Computer Application for Scientific Computing Final Project with LaTeX

Woojeong Park
Department of Mathematical Science
Seoul National University
2015-17231

June 18, 2019

1 Introduction

In this project, we try to solve the non-linear ordinary differential equation called Lotka-Volterra model, which predicts the approximation of preys and predators in reality. And also solve the linear system with huge size of Matrix using several methods and compare what the differences are. There are many numerical methods including direct and iterative things, and we can apply to many mathematical problem in reality. All PseodoCode and graph can be used in FORTRAN90 and MATLABr2018.

2 Project I : Differential Equation System

At first, we will solve the ordinary differential equation system, namely called ODE system. It is well-known how to solve it as an analytic solution, but here we consider it as a numerical problem.

2.1 Lotka-Volterra predatoryprey model system

The differential equation system below is called LotkaVolterra predatorprey model system.

$$\frac{dx}{dt} = Ax(t)(1 - By(t)), \quad x(0) = x_0$$
$$\frac{dy}{dt} = Cy(t)(Dx(t) - 1), \quad y(0) = y_0$$

The variable t denotes time, $x(t)$ the number of prey (e.g., rabbits) at time t, and $y(t)$ the number of predators (e.g., foxes). The positive constants A and C mean prey and predator population growth parameter, and B and D mean the species interaction parameters.
Let $A = 4$, $B = 12$, $C = 3$, $D = 13$ and $x(0) = 3$, $y(0) = 5$ be given. We try to solve that equation on $0 \leq t \leq 5$

$$\frac{dx}{dt} = 4x(t)(1 - 12y(t)), \quad x(0) = 3$$

$$\frac{dy}{dt} = 3y(t)(13x(t) - 1), \quad y(0) = 5$$

For solving this differential equation system, here we use 6 methods, plot them between time t and number of preys x and predators y, and also between x and y.

2.2 Explicit Euler Method

Explicit Euler Method, also called as `Foward Euler`, is very simple numerical method for integration of the first-order ODE. IF we should solve the ODE given:

$$\frac{dy}{dt} = f(t, y)$$

Then the basic algorithm using is

$$y_{n+1} = y_n + hf(y_n, t_n),\ n = 0,\ 1,\ 2,\ 3,\ \cdots$$

where y_n denotes $y(t_n)$. So apply this method to our program, we get

$$x_{n+1} = x_n + h(4x_n - 48x_n y_n)$$

$$y_{n+1} = y_n + h(-3y_n + 39x_n y_n)$$

Here is the pseudo code using this algorithm above :

```
PseudoCode :  Explicit Euler Method
1.  do j=1, iter
2.       x(j+1)=x(j)+(4*x(j)-48*x(j)*y(j))*h
3.       y(j+1)=y(j)+(-3*y(j)+39*x(j)*y(j))*h
4.  enddo
```

We want to solve it on $0 \leq t \leq 5$, so if we are going to set $h = 0.001$, $h = 0.005$, $h = 0.00025$, then the value of the `iter` above that algorithm should be 5000, 10000, and 20000.

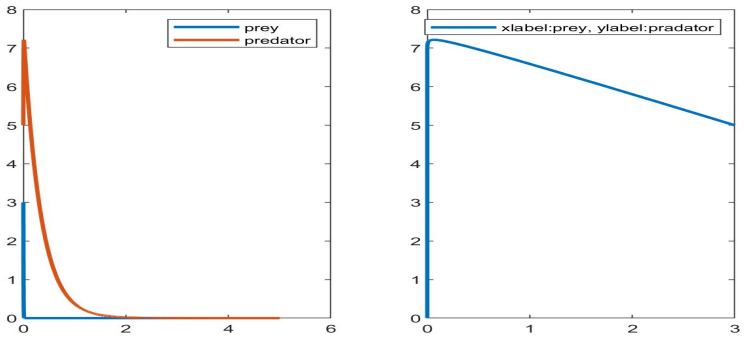

Figure 1: Explicit Euler with $h = 0.001$

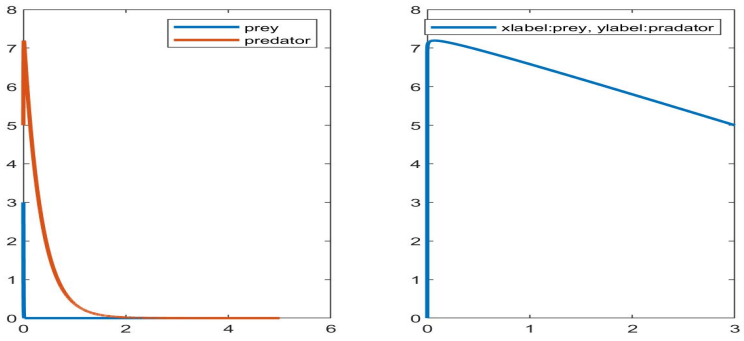

Figure 2: Explicit Euler with $h = 0.0005$

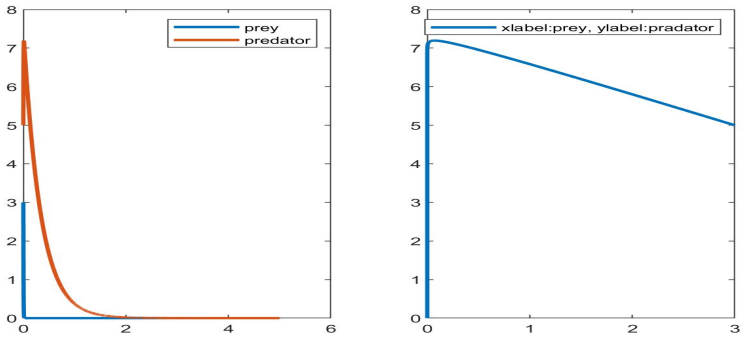

Figure 3: Explicit Euler with $h = 0.00025$

Briefly, we can notice that the number of preys and predators started at 5 and 3 respectively, and they become almost to zero at $t \geq 2$. There are no explicit difference with the different value of h. I think because the number of predators at the begin is larger than preys, so they are predicted all going to be extinct.

2.3 Implicit Euler Method

Implicit Euler Method, also called `Backward Euler`, costs higher than the explicit method before, but it is significantly stable than other methods. The basic algorithm is below :

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1})$$

So our problem is :

$$x_{n+1} = x_n + h(4x_{n+1} - 48x_{n+1}y_n)$$
$$y_{n+1} = y_n + h(-3y_{n+1} + 39x_n y_{n+1})$$

Note that for applying this method to solve that ODE system, we should remain y_n and x_n at the first and second equation. Now describe what x_{n+1} and y_{n+1} are. Just describing only the x_{n+1} and y_{n+1} is below :

$$x_{n+1} = \frac{x_n}{1 - h(4 - 48y_n)}$$
$$y_{n+1} = \frac{y_n}{1 - h(-3 + 39x_n)}$$

Here is the pseudo code using this algorithm above :

```
PseudoCode :  Implicit Euler Method
1.  do j=1, iter
2.      x(j+1)=x(j)/(1-h*(4-48*y(j)))
3.      y(j+1)=y(j)/(1-h*(-3+39*x(j)))
4.  enddo
```

Similar to Explicit Euler method, the number of iteration is decided on the same way, and execute the code using FORTRAN90, we get the following result.

We can figure out that the result from `Implicit Euler Method` is the same as the result from `Explicit Euler Method`.

