The Nuclear Reactor

An application of Monte Carlo simulation methods

This experiment concerned the accurate simulation, via Monte Carlo methods, of an infinite and finite spherical nuclear reactor of homogeneous nature. The main aim being to analyse how the proportions of constituent atoms in the reactor affected it's performace and so how to balance these proportions in order to create an efficent system. Having done this, the effect of finite size of the reactor performance could be determined. It was discovered that there is a well defined set of conditions for maximum efficiency and that the size of the reactor has a significant effect on the reactor performance at macroscopic levels.

Andrew N Jackson

9th May 1995

Introduction

The testing of a nuclear reactor is perhaps one of the best examples of the advantages computational simulation holds over physical experimentation. Once the software is written, countless experiments to examine the factors affecting a reactors performance can be carried out at little cost and with no dangerous side effects.

In this computational experiment we consider a simple reactor model consisting of carbon atoms and two isotopes of uranium (U^{235} and U^{238}), all of which are intimately mixed (ie the reactor can be considered homogenous). The nuclear chain reaction propagates as follows:

- Splitting U^{235} atoms releases high speed neutrons.
- High speed neutrons are slowed down to thermal energies by collisions with carbon atoms.
- Atoms of U^{235} absorb thermal neutrons and fission occurs, creating more high speed neutrons.

However, reality is not quite this simple, in that the travelling neutrons can be absorbed by U^{238} before it can slow right down. Indeed any atom in the reactor stands a chance of absorbing a neutron, thus stopping that neutron from going on to complete it's part of the chain reaction (although resonance absorption by U^{238} is the main culprit). Due to this possibility of neutron loss, the reactors ability to sustain the chain reaction depends strongly on the exact proportions of the different atoms in the reactor. This in turn depends on the ratio of U^{235} atoms to U^{238} atoms (how pure the uranium fuel is), and the ratio of fuel atoms to carbon atoms (the proportion of the reactor which is moderating the speed of the neutrons, and thus moderating the overall reactor performance). We measure the performance of the reactor by it's multiplication factor, k, which is defined as the average number of neutrons created by each high speed neutron in the system. Therefore, in order for the reactor to be self-sustaining, k must be greater than or equal to 1. However, as high values of k can lead to a rapidly accelerating reaction, ie a nuclear bomb, we are only interested in a reactor that is *just* self-sustaining, in other words, where it's multiplication factor is exactly equal to 1.

The aim of this experiment is to find the best balance of the fuel to moderator ratio, and the minimum level of (costly) enrichment of the fuel required to make a reactor self-sustaining.

In order to simplify the simulation, we first investigate the behaviour of an infinite reactor before going on to consider how finite size effects reactor performance.

Theory

The implementation of the Monte Carlo method for this simulation involves following the lives of a large group of neutrons in the reactor, with each neutron taking it's own random walk from release by a splitting U^{235} atom through to either absorption, exiting the reactor, or to causing further fission. If we keep count of how many fissions occurred, we can work out how many neutrons were created by our group of neutrons, and so we can work out the multiplication factor, k (as $^{\text{neutrons out}}/_{\text{neutron in}}$), for the reactor. From this it is possible to work out how the enrichment level and fuel:moderator ratio affect the reactors performance (by doing the above calculation for various configurations of reactor).

A single neutrons random walk through the reactor breaks down into 4 basic steps:

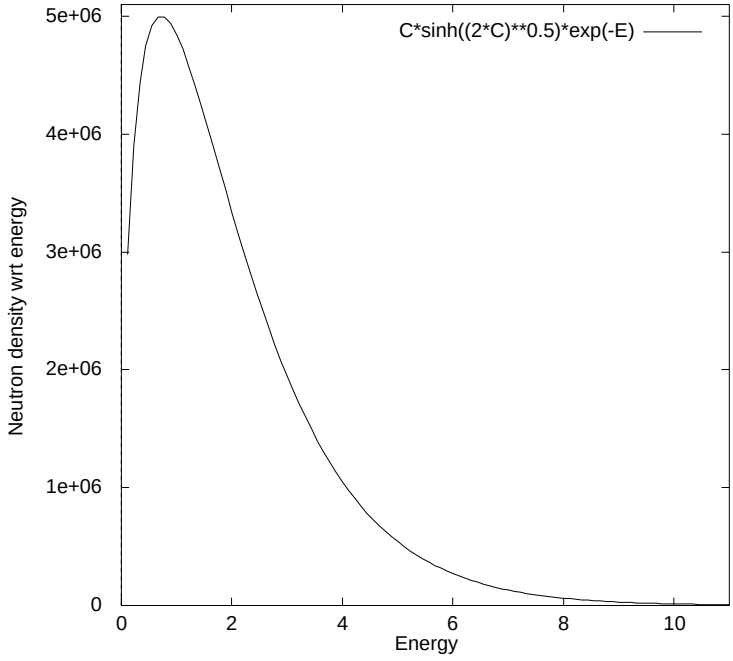
i) Neutron release by U fission

In the simulation, each neutron begins it's random walk as a product of U fission. The statistical distribution (in MeV) for this process is given by:

$$N(E) = C \sinh [\sqrt{(2E)}] e^{-E}$$

Where N(E) dE is the number of neutrons in the energy range E -> E+dE MeV. The neutrons are emitted isotropically, with a mean energy of 2MeV, and the peak of the distribution is at 0.72 MeV (see fig. ii:1).

Figure 1 : Fission neutron's energy distribution:

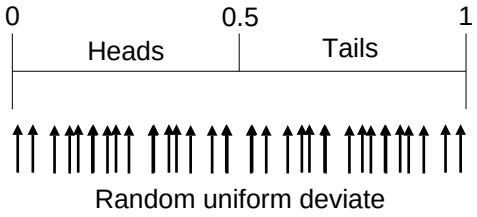


ii) Neutron slowing down

After a neutron has been emitted, it is slowed down to thermal energies(~0.025 eV) by a series of elastic collisions with moderator atoms (ie carbon). As the reactor consists of various types of atoms, for each collision we consider we must make a decision as to whether the neutron hits carbon, U^{235} or U^{238}. Indeed, having made this decision, we need to decide if the atom in question scatters or absorbs the neutron. The process of making these decisions

is based on branching probabilities, where a uniform deviate is used to decide which of all the possible outcomes occurs for this collision. If, for example, the decision to be made concerns whether a hypothetical coin comes up heads or tails, then the branching probability situation is as described in fig. ii:2. As the probability of each is $^1/_2$, we simulate throwing a coin by creating a uniformly distributed random number and seeing which range it lies in. If the number is between 0 and 0.5, we count a head, and if the number lies between 0.5 and 1, we count a tail. For a many coins simulation, this will give the result that the probability of a head or a tail is $^1/_2$. Although this is a pointlessly simple example, this method is very useful for making decisions in more complex systems, and all the decisions made for the reactor simulator work in this way.

Figure 2 : The technique of 'branching probabilities':



The actual probabilities we need to decide between come from the 'cross-sections' of each possibility, where a cross-section represents the effective area that an atoms places in the path of a particle for the appropriate interaction to take place. This usually has the symbol σ, for example; σ_a for absorption, σ_s for scattering and so on. To carry out the calculation, we require the probability per unit length, Σ, given by:

$$\Sigma = N\sigma$$

Where N is the number density of the particular type of atoms in question, which can be calculated from:

$$N = \frac{0.6025\times10^{27} \times \text{Density}}{\text{Atomic weight}}$$

This allows us to calculate the mean free path for the system by finding the total probability of any kind of collision occuring, and simply inverting it (ie dividing 1 by the result). For example, if the total probability of a collision is given by:

$$\Sigma_{total} = N_a\sigma_a + N_s\sigma_s$$

then the mean free path, usually denoted by an λ, is: $\frac{1}{\Sigma_{total}}$

Therefore, the calculation can proceed under the assumption that the neutron travels an average distance λ between each collision, and that we only have to make any decisions about the neutrons random walk at these points. To do this we need to know the branching probabilities of scattering and absorption for a collision, and what the probabilities of the atom being carbon, U^{235} or U^{238} are. These are simple to calculate, for example the probability of scattering occuring is:

$$P(scat) = \frac{\Sigma_s}{\Sigma_{total}}$$

[The cross-sections we require (along with other data) are contained in appendix A.]

As mentioned before, the principle mechanism for neutron loss is through resonance absorption by U^{238}. This is slightly more complicated than before, because while most of the cross-sections involved are independent of reactor configuration, the absorption cross-section for U^{238} depends strongly upon it (more specifically, it depends on Σ_s / N_{238}). This dependence is described by the table of (Σ_s / N_{238}) against σ_a in appendix A, and linear interpolation is used to approximate a value for σ_a from this data. From all of this we can calculate all the probabilities we need.

The whole point of the slowing down phase is for the neutrons energy to be decreased through collisions, and to be able to do this we need to know how the energy is affected by a collision. The energy loss in an elastic collision is related to the angle of deflection, ⬮, through the equation:

$$\frac{E_1}{E_0} = \frac{1 + 2A\cos\text{⬮} + A^2}{(1 + A)^2}$$

Where E_0 and E_1 are the energies before and after collision, and A is the mass number of the scattering atom. It takes around 100 collisions with carbon (low mass number, high energy exchange) to slow a neutron down to thermal energies.

iii) Neutron Diffusion

A neutron cannot cause fission until it has slowed to thermal energies (ie until it is at the same 'temperature' as the reactor). After this point, no net energy loss can occur and so we ignore that possibility as we follow the neutrons randomly scattered path, until it is absorbed (which may possibly lead to fission) or lost from the reactor altogether.

iv) Neutron production through fission

On average, a splitting U^{235} atom releases 2.47 fast neutrons. Therefore if we test N neutron lives, and n of them cause fission, then:

$$k = (2.47 \times n)/N$$

The reactor multiplication constant, k, must be greater than 1 for the reactor to be self-sustaining. However, as mentioned before, we are interested only in the reactor condition where k is equal to 1.

