Ostwald ripening model

Fate of two crystal sizes

Kelechi Akwataghibe

Project #2

Math Department
University of Tennessee ,Knoxville
SPRING 2020
Saturday 18th April, 2020

Contents

0.1 Introduction

An experiment is conducted with a given number of identical crystals immersed in a solvent. We want to find out the effect of the solvent on the crystal size as time increases . It has been experimentally observed that these crystals undergo a process known as Oswald Ripening. This phenomenon predicts that overtime, the large sized crystals are favored over the smaller sized ones; and hence, the smaller particles should dissolve into the solution and "redeposit" to "grow" the larger ones. In this report, we will numerically verify the predictions of the Ostwald Ripening process by considering what happens to a two crystal sizes under certain predefined conditions.

0.2 Problem Statement and Assumptions

We assume in this problem, that we have crystals of different sizes in solution.
The following ODE models its size evolution as time progresses

$$\frac{dx_1}{dt} = k \cdot [c(t) - c^* \cdot e^{\frac{\Gamma}{x_1}}] \tag{1}$$
$$\frac{dx_2}{dt} = k \cdot [c(t) - c^* \cdot e^{\frac{\Gamma}{x_2}}] \tag{2}$$
$$x_1(0) = x_1^* \tag{3}$$
$$x_2(0) = x_2^* \tag{4}$$

where

- $c(t)$ represents the solute concentration in solution at time t
- x_i for $i \in 1, 2$, represent the sizes of the crystals. We assume cubic shape.
- c^*, Γ are constants

The solute concentration is given by,

$$c(t) = c_0 + \sum_{j=1}^{N} \mu_j (x_j^*)^3 - \sum_{j=1}^{N} \mu_j (x_j(t))^3$$

where c_0 is the initial dissolved solute and μ is the conversion factor.
Setting $c_1 = c_0 + \sum_{j=1}^{N} \mu_j (x_j^*)^3 =$ constant, the equation above can be written as

$$c(t) = c_1 - \sum_{j=1}^{N} \mu_j (x_j(t))^3$$

Now, we then look for constant solutions to this model. To enable us to understand the long-term behavior of these crystal sizes, we would seek to find solutions(x-values) to the model that do not change as time changes. These are called steady-state solutions(or equilibria).
Values of the parameters are : $\mu = 1.e-3, c^* = 7.52e-7, \Gamma = 4.e-3, c_0 = 1.05c^*$.

The theory predicts that the larger crystal size (in this case x_2) will exist for all time, whereas, the smaller crystal size(x_1) will dissolve in finite time. After dissolution, our ODE system will be reduced to a single-sized crystal case for the larger crystal size.

$$\frac{dx}{dt} = G(x) = k \cdot [c - c_{crit}] \ , \ x(0) = x(t_{diss})$$

where the equilibria are the roots of $G(x) = 0$

0.3 Methods

We will solve the ODE system numerically using Euler method to determine the time of dissolution of the smaller crystal size. This will be done by increasing the number of iterations of the algorithm until we can precisely when the smaller crystal size dissolves. Once our ODE system reduces to the single-sized crystal case(as shown above), we will then utilize the Newton Method to compute the equilibria solution and then determine what the limiting value of that remaining crystal size will be.
Below is a brief overview of those two numerical methods .

0.3.1 The Newton-Raphson Method

This is an iterative method used to approximate a root of a "smooth" function $f(x)$. The procedure requires a starting value(initial guess) from the user, and then with a defined algorithm , it iteratively calculates subsequent values until the subsequent values become "extremely close" such that there are "approximately the same". At this point, we say that the algorithm converges, and this final value is the solution(root) required .

Algorithm
Let x_0 is the initial guess which the user wants to guess.
$x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$, n $= 0, 1, \cdots$
$\delta x = -\dfrac{f(x_n)}{f'(x_n)}$
Hence, $x_{n+1} = x_n + \delta x$
For this algorithm to work, we need f and f' to be continuous near a root. When this happens, we know that there exists an interval I containing the root such that $\forall x_0 \in I$, the iteration converges to the root r.
If we start close enough to r, then Newton method converges. If it converges, it does so quadratically fast.

$$|e_{n+1}| \leq C \cdot |e_n|^2$$

where $e_n = x_n - r$. This amounts to doubling the number of correct (binary) digits in each iteration

0.3.2 Euler Method

The Euler method is a time-stepping scheme that is used to find approximate solutions to Initial Value Problems(IVP). IVPs are ODEs with initial values given. To start the process, the time-interval given is discretized, and then , through the time-stepping scheme, the approximate values of the function are calculated at those time-intervals given..

