گزارش پروژه اول متلب

درس محاسبات عددی

امیرمرتضی رضائی

810101429

بهار 1402

سوال اول:

ابتدا در سطر اول با نمایش یک پیغام متناسب، طول دنباله یعنی n را از کاربر دریافت می کنیم. سپس در سطر بعدی یک ماتریس به اسم arr شامل یک سطر و n ستون تعریف می کنیم که مقادیر همه اعضای آن در ابتدا برابر با صفر می باشد.

```
1   n=input(sprintf('Enter n: \n'));
2   arr=zeros(1,n);
```

حال باید دنباله اعداد را دریافت کرده و آنها را در ماتریس arr ذخیره کنیم. برای این کار، از یک حلقه for استفاده می کنیم که به ترتیب از خانه ی اول تا خانه ی n ام ماتریس arr حرکت کرده و مقادیر هر یک را با استفاده از دستور input از کاربر دریافت میکند.

```
3   for i=(1:n)
4       arr(1,i)=input(sprintf('Enter the %d number\n',i));
5   end
```

با توجه به اینکه مقدار منفی برای ورودی ها مجاز نمی باشد، نیاز به تابعی داریم که بررسی کند تا اگر ماتریس ما شامل مقادیر منفی بود، پیغام خطایی را به کاربر نمایش دهد. بنابر این ابتدا تابعی به نام negative_check تعریف می کنیم که ورودی های آن به ترتیب برابر با لیست اعداد دریافت شده (numbers_arr) و همچنین تعداد آنها (n) می باشد. در داخل تابع، با استفاده از یک حلقه for به ترتیب از خانه ی اول تا خانه ی n ام ماتریس اعداد حرکت کرده و همه ی درایه ها را بررسی می کنیم؛ در صورتی که به درایه ای منفی برخوردیم با استفاده از تابع error یک پیغام خطا نمایش داده و از برنامه خارج می شود.

```
19  function negative_check(numbers_arr , n)
20      for i=(1:n)
21          if numbers_arr(1,i)<0
22            error('Negative numbers are unallowed!');
23          end
24      end
25  end
```

در خود برنامه نیز این تابع را فراخوانی می کنیم:

```
6   negative_check(arr,n);
```

اکنون باید اعداد غیر یکتا را از دنباله جدا کنیم. برای این مورد نیز تابعی با نام single_num_finder تعریف میکنیم که ورودی های آن به ترتیب برابر با لیست اعداد دریافت شده (numbers_arr) و تعداد آنها (n) و خروجی آن نیز ماتریسی به نام final_arr است که شامل اعداد یکتا می باشد. از آنجا که هر عدد نهایتا دو بار در دنباله تکرار شده است، کافیست تا از اولین درایه شروع کرده و برای هر یک از درایه ها، تساوی آن با درایه های بعدی را بررسی نماییم، در صورت برقراری تساوی، مقدار هر دو درایه که با هم مساوی اند را برابر با یک مقدار غیر صحیح (برای مثال: 2.5 در این سوال) قرار می دهیم تا بتوان در مرحله بعد آنها را از باقی دنباله جدا کرد.

```
27      function final_arr=single_num_finder(numbers_arr , n)
28          for j=(1:n-1)
29              for k=(j+1:n)
30                  if numbers_arr(1,j)~=numbers_arr(1,k)
31                      continue;
32                  else
33                      numbers_arr(1,j)=2.5;
34                      numbers_arr(1,k)=2.5;
35                  end
36              end
37          end
```

حال می خواهیم اعداد یکتا را در ماتریسی به نام final_arr ذخیره کنیم. پس یک ماتریس خالی با این نام تعریف می کنیم. سپس با بهره گیری از یک حلقه for به ترتیب از خانه ی اول تا خانه ی n ام ماتریس numbers_arr حرکت کرده و درایه هایی را که برابر با 2.5 نیستند را توسط تابع horzcat به انتهای ماتریس final_arr اضافه می کنیم.

```
38          final_arr=[];
39          for i=(1:n)
40              if numbers_arr(1,i)~=2.5
41                  final_arr=horzcat(final_arr,numbers_arr(1,i));
42              end
43          end
44      end
```