Method (part 1) : The Infinite Reactor

Before we can even begin to program the actual neutron life simulation, we need to assemble all of the data we need into a form accessible from FORTRAN. As the aim is to see how the enrichment (E) and fuel to moderator ratio (f:m) factors affect the reactor, we need to keep in mind that we will use the program to analyse their effect later. This means that the data splits into two sections; the constant parameters, that always remain the same, and the variable parameters, whose value in some way depends of E and f:m. When creating these constants, and indeed when naming all program variables, I have used a categorisation procedure such that those names beginning with the letters 'a'-'h' or 'j'-'z' refer to double precision data (unless otherwise stated). The 'i' prefix excluded in the above definition is used to identify integer variables.

The constant parameter section consists of most of the data from appendix A, ie the thermal energy value (0.025 eV), the densities and atomic weights of carbon and uranium, the scattering, absorption and absorption leading to fission cross-sections for the three isotopes present (C^{12}, U^{235} and U^{238}), and a conversion factor to convert figures in barns to metres (defined as 1 barn = 10^{-28} m). The second section is mainly calculations for number densities and probabilities. Since the uranium fractions are very small, the number densities can be worked out as follows:

$$N_{12} = \frac{0.6025\times10^{27} \times [\text{Density}]_{\text{carbon}}}{[\text{Atomic Weight}]_{\text{carbon}}}$$

$$N_{235} = F \times N_{12}$$

$$N_{238} = (1 - F) \times N_{12}$$

Where F is the ratio of U^{238} to U^{235}, defined from the enrichment factor multiplied by the natural U^{238}:U^{235} ratio ($\sim{}^{1}/_{138}$). The rest of this section, which concerns cross-sections and probabilities, is mostly straightforward. The only complication is that the absorption cross-section of U^{238} has to be looked up from the tabulated data shown in appendix A. This simply involves a series of if statements and a set of calls to a small double-precision interpolation procedure (called 'dinterp') which, for a function y=f(x), works as follows: If the x interpolation range is x_{low} to x_{high} with corresponding y range y_{low} to y_{high}, then given a value for x inbetween, (called x_{mid}) an approximate value for y_{mid} can be found from:

$$y_{mid} = \frac{(x_{mid} - x_{low})}{(x_{high} - x_{low})} \times (y_{high} - y_{low}) + y_{low}$$

There is, unfortunately, a further complication, in that the table includes an interpolation between 2000 and infinity. This is obviously impossible using the formula above, and finding the correct modification of the formula caused me some problems. At first I though that the problem could be solved by adding an inversing function to all parameters, ie:

$$y_{mid} = \left[\frac{(x_{mid}^{-1} - x_{low}^{-1})}{(x_{high}^{-1} - x_{low}^{-1})} \times (y_{high}^{-1} - y_{low}^{-1}) + y_{low}^{-1} \right]^{-1}$$

This, however, is not the correct method, and later caused my program to produce incorrect results (in comparison to the data from Professor Pert's program). I later discovered my

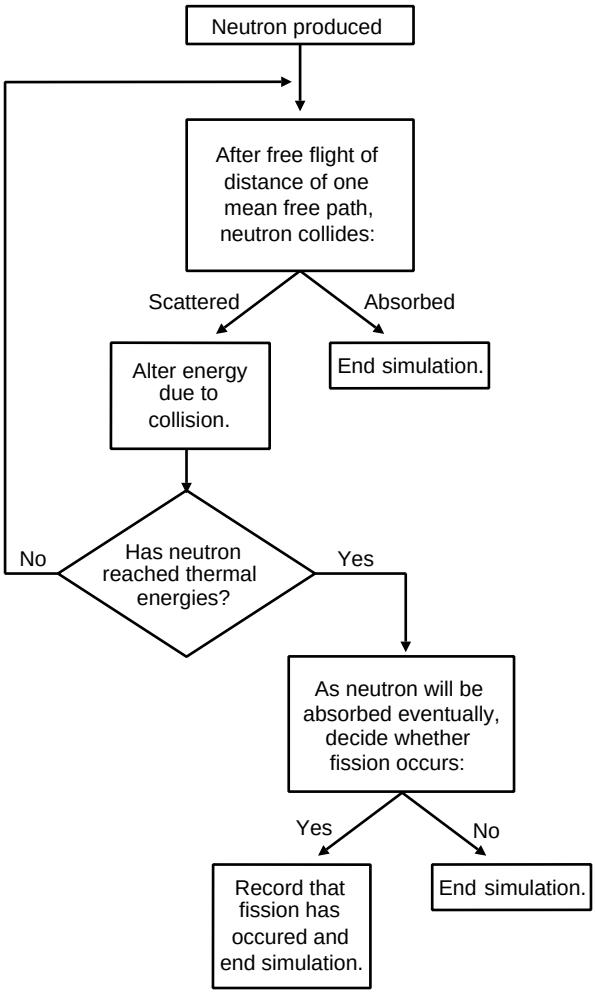
mistake and found that the method only requires you to fit an inverting function onto the x-range part of the data, as below:

$$y_{mid} = \frac{(x_{mid}^{-1} - x_{low}^{-1})}{(x_{high}^{-1} - x_{low}^{-1})} \times (y_{high} - y_{low}) + y_{low}$$

Having found all the neccessary cross-sections and number densities as, it is possible to go on to calculate the probabilities of scattering, absorption and absorption leading to fission at thermal and non-thermal temperatures. Once this is done, we are ready to go on to the simulation itself.

As mentioned before, the reactor simulation consists of running many neutron lives through the reactor and working out the reactors performance from the average fates of these neutrons lives. This means that the most important element of this experiment is the correct simulation of a single neutrons random walk. The program structure needed to do this is illustrated in figure 3 below.

Figure 3 : Flow chart for infinite reactor simulation:



Before we can implement this structure, we require two routines: A random number generator, 'ran1', and the random energy deviate generator, 'disdev' (see fig. 1). Once these have been appended to the code, the first step (neutron production) consists of simple a call to 'disdev' to pick an energy for the current neutron. The free flight/collision loop part requires one branching probability decision to be made, and based on this decision, an energy alteration

and energy check. Once the neutron is thermal it simply diffuses through the reactor, and as this is an infinite medium, it is sure to be absorbed eventually, and so we can assume that it is absorbed and work another branching probability for the chances of fission occuring under those conditions.

Once implemented, the code has the form shown in code fragment 1.

Code Fragment 1 : The branching probability based infinite reactor.

```
c Get energy, converting to eV:
      Energy=disdev(idum)*1.0d6
      iEndLife=0
c Loop over one neutron life.
 666  IF (Energy.GT.ThermSwitch) THEN
c Scatter or absorb:
       Rand=RAN1(idum)
C If absorbed:
       IF (Rand.LE.AbsPrb) THEN
       iEndLife=1
       ELSE
C Else scattered(was it carbon? If not ignore collision):
       Rand=Rand-AbsPrb
       IF (Rand.LE.CarbScat) THEN
        Costheta=-1+2*RAN1(idum)
        AtW=AtWC
        NewEnrg=Energy*(1+2*AtW*Costheta+AtW*AtW)/((1+AtW)*(1+AtW))
c       END IF
       END IF
      ELSE
c in thermal range so; Absorb or fission:
       Rand=RAN1(idum)
c If fission:
       IF (Rand.LE.FissPrb) THEN
       iFissions=iFissions+1
       iEndLife=1
       ELSE
c Else absorbed:
       iEndLife=1
       END IF
      END IF
      Energy=NewEnrg
      IF (iEndLife.NE.1) GOTO 666
```

Where 'AbsPrb' is the probability of absorption out of either absorption or scattering, 'CarbScat' is the probability of hitting a carbon atom given that a collision has occurred and 'FissPrb' is the probability of fission occuring given that the thermal neutron is absorbed.

From this it is plain to see that for a large number of neutrons, 10,000 for example, the program is going to carry out a very large number of calculations, and thus will be very slow to run. There are ways to reduce the number of calculations needed, known as variance reduction techniques, and now that we know how the most basic neutron simulation works, we can go on to apply one of these alternative methods to speed up the calculation. Although we first need to identify where most of the time is lost.

The problem is that when a neutron has been absorbed, we simply drop the calculation we have been doing and forget about it, moving on to the next neutron. As each fission creates

more that one neutron (2.47 on average), then we only need 1 neutron out of every 2.47 to cause fission in order for the reactor to balance at k=1. This means that most of the neutron lives we run end in absorption by U^{238} during the time spent at high energies. From this we can see that cutting down on the number of dumped calculations could be very beneficial for speed, and one of the methods that we can use to improve this situation is survival biasing.