Consider the IVP

$$y_1' = f(t, y_1)$$

$$y_2' = f(t, y_2)$$

$$y_1(t_0) = y_{10}$$

$$y_2(t_0) = y_{20}$$

Below is the Euler scheme algorithm:

- The time interval given is discretized into timestamps. $T = \{t_1, t_2, \cdots, t_n \cdots t_{N_{max}}\}$. An equimesh size is created by the equation below,

  $$\delta t = \frac{t_{end} - t_0}{N_{max}}$$

  Hence for each $n = 0, 1, \cdots, N_{max}$,

  $$t_n = t_0 + n \cdot \delta t$$

- Next, we discretize the derivative
  For $i \in 1, 2$,

  $$y_i'(t_n) = \lim_{\delta t \to 0} \frac{y_i(t_n + \delta t) - y_i(t_n)}{\delta t}$$

  $$\simeq \frac{y_i(t_n + \delta t) - y_i(t_n)}{\delta t}$$
  $$= f(t, y_i(t_n))$$

  Using the approximation,

  $$Y_{in} \simeq y_i(t_n)$$

  We can write that

  $$Y_{i,n+1} = Y_{in} + \delta t \cdot f(t_n, Y_{in})$$

0.4 Solution/Results

a **Dissolution Time and Theory Predictions**

To solve the ODE system using the Euler Method, the tolerance(TOL) was set to 1e-12(double precision). Different values for Nsteps was used: 1000, 10000 but we eventually settled with 1000000 which gave us a very high degree of accuracy. The computation showed that $x_1(t)$ dissolved and the dissolution time(t_1) was calculated to be approximately t= 0.01440127. $x_2(t_1)$, however grew in size such that at time t_1, it is approximated to be about 0.0949002896834991