حال در خود برنامه نیز این تابع را فراخوانی کرده و ماتریس final_arr را ایجاد می کنیم:

```
7       final_arr=single_num_finder(arr , n);
```

از آنجا که اگر هیچ عدد غیر یکتایی در دنباله ورودی وجود نداشته باشد باید عدد 0 را نمایش دهیم، در یک if با استفاده از تابع isempty ، تهی بودن ماتریس اعضای یکتا یعنی final_arr را بررسی می کنیم. در صورت تهی بودن این ماتریس، با استفاده از تابع error ، 0 را نمایش داده و از برنامه خارج می شویم.

```
8     if isempty(final_arr)
9         error('0');
10    end
```

برای آنکه هنگام XOR کردن درایه ها به مشکلی برنخوریم، اعضای آرایه ی final_arr را توسط تابع sort به صورت نزولی مرتب می کنیم.

```
11    final_arr=sort(final_arr,'descend');
```

حال باید مقدار XOR درایه ها را محاسبه نماییم. (برای این کار از تابعی به نام xorcalculator استفاده شده که توضیحات آن در انتها ارائه خواهد شد.) ابتدا باید تعداد درایه های یکتا را بدست آوریم. از آنجا که ماتریس final_arr ماتریسی سطری است، ما به تعداد ستون های آن نیاز داریم. بنابریان ابتدا تعداد ستون های آن را در m ذخیره می کنیم.

```
12    [~,m] = size(final_arr);
```

حال با استفاده از دو حلقه for تو در تو که اولی به ترتیب از خانه ی اول تا خانه ی m ام ماتریس final_arr حرکت کرده و دومی از یک خانه بعد از متغیر قبلی تا خانه پایانی ماتریس final_arr حرکت می کند؛ در هر مرحله حاصل XOR هر دو خانه از ماتریس را با استفاده از تابع xorcalculator یافته و مقدار xor_res را آپدیت می کنیم. در نهایت نیز مقدار نهایی آن را چاپ می کنیم.

```
13    for i=(1:m)
14        for j=(i+1:m)
15            xor_res=xorcalculator(final_arr(1,i),final_arr(1,j));
16            fprintf('XOR(%d , %d)= %d\n', final_arr(1,i),final_arr(1,j), xor_res);
17        end
18    end
```

حال به توضیح تابع xorcalculator می پردازیم:

خروجی این تابع xor_result و ورودی های آن، دو عدد با نام های num1 و num2 می باشند. ابتدا مقدار خروجی را برابر با صفر قرار می دهیم. همچین یک شمارنده با نام i در نظر گرفته و مقدار ابتدایی آن را نیز برابر با صفر قرار می دهیم. از آنجا که می دانیم مقدار باینری یک عدد دسیمال از تقسیم های متوالی آن بر 2 بدست آمده و ارقام آن از انتها برابرند با حاصل باقی مانده آن در هر مرحله بر 2، در هر مرحله ابتدا باقی مانده دو ورودی را بر 2 محاسبه کرده و سپس در صورتی باقی مانده آنها با هم برابر نباشد مقدار xor_result را به صورتی آپدیت می کنیم که با مقدار دسیمال نتیجه در انتهای هر مرحله برابر باشد. بدین شکل که حاصل جمع آن را با 2 به توان شماره مرحله را در خودش میریزیم. (چرا که اگر باقی مانده ها با هم برابر نباشند، در آن مرحله، مقدار 1 به ابتدای مقدار باینری نتیجه اضافه خواهد شد. و از آنجا که ارزش هر خانه برابر است با 2 به توان شماره آن مرحله، این کار را انجام می دهیم تا نتیجه در نهایت به صورت دسیمال بدست آید.) در انتهای هر مرحله مقدار ورودی ها را با مقدار کف (floor) حاصل تقسیم آنها بر 2 آپدیت کرده و همچنین یکی به مقدار i اضافه می کنیم. بدین شکل این تابع مقدار XOR دو عدد را محاسبه می کند. در واقع به جای اینکه ابتدا دو عدد ورودی را به اعداد باینری تبدیل کرده و پس از محاسبه XOR آنها، نتیجه را دوباره از باینری به دسیمال تبدیل کنیم، همه این اعمال را در هر مرحله رقم به رقم انجام داده و نتیجه را آپدیت می کنیم.

```
46      function xor_result = xorcalculator(num1,num2)
47          xor_result=0;
48          i=0;
49          while num1~=0
50              remainder1=mod(num1,2);
51              remainder2=mod(num2,2);
52              if remainder1~=remainder2
53                  xor_result=xor_result+(2^(i));
54              end
55              num1=floor(num1/2);
56              num2=floor(num2/2);
57              i=i+1;
58          end
59      end
```