Survival biasing is based on the simple concept of weighting a neutron life simulation according to how well it is going. In the current system, each neutron carries a weight of 1, as when a neutron causes fission it increases the fissions counter by 1. Under survival biasing, a neutron begins it's life with a weight of 1, but at any point in time after this, the weighting is made to change in accordance with how likely it is that the neutron survived up to that point without being absorbed. For example, if the probability of a neutron being absorbed is 0.1, then the probability of it surviving absorption is 0.9, and so for every collision it undergoes, 90% of that neutron will survive to the next collision. In other words, we just multiply a neutrons weight by the probability of survival for every collision it undergoes. Applying similar arguments to the chance of fission occuring from absorption at thermal energies, and modifying the fission counter to handle real numbers (not just integers), the simulation of one neutron life simplifies to a structure such as that shown in code fragment 2.

Code Fragment 2 : The infinite reactor simulation using survival biasing

```
c Get energy & set initial weighting:
      Energy=disdev(idum)
      weight=1.0d0
c Loop over one neutron life.
 666    weight=weight*ScatPrb
c Scattered, but was it carbon?
        Rand=RAN1(idum)
        IF (Rand.LE.CarbScat) THEN
         Costheta=-1+2*RAN1(idum)
         AtW=AtWC
         Energy=Energy*(1+2*AtW*Costheta+AtW*AtW)/((1+AtW)*(1+AtW))
        END IF
       IF (Energy.GT.MTSwitch) GOTO 666
      AbsLow=AbsLow+weight
```

Where 'ScatPrb' is the probability of the high-energy neutron surviving absorption (simply the probability of scattering), 'CarbScat' is the chance of hitting carbon given that scattering occurs, and 'AbsLow' is the double precision counter to add up the weights of neutrons after absorption at thermal energies. The number of fissions that occurred can be calculated simply as the number of low energy neutrons absorbed (ie AbsLow) multiplied by the probability that such a neutron will cause fission.

While this is a great improvement on the previous scheme, the calculation can be simplified further. As we are only interested in f:m ratio of the order of 1:100 to 1:1000, the chances that a scattered neutron has hit an atom other that carbon is small enough to be treated as negligible, as so we can assume that the neutron always hits a carbon atom. It is also possible to make the calculation of the energy alteration quicker, in that we can break down the formula in order to simplify the collision code.

We have already seen that the equation for the energy change is:

$$\frac{E_1}{E_0} = \frac{1 + 2A\cos\theta + A^2}{(1 + A)^2}$$

Where A is the atomic weight of the atom in the collision. When this is implemented in the program, cosθ is given by:

$$\cos\theta = -1 + 2x$$

Where x is a uniform random deviate between 0 and 1. By substituting this into the energy formula, we get:

$$\frac{E_1}{E_0} = \frac{1 - 2A + 4Ax + A^2}{(1 + A)^2}$$

Or,

$$\frac{E_1}{E_0} = \frac{1 - 2A + A^2}{(1 + A)^2} + \frac{4A}{(1 + A)^2} \cdot x$$

So, if we define two variables as below:

$$E_{sum} = \frac{1 - 2A + A^2}{(1 + A)^2} \qquad E_{ratio} = \frac{4A}{(1 + A)^2}$$

Then the equation for the energy change simply becomes:

$$\frac{E_1}{E_0} = E_{sum} + E_{ratio}.x$$

Once these simplifications are implemented, the neutron simulation code takes the form shown in code fragment 3.

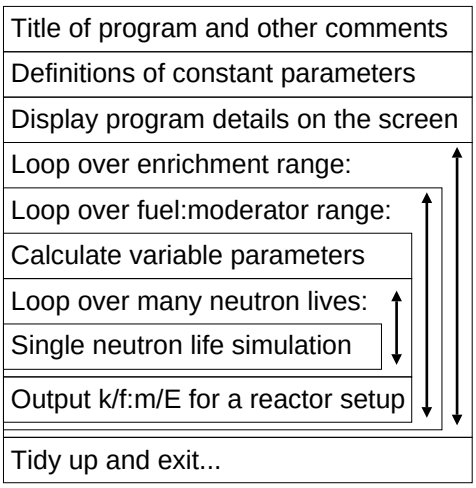
Code Fragment 3 : The infinite reactor simulation using simplified survival biasing

```
c Get energy & set initial weighting:
      Energy=disdev(idum)
      weight=1.0d0
c Loop over one neutron life.
 666    weight=weight*ScatPrb
        Energy=Energy*(ESum+ERatio*RAN1(idum))
        IF (Energy.GT.MTSwitch) GOTO 666
      AbsLow=AbsLow+weight
```

This highly simplified form runs very quickly, making it possible to go on to find accurate estimates of how the enrichment and the fuel to moderator fraction affect the reactor in a reasonable time. The rest of the programming method for the infinite reactor deals with those alterations/additions which are necessary to make correct analysis possible.

Firstly, we need to add a loop over the single neutron life simulation in order to run enough neutrons through the reactor to get a decent result. Then I added two more loops, the first to cycle over the desired range of enrichment (1-10), and the second to cycle through the f:m range (1:1000 - 1:100). In order to appreciate this data, I added the necessary graphics routines to obtain the plot required (k as a function of f:m for a range of E values), and file handling routines to dump the data on to disc. The general structure of the program is given in figure 4 below.

Figure 4 : General structure of the infinite reactor simulator

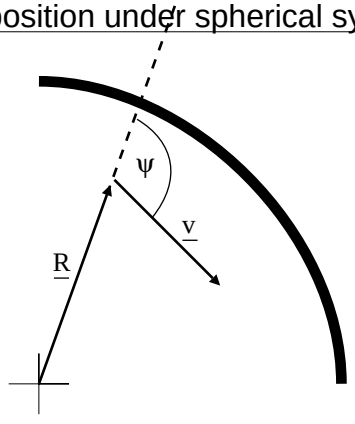


At this stage, the program has all that is required to make a full analysis of the effect of E and f:m on an infinite reactors performance. The full program is supplied in appendix B.

Method (part 2) : The Finite Spherical Reactor

While the general structure of the simulation does not change when considering the finite reactor, we do have to change the neutron life simulation so that it keeps track of the neutrons position and speed, as well as it's energy. In this way we can track the neutrons through the reactor and see if any are lost through the reactors walls. We make the prospect of tracking the neutrons easier by assuming that the reactor is spherically symmetrical, as this means we only need to track radial position, velocity and the angle between the radius and velocity vectors (see fig. 5). Although position and speed are in vector form in the diagram, we only need to know their magnitudes to identify the neutrons position if we have the angle betweeen the vectors. In the program, speed is stored under the guise of the neutrons energy.

Figure 5 : Storing a neutrons position under spherical symmetry

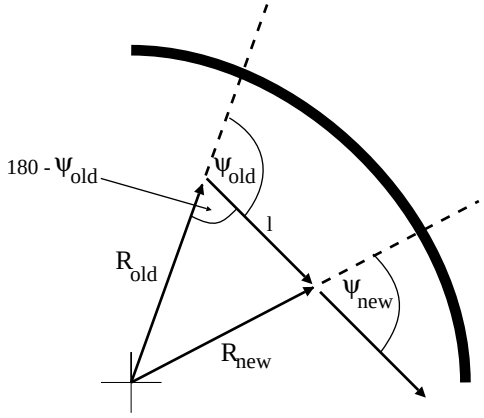


Once this coordinate system has been set up, we need to examine how the neutrons random walk between free flight and collisions affect these coordinates. If the mean free path is given by λ, then we assume that the deviation from this is such that:

$$l = -\ln(x)$$

Where x is uniform random deviate, giving a logarithmic distribution of lengths of path. For one particular loop of the program, we assume that the neutron travelled this distance l before hitting something and re-calculate the position as shown in figure 6.

Figure 6 : Translation of neutron coordinates in free-flight



By applying the cosine rule to the internal triangle in the above diagram, we get:

$$r_{new}^2 = r_{old}^2 + l^2 - 2\, r_{old}\, l \cos(180-\psi_{old})$$
$$= r_{old}^2 + l^2 + 2\, r_{old}\, l \cos(\psi_{old})$$

and,

$$\cos\psi_{new} = \frac{r_{new}^2 + l^2 - r_{old}^2}{2\, r_{new}\, l}$$

Once the neutron has moved to this new position, it then collides with an atom, and so we need to know how to alter the angle between the radial and velocity vectors under this conditions. It can be shown that the angle changes such that;

$$\cos\psi_{coll} = \frac{(1 + A\cos\theta)\cos\psi + A\sin\theta\sin\psi\cos\phi}{\sqrt{[1 + 2A\cos\theta + A^2]}}$$

Where ψ corresponds to the angle between the radius and velocity vectors before collision, and where 3 dimensional scattering is represented be the two random deviates:

$$\cos\theta = -1 + 2x \qquad \text{[as before]}$$

and,

$$\phi = 2\pi x$$

The energy change associated with the collision is the same as before.

While this is all the information needed to track the neutron through the system, there is an extra complication in the finite reactor simulation. The problem is that we can no longer

assume that, once thermal, the neutron will be absorbed eventually because it may escape through the reactor walls before absorption occurs. This means that we need to track the position of the neutron the same way for thermal energies as we do at non-thermal energies, whilst allowing for the possibilities of scattering, absorption or absorption leading to fission.

Once this is implemented, the program is as shown in code fragment 4.