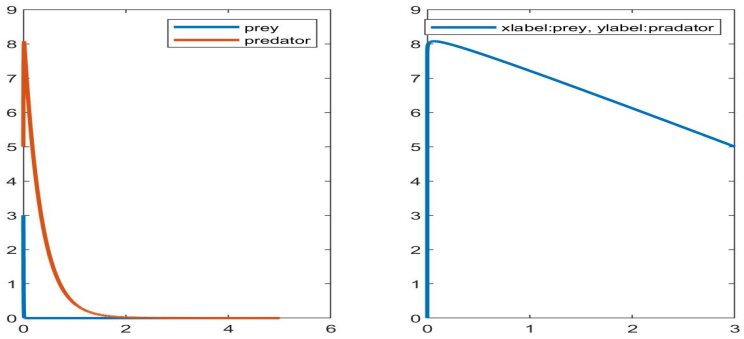

Figure 4: Implicit Euler with $h = 0.001$

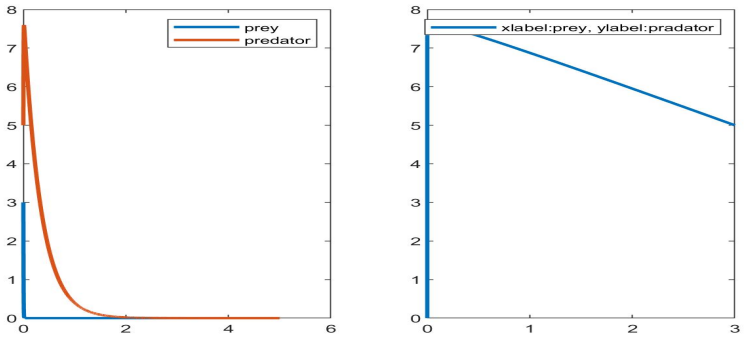

Figure 5: Implicit Euler with $h = 0.0005$

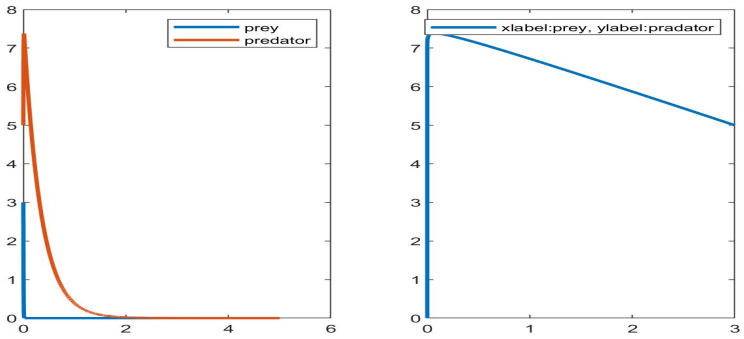

Figure 6: Implicit Euler with $h = 0.00025$

2.4 Improved Euler Method

Improved Euler Method is basically the combination of Explicit and Implicit Methods. There are many combinations for using Improved Euler Methods, we use that relation below : (This method is called Trapezoidal method.)

$$y_{n+1} = y_n + \frac{h}{2}(f(y_n, t_n) + f(y_{n+1}, t_{n+1}))$$

On that problem, similar to implicit method, we should change the form of that relation only for y_{n+1}. Let's see our problem is :

$$x_{n+1} = x_n + \frac{h}{2}(4x_n - 48x_n y_n + 4x_{n+1} - 48x_{n+1}y_n)$$

$$y_{n+1} = y_n + \frac{h}{2}(-3y_n + 39x_n y_n - 3y_{n+1} + 39x_n y_{n+1})$$

So we get

$$x_{n+1} = \frac{x_n(1 + h(2 - 24y_n))}{1 - h(2 - 24y_n)}$$

$$y_{n+1} = \frac{y_n(1 + \frac{h}{2}(-3 + 39x_n))}{1 - \frac{h}{2}(-3 + 39x_n)}$$

It looks kinds of complicated, but it can help to reduce the error of order 3 or higher than explicit or implicit Euler methods. And it also optimizes the computational cost, so if we can get other methods instead of trapezoid, we develop the numerical precision as high as possible.

Here is the pseudo code using this algorithm above :

```
PseudoCode :  Improved Euler Method
1.  do j=1, iter
2.      x(j+1)=x(j)*(1+h*(2-24*y(j)))/(1-h*(2-24*y(j)))
3.      y(j+1)=y(j)*(1+h*(-3+39*x(j))/2)/(1-h*(-3+39*x(j))/2)
4.  enddo
```

And, as we expected, excuting the code using `FORTRAN90`, we get the same result.

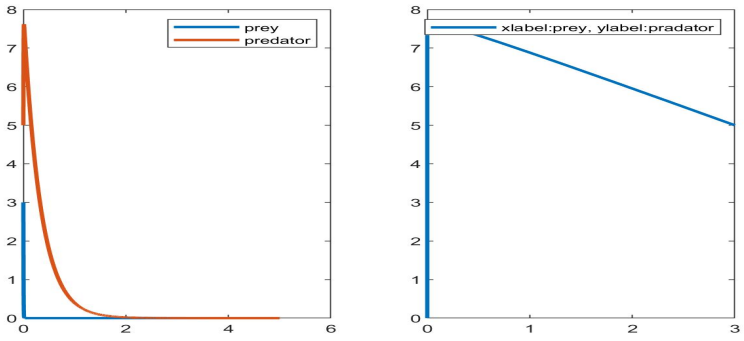

Figure 7: Improved Euler with $h = 0.001$

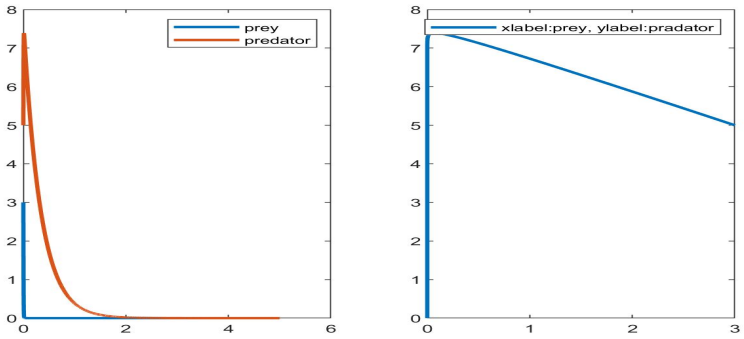

Figure 8: Improved Euler with $h = 0.0005$

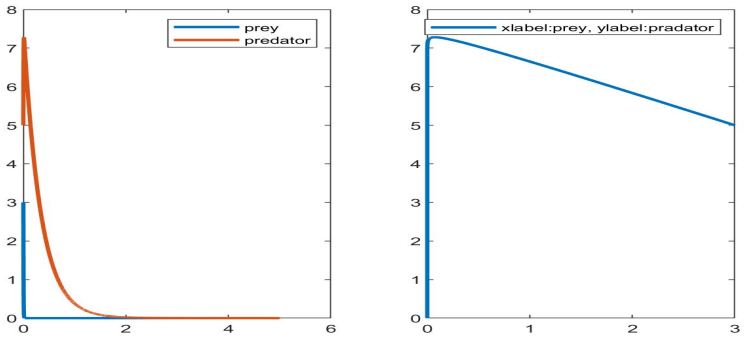

Figure 9: Improved Euler with $h = 0.00025$

2.5 Runge-Kutta Method

Runge-Kutta Method is one of the famous explicit method to solve ODE $\frac{dy}{dt} = f(y,t)$. Runge-Kutta method is to find intermediate point between t_n and t_{n+1} and we can predict the next value y_n more accurately. There 2 kinds of Runge-Kutta method will be introduced. The order means 'how much the error, the difference between approximation and exact value, is reduced'.

