

Warm-Up Question: An Initial Value Problem for ODE and Explicit Euler

Rüştü Erciyes Karakaya

July 2022, 2

1 Problem Definition

$\frac{dx}{dt} = 2y$ and $\frac{dy}{dt} = -2x$, when $t \in [0.0, 10.0]$ and $x(0)=0$, $y(0)=1$;

$x(t+h) = x(t) + hx'$ and $y(t+h) = y(t) + hy'$ by Euler, while $t \rightarrow t+h$

Start with $h=\frac{1}{10}$. **Aim:** Observe $x(10)$ and $y(10)$ with respect to h .

2 Algorithm Design¹

Firstly, I set the terminal points of t (0 and 10), initial values of x and y , h and number of times of iteration accordingly. Additionally, the expressions used in algorithm design are directly implemented what explained in problem definition above.

As a last, I'm just putting all design in a for loop which iterates for the different h values to create table after all. You can see the algorithm in following page as MATLAB codes.

3 Comments and Outcome

That's not a so sophisticated algorithm and we did same things in Numerical Analysis Course actually; that's why the total time I wasted for algorithm design and implementation: 25-30 min.

outcomes = 5x3 table

	h	last_value_of_y	last_value_of_x
1	0.1000	4.4730	5.5224
2	0.0100	0.5014	1.1137
3	0.0050	0.4517	1.0087
4	0.0025	0.4292	0.9597
5	0.0001	0.4089	0.9148

Figure 1: Outcomes from MATLAB

¹The textbook I used in this part: Numerical Analysis, 9th Edition. Richard L. Burden and J. Douglas Faires.

```

%initial point settings
a=0.0;b=10.0;
%h=0.1;
w_for_x(1)=0.0;w_for_y(1)=1.0;
t=0;
h=[0.1,0.01,0.005,0.0025,0.001,0.0005,0.00001];
C=cell(2,length(h));
for j=1:length(h)
    % # of times loop iteration
    N=(b-a)/h(j);
    w_for_y=[];w_for_x=[];
    t=0;w_for_x(1)=0.0;w_for_y(1)=1.0;
    for i=1:N
        w_for_x(i+1)=w_for_x(i)+h(j)*(2*w_for_y(i));
        w_for_y(i+1)=w_for_y(i)+h(j)*(-2*w_for_x(i));
        t=t+h(j);
    end
    last_value_of_x(j)=w_for_x(N+1);
    last_value_of_y(j)=w_for_y(N+1);

    C(1,j)={w_for_x};
    C(2,j)={w_for_y};
end
h=transpose(h);
y_1=transpose(last_value_of_y);
x_1=transpose(last_value_of_x);
outcomes=table(h,y_1,x_1)
solution_of_error=log(sqrt(((cos(20))-last_value_of_y).^2+((sin(20))-last_value_of_x).^2));
new_h=log(transpose(h));
%loglog(new_h,solution_of_error)
plot(new_h,solution_of_error)

```

Figure 2: Algorithm Implementation

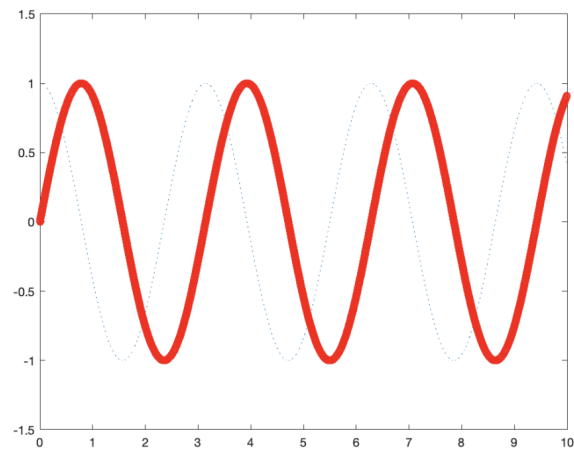


Figure 3: Functions $(x(t))$ and $y(t)$ as conclusion of Numerical Analysis

4 Analytical Solution

Assume $x(t) = B_1 \cos(t\omega) + B_2 \sin(t\omega)$.

By using given ODEs in problems definition, what we obtained is the following:

$$y = \dot{x}/2 \Rightarrow \dot{y} = \ddot{x}/2 = -2x \Rightarrow \ddot{x} + 4x = 0 \quad (*)$$

If we put our x definition in assumption into the equation (*) when $t=0$:

$$\begin{aligned} -B\omega^2 \cos(0) + 4B \cos(0) &= 0 \Rightarrow \omega = 2, B_1 = 0 \quad (\text{by using } x(0)=0) \\ -2B_1 \sin(2t) + 2B_2 \cos(2t) &= 1 \Rightarrow B_2 = 1 \quad (\text{by using } y(0)=1) \end{aligned}$$

Conclusion-I: $x(t) = \sin(2t)$

Assume $y(t) = B_1 \cos(2t) + B_2 \sin(2t)$.