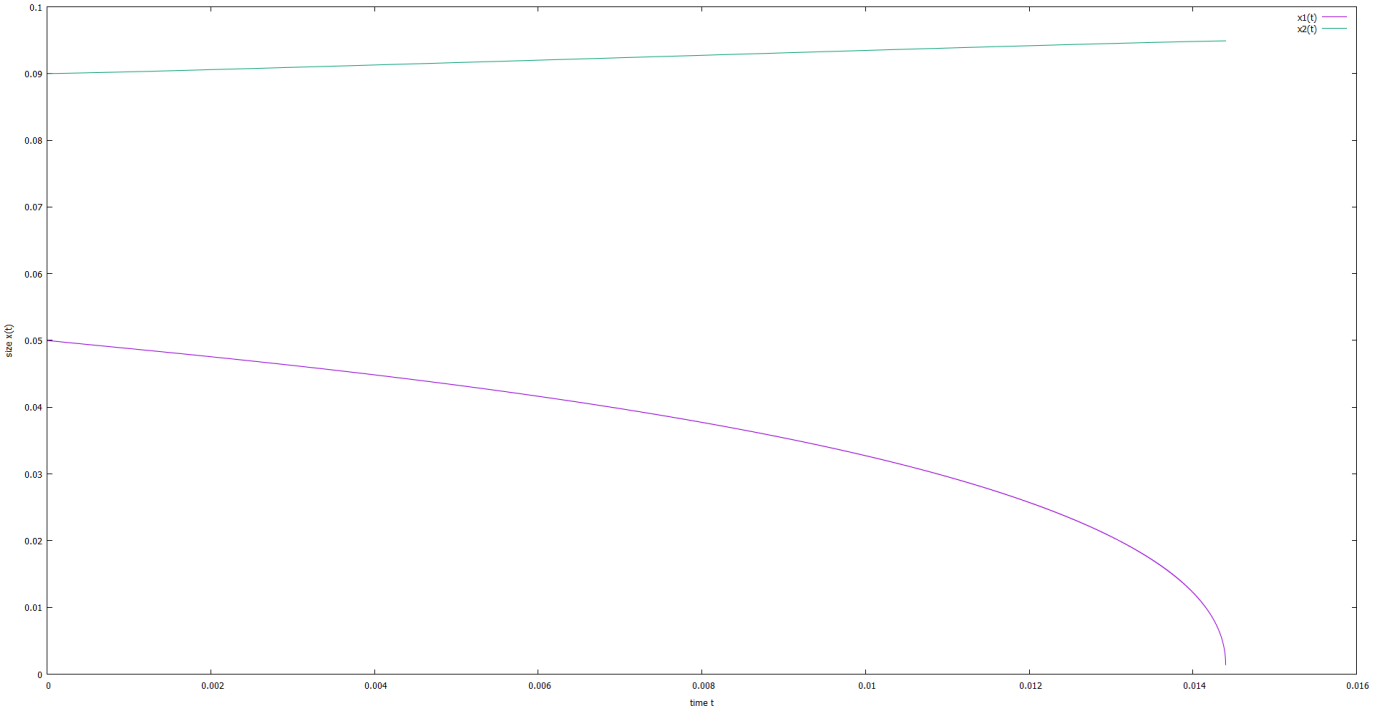

Figure 1: Evolution of sizes $x_1(t)$ and $x_2(t)$, $x_1(t)$ dissolves in finite time as theory predicts

After the time of dissolution, we just have a single-sized crystal($x_2(t)$) in the solution with initial size $x^* = 0.0949002896834991$. Using the Newton method, we calculated the equilibria to be

- x_{21}= 5.113525881614132e-03
- x_{22} = 9.509583125726935e-02

  Here $x_1 < x^* < x_2$

Since, the initial size of the crystal is in between both equilibrium sizes, from our theory, it will grow and tend to the larger equilibrium size (x_{22}) asymptotically. Furthermore, we can observe here that the larger equilibrium is very close to the current size of $x_2(t)$. Therefore, it is necessary that the numerical size is computed with high accuracy, as if it is slightly above the equilibrium, based on theory, our prediction will also have to change.

b **Time evolution of the size(Using Euler with $N = 10^5$ timesteps)**
Utilizing the Euler method , we numerically determined that the remaining crystal increases overtime(monotonically) and finally converges to the larger equilibrium as predicted by the equilibrium analysis, as seen in Fig.3.
From approximately time $t = 0.031913$ onwards, the crystal size $x_1(t)$ remains at 0.095095831257258 which is approximately equal to the equilibrium as computed by the Newton method.

