute average power
      avgpower = (ipke % N(1,i) + ipke % N(1,i+1)) / 2.0_8

      ! solve for precursors
      ipke % N(2:NUM_PRECS+1,i+1) = ipke % N(2:NUM_PRECS+1,i)*exp(-lambda*dt) +&
      beta/(lambda*pnl)*(avgpower - avgpower*exp(-lambda*dt))

      ! solve for reactivity
      ipke % react(i+1) = pnl/(ipke % N(1,i+1)) *                              &
                          ((ipke % N(1,i+1) - ipke % N(1,i))/dt) +             &
                          sum(beta) - pnl/ipke % N(1,i+1) *                    &
                          sum(lambda*ipke %N(2:NUM_PRECS+1,i+1))

      ! change to dollars
      ipke % react(i+1) = ipke % react(i+1)/sum(beta)

    end do

  end subroutine run_invkinetics

!=======================================================================
! SET_POWER
!=======================================================================

  subroutine set_power(i,dt)

!---external references

    use constants,  only: beta, pnl
    use global,     only: ipke

!---arguments

    integer :: i   ! current time step
    real(8) :: dt  ! current dt

!---local variables

    integer :: idx ! interpolation index
```

```
!---begin execution

    ! check if index should be moved in input vectors
    if (i > sum(ipke % nt(1:ipke % idx))) ipke % idx = ipke % idx + 1
    idx = ipke % idx

    ! compute current time
    dt = ipke % dt(idx)
    ipke % time(i+1) = ipke % time(i) + dt

    ! interpolate on power
    ipke % N(1,i+1) = ((ipke % power(idx+1) - ipke % power(idx))  /  &
                       (ipke % t(idx + 1) - ipke % t(idx)))   *  &
                       (ipke % time(i+1) - ipke % t(idx)) + &
                       ipke % power(idx)

  end subroutine set_power

end module physics
```

Matrix Exponential Solver - Pade Approximation

```
module solvers

!-module options

  implicit none
  private
  public :: expm_pade

contains

!=======================================================================
! EXPM_PADE
!=======================================================================

  subroutine expm_pade(A,N,dt,EXPM)

!---external references

    use constants,  only: ZERO, ONE
    use math,       only: norm_inf, ilog2

!---arguments

    integer :: N          ! dimension of square matrix
```

```
    real(8) :: A(N,N)      ! the coefficient matrix
    real(8) :: dt          ! the time step
    real(8) :: EXPM(N,N)   ! output matrix

!---local variables

    integer :: ex   ! this is the integer exponent that goes into th scaling
    integer :: s    ! this is the scaling parameter
    integer :: i    ! loop counter
    integer :: j    ! Pade order counter
    integer :: q    ! Pade approximation order
    integer :: info ! did LAPACK work?
    integer, allocatable :: IPIV(:) ! pivot elements from LU factorization
    real(8) :: norm   ! the norm of the coefficient matrix for scaling
    real(8) :: c      ! Pade coefficient
    real(8) :: shift  ! -1 or +1 and a function of J for D term
    real(8), allocatable :: At(:,:)  ! the exponential being approximated
    real(8), allocatable :: X(:,:)   ! accumulates multiplication of A
    real(8), allocatable :: Xt(:,:)  ! temporary matrix for X
    real(8), allocatable :: E(:,:)   ! numerator of Pade
    real(8), allocatable :: D(:,:)   ! denominator of Pade

!---begin execution

    ! allocate temporary matrices
    allocate(At(N,N))
    allocate(X(N,N))
    allocate(Xt(N,N))
    allocate(E(N,N))
    allocate(D(N,N))
    allocate(IPIV(N))

    ! set matrix
    At = A

    ! get matrix to be approximated in exponential
    At = At * dt

    ! compute norm for scaling
    norm = norm_inf(At,N)

    ! compute integer exponent of formula norm = f*2**(ex)
    ex = ilog2(norm)

    ! determine scaling parameter
    s = max(0,ex+1)
```

```
! scale matrix by scaling parameter
At = At / (2**s)

! begin pade approximation for numerator and denominator of rational fun.
! Pade order 6, diagonal approximates
q = 6

! can easily set j=0 and j=1 parts right away
! begin loop to set the A^0 terms in each E and D, not this is a diag of ones
E = ZERO
D = ZERO
do i=1,N
  E(i,i) = ONE
  D(i,i) = ONE
end do
c = ONE
c = 0.5_8    ! this is j=1 Pade coefficient (need to do this before loop)
E = E + c*At ! accmulate in sum
D = D - c*At ! accumulate in sum