Code Fragment 4 : A neutron life in the finite spherical reactor

```
c Get energy & set initial weighting:
      Energy=disdev(idum)
      NeuCost=COS(6.283185307d0*RAN1(idum))
      weight=1.0d0
c Loop over one neutron life, above thermal energies first.
 666  IF (Energy.GT.MTSwitch .AND. RadPos.LE.Rad) THEN
        weight=weight*ScatPrb
        costheta=-1.0d0+2.0d0*RAN1(idum)
        l=-MeanPath*LOG(RAN1(idum))
        IF (l.GT.0.0d0) THEN
c Alter radial position and angle through translation:
         NewRadPos=DSQRT(RadPos*RadPos+l*l+2.0*RadPos*l*NeuCost)
         NeuCost=(NewRadPos*NewRadPos+l*l-RadPos*RadPos)
     +/(2.0d0*NewRadPos*l)
         RadPos=NewRadPos
c Alter neutron velocity angle as in a collision:
         cosphi=DCOS(6.283185307d0*RAN1(idum))
         sintheta=DSIN(DACOS(costheta))
         NeuSint=DSIN(DACOS(NeuCost))
         NeuCost=(1.0+AtWCol*costheta)*NeuCost+
     +AtWCol*sintheta*NeuSint*cosphi
         Efactor=1.0+2.0*AtWCol*costheta+AtWCol*AtwCol
         NeuCost=NeuCost/DSQRT(Efactor)
         Energy=Energy*Efactor/((1.0+AtwCol)*(1.0+AtWCol))
        END IF
        GOTO 666
      END IF
      AbsLow=AbsLow+weight
c Below thermal energies:
 667  IF (Energy.LE.MTSwitch .AND. RadPos.LE.Rad) THEN
       branch=RAN1(idum)
       IF (branch.LE.tAbsPrb) THEN
c Neutron absorbed, weight fission chance and end calc:
        Fissions=Fissions+weight*FissPrb
c End calculation by throwing neutron out of the reactor:
        RadPos=2.0*Rad
       ELSE
c Scattered, calculate the movement through reactor:
        costheta=-1.0d0+2.0d0*RAN1(idum)
        l=-MeanPath*LOG(RAN1(idum))
        IF (l.GT.0.0d0) THEN
c Alter radial position and angle through translation:
         NewRadPos=DSQRT(RadPos*RadPos+l*l+2.0*RadPos*l*NeuCost)
         NeuCost=(NewRadPos*NewRadPos+l*l-RadPos*RadPos)
     +/(2.0*NewRadPos*l)
         RadPos=NewRadPos
c Alter neutron velocity angle as in a collision:
         cosphi=DCOS(6.283185307d0*RAN1(idum))
         sintheta=DSIN(DACOS(costheta))
```

```
    NeuSint=DSIN(DACOS(NeuCost))
    NeuCost=(1.0+AtWCol*costheta)*NeuCost+
+AtWCol*sintheta*NeuSint*cosphi
    Efactor=1.0+2.0*AtWCol*costheta+AtWCol*AtwCol
    NeuCost=NeuCost/DSQRT(Efactor)
   END IF
  END IF
  GOTO 667
 END IF
```

However there is one more problem to be solved before we can begin simulation of the reactor, which is that we don't know where to start the neutrons from. Obviously, starting them from the same place, or even having them uniformly distributed over the sphere will give us inaccurate results. It can be analytically shown that diffusive loss gives rise to a neutron density profile (see fig. 7):

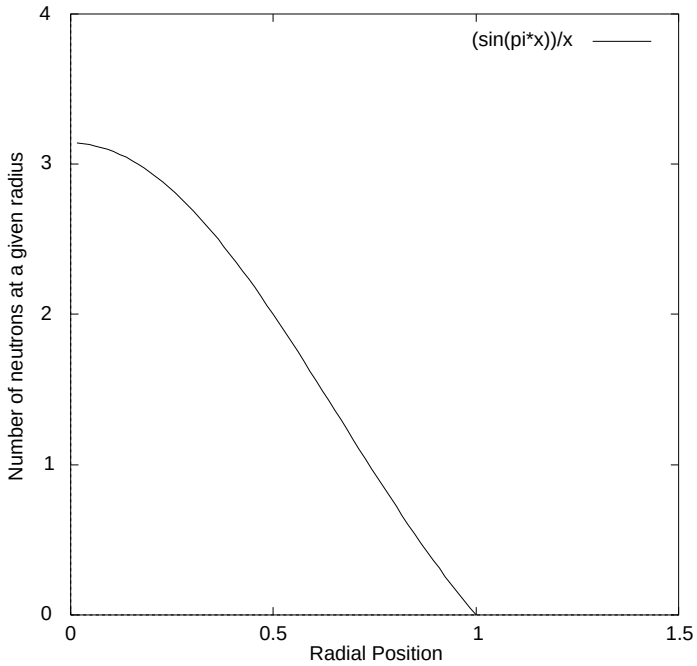
$$n = n_0/R \sin[\pi R/R_e]$$

at radius R, where R_e is determined by the sphere radius, R_0, and the thermal neutron mean free path,λ :

$$R_e = R_0 + 0.71\lambda$$

The implemented code for the finite reactor is given in appendix B.

Figure 7 : Neutron density distribution (at arbitrary x/y scales)



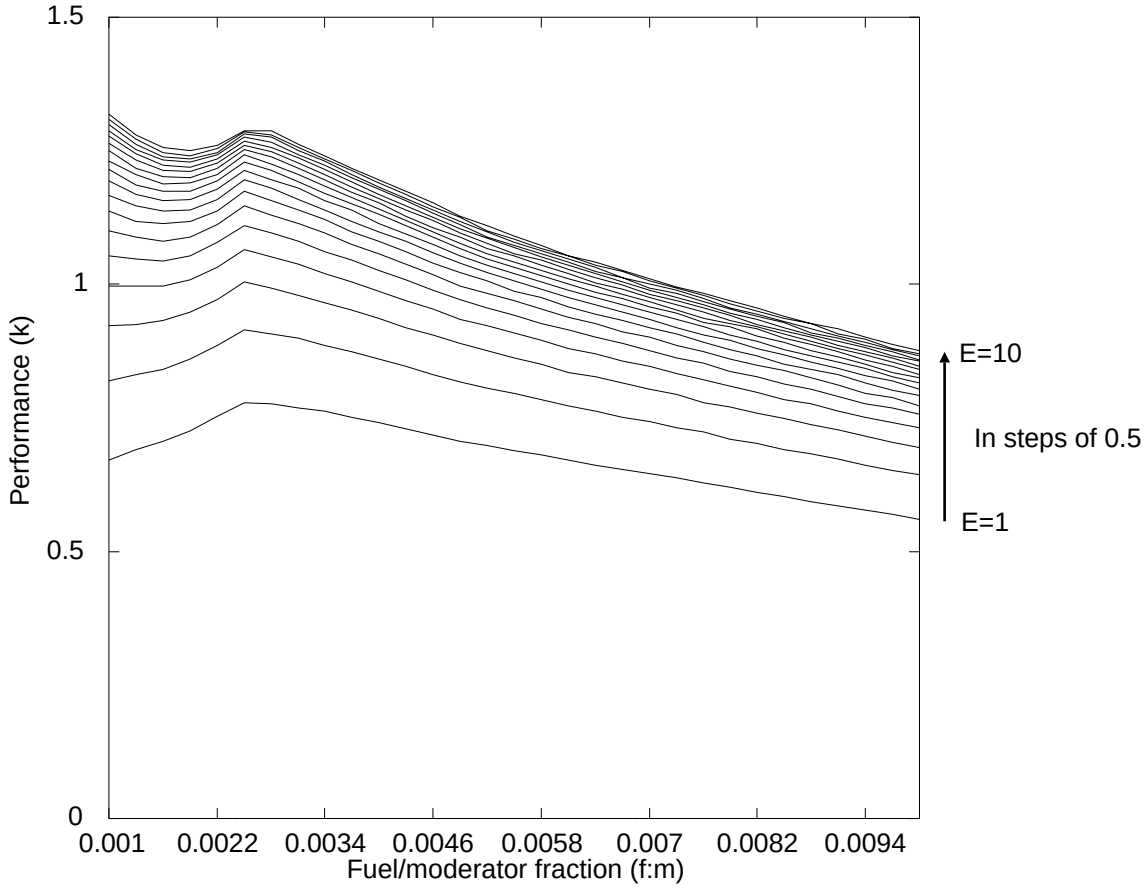
Notes on reactor analysis

While the programs supplied in the appendices contain all the important elements of the simulation, the programs were modified to give the desired results. Simple alterations were made to pick out different information, such as adding line that would pick out the optimum performance values for f:m and plotting a graph of them. Details of these minor modifications will be given in the results section as they arise.

Results (part 1) : The Infinite Reactor

The main infinite reactor program, as supplied in appendix B, which examines reactor performance in relation to fuel:moderator ratio and enrichment give the output in figure 8.