سوال دوم:

الف)

توضیح مبحث تئوری:

برای حل معادلات خطی به روش حذفی گاوس، می بایست ابتدا ماتریس ضرایب و پاسخ ها مجهولات را بدست آوریم(مانند: AX=B). حال باید ماتریس ضرایب و پاسخ ها را کنار هم قرار دهیم ([A:B]) و سپس با استفاده از عملیات سطری مقدماتی و با استفاده از عناصر روی قطر اصلی ، در هر ستون عناصر زیر قطر اصلی را به صفر تبدیل می کنیم تا ماتریس ضرایب ما به یک ماتریس بالا مثلثی تبدیل شود. اما باید توجه داشت که برای همگرایی، باید قبل از بالا مثلثی کردن ماتریس، آنرا محورگیری کنیم. بدین منظور باید داده ها را به نحوی جابجا کنیم که در هر ستون، عناصر روی قطر اصلی از عناصر زیرین خود بزرگتر باشند.

حل سوال:

در سطر ابتدایی داده های مسئله را در یک ماتریس به نام variables ذخیره می کنیم. (از آنجا که توابع نوشته شده برای این سوال، برای n داده دلخواه نوشته شده اند، برای افزودن متغیر های جدید کافیست تا مقادیر جدید زمان را در ستون اول و مقادیر سرعت متناظر آنها را در ستون دوم اضافه کنیم.)

```
1    variables=[5,106.8 ; 8,177.2 ; 12,279.2];
```

حال برای محور گیری و مرتب کردن داده ها از بزرگ به کوچک، از تابع sortrows استفاده کرده و آرگومان دوم آنرا برابر با 1- قرار می دهیم که ماتریس ورودی را بر اساس ستون اول آن و به صورت نزولی مرتب می کند. در نهایت نیز ماتریس مرتب شده را در ماتریسی به نام sortedvars ذخیره می کنیم.

```
2    [sortedvars]=sortrows(variables,-1);
```

حال باید ماتریس ضرایب و ماتریس پاسخ ها را ایجاد کنیم. برای اینکار تابعی به نام AB_finder تعریف کرده که مقدار ورودی آن همان ماتریس مرتب شده مرحله قبل (sortedvars) و خروجی آن، ماتریس های A و B می باشد. در این تابع ابتدا تعداد متغیر ها را با استفاده از تابع size بدست آورده و آنرا در n ذخیره می کنیم. سپس ماتریس های A و B را به ترتیب ماتریس های صفری به ابعاد n در n و nدر 1 تعریف میکنیم. حال در دو حلقه با متغیر های i و j از 1 تا n حرکت میکنیم. i شمارنده ی سطر ها و j شمارنده ی ستون ها می باشد.