! begin rest of pade loop
X = At       ! set A to X, X will accumulated the A^j in the Pade eq.
shift = ONE ! for order 2 term, the shift for denominator is +1
do j = 2,q

  ! compute Pade coefficient
  c = c * dble(q-j+1) / dble(j*(2*q-j+1))

  ! multiply A^(j-1) by LAPACK ROUTINE
  call DGEMM('N','N',N,N,N,ONE,At,N,X,N,ZERO,Xt,N)

  ! move temporary to actual
  X = Xt

  ! accumulate into numerator and denominator
  E = E + c*X
  D = D + shift*c*X
  shift = -ONE

end do

! compute rational function (E is R after the next two lines)
call DGETRF(N,N,D,N,IPIV,info)          ! LU factorization of D, LAPACK
call DGETRS('N',N,N,D,N,IPIV,E,N,info)  ! solves E = D^-1*E, LAPACK

! perform squaring
do i = 1,s
```

```
      call DGEMM('N','N',N,N,N,ONE,E,N,E,N,ZERO,Xt,N) ! does E*E LAPACK
      E = Xt
    end do

    ! set output matrix
    EXPM  = E  ! E is really R, didnt want to make a separate matrix

    ! deallocate temporary matrices
    deallocate(At)
    deallocate(X)
    deallocate(Xt)
    deallocate(E)
    deallocate(D)
    deallocate(IPIV)

  end subroutine expm_pade

end module solvers
```

Part B Runs

Point Kinetics Input

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

  <!-- Reactivity Shape -->
  <reactivity>
    <time> 0.0 5.0 7.0 27.0 29.0 100.0 </time>
    <rho>  0.0 0.0 0.1 0.1  0.0  0.0 </rho>
    <timestep> .01 .01 .01 .01 .01 </timestep>
  </reactivity>

</input>
```

Point Kinetics Output

```
 ______   ___   ___            ______   ______   __       __   __   ______
/_____/\ /___/\/__/\          /_____/\ /_____/\ /_/\     /_/\ /_/\ /_____/\
\:::_ \ \\::.\ \\ \ \  _______\:::::_\/_\:::_ \ \\:\ \    \:\ \\ \ \\::::_\/_
 \:(_) \ \\:: \/_) \ \/______/\\:\/___/\\:\ \ \ \\:\ \    \:\ \\ \ \\:\/___/\
  \: ___\/ \:. __  ( (\__::::\/ \_::._\:\\:\ \ \ \\:\ \____\:\_/.:\ \\::___\/_
   \ \ \    \: \ )  \ \           /____\:\\:\_\ \ \\:\/___/\\ ..::/ / \:\____/\
    \_\/     \__\/\__\/           \_____\/ \_____\/ \_____\/ \___/_(   \_____\/
```

```
      Developed At:  Massachusetts Institute of Technology
      Version:       0.1
      Git SHA1:      029eecfb04ec8c268d2584929167b6a17c1af2fc
      Date/Time:     2012-09-15 11:33:19

 ===========================================================================
 ========================>     INITIALIZATION     <=========================
 ===========================================================================

 Reading settings XML file...

 INPUT SUMMARY
 ---------------------------------------------------------------------------

Number of time steps:        10000

Time (s)           Rho ($)
--------           -------
0.000E+00          0.000E+00
5.000E+00          0.000E+00
7.000E+00          1.000E-01
2.700E+01          1.000E-01
2.900E+01          0.000E+00
1.000E+02          0.000E+00

 PHYSICS SUMMARY
 ---------------------------------------------------------------------------

Number of Precursor Groups: 8

beta               lambda
----               ------
2.180E-04          1.247E-02
1.023E-03          2.829E-02
6.050E-04          4.252E-02
1.310E-03          1.330E-01
2.200E-03          2.925E-01
6.000E-04          6.665E-01
5.400E-04          1.635E+00
1.520E-04          3.555E+00

Total Delayed Neutron Fraction  0.006648

Prompt Neutron Lifetime [s]:  1.866E-05
```

```
 ============================================================================
 ==================>     POINT KINETICS SIMULATION     <==================
 ============================================================================

Plotting results with GNUPLOT...

 SIMULATION SUMMARY
 ----------------------------------------------------------------------------

 Total simulation time            =  6.9000E-01 seconds

Simulation Finished.
```