By using given ODEs in problems definition, what we obtained is the following:

$$x = \dot{y}/-2 \Rightarrow \dot{x} = \ddot{y}/-2 = -2y \Rightarrow \ddot{y} - 4y = 0 \quad (*)$$

If we put our y definition in assumption into the equation (*) when $t=0$:

$$\begin{aligned} -2B_1 \sin(2t) + 2B_2 \cos(2t) &= 0 \Rightarrow B_2 = 0 \quad (\text{by using } x(0)=0), \\ \text{as known we found } \omega=2 \text{ and } y(0) &= 1 \Rightarrow B_1 = 1 \end{aligned}$$

Conclusion-II: $y(t) = \cos(2t)$

Remarkable Point: If you try to solve this IVP by using separable equation then you will get $x(t)^2 + y(t)^2 = 1$. That's why I tried sinusoidal function in a first for the assumption.

Solutions of Equations in One Variable, Numerical Root-Finding

Rüştü Erciyes Karakaya

July 2022, 7

1 Problem Definition

Aim: Compute $\frac{\pi}{2}$.

Method: By finding the root, $\sin(\pi)=0$, of sinusoidal function which is numerically generated before.¹

2 Algorithm Design²

Take a and b which satisfies $\sin(a) < \sin(\pi)=0 < \sin(b)$;

Then calculate take $\sin(\frac{a+b}{2})$ check whether is zero or not, if it's not;

- if $\sin(\frac{a+b}{2}) > 0$ update $b = \frac{a+b}{2}$; else $a = \frac{a+b}{2}$

Execute this algorithm until $\sin(\frac{a+b}{2})=0$ or is close to 0 at least around 5-6 digits.

Actually this algorithm is basically the method of *Bisection* which is based on Intermediate Value Theorem and Binary searching. According to IVT; let $f(x)$ be continuous function and it has root in $[a,b]$ definitely, when a and b are defined above. In our case, we have to modify the bisection method because of our $f(x)$, which is not continuous and $f(x)=\sin(x)$ is numerically derived discrete function.

Modification on Bisection Algorithm: Take a and b as index of numerically generated $\sin(x)$ function. Take $p = \lceil \frac{a+b}{2} \rceil$.

In this way, we can visit values of the numerically generated discrete function properly, you can see the details in MATLAB codes clearly. MATLAB codes is just implementation of modified version of Bisection Method.

Conclusions: The first answer of next page represent the approximation for $\frac{\pi}{2}$ and the second ans. refer to approximation for root of $\sin(x)$, as seen it is close to 0 around 7-digits.

What can be faster method?: As a known fact, Newton's Method for root-finding is always suggested as fastest root-finder numerically. In some cases, The method we used require so many iteration.³

¹Check the first week (July, 2) document for numerically generated sinusoidal function.

²The textbook used in this part: Numerical Analysis, 9th Edition. Richard L. Burden and J. Douglas Faires.

³If your points around root are increasing, namely h is decreasing, number of iteration we require will begin to increase drastically. It's problem for more precise measurements actually.

```

...% previous code
...

%implementing the bisection algorithm with discrete function
a=find(t==1);b=find(t==2);
tol=1;N0=20;
i=0;FA=w_for_x(a);
while(i<N0)
    p=ceil((b+a)/2);
    FP=w_for_x(p);
    if(FP==0 || (b-a)/2 < tol)
        value=p;
        break
    end
    i=i+1;
    if FA*FP > 0
        a=p;FA=FP;
    else
        b=p;
    end
end
t(p)

ans = 1.5708

w_for_x(p)

ans = 6.5360e-07

```

Figure 1: Supplementary part of the previous week's MATLAB program.

Further Investigation on IVP for ODE: Local Extrapolation

Rüştü Erciyes Karakaya

July 2022, 14

1 Problem Definition

Main Question: How can we speed up the integration?

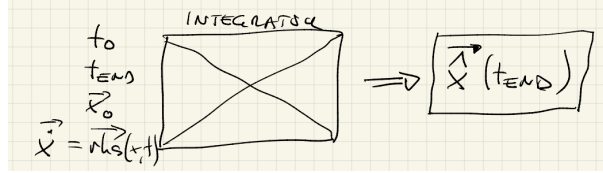


Figure 1: Integration

As seen in Figure-I, Integrator take starting-end points, initial values and differential equation as input and gives the value of function at end-point. Previously, we designed Integrator by using Explicit Euler method which is *not optimal*.

2 Algorithm Design

Midpoint Rule¹: It's based on *Midpoint Rule* for Integration which can be formulated as $\omega_{i+1} = \omega_{i-1} + 2h\omega_i$. This is actually extrapolating to solve ODE properly.² As seen in equation, in this method, we need two initial point; that's why one of them can be derived by Euler's Method to start algorithm while the another is given. Now, we are in the 2nd order method and slope of the loglog plot is expected accordingly. Additionally, time consume of algorithm is the our another concern of course.