2.5.1 RK-2nd Order

The general form of Runge-Kutta Method 2nd Order is :

$$y_{n+1} = y_n + \gamma_1 k_1 + \gamma_1 k_2$$

where γ_1, γ_2 is constant and

$$k_1 = hf(y_n, t_n), \quad k_2 = hf(y_n + \beta k_1, t_n + \alpha h)$$

We're going to use the most popular form. Let $\gamma_1 = 0$, $\gamma_2 = 1$, $\alpha = \beta = \frac{1}{2}$, then we get

$$y_{n+1} = y_n + hf(y_n + \frac{1}{2}hf(y_n, t_n), t_n + \frac{1}{2}h)$$

Apply to our problem, then the final form is :

$$k_{1x} = h(4x_n - 48x_n y_n)$$

$$k_{1y} = h(-3y_n + 39x_n y_n)$$

$$k_{2x} = h(4(x_n + \frac{1}{2}k_{1x}) - 48(x_n + \frac{1}{2}k_{1x})y_n)$$

$$k_{2y} = h(-3(y_n + \frac{1}{2}k_{1y}) + 39(y_n + \frac{1}{2}k_{1y})x_n)$$

$$x_{n+1} = x_n + k_{2x}, \qquad y_{n+1} = y_n + k_{2y}$$

Here is the pseudo code using this algorithm above :

```
PseudoCode :  Runge-Kutta 2nd Order
1.  do j=1, iter
2.      k1x=h*(4*x(j)-48*x(j)*y(j))
3.      k1y=h*(-3*y(j)+39*y(j)*x(j))
4.      k2x=h*(4*(x(j)+k1x/2)-48*(x(j)+k1x/2)*y(j))
5.      k2y=h*(-3*(y(j)+k1y/2)+39*(y(j)+k1y/2)*x(j))
6.      x(j+1)=x(j)+k2x
7.      y(j+1)=y(j)+k2y
8.  enddo
```

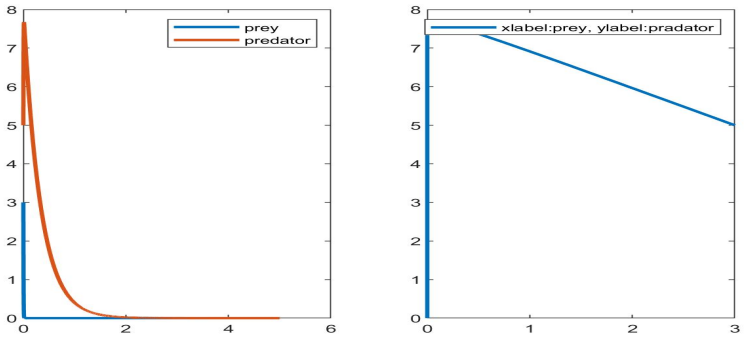

Figure 10: Runge-Kutta 2nd Order with $h = 0.001$

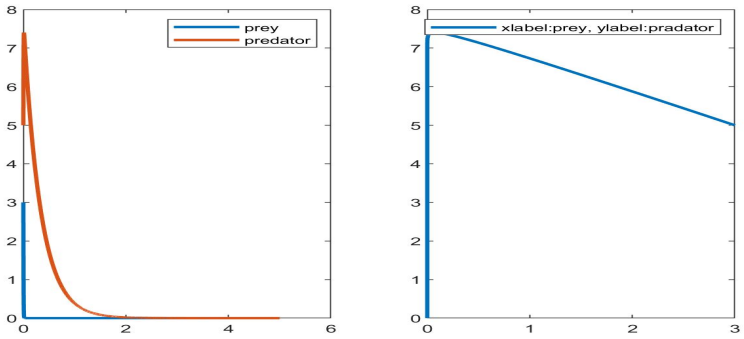

Figure 11: Runge-Kutta 2nd Order with $h = 0.0005$

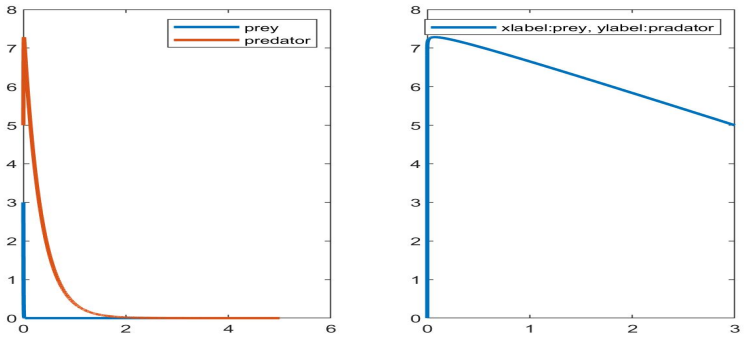

Figure 12: Runge-Kutta 2nd Order with $h = 0.00025$

2.5.2 RK-4th Order

Runge-Kutta fourth order method is much more precisely than any other numerical ode solver. So many automatic solver including MATLAB or MATHEMETICA is programmed with using Runge-Kutta 4-th Order method. The general form is :

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = f(y_n, t_n) \qquad k_2 = f(y_n + \frac{1}{2}k_1, t_n + \frac{h}{2})$$

$$k_3 = f(y_n + \frac{1}{2}k_2, t_n + \frac{h}{2}) \qquad k_4 = f(y_n + k_3, t_n + h)$$

Similar to RK-2nd Order method, we can apply that basic algorithm into our problem. Then the pseudo code is :

```
PseudoCode :  Runge-Kutta 4th Order
1.  do j=1, iter
2.      k1x=4*x(j)-48*x(j)*y(j)
3.      k1y=-3*y(j)+39*x(j)*y(j)
4.      k2x=4*(x(j)+k1x*h/2)-48*(x(j)+k1x*h/2)*y(j)
5.      k2y=-3*(y(j)+k1y*h/2)+39*x(j)*(y(j)+k1y*h/2)
6.      k3x=4*(x(j)+k2x*h/2)-48*(x(j)+k2x*h/2)*y(j)
7.      k3y=-3*(y(j)+k2y*h/2)+39*x(j)*(y(j)+k2y*h/2)
8.      k4x=4*(x(j)+k3x*h)-48*(x(j)+k3x*h)*y(j)
9.      k4y=-3*(y(j)+k3y*h)+39*x(j)*(y(j)+k3y*h)
10.     x(j+1)=x(j)+h*(k1x+k2x+k3x+k4x)/6
11.     y(j+1)=y(j)+h*(k1y+k2y+k3y+k4y)/6
12.  enddo
```