Figure 8 : k against f:m for a range of enrichment (E)

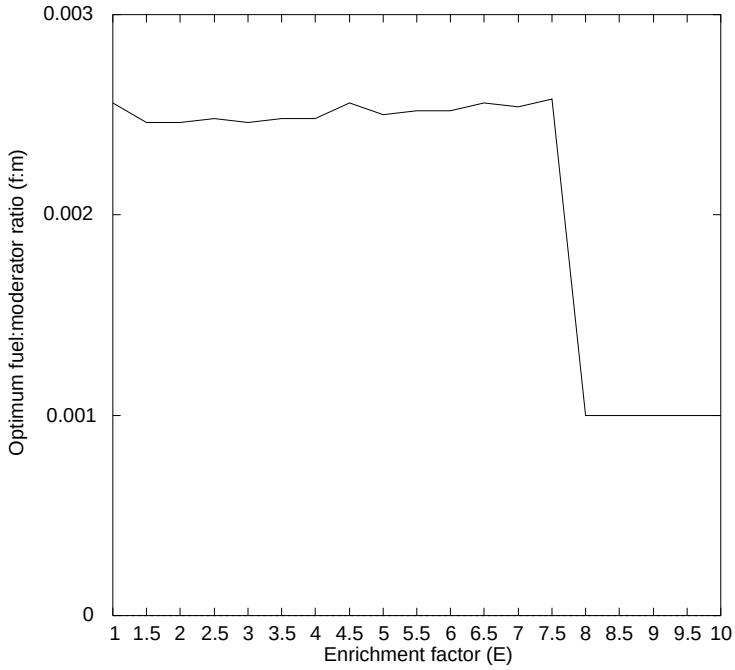


The way in which reactor performance varies with f:m can be thought of as the balancing process between there being too little carbon to slow the neutrons down before absorption and there being too much carbon for the neutron to stand a chance of hitting a U^{235} atom and causing fission before being absorbed. These two effects are combined to produce a balance point, ie a maximum at approximately f:m = 0.0024, or 2.4 uranium atoms to every 1000 of carbon. As the fuel is enriched (shown by the progression of curves for E=1-10 in steps of 0.5), the peak becomes less well defined (although stays in roughly the same place), and the level of enrichment makes less and less difference at higher E, leading to saturation by the level of around E=9. One interesting feature is the way that, for high levels of enrichment, 0.001 takes over as the optimum f:m. This is because the main loss of neutrons is via resonance absorption by U^{238}, and when the fuel is highly enriched there are so few U^{238} atoms that the neutron stands little chance of absorption in the slowing down phase, and is a lot more likely to cause fission when finally thermal, no matter how long it takes to slow down (which corresponds to how many carbons there are). In the limit, ie at infinite enrichment, I would suggests that the curve forms a $^1/_x$ shape, tending to infinity at zero (because as long is there is *some* carbon, the neutron will eventually slow down and be absorbed often leading to fission).

In order to get the most out of our reactor, we should be running it at it's most efficient, ie at

the fuel:moderator ratio that corresponds to the maximum on the graphs. We can find this turning point by altering the program to pick out the values of f:m that give the maximum performance for a given E, and plotting a graph of the best f:m value against E. When suitable alterations have been made, the output is as shown below (fig.9).

Figure 9 : Optimum f:m against E



The graph shows the switch to a lower f:m ratio at high enrichment levels well, as well as showing that the maximum is rougly stable for low enrichment. As I have said, we are only interested in the point where the reactor is *just* stable, and from figure 8 we can see that, if we assume we have the reactor working at maximum efficiency then the required level of enrichment is approximatly at E=2.0. This means that we are only interested in the graphs maximum at low E and so an average of the data from the enrichment range 1.0-3.0 will give us a good estimate of the optimum f:m value:

E	Optimum f:m
1.00000000000	2.560000030790E-03
1.50000000000	2.460000031861E-03
2.00000000000	2.460000031861E-03
2.50000000000	2.480000031646E-03
3.00000000000	2.460000031861E-03

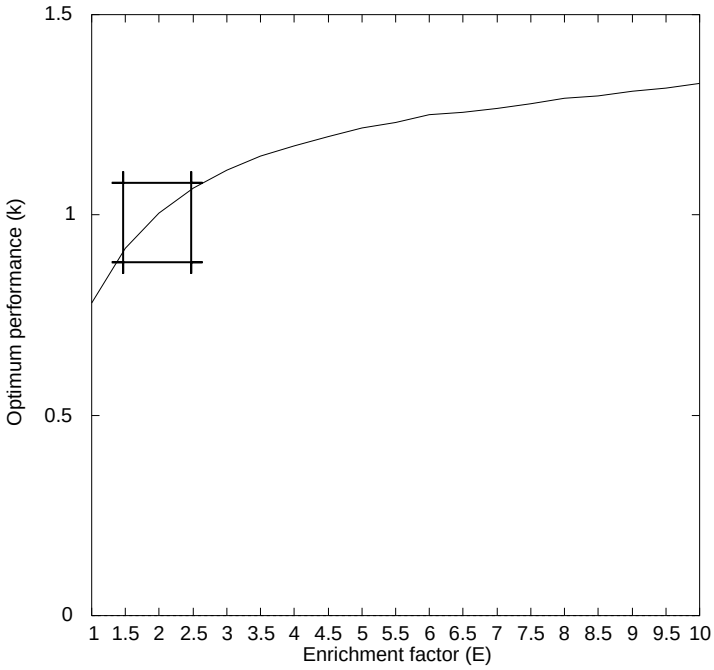
Therefore the optimum f:m ratio is approximatly

$f{:}m_{opti} = 2.484\times10^{-3} \pm 0.076\times10^{-3}$ (3.1% error)

Where the error given is big enough to cover all the values in the table above (ie go up to 2.56×10^{-3}).

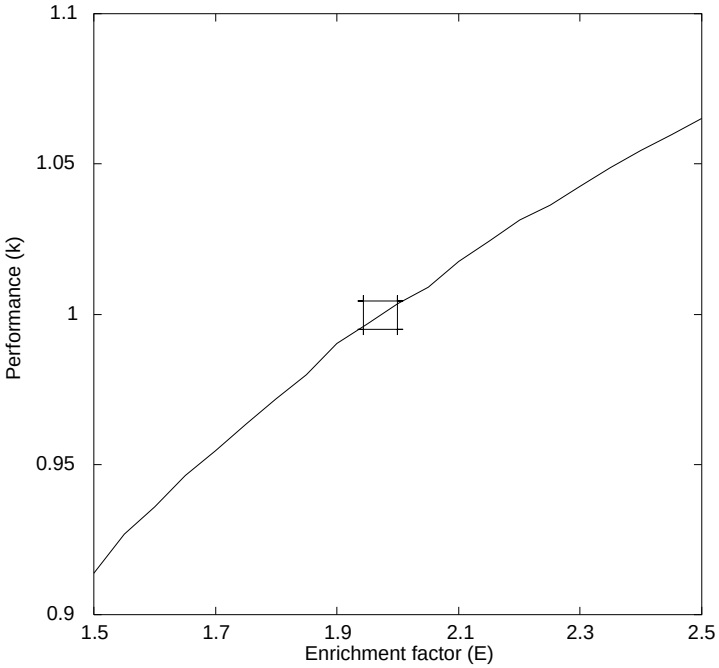
Now we know the optimum f:m ratio (2.498 fuel for every 1000 carbon), we can go on to find the minimum enrichment level needed to get k=1. To do this we alter the program such that f:m is set to the constant value of 2.498×10^{-3} and the enrichment factor has the range 1.0-10.0 and the output is of the graph performance against enrichment (see fig. 10).

Figure 10 : Optimum k against full enrichment range



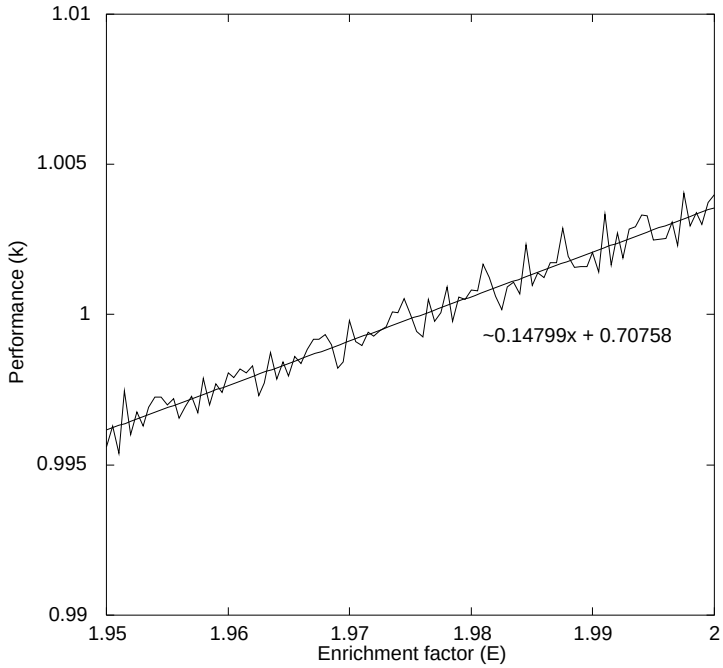
By altering the program slightly, we can 'zoom in' on the marked cross over point (ie where k passes through 1) and get the graph in figure 11.

Figure 11 : Optimum k against E close-up



And by zooming in once more, we get the final output as in figure 12.