Important Analytical Outcome: $x_{real} = 2\hat{x}_{h/2} - \hat{x}_h$ ³

Further Question: How can we speed up root-finding?

¹I used this method to speed up instead of using Euler-Richardson Algorithm.

²Check Figure-4 for more visualization to understand the concept of local extrapolation.

³ \hat{x} 's in there are just Taylor Expansion with given h values and quadratic error term. This is nothing but just a basic conclusion from Euler's for local extrapolation. This is actually main structure of Euler-Richardson Algorithm.

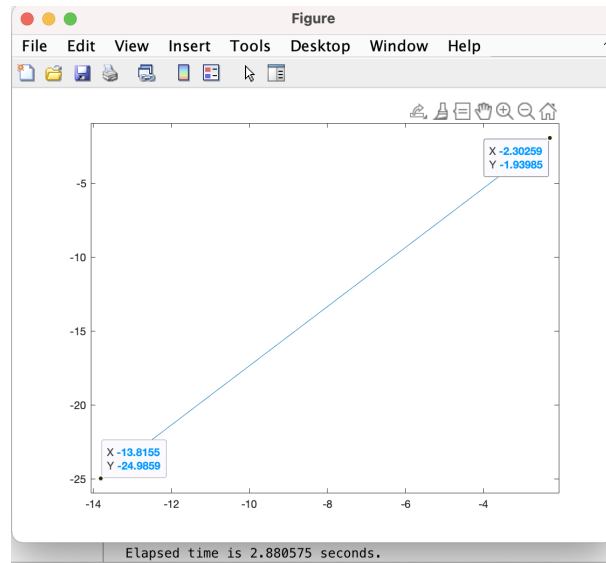


Figure 2: Outputs, Loglog Plot with terminal points and Elapsed Time of the program.

```
tic
%f=x'=2*y(t)
%initial point settings
a=0.0;b=10.0;
%h=0.1;
w_for_x(1)=0.0;w_for_y(1)=1.0;
t=0;
h=[0.1,0.01,0.005,0.0025,0.001,0.0005,0.00001];

for j=1:length(h)
    % # of times loop iteration
    Nh=(b-a)/h(j);
    w_for_yh=[];w_for_xh=[];

    t=0;w_for_xh(1)=0.0;w_for_yh(1)=1.0;
    w_for_xh(2)=w_for_xh(1)+h(j)*(2*w_for_yh(1));
    w_for_yh(2)=w_for_yh(1)+h(j)*(-2*w_for_xh(1));
    for i=2:Nh
        w_for_xh(i+1)=w_for_xh(i)+2*h(j)*(2*w_for_yh(i));
        w_for_yh(i+1)=w_for_yh(i)+2*h(j)*(-2*w_for_xh(i));
        t=t+h(j);
    end

    last_value_of_y(j)=w_for_yh(Nh+1);
    last_value_of_x(j)=w_for_xh(Nh+1);
end
h=transpose(h);
y_1=transpose(last_value_of_y);
x_1=transpose(last_value_of_x);
outcomes=table(h,y_1,x_1)
solution_of_error=log(sqrt(((cos(20))-last_value_of_y).^2+((sin(20))-last_value_of_x).^2));
new_h=log(transpose(h));
%loglog(new_h,solution_of_error)
plot(new_h,solution_of_error)
toc
```

Figure 3: Implementation of the Algorithm and Sketching Graph

3 APPENDIX

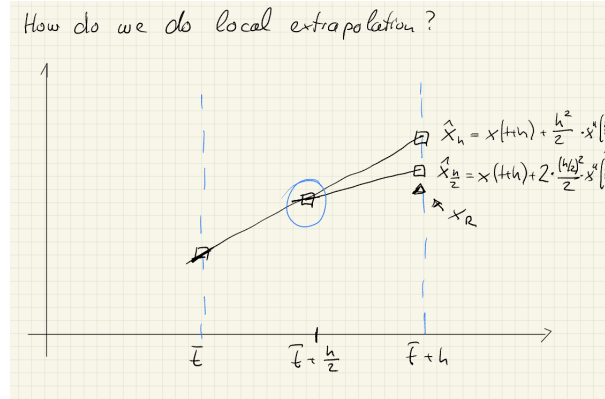


Figure 4: Locally Extrapolating for the point $t+h$

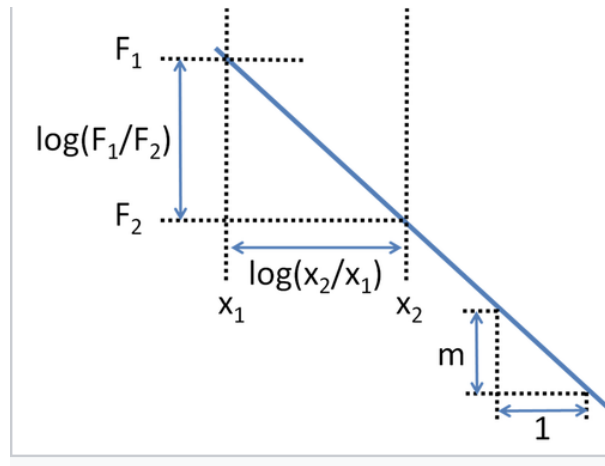


Figure 5: Slope of the Loglog plot gives the order of accuracy of the algorithm. As seen in Figure-2, its slope is roughly 2 which is the order of Midpoint Rule actually.