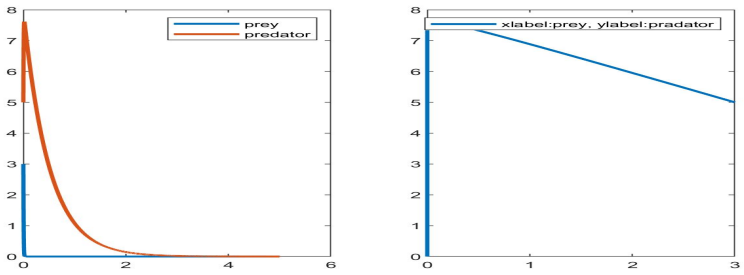

Figure 13: Runge-Kutta 4th Order with $h = 0.001$

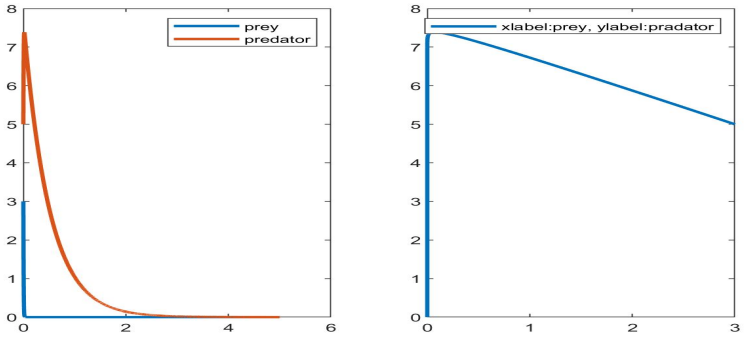

Figure 14: Runge-Kutta 4th Order with $h = 0.0005$

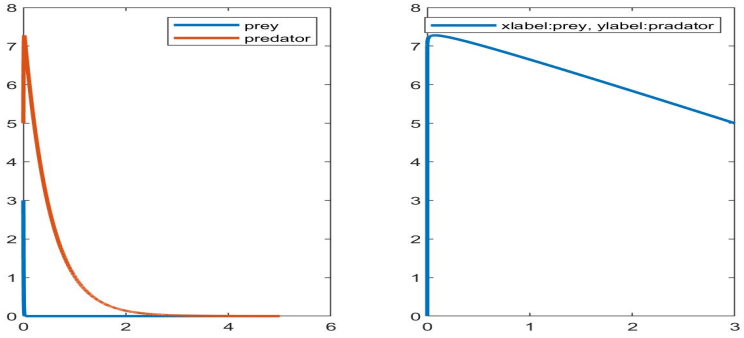

Figure 15: Runge-Kutta 4th Order with $h = 0.00025$

2.6 Leapfrog Method

Leapfrog Method is the one of the multi-step method which has second-order accurate globally. This method is derived by applying the central difference formula. By subtract these two equations below,

$$y_{n+1} = y_n + hy'_n + \frac{h^2}{2}y''_n + \cdots$$

$$y_{n-1} = y_n - hy'_n + \frac{h^2}{2}yll_n + \cdots$$

we get

$$y_{n+1} = y_{n-1} + 2hf(y_n, t_n) + O(h^3)$$

Here is the pseudo code using this algorithm above :

```
PseudoCode :  Leapfrog Method
1.  do j=1, iter
2.       x(j+1)=x(j-1)+(4*x(j-1)-48*x(j-1)*y(j-1))*h*2
3.       y(j+1)=y(j-1)+(-3*y(j-1)+39*x(j-1)*y(j-1))*h*2
4.  enddo
```

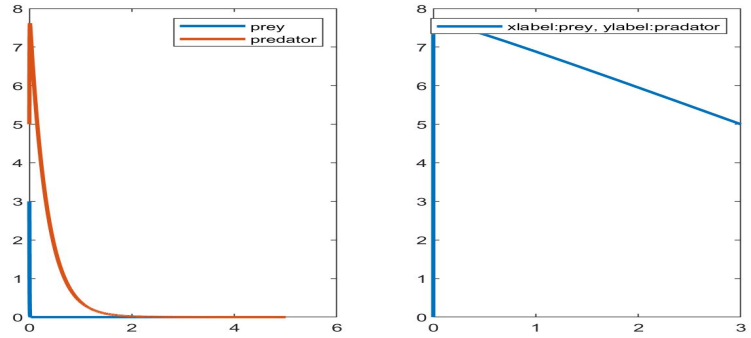

Figure 16: Leapfrog Method with $h = 0.001$

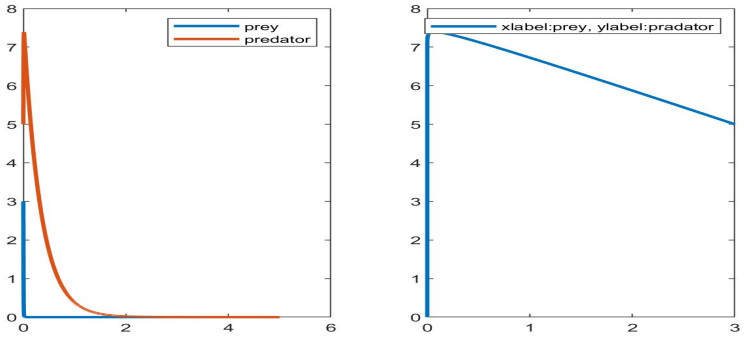

Figure 17: Leapfrog Method with $h = 0.0005$

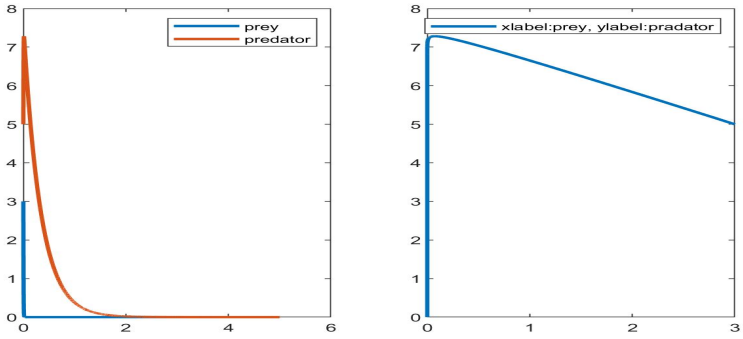

Figure 18: Leapfrog Method with $h = 0.00025$

As we noticed, because the results of all the methods are same, so we didn't make any mistakes, and execute programming code right.

2.7 Result and Discussion

As we can see, the result of all 6 methods, number of preys and predator plot per time and plot of them, are almost the same. Especially, by changing the value of h from 0.001 to 0.00025, the maximum value of the predators decrease a little, which means the precision is being higher. The same thing is noticed in the second plot between predators and preys. So if we choose h much smaller, we can easily find the critical point or equilibrium point of that differential equation system. In usual, when solving Lotka-Volterra predatory prey model, the solution function looks like kinds of something periodic. For find the periodic part of that differential equation, I use MATHEMATICA to use the internal function called ode45 and plot it on the extended domain. Let's see that `Figure 19`.

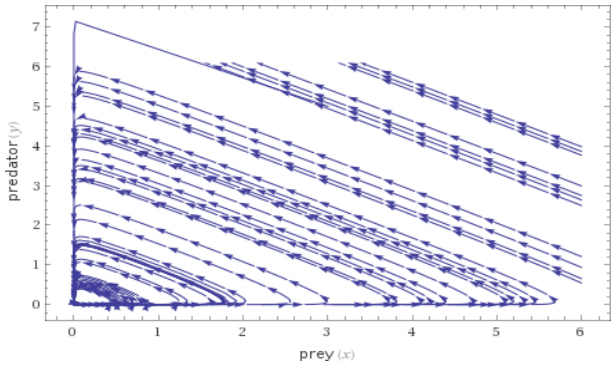

Figure 19: The extended solution of Lotka-Volterra model

At the extended time series, we finally notice that the number of predators and preys are circulating each other. At the result we solved above, at $0 \leq t \leq 5$, we may predict the two species would be extinct. Maybe we can find the appropriate values of coefficients when considering the + and - of each derivative. As we can see from the table below, we can notice that as there are some conditions of A, B, C and D satisfied, we can get appropriate solutions what we want.

-	$\frac{dx}{dt}$	$\frac{dy}{dt}$	Direction
I	+	−	↘
II	+	+	↗
III	−	+	↖
IV	−	−	↙

We can easily notice that the sign of each derivative changes when x and y are larger(or smaller) than $\frac{1}{D}$ and $\frac{1}{B}$. So we can see the cycle when the appropriate condition of coefficients A, B, C and D and initial value given.

cycle.jpg

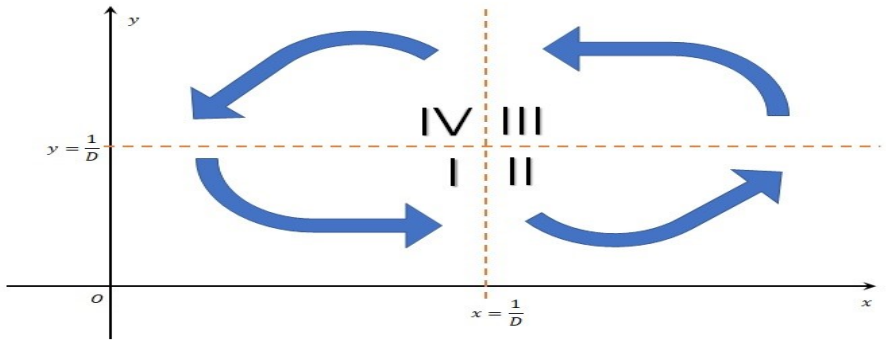

Figure 20: Lotka-Volterra critical point and Cycle

Now, let's change the coefficients in many cases. As we can see, if the coefficients get closer to the equilbrium point, $(\frac{1}{D}, \frac{1}{B})$ as possible, there are much numbers of cycles near that point. At the initial value is at distance sufficiently, like $x_0 = y_0 = 5$, we cannot find any cycle between preys and predators. So we need to set a initial value appropriately.