با توجه به اینکه توان هر ستون از ماتریس ضرایب (A) به ترتیب از n-1 شروع شده و تا 0 ادامه پیدا می کند، (یعنی توان هر ستون برابر است با اختلاف n و شماره ی ستون) ، ابتدا در هر سطر برای هر یک از ستون ها ، توان های متناسبی از مقادیر موجود در ماتریس ورودی را در درایه ی مورد نظر قرار می دهیم. حال برای ماتریس پاسخ ها (B) مقدار متناظر با زمان هر سطر را در درایه ی متناظر آن قرار می دهیم. بدین ترتیب ماتریس ضرایب و پاسخ ها بدست می آیند.

```
10      function [A,B]=AB_finder(sortedvars)
11          n=size(sortedvars,1);
12          A=zeros(n,n);
13          B=zeros(n,1);
14          for i=(1:n)
15              for j=(1:n)
16                  A(i,j)=(sortedvars(i,1))^(n-j);
17              end
18              B(i,1)=sortedvars(i,2);
19          end
20      end
```

در خود برنامه نیز این تابع را فراخوانی کرده و ماتریس های A و B را ایجاد می کنیم:

```
3       [A,B]=AB_finder(sortedvars);
```

حال باید ماتریس A را بالامثلثی گردانیم. بنابراین ابتدا یک تابع به نام upper_triangularer تعریف می کنیم که ماتریس های A و B تغییر یافته (A_new و B_new)خروجی آن و ماتریس های A و B دو ورودی آن باشند. ابتدا تعداد متغیر ها را توسط تابع size یافته و در متغیر n ذخیره می کنیم. سپس مقادیر A_new و B_new را برابر با مقادیر A و B قرار می دهیم. حال دو حلقه می زنیم که در اولی متغیر i از سطر اول تا سطر n-1 ام تغییر کرده و در حلقه ی درونی، متغیر j در هر مرحله از سطر i+1 ام تا سطر آخر (n ام) تغییر می کند. حال برای هر سطر عاملی به نام z را برابر با مقدار تقسیم درایه ای که قصد داریم آن را صفر کنیم بر درایه روی قطر اصلی در همان ستون قرار می دهیم. و ضرب z در تمام درایه های سطری که با درایه ی روی قطر اصلی آن می خواهیم درایه ی مورد نظر را صفر کنیم را از سطری که درایه مورد نظر در آن قرار دارد کم می کنیم. بدین ترتیب ماتریس A به ماتریسی بالا مثلثی تبدیل می گردد. همچنین در سطر پایانی نیز همین اعمال را بر روی ماتریس پاسخ های B انجام می دهیم.

```
21      function [A_new , B_new]=upper_triangularer(A,B)
22          n=size(A,1);
23          A_new=A;
24          B_new=B;
25          for i=(1:n-1)
26              for j=(i+1:n)
27                  z=A_new(j,i)/A_new(i,i);
28                  A_new(j,i:n) = A_new(j,i:n)-z*A_new(i,i:n);
29                  B_new(j) = B_new(j)-z*B_new(i);
30              end
31          end
32      end
```

در خود برنامه نیز این تابع را فراخوانی کرده و ماتریس های بالا مثلثی شده ی A_new و B_new را ایجاد می کنیم:

```
4       [A_new,B_new]=upper_triangularer(A,B);
```

اکنون باید پاسخ ماتریس مجهولات X را بیابیم. با توجه به اینکه ماتریس A بالا مثلثی شده است، می توان اظهار داشت که حاصل ضرب تنها درایه ی غیر صفر سطر n ام ماتریس A در درایه n ام ماتریس X برابر است با درایه ی سطر n ام ماتریس B. پس طبق همین رابطه مقدار ضریب c را بدست می آوریم. سپس برای بدست آوردن باقی ضرایب مجهول، در یک حلقه که متغیر i در آن از سطر یکی مانده به آخر یکی یکی به سطر اول باز