Figure 12 : Optimum k against E very close up



At this level of examination the output starts to break up, but by fitting a linear curve to the data, we get that the optimum performance at this enrichment goes as:

$$k = 0.14799E + 0.70758$$

Therefore, for the ideal value of k=1,

$$E = (1 - 0.70758)/0.14799$$
$$= 1.975944$$

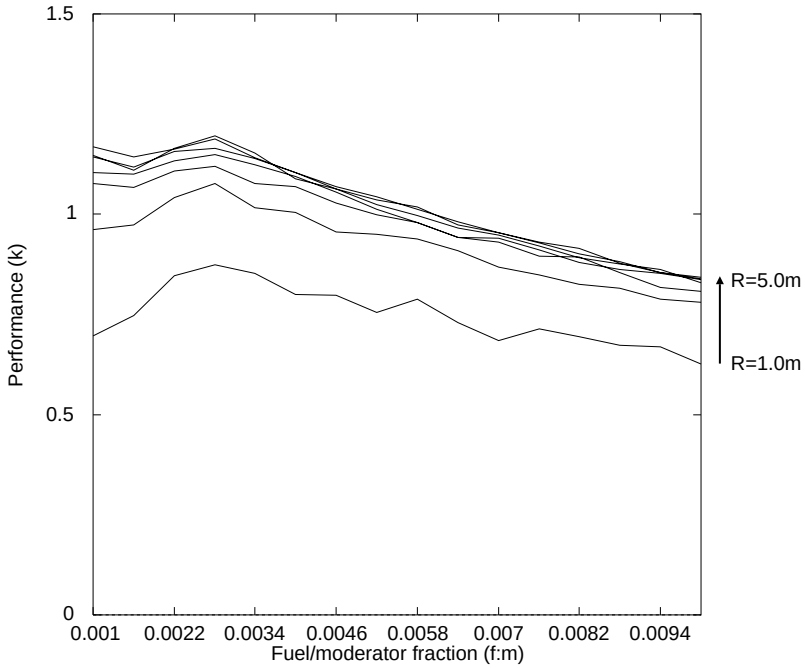
The linear curve fitter gives errors of $\pm 3.2\times10^{-3}$ (2.1%) in the slope and $\pm 6.4\times10^{-3}$ (0.6%) in the intercept, and so;

$$E = 1.976 \pm 0.046 \quad (2.3\%)$$

Results (part 2) : The Finite Spherical Reactor

We can begin to analyse how the radius affects the performance in the same way that we analysed how the enrichment affected it. Just by plotting many graphs on top of each other, but instead of going through a range of E, keep E constant and go through a range of radii (1.0m to 5.0m). The suitable altered finite reactor program produces the output in figure 13.

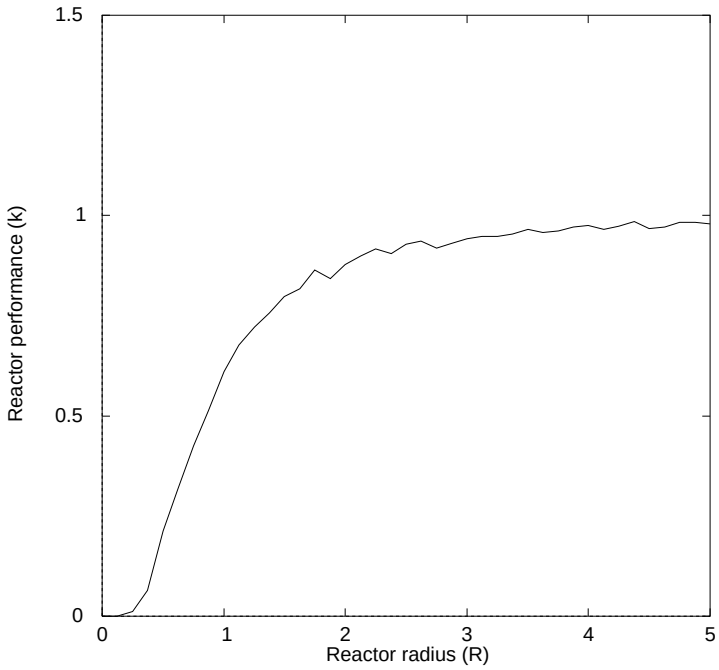
Figure 13 : k against f:m for a range of radii (R)



While the plots are a little rough (as a consequence of drastically cutting down on the number of neutron lives per reactor simulation in order for the program to run in a reasonable time), they show the general behaviour changes as radius increases. For the lower reactor sizes, a little extra radius makes big difference to the final result, but for the higher reactor sizes, the system saturates and there is no point in further increasing the reactor size. As we would expect, the graph tends to it's infinite reactor equivalent at high R (the above graph is for E=5), as indeed it must in order for the finite/infinite programs to be consistant.

To better see what is going on, we can alter the program so that it plots the performance (k) against radius (R). Once this is implemented, the results for the optimum reactor conditions (from the infinite reactor, ie giving k=1) are as in figure 14 (overleaf).

Figure 14 : k against R for the optimum infinite reactor performance



As suggested before, the gradient at low R is high, but tails off to a plateau approaching the k=1 level asymptotically as the radius tends to infinity, thus this result is consistent with the infinite reactor result.

Discussion and Conclusions

While the basic infinite reactor simulation make a large number of assumptions and simplifications, it's results appear to make good physical sense, and so we conclude that given a sufficiently large homogeneous reactor, we need to set the fuel:moderator ratio to 2.484 uranium per 1000 carbon and the enrichment to 1.976 times the natural level (ie $1.976 \times {}^{1}/_{138} = 0.014$, or 1.4 U^{235} for every 100 u^{238}) for the reactor to run efficently at k=1 (with an error of 3.1% in the first figure and of 2.3% in the second). Without new theoretical information, physical experiments would be required in order to verify these conclusions.

When considering the effect that the radius has on reactor performance, the sitation is best describe by the output in figure 14. However, a rough rule as to how the radius affects the performance can be read off data at various points such that:

For 50% performance (k=0.5), R needs to be around 1.0m,
for 75% performance (k=0.75), R needs to be around 1.5m,
for 90% performance (k=0.9), R needs to be around 2.5m, and,
for 95% performance (k=0.95), R needs to be around 3.5m.

In order to be more accurate about the effect of the radius, it would be neccessary to know the limitations on radius (ie what kind of radius is required/reasonable etc). Once this is know, it would be possible to work back and see what level of enrichment is required to get a reactor of the prescribed size to perform efficiently such that k=1. Thus, from the information given in this report, it would be possible to give reasonable estimated of the atomic proportions and overall size neccessary to make a nuclear reactor of the carbon/uranium type efficient as well as self-sufficent, so that we get the most out of it whilst minimising the cost due to enrichment of the fuel. However, this would require system specific information and so does not come within the scope of these experiments.

Appendix A : Data needed for the simulation

Densities:	graphite	1.63×10^{3} kg m^{-3}
	uranium	19.05×10^{3} kg m^{-3}
Atomic masses:	graphite	12.01
	uranium	238.07

Natural uranium fraction (before enrichment): $U^{235}/U^{238} = {}^{1}/_{138}$

Enriched uranium fraction (with enrichment E): $U^{235}/U^{238} = E \times {}^{1}/_{138}$

Cross sections [1 barn = 10^{-28} sq m]

For thermal neutrons (velocity ~2200 m/s, energy ~0.025 eV):

Graphite
σ_s = 4.8 barns $\sigma_a = 3.2\times10^{-3}$ barns
Uranium, isotope U^{235}
σ_s = 10 barns σ_f = 582 barns σ_a = 694 barns †
Uranium, isotope U^{238}
σ_s = 8.3 barns σ_a = 2.73 barns

Number of neutrons released per fission = 2.47

[† includes absorption leading to fission]

Resonant absorption by uranium 238

Σ_s/N_{238}	8.3	50	100	300	500	800	1000	2000	∞
σ_a	0.601	1.066	1.506	2.493	3.096	3.754	4.138	5.400	15.936