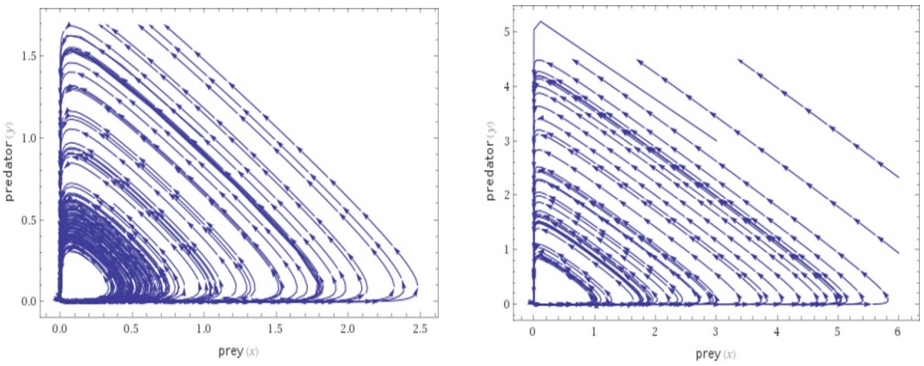

Figure 21: $x_0 = y_0 = 1$ and $x_0 = y_0 = 3$

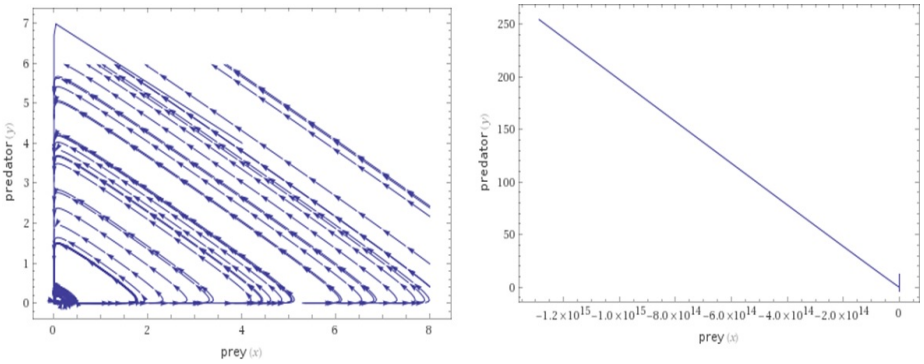

Figure 22: $x_0 = y_0 = 4$ and $x_0 = y_0 = 5$

Among the 6 methods for solving ODE, the well-known fact is that Runge-kutta is the most popular and has significant preciion almost higher than other methods, but it is very complicated to execute in programming code. However, Leapfrog method is easy to understand and execute in programming code, and also make the precision higher for expanding taylor series more longer. The best way to solve differential equation is, at first, we should know how much precise we want, and choose the easier way to find the solution satisfying our purpose.

3 Project II : Linear system

All the linear system can be identified with linear map equation (or matrix equation) following :

$$Ax = b$$

Where A is $m \times n$ matrix, and x and b is $n \times 1$ matrix. As we know, if given matrix A is invertible, then that equation above has a unique solution $x = A^{-1}b$. But, in many cases A is not a square matrix, so we cannot find the exact solution. There are many ways to decompose that given matrix A into several useful things, so in this project, we apply several methods and find the approximate solution, and measure the executing time for which method is much faster than others.
Here are some information of that given matrix we try to analyze.

Matrix Properties	
Number of rows	494
Number of columns	494
nonzeros	1,666
structural full rank?	yes
structural rank	494
Number of blocks from dmperm	1
Number of strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate	yes
positive definite	yes

3.1 Direct Method

Direct Methods is, at first, to decompose the given matrix A into very useful form. For solving the linear system, when A is triangular matrix then it'll be helpful for doing that. We introduce two methods, one called `PLU-Decomposition` and `QR-Decomposition`.

3.1.1 LU decomposition - Partial Pivoting

There are so many methods related with LU decomposition, which solve using Lower and Upper Triangular matrix. We'll introduce the simplest and strongest one among them.
It is well known that Gauss elimination can make given matrix A upper triangular. But in some cases, when the elements on diagonal can sometimes make error on solving the linear system. So we prevent it by pivoting diagonal entires with other entries. In detail, we should change the j-th and j_s-th row where $|a_{j_s j}| \geq |a_{kj}|$ for $k \geq j$ for more exact solution.
Here is the algorithm using LU decomposition with partial pivoting.

```
PseudoCode :  LU decomposition with Partial Pivoting
1.  do j=1, n-1
2.       jsave=MAXLOC(ABS(a(j:n,j)),1)
3.       js=jsave+j-1
4.       change a(j,:)  and a(js,:)
5.       change b(j) and b(js)
6.       do k=j+1, n
7.         m=a(k,j)/a(j,j)
8.         a(k,k:n)=a(k,k:n)-m*a(j,k:n)
9.         b(k)=b(k)-m*b(j)
10.      enddo
11.enddo
```

Our matrix size is $n = 494$, so we can apply this algorithm and execute to find the solution x. The costed cputime is :

$$t_{PLU} = 13.340850(s)$$

and we can also the plot of the x with its index on x-label.

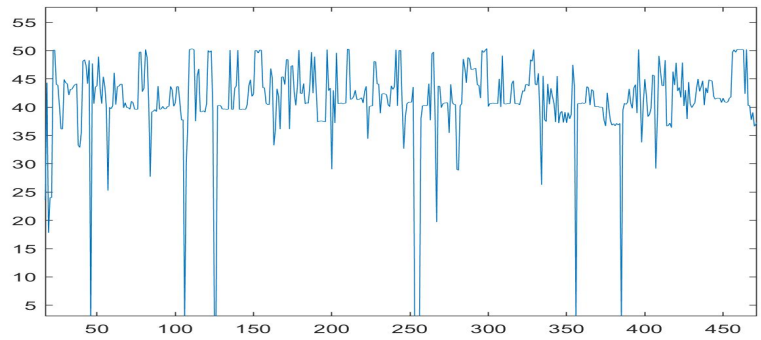

Figure 23: Solution from LU decomposition with Partial Pivoting

3.1.2 QR-Decomposition

QR decomposition is basically make each column of given matrix mutually orthogonal. So it costs a lot more than LU decomposition, but it is quite useful when the given matrix is non-square. (In fortunate, our matrix is 494×494 square matrix). there are three famous QR decomposition, which are `Gram-Schmidt Orthogonalization`, `Householder transformation`, and `Givens Rotation`. We use Givens rotation to solve that given system.
Givens rotation makes columns of the matrix being orthogonal by rotation on the hyperplane. So, at first, we should find *theta* of the k-th and l-th coordinates as follow :

```
PseudoCode :  Givens Rotation I : find θ
1.  v(:)=a(k,:)
2.  w(:)=a(l,:)
3.  a(k,:)=cos θ*v(:)+sin θ*w(:)
4.  a(l,:)=− sin θ*v(:)+cos θ*w(:)
```