SIR Models without Vital Dynamics

Rüştü Erciyes Karakaya

July 2022, 21

1 Introduction

This type of models are used to model infectious diseases, actually very general mathematical modelling tool. The abbreviation of SIR is meaning respectively that Susceptible, Infectious, or Recovered. Each variable (S, I, R) has ODE with initial values and if we formulate the problem:

Let $N := \text{population} = S(t) + I(t) + R(t) = 1000$, infectious rate $= \beta = 0.4$, recovery rate $= \gamma = 0.04$; with initial values $S(0) = 997$, $I(0) = 3$, $R(0) = 0$, then the ODEs are:

$$\dot{S} = -\frac{\beta SI}{N}, \quad \dot{I} = \frac{\beta SI}{N} - \gamma I, \quad \dot{R} = \gamma I$$

Interpretation:

\dot{S} can be interpreted that the interaction between Susceptible and Infectious peoples decrease the number of susceptible peoples with rate of infectious.

\dot{I} can be interpreted that the interaction between Susceptible and Infectious peoples increase the number of infectious peoples with rate of infectious and decreases with rate of recovery.

\dot{R} can be interpreted that the infectious people increase the number of recovered people with rate of recovery.

Aim: Obtaining the $S(t)$, $I(t)$ and $R(t)$ by using our integrator.¹

Further Investigation: As seen we are working with fixed population and coefficients (β, γ) . However, population and coefficients may differ. For example, infectious rate and population is bigger in Paris than Heidelberg, we can encounter many different scenario like that. At the end of the day, can we design the algorithm which obtaining the SIR Model-curves of the scenario by estimating coefficients just with given data?²

¹Integrator is designed in previous weeks, you can check the documents.

²If you check the Figure-I, you can see the further design to attack on this problem. In this week, we just focus on first part of the design in Figure-I.

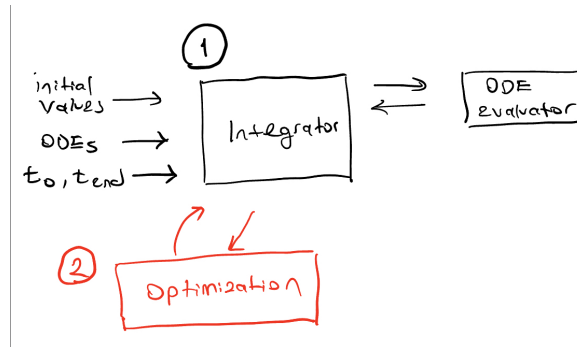


Figure 1: In this week's problem we just focus on first part of this design. However in overall; Integrator works as sub-routine of Optimization, while ODE evaluator works as sub-routine of Integrator.

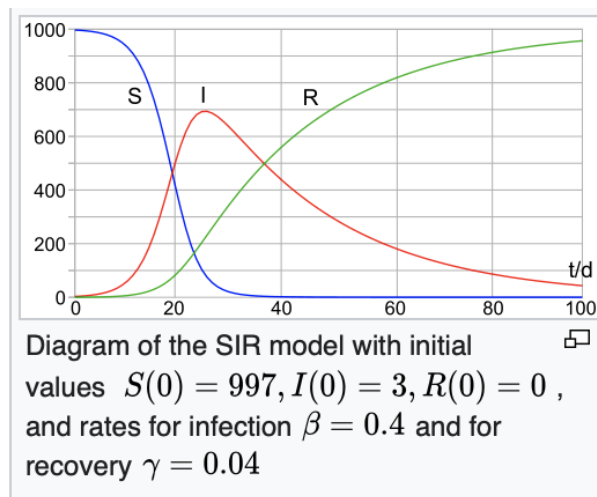
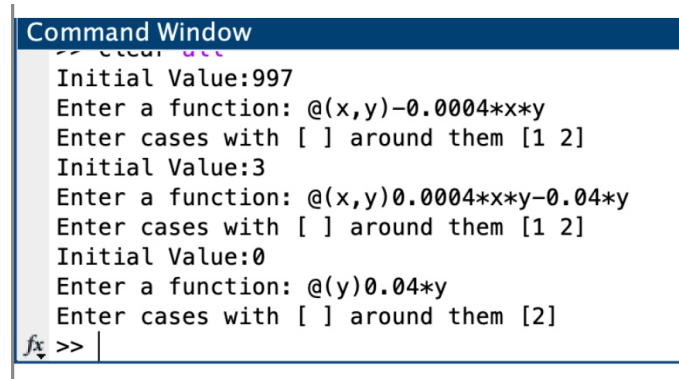


Figure 2: Expected Outcome with fixed population and coefficients. It's taken from Wikipedia.