می گردد، عنصر i ام ماتریس X برابر است با حاصل تفریق عنصر i ام ماتریس B با حاصل ضرب درایه ها ی سطر متناظر ماتریس A در درایه های بعدی ماتریس X تقسیم بر عنصر روی قطر اصلی سطر i ام ماتریس A. بنابراین تمام اینها را در تابعی به نام gaussian_elimination که ماتریس های A و B ورودی های آن و ماتریس X خروجی آن می باشد، پیاده سازی کرده و در خود برنامه نیز این تابع را فراخونده و ماتریس مجهولات X را تولید میکنیم:

```
33    function X = gaussian_elimination(A,B)
34        n=size(A,1);
35        X=zeros(n,1);
36        X(n) = B(n)/A(n,n);
37        for i = n-1:-1:1
38            X(i) = (B(i)-A(i,i+1:n)*X(i+1:n))/A(i,i);
39        end
40    end
```

```
5    X = gaussian_elimination(A_new,B_new);
```

در دو سطر آخر نیز برای چاپ مقادیر ضرایب ، ابتدا دوباره تعداد مقادیر را بدست آورده و در متغیر n ذخیره می کنیم. سپس در یک حلقه که به تعداد n بار تکرار می شود، مقادیر درایه های ماتریس مجهولات X را به ترتیب از اول تا انتها چاپ می کنیم. ولی در ابتدا فرمت آنها را از عدد به رشته تبدیل می کنیم. باید توجه داشت که مقادیر چاپ شده به ترتیب بوده و به ترتیبِ ضرایب توان بیشتر به کمتر مرتب شده اند.

```
6    n=size(X,1);
7    for i=(1:n)
8        disp(['the coefficient ', num2str(i), ' is : ',num2str(X(i,1)) ]);
9    end
```

در نهایت، نتایج بدست آمده به صورت زیر خواهد بود:

```
the coefficient 1 is : 0.29048
the coefficient 2 is : 19.6905
the coefficient 3 is : 1.0857
```

ب)

توضیح مبحث تئوری:

می دانیم در روش ژاکوبی، اگر ماتریس $A = [a_{ij}]$ ماتریس ضرایب و ماتریس $B = [b_{ij}]$ ماتریس پاسخ ها و ماتریس X ، ماتریس مجهولات باشد و برای مثال برای 3 مجهول، پس از محورگیری و پس از آنکه ماتریس الحاقی $\{A|B\}$ را به ماتریس قطری غالب تبدیل کردیم، داشته باشیم:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \end{cases}$$

در مرحله ی k ام، مقادیر مجهولات از روابط زیر بدست می آیند:

$$\begin{cases} x_1^{(k+1)} = \dfrac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}}{a_{11}} \\ x_2^{(k+1)} = \dfrac{b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)}}{a_{22}} \\ x_3^{(k+1)} = \dfrac{b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)}}{a_{33}} \end{cases}$$

حل سوال:

همانند قسمت قبل، در سطر ابتدایی داده های مسئله را در یک ماتریس به نام variables ذخیره می کنیم. (از آنجا که توابع نوشته شده برای این سوال، برای n داده دلخواه نوشته شده اند، برای افزودن متغیر های جدید کافیست تا مقادیر جدید زمان را در ستون اول و مقادیر سرعت متناظر آنها را در ستون دوم اضافه کنیم.)

```
1    variables=[5,106.8 ; 8,177.2 ; 12,279.2];
```

حال برای محور گیری و مرتب کردن داده ها از بزرگ به کوچک، از تابع sortrows استفاده کرده و آرگومان دوم آنرا برابر با 1- قرار می دهیم که ماتریس ورودی را بر اساس ستون اول آن و به صورت نزولی مرتب می کند. در نهایت نیز ماتریس مرتب شده را در ماتریسی به نام sortedvars ذخیره می کنیم.

```
2    [sortedvars]=sortrows(variables,-1);
```