We can identify the above algorithm making column like that with multiplying the matrix G_j on left side of the A. The G_j is following : (c_j and s_j means $\cos\theta_j$, $\sin\theta_j$ respectively.)

$$\begin{pmatrix} & \vdots & & \vdots & \\ \cdots & c_j & \cdots & s_j & \cdots \\ & \vdots & & \vdots & \\ \cdots & -s_j & \cdots & c_j & \cdots \\ & \vdots & & \vdots & \end{pmatrix}$$

For more stability, we compute $\sqrt{1-s_j^2}$ than $sqrt1 - c_j^2$ if $|s_j| \leq |c_j|$. For this, define $\rho(j)$ as

$$\rho(j) = \begin{cases} 1 & (c_j = 0) \\ \frac{s_j}{2} & (|s_j| < |c_j|) \\ \frac{2}{c_j} & (|c_j| \leq |s_j| \end{cases}$$

Now, put $k'(j) = sign(c_j)k(j)$, $l'(j) = sign(s_j)l(j)$, and store only $(k'(j), l'(j), \rho(j))$.
For the recovery of G_j, use the following algorithm.

```
PseudoCode :  Givens Rotation II : find G_j
1.  if(ρ(j) == 1) then c_j = 0; s_j =sign l'(j)
2.  elseif(|ρ(j)| < 1) then s_j = 2ρ(j),c_j =sign k'(j)√(1 − s_j^2)
3.  else c_k = 2/ρ(j); s_j =sign l'(j)√(1 − c_j^2)
4.  endif
```

It may be easier for using two subroutine and apply above algorithms appropriately. The executed result is below :

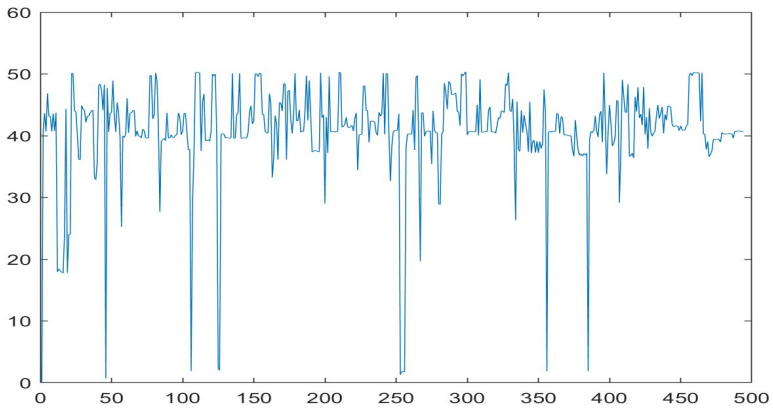

Figure 24: Solution from Givens Rotation

and the calculated time is

$$t_{QR} = 18.862775(s)$$

As we can see, the result is very similar to LU decomposition solution, so we can check we programmed that method right. As expected, the costed time of QR decomposition is larger than of LU decomposition, because the amount of calculation of QR decomposition is as twice as of LU decomposition. Not only for factorizing, but also for making their columns mutually orthogonal advantages us for analyzing it.

3.2 Iterative Method

In many cases, linear system has no exact solution, so we should solve 'least square' problem, finding the most appropriate solution fitting to our system. `iterative Method` is the way we can find the solution within the error bound we already set by repeating an algorithm. In this section, we will introduce 3 iterative methods, `Jacobi`, `Gauss-Seidel`, and `SOR` methods.

3.2.1 Jacobi Method

Jacobi method is an iterative method with initial approximation $x^{(0)}$. Decompose $A = P + N$ where $P = D$ the diagonal part of the given matrix A and N is the other part. Let $r^{(j)} = b - Ax^{(j)}$ be the jth residual. Assume D is invertible and satisfies certain conditions for A such that Jacobi method converges. We can simplify the Jacobi method algorithm below :

```
PseudoCode :  Jacobi Method
1.  initial value x(:)
2.  do j=1, max-iter
3.    do k=1, n
4.      x_next(k) = (b(k)-dot-product(a(k,1:k-1),x(1:k-1))-dot-product(a(k,k+1:n),x(k+1:n)))/a(k,k)
5.    enddo
6.  x = x_next
7.  enddo
```

Then, let max-iter=20000 be given and compare the solution before.
Here is a cputime costs and iterative solution.

$$t_{Jacobi} = 545.863482(s)$$

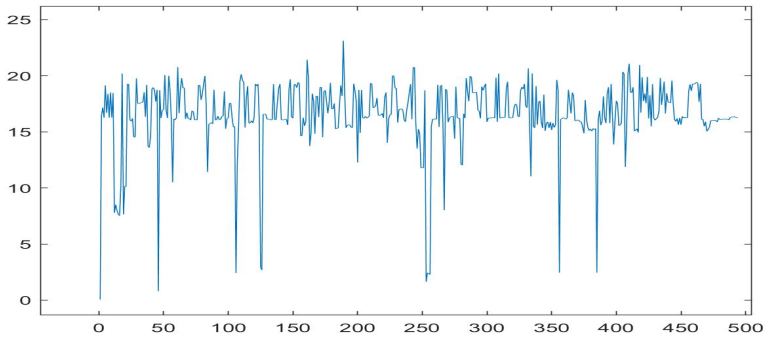

Figure 25: Iterative solution from Jacobi method.(iter=20000)

As we can see, the value of the solution is quite different from before direct solution, but the variance among components in solution look quite similar overall.

3.2.2 Gauss-Seidel Method

Gauss-Seidel Method is very similar to Jacobi, but more accurate. There is one more decomposition than Jacobi method, that is $P = D + L$ and $N = U$ where L and U are strictly lower and upper triangular matrix of the given A. If we consider the lower triangular part L, then we can update the lower part in Jacobi method.
Here is the algorithm using Gauss-Seidel method below :

```
PseudoCode :  Gauss-Seidel Method
1.  initial value x(:)
2.  do j=1, max-iter
3.    do k=1, n
4.      x(k) = (b(k)-dot-product(a(k,1:k-1),x(1:k-1))-dot-product(a(k,k+1:n),x(k+1:n)))/a(k,k)
5.    enddo
6.  enddo
```

Then, apply this algorithm to our program with the given matrix A. Then, we get the cputime cost is :

$$t_{Gauss-Seidel} = 418.422275(s)$$

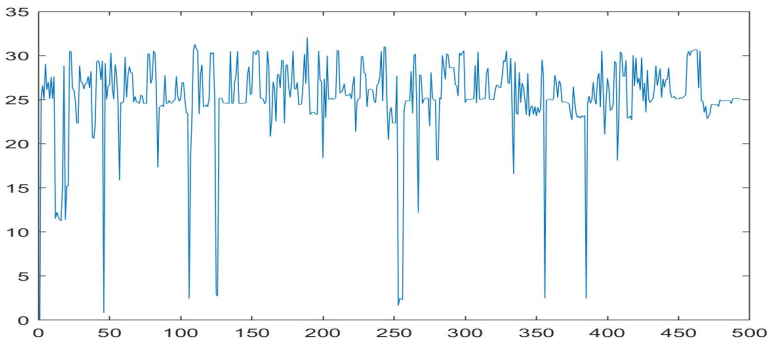

Figure 26: Iterative solution from Gauss-Seidel Method(iter=20000)

As we can see, the result of the solution is quite similar to before, and it is much faster and more accurate than Jacobi methods compared to the solution from direct method.

