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 dense Pade approximation available in the external solver package *EXPOFIT*. We can discretize the system for any time step, k, as

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

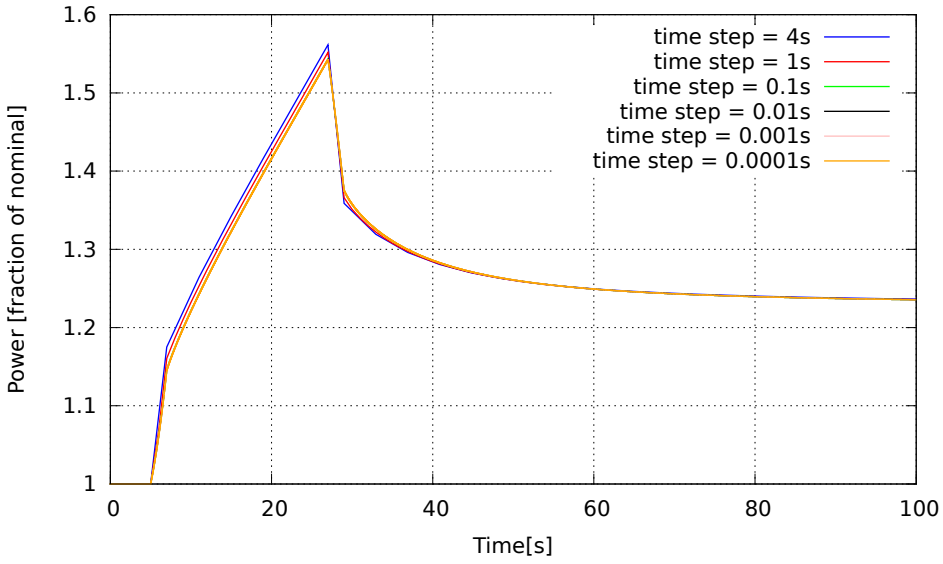
The source code to the main point kinetics is listed in the Appendix.

Part B

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

$$\beta = \qquad \Lambda = .$$

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



Source Code

Point Kinetics Main Physics Solver

```
!==========================================================================!
! MODULE: global
!
!> @author Bryan Herman
!>
!> @brief Contains all of the global variables
!==========================================================================!
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
    use output,     only: header

!---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)

      ! compute exponential matrix
      call set_reactivity(i,dt)

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

      ! 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

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("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
```