2 APPENDIX

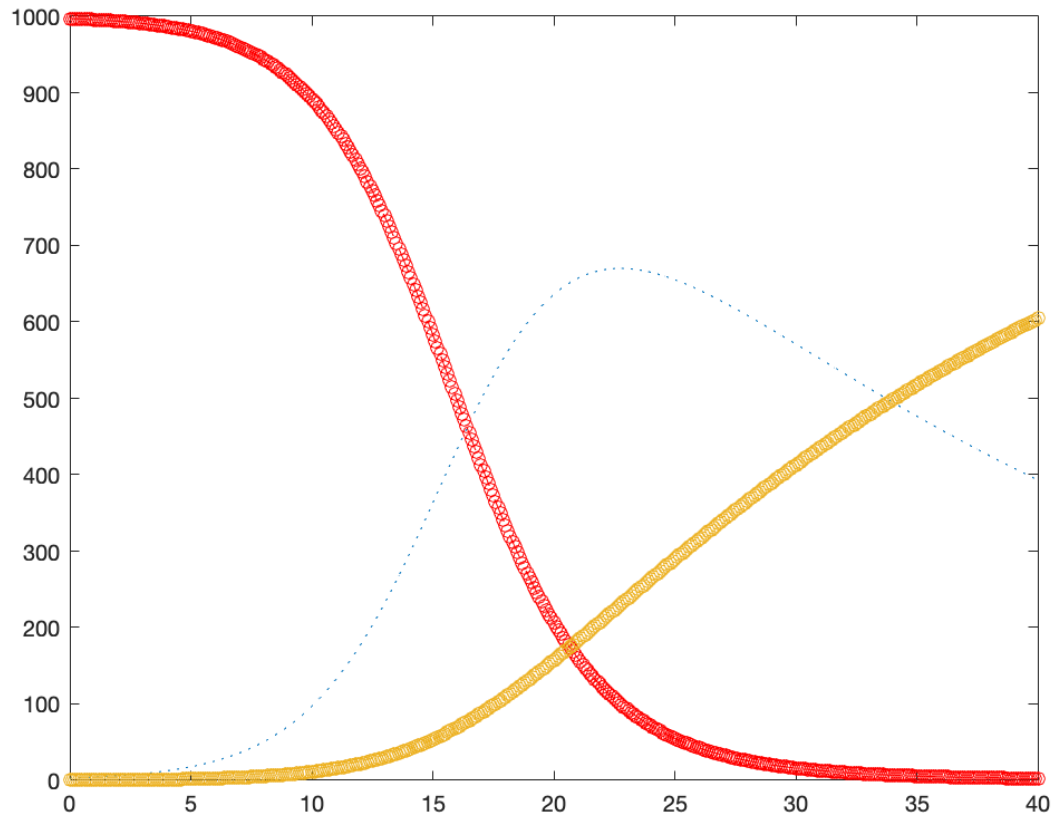
A screenshot of the MATLAB Command Window. The window has a blue title bar that says "Command Window". The text inside is as follows:

```
>> clear all  
Initial Value:997  
Enter a function: @(x,y)-0.0004*x*y  
Enter cases with [ ] around them [1 2]  
Initial Value:3  
Enter a function: @(x,y)0.0004*x*y-0.04*y  
Enter cases with [ ] around them [1 2]  
Initial Value:0  
Enter a function: @(y)0.04*y  
Enter cases with [ ] around them [2]  
fx >> |
```

Figure 3: Input format for the ODE Solver in MATLAB

For the MATLAB part of the APPENDIX, please visit the following pages!

```
tic
IVP_ODE_solver(3,0,40);
```



```
toc
```