3.2.3 SOR Method

The SOR Method, Successive Overrelaxation Scheme, we should choose a good P because it will result in significantly different convergence depending on A. Also a modification of iterative vector may provide significant improvement in convergence. SOR methods is the modification of Jacobi or Gauss-Seidel method by using a 'weighted' parameter w.
We can apply the parameter w on the Gauss-Seidel method, then we get

$$x^{(j+1)}(k) = w(b_k - \sum_{l=1}^{k-1} a_{kl}x_l^{(j+1)} - \sum_{l=k+1}^{n} a_{kl}x^(j)_l)/a_{kk} + (1-w)x_k^{(j)}$$

This can be written as

$$a_{kk}x_k^{(j+1)} + w\sum_{l=1}^{k-1} a_{kl}x_l^{(j+1)} = w(b_k - \sum_{l=k+1}^{n} a_{kl}x_l^{(j)}) + (1-w)a_{kk}x_k^{(j)}$$

In the vector form, they are equal to

$$(D + wL)x^{(j+1)} = w(b - Ux^{(j)} + (1-w)Dx^{(j)}$$

Now, put P and N such that

$$P_w = \frac{1}{w}(D + wL) \qquad N_w = \frac{1}{w}\{(1-w)D - wU\}$$

Then we can write the final form of SOR

$$P_w x^{(j+1)} = N_w x^{(j)} + b$$

We already know that the given 494×494 matrix A is symmetric, so for more saving in cputime costs, we can use the Symmetrized SOR in the algorithm.
Here is the algorithm using SSOR. First, update the vector x, from the first to last components. In the second step update it from the last to first components. With a parameter , let us consider the following modification of the jth iterative vector.

```
PseudoCode :  SOR Method(Symmetrized)
1.  initial value x(:)
2.  do j=1, max-iter
3.    do k=1, n
4.       x(k) = w(frac{b(k)-sum_{l=1}^{k-1} a_kl x(l)-sum_{l=k+1}^{n} a_kl x(l)}{a(k,k)}) + (1-w)x(k)
5.    enddo
6.    do k=n,1,-1
7.       x(k) = w(frac{b(k)-sum_{l=1}^{k-1} a_kl x(l)-sum_{l=k+1}^{n} a_kl x(l)}{a(k,k)}) + (1-w)x(k)
8.    enddo
9.  enddo
```

Now we apply this algorithm to our linear system while changing the value of w and find which w is the best to solve it.

$$t_{0.3} = 384.045643(s)$$

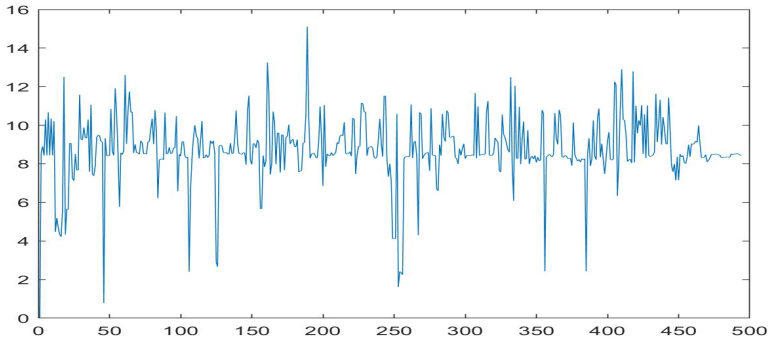

Figure 27: Iterative solution from SOR $w = 0.3$(iter=20000)

$$t_{0.7} = 570.291953(s)$$

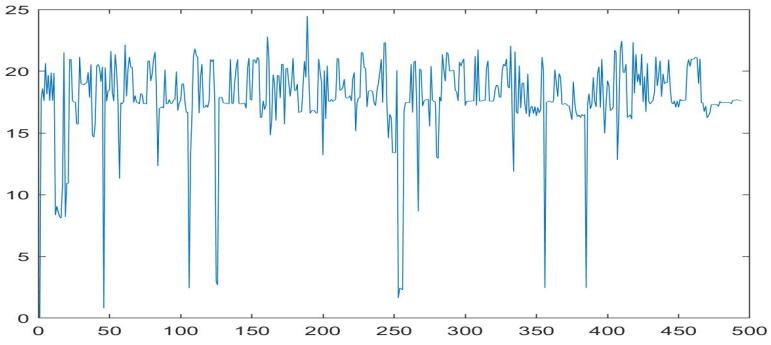

Figure 28: Iterative solution from SOR $w = 0.7$(iter=20000)

$$t_{1.0} = 560.195636(s)$$

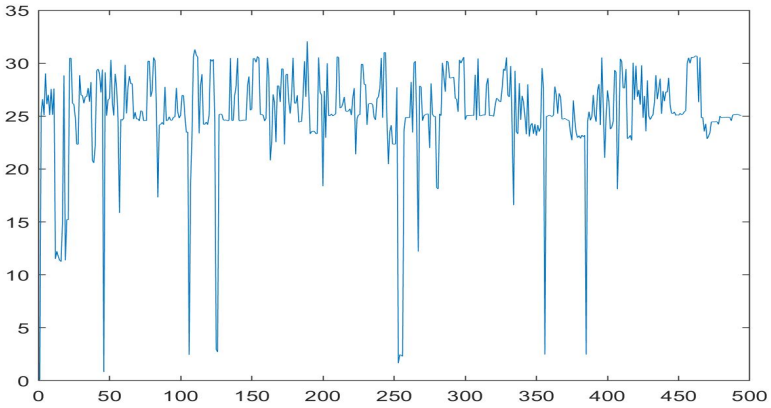

Figure 29: Iterative solution from SOR $w = 1.0$(iter=20000)

$$t_{1.3} = 523.870957(s)$$

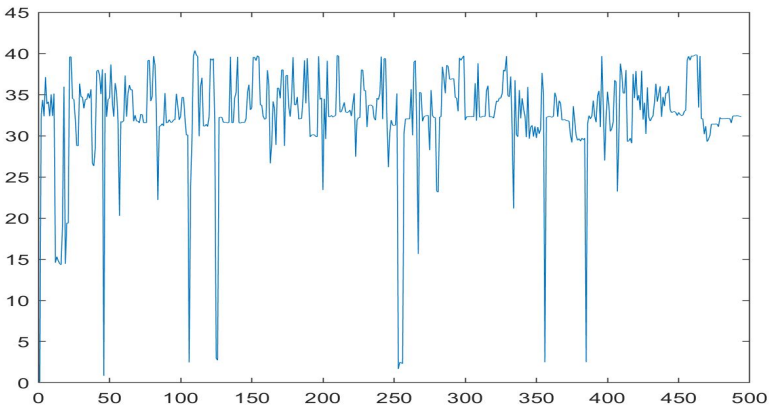

Figure 30: Iterative solution from SOR $w = 1.3$(iter=20000)

$$t_{1.7} = 317.516215(s)$$

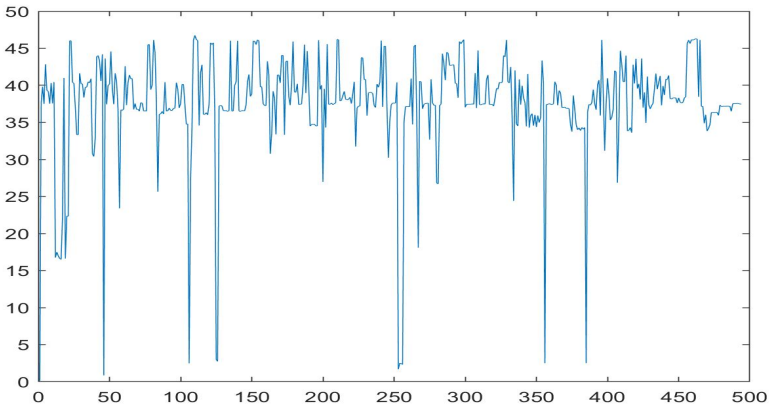

Figure 31: Iterative solution from SOR $w = 1.7$(iter=20000)

$$t_{1.9} = 305.731392(s)$$

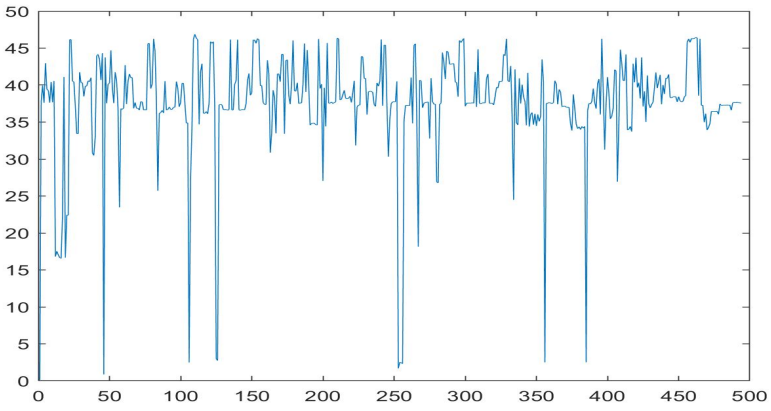

Figure 32: Iterative solution from SOR $w = 1.9$(iter=20000)

As we can see, the larger the value of w is, the solution is getting similar to the solution from direct methods.(When looking considerately at y-label.)