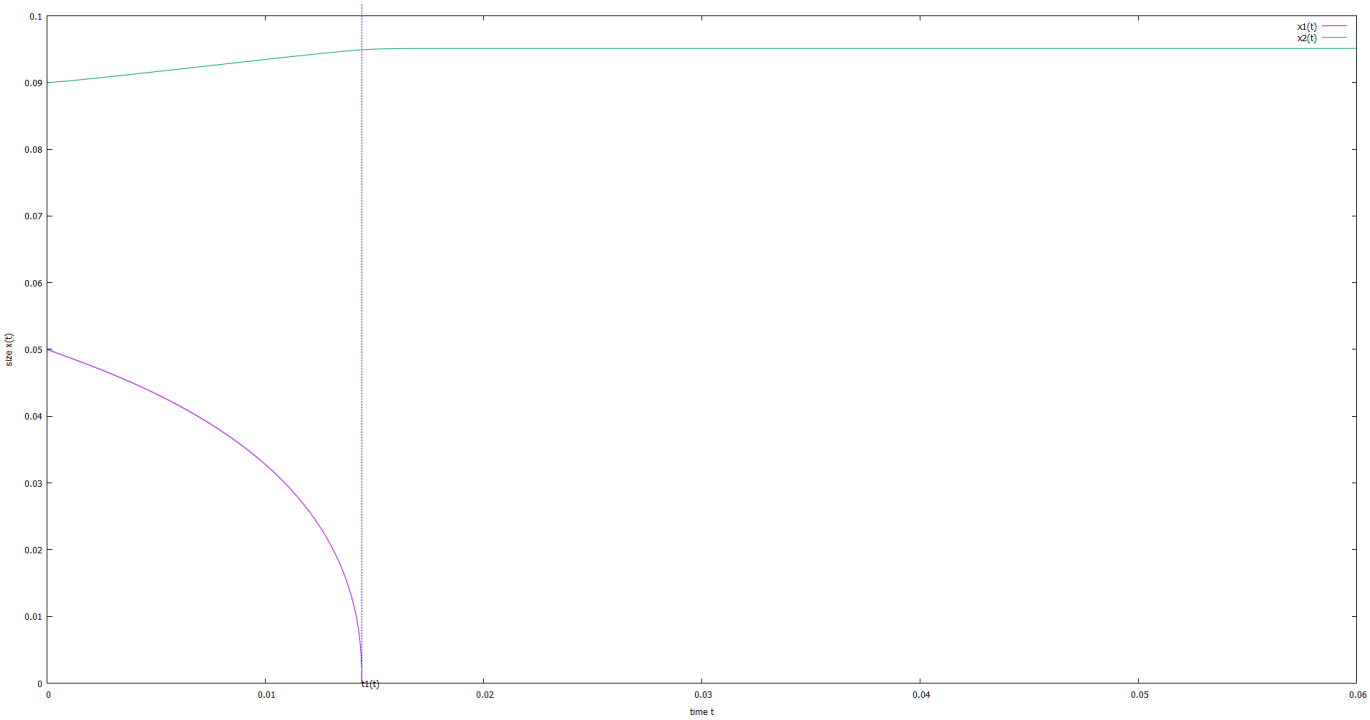

Figure 2: Evolution of both crystal sizes $x_1(t)$ and $x_2(t)$, $x_1(t)$ dissolves in finite time, while $x_2(t)$ tends to the larger equilibrium x_{22} asymptotically, as theory predicts