Elapsed time is 482.717084 seconds.

```
function [vector]=rhs(c,C,w)
dimension=size(w,1);
N=size(w,2);
ode=zeros(dimension,N+1);
for i=1:dimension
    f=str2func(C{i});
    for j=1:N
        c1=cell2mat(c{i});
        b=[];
        for k=1:length(c1)
            b(k)=w(c1(k),j);
        end
```

```

        cell11=num2cell(b);
        ode(i,j)=f(cell11{:});
    end
end
vector=ode;
end
function IVP_ODE_solver(dimension,t0,tend)
    C = {};c = {};
    for i=1:dimension
        prompt = "Initial Value:";
        x = input(prompt);
        initial_values(i)=x;
        func = input('Enter a function: ','s');
        C{i}=func;
        %Which independent variables are used in equation of ODE
        Cases = input('Enter cases with [ ] around them ');
        c{i}=num2cell(Cases);
    end
    t=0;
    %h=[0.1,0.01,0.005,0.0025,0.0001,0.00005,0.000001];
    h=[0.1];
    for j=1:length(h)
        Nh=(tend-t0)/h(j);
        w=zeros(dimension,Nh+1);
        for i=1:dimension
            w(i,1)=initial_values(i);
        end
        %update the ode's
        ode=rhs(c,C,w);
        %assign the second initial values
        for i=1:dimension
            w(i,2)=w(i,1)+h(j)*(ode(i,1));
        end
        %update the ode's
        ode=rhs(c,C,w);
        t=0;
        for i=2:Nh
            for k=1:dimension
                w(k,i+1)=w(k,i-1)+2*h(j)*(ode(k,i));
                %update the ode's
                ode=rhs(c,C,w);
            end
            t=t+h(j);
        end
        for i=1:dimension
            last_values(i,j)=w(i,Nh+1);
        end
    end
end

```

```

        end
    end
    h=transpose(h);
    y_1=transpose(last_values(2,:));
    x_1=transpose(last_values(1,:));
    %outcomes=table(h,y_1,x_1)

    figure
    t=t0:h(1):tend;
    y = w(2,:);
    plot(t,y,':')

    hold on
    y2 = w(1,:);
    plot(t,y2,'--ro')
    y3=w(3,:);
    scatter(t,y3)
    hold off
end

```

Gauss-Newton Algorithm for an Non-linear Least Square Problem

Rüstü Erciyes Karakaya

July 2022, 28

1 Problem Definition¹

Aim: $\min RSS = \sum \frac{\|f(x_i) - \eta(x_i)\|^2}{\omega_i}$ with respect to parameters,
and residual vector is $r(\beta) = f(x_i) - \eta(x_i)$, when parameters $= \beta$.
i.e our aim is to find minimizer, β^* .

$f(x_i)$:= Given data points. Once again, our aim is try to find a curve by updating the parameters and obtaining a curve almost fits these data points.

$\eta(x_i)$:= This is numerically generated values by Integrator with given set of ODE, required parameters, β , and initial values as inputs.²

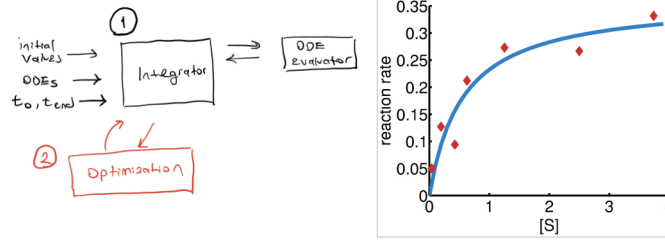


Figure 1: Output of the Integrator part is $\eta(x_i)$ above, namely blue curve in the sketch. The red points in sketching are referring to $f(x_i)$, namely given data points. In optimization part of the our program is aim to fit these point by using integrator; and to do that, integrator is feed by output of optimization part. Sketching is taken from Wikipedia.

¹Actually, it's referred in the 'Further Investigation' part of the Week-4's document.

²You can check the document of Week-4. In previous week, I implemented the function which take set of ODE with parameters as input and give numerical solutions in a high accuracy and low time consumption.

2 Method for The Solution

Gauss-Newton Method: Basically, it's an iterative algorithm to find roots of non-linear sum of squared residual, namely minimizing the error in our case. Let say given m data values-point, in this case residuals are as follows:

$$r = (r_1, r_2, \dots, r_m) \text{ and } RSS = \sum r_i(\beta)^2, \text{ and } \beta = (\beta_1, \beta_2, \dots, \beta_n), \text{ when } m \geq n.$$

Equation for Iteration: $\beta^{(s+1)} = \beta^{(s)} - (J_r^T \cdot J_r)^{-1} \cdot J_r^T \cdot r(\beta^{(s)})$
and Jacobian Matrix $:= (J_r)_{ij} = \frac{\partial r_i(\beta^{(s)})}{\partial \beta_j}$

The explanation above is the process of finding the minimizer (or maximizer) of an objective function in the settings of non-linear least square problems with Gauss-Newton Iteration. Actually, the iteration can be derived from second degree of Taylor Expansion, which is:

$$T^{(2)} = f(x) + (y - x)^T \cdot \nabla f(x) + \frac{1}{2} \cdot (y - x)^T \cdot \nabla^2 f(x) \cdot (y - x)$$

Actually, we prefer Newton's method because of its fewer number of initial values to execute algorithm in comparison to other options, and we know that it can converge with high accuracy in a few seconds. As an example; if we use Inverse Quadratic Interpolation, we need three initial point to start but in the case that one dimensional Newton's Approximation, we just need one initial value and its derivative.