3.3 Result and Discussion

At first, we should notice that the costed cpu times are various with each methods, respectively. (Note that the costed time of iterative methods are based on iteration 20000.)

Method Name	costed CPU time(seconds)
LU Poviting	13.340850
QR Givens	18.862775
Jacobi Iteration	545.863842
Gauss Seidel	418.422275
SOR($w = 0.3$)	384.0.45643
SOR($w = 0.7$)	570.291953
SOR($w = 1.0$)	560.195636
SOR($w = 1.3$)	523.870957
SOR($w = 1.7$)	317.516215
SOR($w = 1.9$)	305.731392

We easily notice that direct methods are much faster than iterative methods. It's because the given 494×494 matrix A is square and also invertible. So, if the given matrix is square(in many cases matrix having sufficiently large size is hard not to be invertible), direct method will be useful for solving that linear system. Let's see two iterative methods, Jacobi and Gauss-Seidel. Gauss- Seidel is a little faster than Jacobi method, namely, converges to a exact solution faster. I plot the graph between x-axis be iter(from 1 to 20000) and y-axis be l^2-norm error value.

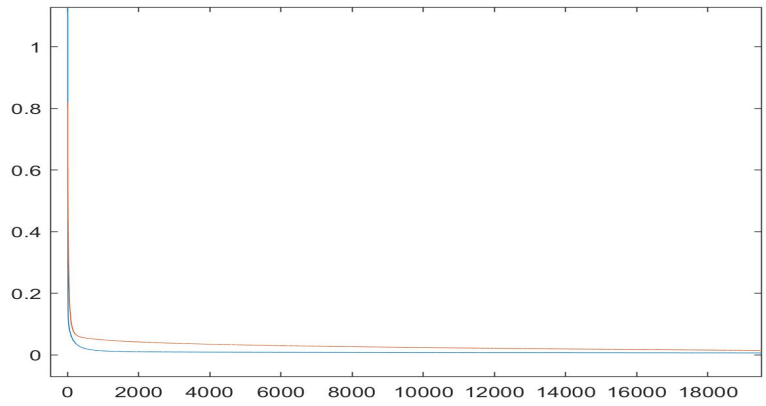

Figure 33: Jacobi speed versus Gauss-Seidel speed

The blue line of that figure is, as we expected, is Gauss-Seidel method and the other red one is Jacobi method. After the number of iteration 300, the speed of convergence of Gauss-Seidel method is twice faster than Jacobi method, and it is because Gauss-seidel algorithm update the solution everytime they get new one, while Jacobi doesn't. We can also compare the speed of convergence among SOR methods. We can expect that in this matrix A, the larger the value of w is, the faster the convergence of exact solution is. Here is the plot between the number of iteration and l^2-norm error value, combining all the cases of SOR changing w respectively.

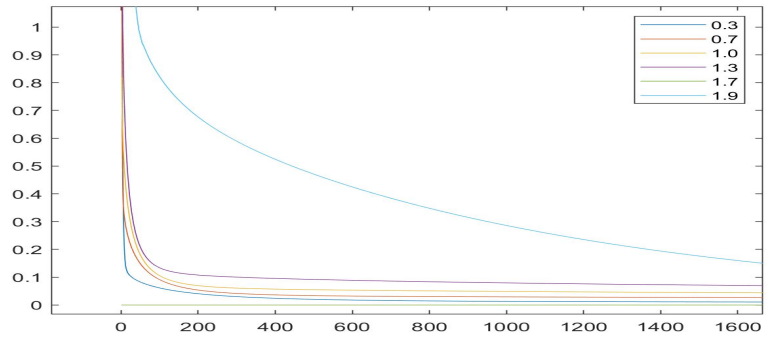

Figure 34: SOR error comparision (Part I)

On the number of iteration between 1 and 1600, the speed $w = 1.9$ is quite slower than other values of w, while $w = 1.7$ is the fastest one among them. But, note that if we set the error criterion be 10^{-4}, no one cannot satisfy the being-inner error condition.

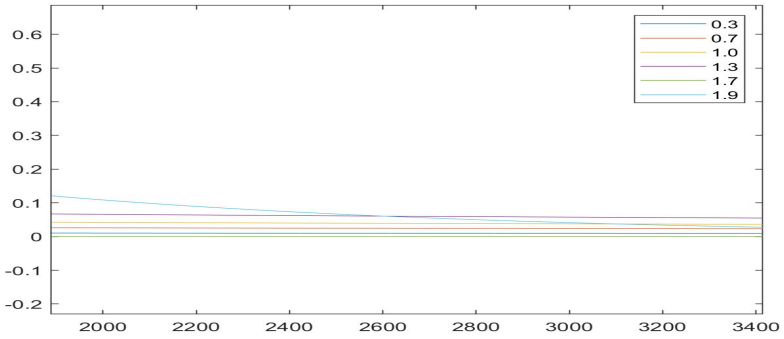

Figure 35: SOR error comparision (Part II)

We should notice that after the number of iteration goes 2600 or above, $w = 1.9$ blue line surpass all the other lines and start to become the fastest convergence speed among them. So in our problem, choose $w = 1.9$ as a relaxation parameter is the best way to get a solution.
In fact, if $w = 1$, the SOR method is exactly same to the Gauss-Seidel method. It is well-known that SOR fails to converge if w is outside the interval $(0, 2)$(`Kahan's theorem`). When w is in the interval $(0, 1)$, then the value of w is called `underrelaxation parameter`, and the other case is called `overrelaxation parameter`. And as we can see, if we choose the optimal w, then it is possible to save the expense of such amount of computations.
Above the table of the information of the given 494×494 matrix show that A can be decomposed by Cholesky method. It means A is positive-definite and symmetric, and SOR iterative method guaranteed to converge for any value of w between 0 and 2, so we can get a similar result from all of them. In principle, if we know the spectral radius ρ of Jacobi iteration matrix, then one can determine a priori the theoratically optimal value of w for SOR :

$$w_{opt} = \frac{2}{1 + \sqrt{1 - \rho^2}}$$

4 End of the project

I think that the strongest merit of numerical analysis on complicated mathematical problem is to visualize the problem and solution of that. Although we failed to get a exact solution or approximation of that, we understand the problem one more and correct the wrong part. The most important thing is, when solving differential equation system or linear system, we should start to discretize the problem using matrix and identify with the matrix least square problem. In some cases, direct methods like PLU or QR can be the strongest way, but in the other cases, optimized iteration method will be the best. Also, if we get non-linear system when discretizing differential equation, such as Lotka-Volterra predatory and prey model, we can use many integeration method to solve it. So, at first we should understand the mathematical principle on them and apply to our problems by choosing a appropriate way.