Part C Runs

Inverse Point Kinetics Input

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

  <!-- Power Shape -->
  <time>   0.0 2.0 2.01    3.0     10.0    12.0  20.0   </time>
  <power>  1.0 1.0 10000.0 10000.0 10000.0 100.0 100.0  </power>
  <timestep> 0.1 3e-8 3e-5 1e-3 1e-4 0.1 </timestep>

</input>
```

Inverse Point Kinetics Output

```
       __  .__   __. ____    ____      .______    __  ___  _______
      |  | |  \ |  | \   \  /   /      |   _  \  |  |/  / |   ____|
      |  | |   \|  |  \   \/   / ______|  |_)  | |  "  /  |  |__
      |  | |  . `  |   \      / |______|   ___/  |    <   |   __|
      |  | |  |\   |    \    /         |  |      |  .  \  |  |____
      |__| |__| \__|     \__/          | _|      |__|\__\ |_______|

      Developed At:  Massachusetts Institute of Technology
      Version:       0.1
      Git SHA1:      6ad3e02a086a0c13c2b75274e997addef77cc7e7
      Date/Time:     2012-09-15 12:04:56

 ============================================================================
 ======================>     INITIALIZATION     <=======================
 ============================================================================
```

```
 Reading settings XML file...

 INPUT SUMMARY
 ------------------------------------------------------------------------

Number of time steps:        393433

Time (s)           Rho ($)
--------           -------
0.000E+00          1.000E+00
2.000E+00          1.000E+00
2.010E+00          1.000E+04
3.000E+00          1.000E+04
1.000E+01          1.000E+04
1.200E+01          1.000E+02
2.000E+01          1.000E+02

 PHYSICS SUMMARY
 ------------------------------------------------------------------------

Number of Precursor Groups: 8

beta               lambda
----               ------
2.180E-04          1.247E-02
1.023E-03          2.829E-02
6.050E-04          4.252E-02
1.310E-03          1.330E-01
2.200E-03          2.925E-01
6.000E-04          6.665E-01
5.400E-04          1.635E+00
1.520E-04          3.555E+00

 ========================================================================
 ===============>     INVERSE POINT KINETICS SIMULATION     <===============
 ========================================================================

Plotting results with GNUPLOT...

 SIMULATION SUMMARY
 ------------------------------------------------------------------------

 Total simulation time            =  2.4000E-01 seconds
```

```
Simulation Finished.
```

Point Kinetics Input

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

  <restart> .TRUE. </restart>

</input>
```

Point Kinetics Output

```
 ______   ___   ___            ______   ______   __       __   __   ______
/_____/\ /___/\/__/\          /_____/\ /_____/\ /_/\     /_/\ /_/\ /_____/\
\:::_ \ \\::.\ \\ \ \  _______\:::_\/_\:::_ \ \\:\ \    \:\ \\ \ \\:::_\/_
 \:(_) \ \\:: \/_) \ \/______/\\:\/___/\\:\ \ \ \\:\ \    \:\ \\ \ \\:\/___/\
  \: ___\/ \:. __  ( (\__::::\/ \_::._\:\\:\ \ \ \\:\ \____\:\_/.:\ \\::___\/_
   \ \ \    \: \ )  \ \           /____\:\\:\_\ \ \\:\/___/\\ ..::/ / \:\____/\
    \_\/     \__\/\__\/           \_____\/ \_____\/ \_____\/ \___/_(   \_____\/

      Developed At:  Massachusetts Institute of Technology
      Version:       0.1
      Git SHA1:      029eecfb04ec8c268d2584929167b6a17c1af2fc
      Date/Time:     2012-09-15 12:06:35

 ============================================================================
 ========================>     INITIALIZATION     <=========================
 ============================================================================

 Reading settings XML file...

 INPUT SUMMARY
 ----------------------------------------------------------------------------

Number of time steps:        393433

Restarted from inverse kinetics...


 PHYSICS SUMMARY
 ----------------------------------------------------------------------------

Number of Precursor Groups: 8
```

```
beta               lambda
----               ------
2.180E-04          1.247E-02
1.023E-03          2.829E-02
6.050E-04          4.252E-02
1.310E-03          1.330E-01
2.200E-03          2.925E-01
6.000E-04          6.665E-01
5.400E-04          1.635E+00
1.520E-04          3.555E+00

Total Delayed Neutron Fraction  0.006648

Prompt Neutron Lifetime [s]:  1.866E-05

 ================================================================================
 ==================>     POINT KINETICS SIMULATION     <==================
 ================================================================================

Plotting results with GNUPLOT...

 SIMULATION SUMMARY
 --------------------------------------------------------------------------------

 Total simulation time            =  4.8745E+02 seconds

Simulation Finished.
```