3 Implementation of Algorithm

Generating the Residuals and RSS:

```
%calculate residue
for j=1:7
    r(j)=y(1,index(j))-numerical_values(1,index(j));
end
rss(i)=sum(r.^2);
r=transpose(r);
```

Figure 2: In algorithm, r and β vectors are assumed as column vectors.

In this implementation, I used SIR-Model example which has;
 $t_0 = 0$, $t_{end} = 100$, parameters = $[\beta_1, \beta_2, \text{population size}]$
and there are seven data point we try to find appropriate parameters of β by iterations to almost fit these data points.

Obtaining the Jacobian Matrix:

```
%calculate jacobian
h=0.001;
p_1_1=[beta(1)+h beta(2) 1000];
p_1_2=[beta(1)-h beta(2) 1000];
x1_1=IVP_ODE_solver(0,100,option,p_1_1,initial_values);
x1_2=IVP_ODE_solver(0,100,option,p_1_2,initial_values);
for k=1:7
    diff1(k)=(-x1_1(1,index(k))+x1_2(1,index(k)))./(2*h);%symmetric difference
end

h=0.001;
p_2_1=[beta(1) beta(2)+h 1000];
p_2_2=[beta(1) beta(2)-h 1000];
x2_1=IVP_ODE_solver(0,100,option,p_2_1,initial_values);
x2_2=IVP_ODE_solver(0,100,option,p_2_2,initial_values);
for k=1:7
    diff2(k)=(-x2_1(1,index(k))+x2_2(1,index(k)))./(2*h);%symmetric difference
end

change_of_r_1=transpose(diff1);
change_of_r_2=transpose(diff2);
J=[change_of_r_1 change_of_r_2];% speed of change in a sense of derivative
```

Figure 3: In definition above, Jacobian matrix is based on partial derivatives; however, I have obtained the functions numerically instead of analytical outcome by using Integrator. That's why I used "Symmetric Difference Quotient" which is one of numerical differentiation method.

Implementing the Iteration:

```
%put iteration here
beta=transpose(beta);|
beta=beta-pinv(J)*r;
beta=transpose(beta);
i=i+1;
```

Figure 4: It's totally same with definition above, but 'pinv' is just an abbreviation of pseudo-inverse.

4 Example-I: SIR Model³

³We started with parameters = $[\beta_1=0.3, \beta_2=0.07]$ to approximate $[\beta_1=0.4, \beta_2=0.04]$ with given 7 data points. Population size is fixed as 1000.

```

%firstly put the general inputs
%func = input('Enter a model option: ','s');
option="SIR Model";
%initial_values = input('Enter the initials with [ ] around them ');
initial_values=[997 3 0];

%generate data for just an trial
tspan= 0:0.01:100;
[t,y] = ode45(@vdp_SIR,tspan,[997;3;0]);
y=transpose(y);
index = randi([1 length(t)],1,7);

beta=[0.3 0.07]

```

```

beta = 1×2
    0.3000    0.0700

```

```

i=1;
while i<5
    %calculate numerical outcome with beta
    p=[beta 1000];
    numerical_values=IVP_ODE_solver(0,100,option,p,initial_values);

    %calculate residue
    for j=1:7
        r(j)=y(1,index(j))-numerical_values(1,index(j));
    end
    rss(i)=sum(r.^2);
    r=transpose(r);
    while (size(r,2)==7)
        r=transpose(r);
    end

    %calculate jacobian
    h=0.001;
    p_1_1=[beta(1)+h beta(2) 1000];
    p_1_2=[beta(1)-h beta(2) 1000];
    x1_1=IVP_ODE_solver(0,100,option,p_1_1,initial_values);
    x1_2=IVP_ODE_solver(0,100,option,p_1_2,initial_values);
    for k=1:7
        diff1(k)=(-x1_1(1,index(k))+x1_2(1,index(k)))/(2*h);%symmetric
difference
    end

    h=0.001;

```

```

p_2_1=[beta(1) beta(2)+h 1000];
p_2_2=[beta(1) beta(2)-h 1000];
x2_1=IVP_ODE_solver(0,100,option,p_2_1,initial_values);
x2_2=IVP_ODE_solver(0,100,option,p_2_2,initial_values);
for k=1:7
    diff2(k)=(-x2_1(1,index(k))+x2_2(1,index(k)))./(2*h);%symmetric
difference
end

change_of_r_1=transpose(diff1);
change_of_r_2=transpose(diff2);
J=[change_of_r_1 change_of_r_2];% speed of change in a sense of derivative

%put iteration here
beta=transpose(beta);
beta=beta-pinv(J)*r;
beta=transpose(beta);
i=i+1;
end
beta

```

```

beta = 1×2
    0.4106    0.0485

```

rss

```

rss = 1×4
105 ×
    2.6128    0.2595    0.0130    0.0006

```

```

function dydt = vdp_SIR(t,y)
dydt = [(-0.4/1000)*y(1)*y(2); 0.4/1000*y(1)*y(2)-0.04*y(2);0.04*y(2)];
end

```

5 Example-II: Sinusoidal Model

We started with parameters $[\beta_1=1.5, \beta_2=2.3]$ to approximate $[\beta_1=2, \beta_2=2]$ with given 7 data points. (See the following pages!)