Appendix B (part 1) : Infinite reactor simulation program

```
      Program React1
C Simulation of many neutron lives in an infinite homogeneous reactor.
C   (using survival biasing)
C
C Aim:  To investigate how the yield factor k depends on the
C       fuel/moderator fraction and the enrichment factor.
C
C v1:28/2/95: By AN Jackson.
C
      Implicit double precision (a-h,j-z)
      REAL ran1
      Integer*2 ikey
C Defining constant parameters:
      idum=-1
C Number of neutron lives to run:
      iN=5000
C Quality factors:
      NatFuelFrac=1.0d0/138.0d0
      ModFuelFrac=1000.0d0
      FuelModFrac=1.0d0/ModFuelFrac
      EnrichFact=1.25d0
      EFLow=1.0
      EFHigh=10.0
      iEFStep=18
      PureFuelFrac=EnrichFact*NatFuelFrac
      Neutronperfission=2.47d0
      FMFLow=1.0d0/1000.0d0
      FMFHigh=1.0d0/100.0d0
      iFMFStep=30
C Energy parameters:
      ThermSwitch=0.025d0
      MTSwitch=ThermSwitch/1.0d6
C Densities:
      DensC=1.65d3
      DensU=19.05d3
C Atomic weight:
      AtWC=12.01d0
      AtWNatU=238.07d0
C Conversion factors:
      barn=1.0d-28
C Cross-Sections (all converted to metres).
C thermal levels first:
      tSig12s=4.8*barn
      tSig12f=0.0
      tSig12a=3.2d-3*barn
      tSig235s=10*barn
      tSig235f=582*barn
      tSig235a=694*barn
      tSig238s=8.3*barn
      tSig238f=0.0
      tSig238a=2.73*barn
      tSigAbs=0.0
      tSigFiss=0.0
      SigScat=0.0
      SigAbs=0.0
      Sig238a=0.0
c
C Prompt/Frontend for user:
      WRITE(*,*)'Infinite Homogeneous Nuclear Reactor Simulation'
      WRITE(*,*)'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
      WRITE(*,*)' '
      WRITE(*,*)'Evaluate the effect of the fuel/moderator'
      WRITE(*,*)'fraction and the enrichment factor on the'
      WRITE(*,*)'yield of the reactor.'
      WRITE(*,*)' '
      WRITE(*,*)'Press `p` to print the graph, post-plotting.'
      WRITE(*,*)' '
      WRITE(*,*)'Press any key to continue...'
      CALL GET_KEY@(ikey)
C
C Open output file:
      OPEN (UNIT=66,FILE='REACT.OUT',STATUS='UNKNOWN')
```

```
      WRITE(66,*) '# Full data output for infinite reactor.'
      WRITE(66,*) ' '
C
C Call up graphics routines:
      CALL hprepit
      CALL hdefwin(1,REAL(FMFLow),REAL(FMFHigh),0.0,1.5,
     +200,500,50,300)
      CALL hsetwin(1)
      CALL hsetcol(7)
      CALL hplotax('Fuel:moderator fraction.','k')
      CALL hsetcol(1)
C
C Loop over enrich factor:
      DO iEF=0,iEFStep
      EnrichFact=EFLow+(EFHigh-EFLow)*iEF/iEFStep
      CALL hmoveto(0.0,0.0)
      WRITE(66,*) ' '
C
C Loop over FuelMod values
      DO iFMF=0,iFMFStep
      FuelModFrac=FMFLow+(FMFHigh-FMFLow)*iFMF/iFMFStep
C
C Calculate the variable parameters.
C Molecular Densities:
      PureFuelFrac=EnrichFact*NatFuelFrac
      N12=(0.6025d27)*DensC/AtWC
      Nu=FuelModFrac*N12
      N235=PureFuelFrac*Nu
      N238=(1.0d0-PureFuelFrac)*Nu
C Cross-sections:
      SigScat=N12*tSig12s+N235*tSig235s+N238*tsig238s
      tmp=(SigScat/N238)/barn
C Interpolate through the table for Siga:
      IF(tmp.LE.8.3) Siga=0.601
      IF(tmp.GT.8.3 .AND. tmp.LE.50.0)
     +Siga=dinterp(tmp,8.3,50.0,0.601,1.066)
      IF(tmp.GT.50.0 .AND. tmp.LE.100.0)
     +Siga=dinterp(tmp,50.0,100.0,1.066,1.506)
      IF(tmp.GT.100.0 .AND. tmp.LE.300.0)
     +Siga=dinterp(tmp,100.0,300.0,1.506,2.493)
      IF(tmp.GT.300.0 .AND. tmp.LE.500.0)
     +Siga=dinterp(tmp,300.0,500.0,2.493,3.096)
      IF(tmp.GT.500.0 .AND. tmp.LE.800.0)
     +Siga=dinterp(tmp,500.0,800.0,3.096,3.754)
      IF(tmp.GT.800.0 .AND. tmp.LE.1000.0)
     +Siga=dinterp(tmp,800.0,1000.0,3.754,4.138)
      IF(tmp.GT.1000.0 .AND. tmp.LE.2000.0)
     +Siga=dinterp(tmp,1000.0,2000.0,4.138,5.400)
      IF(tmp.GT.2000.0) THEN
       Siga=dinterp(1.0/tmp,1.0/2000.0,0.0,5.400,15.936)
      END IF
      Sig238a=Siga*barn
      tSigAbs=N12*tSig12a+N238*tSig238a+N235*tSig235a
      tSigFiss=N235*tSig235f
      SigAbs=N12*tSig12a+N238*Sig238a
C Probabilities:
      AbsPrb=SigAbs/(SigScat+SigAbs)
      ScatPrb=SigScat/(SigScat+SigAbs)
      tAbsUPrb=(N238*tSig238a+N235*tSig235a)/tSigAbs
      FissPrb=tSigFiss/tSigAbs
C Collision calculation outside:
      AtWCol=AtWC
      ERecip=1.0d0/((1.0d0+AtWCol)*(1.0d0+AtWCol))
      ESum=(1.0d0-AtWCol-AtWCol+AtWCol*AtWCol)*ERecip
      ERatio=(4.0d0*AtWCol)*ERecip
C Correct for small fraction that is not moderator:
      ESum=(1+FuelModFrac)*ESum
      ERatio=(1+FuelModFrac)*ERatio
C
C MAIN LOOP: Over iN neutron lives:
      DO i=1,iN
c Get energy & set initial weighting:
      Energy=disdev(idum)
      weight=1.0d0
c Loop over one neutron life.
```

```
 666    weight=weight*ScatPrb
        Energy=Energy*(ESum+ERatio*RAN1(idum))
        IF (Energy.GT.MTSwitch) GOTO 666
      AbsLow=AbsLow+weight
      END DO
      AbsHigh=iN-AbsLow
      Fissions=FissPrb*AbsLow
      k=Fissions*Neutronperfissions/iN
      CALL hlineto(REAL(FuelModFrac),REAL(k))
      WRITE(66,*) EnrichFact,FuelModFrac,k
      END DO
      END DO
C Clean up and exit
      CLOSE (UNIT=66)
      CALL GET_KEY@(ikey)
      IF (ikey.EQ.80 .OR. ikey.EQ.112) CALL hprintgs
      CALL hfinish
      WRITE(*,*)'Best fuel/moderator fraction: ',BestFMF
      END
c
      double precision function dinterp(X,Xl,Xh,Yl,Yh)
      double precision X
      real Xl,Xh,Yl,Yh
      dinterp=((X-Xl)/(Xh-Xl))*(Yh-Yl)+Yl
      return
      end
```