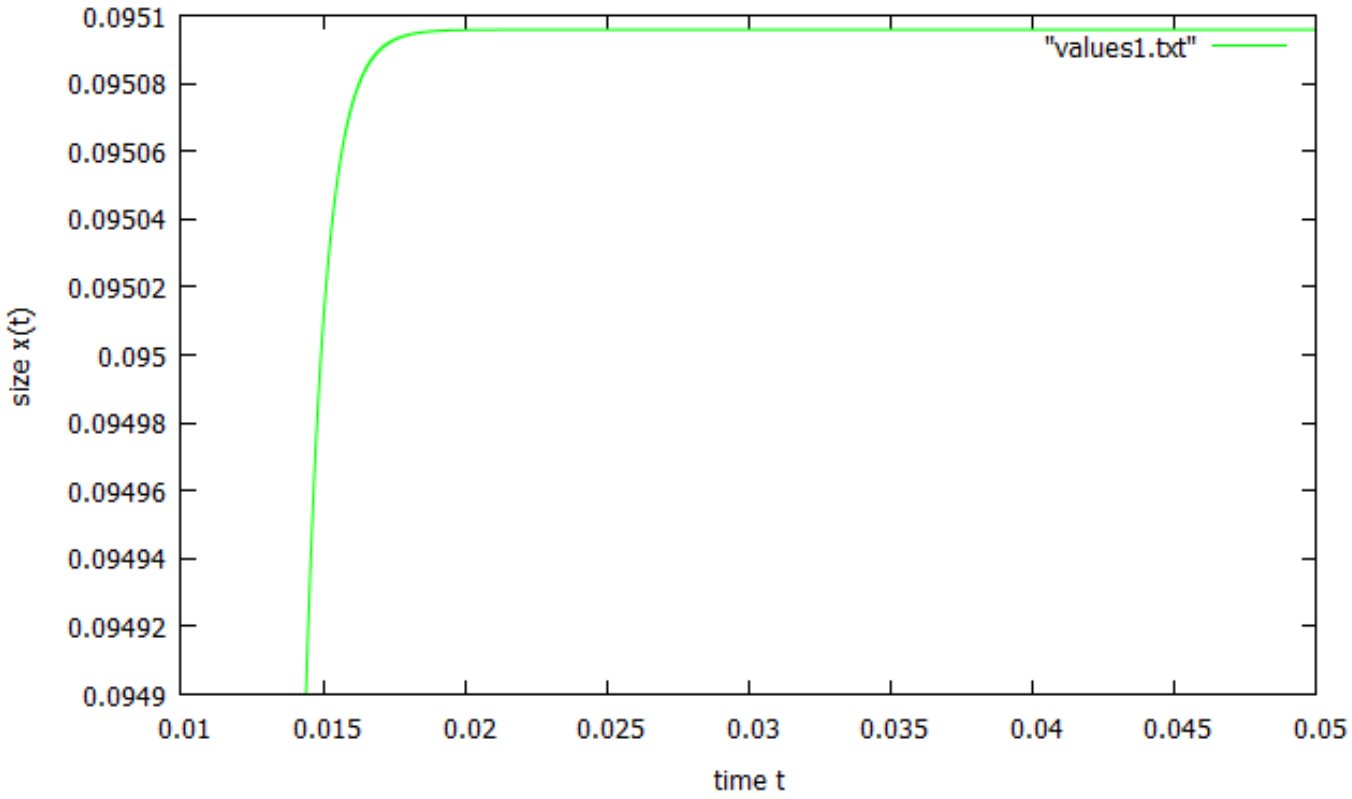

Figure 3: Evolution of size $x_1(t)$, $x_1(t)$ tends to the larger equilibrium x_{22} asymptotically, as theory predicts

0.5 Discussion and Conclusions

The results show in general that for two crystal sizes, the smaller crystal size dissolves completely; otherwise, it would tend asymptotically, to the larger equilibrium size.
This process can be extended to any number of crystal sizes, and can be used to monitor the long term behavior of individual crystals. An application can be found in pharmaceutics, particularly to model the amount of drug dosage required to achieve a certain solute concentration in the body.

For all computations in this project, we employed double precision to improve the accuracy of our results. This was sufficient to perfectly delineate the trends in all the cases although , some of the approximations(especially in the third) case were off.
In the future, we will consider switching to more robust algorithms to improve our accuracy. The improved Euler(Heun) method which is of second order, generates a lower relative error than the classical Euler method which we employed here. This would be a good candidate for any future iterations on our solution

0.6 Appendix

0.6.1 Newton Method

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>

//Double is used instead of float to allow for more precision
//Declared global variable to allow for easy access from different remote subroutines.

double xn, Fn, DFn;
int main() {
   double x0,TOL;
   int maxIT;


   printf("Enter a guess for the root: ");
   scanf("%lf", &x0);
   printf("What is the maximum number of iterations to be performed: ");
   scanf("%d", &maxIT);
   printf("Enter the tolerance: ");
   scanf("%lf", &TOL);


   NewtonID(x0, TOL, maxIT);



   return 1;
}

NewtonID(double x0, double TOL, int maxIT) {
   xn = x0;
   double dx = 1000;
   int n;
   int found = 0; //A flag to indicate whether or not the root has been found
   printf("\n\n#n\t xn \t Fn\n");

   for ( n = 0; n < maxIT; n++)
   {
      FCN();
      printf("%d\t %e \t %e\n", n, xn, Fn); //Printing the values one line at a time
      if (fabs(dx) < TOL)
      {
         if (fabs(Fn) < TOL) {
            printf("\nDONE\nroot= %e\nF=%.15e\n in %d iterations", xn, Fn, n);
            found = 1;
            break;
         }
         else {
            printf("\nSTUCK\ndx<TOL BUT\n residual = %f > %f", Fn, TOL);
            break;
         }
      }
```

```
      dx = 0-Fn / DFn; //Newton's step
      xn = xn + dx;

   }

   //If the root has not been found, then execute said code
   if (found = 0) {
      printf("BAD\nReached %d iterations without a solution", maxIT);
   }

}

 FCN() {

    double cStar = 7.52e-7;
    double xStar = 0.05;
    double gamma = 4.e-3;
    double mu = 1.e-3;

    Fn = (mu)* (xn*xn*xn) + (cStar)*exp((gamma) / xn) - (1.05*cStar +
        mu*xStar*xStar*xStar); //The function we were asked to write

    DFn = 3*(mu)* (xn*xn) - (cStar* gamma/(xn*xn))*exp(gamma / xn) ;
 //The derivative of the said function