6 APPENDIX

What was the beta in SIR Model?:

Once again, $N:=\text{population}=S(t)+I(t)+R(t)=1000$, infectious rate: $=\beta=0.4$, recovery rate: $=\gamma=0.04$; with initial values $S(0)=997, I(0)=3, R(0)=0$, then the ODEs are:

$$\dot{S} = \frac{-\beta SI}{N}, \quad \dot{I} = \frac{\beta SI}{N} - \gamma I, \quad \dot{R} = \gamma I$$

Beta parameters $[\beta, \gamma]$, which must be $[0.4, 0.04]$ in this case.

What was the beta in Sinusoidal Model?:

Once again, $\frac{dx}{dt} = 2y$ and $\frac{dy}{dt} = -2x$, $t \in [0.0, 10.0]$ and $x(0)=0, y(0)=1$.

Analytical Conclusions: $x(t) = \sin(2t)$ and $y(t) = \cos(2t)$

Beta parameters: Inner coefficients of Sinusoidal functions, which must be $[2, 2]$ in this case.

These two model above were investigated in previous weeks in detail, for further information please revisit them!

How to avoid ill-conditioned case of Jacobian Matrices in the Iteration?:

If we define $\Delta = \beta^{(s+1)} - \beta^{(s)}$ and then if we update the our iteration;
 $J_r^T \cdot J_r \cdot \Delta = J_r^T \cdot r(\beta^{(s)})$ (*)

In this way, we can solve the equation (*) numerically without considering whether is $J_r^T \cdot J_r$ is invertible or not.

```

%firstly put the general inputs
%func = input('Enter a model option: ','s');
option="Sinusoidal Model";
%initial_values = input('Enter the initials with [ ] around them ');
initial_values=[0 1];

%generate data for just an trial
tspan= 0:0.01:10;
[t,y] = ode45(@vdp_Sinusoidal,tspan,[0;1]);
y=transpose(y);
index = randi([1 length(t)],1,7);

beta=[1.5 2.3]

```

```

beta = 1×2
    1.5000    2.3000

```

```

i=1;
while i<15

    %calculate numerical outcome with beta
    p=[beta];
    numerical_values=IVP_ODE_solver(0,10,option,p,initial_values);

    %calculate residue
    for j=1:7
        r(j)=y(1,index(j))-numerical_values(1,index(j));
    end
    rss(i)=sum(r.^2);
    r=transpose(r);
    while (size(r,2)==7)
        r=transpose(r);
    end

    %calculate jacobian
    h=0.001;
    p_1_1=[beta(1)+h beta(2)];
    p_1_2=[beta(1)-h beta(2)];
    x1_1=IVP_ODE_solver(0,10,option,p_1_1,initial_values);
    x1_2=IVP_ODE_solver(0,10,option,p_1_2,initial_values);
    for k=1:7
        diff1(k)=(-x1_1(1,index(k))+x1_2(1,index(k)))/(2*h);%symmetric
difference
    end
end

```

```

h=0.001;
p_2_1=[beta(1) beta(2)+h];
p_2_2=[beta(1) beta(2)-h];
x2_1=IVP_ODE_solver(0,10,option,p_2_1,initial_values);
x2_2=IVP_ODE_solver(0,10,option,p_2_2,initial_values);
for k=1:7
    diff2(k)=(-x2_1(1,index(k))+x2_2(1,index(k)))./(2*h);%symmetric
difference
end

change_of_r_1=transpose(diff1);
change_of_r_2=transpose(diff2);
J=[change_of_r_1 change_of_r_2];% speed of change in a sense of derivative

%put iteration here
beta=transpose(beta);
beta=beta-pinv(J)*r;
beta=transpose(beta);
i=i+1;
end
beta

```

```

beta = 1×2
    1.9996    2.0006

```

rss

```

rss = 1×14
    2.6945    0.2831    1.6501    0.8866    0.0375    0.0084    0.0000 ...

```

```

function dydt = vdp_Sinusoidal(t,y)
dydt = [2*y(2); -2*y(1)];
end

```

Literature suggestions:⁴

- Parallel and Distributed Computation: Numerical Methods by Dimitri P. Bertsekas.
- Practical Methods of Optimization by R. Fletcher.
- Variational Calculus with Elementary Convexity by John L. Troutman.
- Mathematical Modeling and Simulation: Introduction for Scientists and Engineers by Kai Velten.
- Numerical Optimization Jorge Nocedal and Stephen Wright.

⁴Just some brilliant sources for Optimization Theory and Modeling!