حال باید ماتریس ضرایب و ماتریس پاسخ ها را ایجاد کنیم. برای اینکار تابعی به نام AB_finder تعریف کرده که مقدار ورودی آن همان ماتریس مرتب شده مرحله قبل (sortedvars) و خروجی آن، ماتریس های A و B می باشد. در این تابع ابتدا تعداد متغیر ها را با استفاده از تابع size بدست آورده و آنرا در n ذخیره می کنیم. سپس ماتریس های A و B را به ترتیب ماتریس های صفری به ابعاد n در n و nدر 1 تعریف میکنیم. حال در دو حلقه با متغیر های i و j از 1 تا n حرکت میکنیم. i شمارنده ی سطر ها و j شمارنده ی ستون ها می باشد.

با توجه به اینکه توان هر ستون از ماتریس ضرایب (A) به ترتیب از n-1 شروع شده و تا 0 ادامه پیدا می کند(یعنی توان هر ستون برابر است با اختلاف n و شماره ی ستون)، ابتدا برای هر یک از ستون های هر سطر، توان های متناسب از مقادیر موجود در ماتریس ورودی را در درایه ی مورد نظر قرار می دهیم. حال برای ماتریس پاسخ ها (B) مقدار متناظر با زمان هر سطر را در درایه ی متناظر قرار می دهیم. بدین ترتیب ماتریس ضرایب و پاسخ ها بدست می آیند.

```
10      function [A,B]=AB_finder(sortedvars)
11          n=size(sortedvars,1);
12          A=zeros(n,n);
13          B=zeros(n,1);
14          for i=(1:n)
15              for j=(1:n)
16                  A(i,j)=(sortedvars(i,1))^(n-j);
17              end
18              B(i,1)=sortedvars(i,2);
19          end
20      end
```

حال ماتریس مجهولات X را تعریف کرده و مقادیر اولیه دلخواه را در آن قرار می دهیم.(با توجه به نتایج قسمت قبل، مقادیر اولیه را به ترتیب برابر با 0 و 20 و 1 قرار می دهیم.) حال توسط تابع size تعداد ورودی ها را در متغیری به نام n ذخیره می کنیم.

```
4       X=[0;20;1];
5       n=size(X,1);
```

اکنون برای یافتن مقادیر x_n در هر مرحله، تابعی با نام jacobi تعریف می کنیم که تعداد مجهولات(n) و ماتریس های ضرایب (A) ، پاسخ ها (B) و مجهولات (X) ورودی های آن و ماتریسی به نام X_out که مقادیر نهایی بدست آمده از روش ژاکوبی در آن ذخیره می شود، خروجی آن می باشد. ابتدا مقادیر اولیه موجود در ماتریس X را در ماتریسی به نام X_pre_values ذخیره می کنیم.همچنین برای ذخیره ی نتایج در هر مرحله، ماتریس دیگری با نام X_new تعریف کرده و به صورت پیش فرض، مقادیر آن را برابر با صفر در نظر میگیریم.

ابتدا یک حلقه ی تکرار برای تکرار روش ژاکوبی به تعداد 20 بار تعریف میکنیم. سپس به یک حلقه ی تکرار دیگر برای حرکت روی هر یک از مجهولات به تعداد مجهولات نیاز داریم. در ابتدا مقدار هر یک از مجهولات را برابر با مقدار متناظر در ماتریس پاسخ ها قرار می دهیم. سپس با توجه به فرمول روش ژاکوبی که بالاتر عنوان شد، توسط یک حلقه، مقدار حاصل ضرب ضرایب دیگر در مقدار مجهول اولیه ی متناظر آنها را باید از مقدار اولیه ی مجهول کنونی کم کنیم. اما باید توجه شد که نباید این حاصل ضرب را برای مجهولی که در حال محاسبه ی آن هستیم انجام دهیم. برای این مورد هم از یک if بهره میگیریم. در پایان محاسبه هر یک از مجهولات، مقدار بدست آمده را بر ضریب متناظر آن مجهول تقسیم می کنیم. در پایان هر تکرار نیز مقدار ماتریس X_pre_values را با مقادیر جدید (X_new) آپدیت کرده و دوباره مقادیر X_new را برابر با صفر قرار می دهیم. بعد از انجام 20 تکرار نیز مقادیر نهایی را که در ماتریس X_pre_values ذخیره شده اند، در ماتریس X_out ریخته و خروجی می دهیم.

```
21      function X_out=jacobi(n, A ,B, X)
22          X_pre_values=X;
23          X_new=zeros(n,1);
24          for m=(1:20)
25              for i=(1:n)
26                  X_new(i,1)=B(i,1);
27                  for j=(1:n)
28                      if (j~=i)
29                          X_new(i,1)=X_new(i,1)-A(i,j)*X_pre_values(j,1);
30                      end
31                  end
32                  X_new(i,1)=X_new(i,1)/A(i,i);
33              end
34              X_pre_values=X_new;
35              X_new=zeros(n,1);
36          end
37          X_out=X_pre_values;
38      end
```