}

}
```

0.6.2 Euler Method for One Crystal Size

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>

//Double is used instead of float to allow for more precision
//Declared global variable to allow for easy access from different remote subroutines.


double FCN(double, double);
double formula(double);




int main() {
   double xStar;
   double tEnd;
   int t0;
   int Nmax;

   printf("Enter the initial value of the system: ");
   scanf("%lf", &xStar);
```

```
   printf("What is the maximum number of iterations to be performed: ");
   scanf("%d", &Nmax);
   printf("Enter an initial time: ");
   scanf("%d", &t0);
   printf("Enter the final time : ");
   scanf("%lf", &tEnd);

   EulerScheme(t0, tEnd, Nmax,xStar);


   return 1;
}

EulerScheme(int t0, double tEnd, int Nmax, double xZero) {
   FILE *f; //Creating File to Store Output
   f = fopen("values3.txt", "w"); //Creating/Opening the file to read values

   int n;
   double xn;
   xn= xZero;
   double tn = t0;
   double dt = (tEnd - t0) /(double) Nmax;
   double errN, errMax, yExact;
   errMax = 0.0;

   for (n = 0; n < Nmax+1; n++) {
      if(xn > 0) {
      xn = xn + dt * FCN(tn, xn);
      tn = t0 + n*dt;

      fprintf(f, "%e\t%0.15e\n", tn, xn); //Printing pair to the file

       }
       else{
          fprintf(f, "%e\t%0.15e\n", tn, xn); //Printing pair to the file
         break;
       }
   }

   fclose(f); //Close the file that was opened
   printf("\nYou output has been successful written to the file values.txt");//Inform
       user that output was written



}

double FCN(double t, double xn) {

   double cStar = 7.52e-7;
   double xStar = 0.08;
   double gamma = 4.e-3;
   double mu = 1.e-5;
   double Fn = 0.0, DFn=0.0;
   double k = 5.e7;
```

```
   DFn = k*(1.05*cStar + mu*xStar*xStar*xStar - (mu)* (xn*xn*xn) - (cStar)*exp((gamma) /
       xn));


   return DFn;
}

double formula(double tn) {

   return exp(tn);
}
```

0.6.3 Euler Method for Two Crystal Sizes

```
struct Func FCN(double, double, double);
double formula(double);

//Double is used instead of float to allow for more precision
//Declared global variable to allow for easy access from different remote subroutines.

double x1n,x2n;
double arr[2];
FILE *file;
//Since C does not return multiple values, we create a struct in order to hold both F1
    and F2 as they are returned from the FCN routine.
struct Func {
   double F1, F2;
};
int main() {
   double tEnd;
   double x10,x20;
   int t0, Nmax;
   file = fopen("output.txt", "w");
   printf("Enter the initial value of the first ODE(x10): ");
   scanf("%lf", &x10);
   printf("Enter the initial value of the second ODE(x20): ");
   scanf("%lf", &x20);
   printf("What is the maximum number of iterations to be performed: ");
   scanf("%d", &Nmax);
   printf("Enter an initial time: ");
   scanf("%d", &t0);
   printf("Enter the final time : ");
   scanf("%lf", &tEnd);

   EulerScheme1(t0, tEnd, Nmax,x10,x20);
   fclose(file);

   return 1;
}

EulerScheme1(int t0, double tEnd, int Nmax, double x10, double x20) {
   int n;
```

```
   x1n= x10;
   x2n = x20;
   double tn = t0;
   double dt = (tEnd - t0) / Nmax;
   double errN, errMax, y1Exact;
   errMax = 0.0;

   for (n = 0; n < Nmax+1; n++) {
      struct Func fcns = FCN(tn, x1n, x2n);
        if(x1n > 0 & x2n >0){
      x1n = x1n + dt * fcns.F1;
      x2n = x2n + dt * fcns.F2;
      tn = t0 + n*dt;

      fprintf(file,"%0.15e\t%0.15e\t%0.15e\n",tn, x1n, x2n);
        }
        else{
            fprintf(file,"%0.15e\t%0.15e\t%0.15e\n",tn, x1n, x2n);
            break;
        }
   }



}


struct Func FCN(double t, double x1, double x2) {

   double cStar = 7.52e-7;
   double x1Star = 0.05;
   double x2Star = 0.09;
   double gamma = 4.e-3;
   double mu = 1.e-3;
   double k = 5.e7;
   double  DF1n = 0.0, DF2n = 0.0;
   double c = 0.0;

   c = 1.05*cStar + mu * (x1Star*x1Star *x1Star + x2Star*x2Star*x2Star) - mu * (x1*x1*x1
       + x2*x2*x2);

   DF1n = k*(c-cStar * exp((gamma)/x1n));
   DF2n = k*(c - cStar * exp((gamma) / x2n));

   struct Func fcts = { DF1n,DF2n };
   return fcts;

}
```