Appendix B (part 2) : The spherical finite reactor program

```
      Program React2
C Simulation of many neutron lives in a finite homogeneous reactor.
C
C Aim:  To investigate how the yield factor k depends on the
C       fuel/moderator fraction and the enrichment factor and
C       on the reactor radius.
C
C v1:12/3/95: By AN Jackson.
C
      Implicit double precision (a-h,j-z)
      REAL ran1
      Integer*2 ikey
C Defining parameters:
      idum=-1
C Number of neutron lives to run:
      iN=500
C Size of reactor(m):
      Rad=1.0
      RadHigh=5.25
      RadLow=0.25
      iRadStep=10
C Quality factors:
      NatFuelFrac=1.0d0/138.0d0
      ModFuelFrac=1000.0d0
      FuelModFrac=1.0d0/ModFuelFrac
      EnrichFact=1.00d0
      EFLow=1.0
      EFHigh=10.0
      iEFStep=9
      PureFuelFrac=EnrichFact*NatFuelFrac
      Neutronperfission=2.47d0
      FMFLow=0.001d0
      FMFHigh=0.01d0
      iFMFStep=15
C Energy parameters:
      ThermSwitch=0.025d0
      MTSwitch=ThermSwitch/1.0d6
      Energy=0.0d0
      ERatio=1.0d0
C Densities:
      DensC=1.65d3
      DensU=19.05d3
C Atomic weight:
      AtWC=12.01d0
      AtWNatU=238.07d0
C Conversion factors:
      barn=1.0d-28
C Cross-Sections (all converted to metres).
C thermal levels first:
      tSig12s=4.8*barn
      tSig12f=0.0
      tSig12a=3.2d-3*barn
      tSig235s=10*barn
      tSig235f=582*barn
      tSig235a=694*barn
      tSig238s=8.3*barn
      tSig238f=0.0
      tSig238a=2.73*barn
      tSigAbs=0.0
      tSigFiss=0.0
      SigScat=0.0
      SigAbs=0.0
      Sig238a=0.0
c
C Prompt/Frontend for user:
      WRITE(*,*)'Infinite Homogeneous Nuclear Reactor Simulation'
      WRITE(*,*)'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
      WRITE(*,*)' '
      WRITE(*,*)'Evaluate the effect of the fuel/moderator'
      WRITE(*,*)'fraction and the enrichment factor on the'
      WRITE(*,*)'yield of the reactor.'
      WRITE(*,*)' '
```

```
      WRITE(*,*)'Press `p` to print the graph, post-plotting.'
      WRITE(*,*)' '
      WRITE(*,*)'Press any key to continue...'
      CALL GET_KEY@(ikey)
      WRITE(*,*) ' '
      WRITE(*,*) 'Calculating...'
C
C Open output file:
      OPEN (UNIT=66,FILE='REACT.OUT',STATUS='UNKNOWN')
C
C Call up graphics routines:
      CALL hprepit
      CALL hdefwin(1,REAL(RadLow),REAL(RadHigh),0.0,1.5,
     +200,500,50,300)
      CALL hsetwin(1)
      CALL hsetcol(7)
      CALL hplotax('Reactor radius ','Best k ')
      CALL hsetcol(1)
C
C Loop over radius range:
      DO iRad=0,iRadStep
      Rad=RadLow+(RadHigh-RadLow)*iRad/iRadStep
       WRITE(66,*) ' '
C
C Calculate the rest of the parameters.
C Parameters to count up success/failures:
      AbsHigh=0.0d0
      AbsLow=0.0d0
      Fissions=0.0d0
C Molecular Densities:
      PureFuelFrac=EnrichFact*NatFuelFrac
      N12=(0.6025d27)*DensC/AtWC
      Nu=FuelModFrac*N12
      N235=PureFuelFrac*Nu
      N238=(1.0d0-PureFuelFrac)*Nu
C Cross-sections:
      SigScat=N12*tSig12s+N235*tSig235s+N238*tsig238s
      ScatPrbA=N12*tSig12s/SigScat
      ScatPrbB=ScatPrbA+(N235*tSig235s)/SigScat
      tmp=(SigScat/N238)/barn
C Interpolate through the table for Siga:
      IF(tmp.LE.8.3) Siga=0.601
      IF(tmp.GT.8.3 .AND. tmp.LE.50.0)
     +Siga=dinterp(tmp,8.3,50.0,0.601,1.066)
      IF(tmp.GT.50.0 .AND. tmp.LE.100.0)
     +Siga=dinterp(tmp,50.0,100.0,1.066,1.506)
      IF(tmp.GT.100.0 .AND. tmp.LE.300.0)
     +Siga=dinterp(tmp,100.0,300.0,1.506,2.493)
      IF(tmp.GT.300.0 .AND. tmp.LE.500.0)
     +Siga=dinterp(tmp,300.0,500.0,2.493,3.096)
      IF(tmp.GT.500.0 .AND. tmp.LE.800.0)
     +Siga=dinterp(tmp,500.0,800.0,3.096,3.754)
      IF(tmp.GT.800.0 .AND. tmp.LE.1000.0)
     +Siga=dinterp(tmp,800.0,1000.0,3.754,4.138)
      IF(tmp.GT.1000.0 .AND. tmp.LE.2000.0)
     +Siga=dinterp(tmp,1000.0,2000.0,4.138,5.400)
      IF(tmp.GT.2000.0) THEN
       Siga=dinterp(1.0/tmp,1.0/2000.0,0.0,5.400,15.936)
      END IF
      Sig238a=Siga*barn
      tSigAbs=N12*tSig12a+N238*tSig238a+N235*tSig235a
      tSigFiss=N235*tSig235f
      SigAbs=N12*tSig12a+N238*Sig238a
      MeanPath=1.0d0/(SigAbs+SigScat)
      tMeanPath=1.0d0/(tSigAbs+SigScat)
C Probabilities:
      AbsPrb=SigAbs/(SigScat+SigAbs)
      tAbsPrb=tSigAbs/(SigScat+tSigAbs)
      ScatPrb=SigScat/(SigScat+SigAbs)
c      tAbsUPrb=(N238*tSig238a+N235*tSig235a)/tSigAbs
      FissPrb=tSigFiss/tSigAbs
C
C MAIN LOOP: Over iN neutron lives:
      DO i=1,iN
c Get energy & set initial weighting:
```

```
      Energy=disdev(idum)
      RadPos=dendev(idum)*(Rad+0.71*MeanPath)/3.141592654d0
      NeuCost=COS(6.283185307d0*RAN1(idum))
      weight=1.0d0
c Loop over one neutron life, above thermal energies first.
 666  IF (Energy.GT.MTSwitch .AND. RadPos.LE.Rad) THEN
        weight=weight*ScatPrb
        costheta=-1.0d0+2.0d0*RAN1(idum)
        l=-MeanPath*LOG(RAN1(idum))
        IF (l.GT.0.0d0) THEN
c Alter radial position and angle through translation:
         NewRadPos=DSQRT(RadPos*RadPos+l*l+2.0*RadPos*l*NeuCost)
         NeuCost=(NewRadPos*NewRadPos+l*l-RadPos*RadPos)
     +/(2.0d0*NewRadPos*l)
         RadPos=NewRadPos
c Alter neutron velocity angle as in a collision:
         cosphi=DCOS(6.283185307d0*RAN1(idum))
         sintheta=DSIN(DACOS(costheta))
         NeuSint=DSIN(DACOS(NeuCost))
         NeuCost=(1.0+AtWCol*costheta)*NeuCost+
     +AtWCol*sintheta*NeuSint*cosphi
         Efactor=1.0+2.0*AtWCol*costheta+AtWCol*AtwCol
         NeuCost=NeuCost/DSQRT(Efactor)
         Energy=Energy*Efactor/((1.0+AtwCol)*(1.0+AtWCol))
        END IF
        GOTO 666
      END IF
      AbsLow=AbsLow+weight
c Below thermal energies:
 667  IF (Energy.LE.MTSwitch .AND. RadPos.LE.Rad) THEN
       branch=RAN1(idum)
       IF (branch.LE.tAbsPrb) THEN
c Neutron absorbed, weight fission chance and end calc:
        Fissions=Fissions+weight*FissPrb
c End calculation by throwing neutron out of the reactor:
        RadPos=2.0*Rad
       ELSE
c Scattered, calculate the movement through reactor:
        costheta=-1.0d0+2.0d0*RAN1(idum)
        l=-MeanPath*LOG(RAN1(idum))
        IF (l.GT.0.0d0) THEN
c Alter radial position and angle through translation:
         NewRadPos=DSQRT(RadPos*RadPos+l*l+2.0*RadPos*l*NeuCost)
         NeuCost=(NewRadPos*NewRadPos+l*l-RadPos*RadPos)
     +/(2.0*NewRadPos*l)
         RadPos=NewRadPos
c Alter neutron velocity angle as in a collision:
         cosphi=DCOS(6.283185307d0*RAN1(idum))
         sintheta=DSIN(DACOS(costheta))
         NeuSint=DSIN(DACOS(NeuCost))
         NeuCost=(1.0+AtWCol*costheta)*NeuCost+
     +AtWCol*sintheta*NeuSint*cosphi
         Efactor=1.0+2.0*AtWCol*costheta+AtWCol*AtwCol
         NeuCost=NeuCost/DSQRT(Efactor)
        END IF
        END IF
       GOTO 667
      END IF
      END DO
      n=Neutronperfission
      e=Fissions/iN
      k=n*e
c End of radial loop:
      CALL hlineto(REAL(Rad),REAL(highk))
      WRITE(66,*) Rad,k
      END DO
C Clean up and exit
      CLOSE (UNIT=66)
      CALL GET_KEY@(ikey)
      IF (ikey.EQ.80 .OR. ikey.EQ.112) CALL hprintgs
      CALL hfinish
      WRITE(*,*)'Best fuel/moderator fraction: ',BestFMF
      END
```

```
      function ran1(idum)
c     Returns a uniform deviate between 0.0 and 1.0. Set IDUM
c     to any negative value to initialise or reinintialise
c     the sequence
c
      dimension r(97)
      parameter (m1=259200,ia1=7141,ic1=54773,rm1=1.0/m1)
      parameter (m2=134456,ia2=8121,ic2=28411,rm2=1.0/m2)
      parameter (m3=243000,ia3=4561,ic3=51349)
      save r,iff,ix1,ix2,ix3
      data iff /0/
c     Initialise on first call even if IDUM is not zero
      if (idum.lt.0.or.iff.eq.0) then
      iff=1
c     Seed the first routine
      ix1=mod((ic1-idum),m1)
      ix1=mod((ia1*ix1+ic1),m1)
c     and use it to seed the second
      ix2=mod(ix1,m2)
      ix1=mod((ia1*ix1+ic1),m1)
c     and the third routines
      ix3=mod(ix1,m3)
c     Fill the table with sequential uniform deviates generated
c     by the first two routines
      do 11 j=1,97
         ix1=mod((ia1*ix1+ic1),m1)
         ix2=mod((ia2*ix2+ic2),m2)
c     Low and high order pieces combined here
         r(j)=(float(ix1)+float(ix2)*rm2)*rm1
   11 continue
      idum=1
      endif
c     Except when initialising this is where we start.
c     Generate the next number for each sequence
      ix1=mod((ia1*ix1+ic1),m1)
      ix2=mod((ia2*ix2+ic2),m2)
      ix3=mod((ia3*ix3+ic3),m3)
c     Use the third sequence to get an integer between 1 and 97
      j=1+(97*ix3)/m3
      if (j.gt.97.or.j.lt.1) write (*,*) ' failure in j'
c     Return that table entry
      ran1=r(j)
c     and refill it
      r(j)=(float(ix1)+float(ix2)*rm2)*rm1
      return
      end
C
C
      double precision function disdev(idum)
c
c   Calculates random deviates to the neutron fission product
c   energy probability distribution:
c           N(E)=C*SINH(SQRT(2E))*EXP(-E)
c
c   The deviate is obtained by the rejection method using
c   an exponential deviate as comparison
c
c  Calculate the exponential deviate
    1 y=-alog(ran1(idum))
c  The scaling factor 2.0464 is chosen to be the most
c  efficient value compatable with the ratio function f always
c  being less than 1.
      x=2.0464*y
c  Form comparison function
      f=0.76648*sinh(sqrt(x+x))*exp(-0.51134*x)
c  Reject if comparison function less than uniform deviate
      if (f.lt.ran1(idum)) go to 1
      disdev=x
      return
      end
      double precision function dendev(idum)
```

```
c
c   Calculates random deviates to the neutron fission product
c   initial density probability distribution:
c           N(R)=C*SIN(R)/R
c   in the range 0 to PI
c
c   The deviate is obtained by the rejection method using
c   a Lorentzian deviate as comparison over a restricted range
c
c  Calculate the Lorentzian deviate limited in range
    1 y=-0.334750207+0.578242494*ran1(idum)
      y=tan(3.141592654*y)
c  Shift mean to peak of density distribution
c  The scaling factor 1.159294016 is chosen to be the smallest
c  value compatable with the ratio function f always being
c  less than 1.
      x=1.159294016*y+2.028757838
c  Reject if value negative or value greater than PI
      if ((x.lt.0.0).or.(x.gt.3.141592654)) go to 1
c  Form comparison function
      f=0.425183296*x*sin(x)*(1.0+y*y)
c  Reject if comparison function greater than uniform deviate
      if (f.lt.ran1(idum)) go to 1
      dendev=x
      return
      end
8
```