در متن برنامه نیز با فراخوانی تابع jacobi ، مقادیر بدست آمده از 20 بار تکرار روش ژاکوبی را در ماتریس X_final ذخیره می کنیم.

```
6    X_final=jacobi(n, A,B,X);
```

در دو سطر آخر نیز برای چاپ مقادیر ضرایب ، در یک حلقه که به تعداد n بار (تعداد ورودی ها) تکرار می شود، مقادیر درایه های ماتریس مجهولات X_final را به ترتیب از اول تا انتها چاپ می کنیم. ولی در ابتدا فرمت آنها را از عدد به رشته تبدیل می کنیم. باید توجه داشت که مقادیر چاپ شده به ترتیب بوده و به ترتیبِ ضرایب توان بیشتر به کمتر مرتب شده اند.

```
7    for i=(1:n)
8        disp(['the coefficient ', num2str(i), ' is : ',num2str(X_final(i,1)) ]);
9    end
```

در نهایت مشاهده می شود از آنجا که ماتریس ضرایب قطری غالب نشده بود، پاسخ ها در این روش همگرا نمی شوند.

```
the coefficient 1 is : -37.5984
the coefficient 2 is : -410.2231
the coefficient 3 is : -2272.2491
```

ج)

توضیح مبحث تئوری:

می دانیم در درونیابی به روش لاگرانژ اگر x_0 تا x_n مجموعه نقاط داده شده از یک تابع و f_0 تا f_n نقاط متناظر آن نقاط روی تابع اصلی باشند، می توان چند جمله ای $P_n(x)$ را که چند جمله ای متناظر با تابع $f(x)$ است، به صورت زیر در نظر گرفت:

$$f(x) \cong P_n(x) = \sum_{i=0}^{n} f(x_i)L_i(x)$$

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

حل سوال:

در 2 سطر ابتدایی داده های مسئله را در دو ماتریس به نام هایT برای زمان ها و V برای سرعت های متناظر آنها ذخیره می کنیم. (از آنجا که توابع نوشته شده برای این سوال، برای n داده دلخواه نوشته شده اند، برای افزودن متغیر های جدید کافیست تا مقادیر جدید زمان را به ترتیب در ماتریس T و مقادیر سرعت متناظر آنها را به ترتیب در ماتریس V اضافه کنیم.)

```
1    T=[5;8;12];
2    V=[106.8;177.2;279.2];
```

اکنون برای بدست آوردن ضرایب L_i ، تابعی به نام lagrange تعریف میکنیم که ورودی آن ماتریس زمان ها (T) ورودی آن و ماتریسی به نام L_out که شامل ضرایب L_i است، خروجی آن می باشد. ابتدا تعداد مقادیر ورودی را توسط تابع size در متغیری به نام n ذخیره می کنیم. حال متغیری نمادین به نام t را تعریف کرده و سپس ماتریس L_out را با n سطر و 1 ستون و با مقادیر پیش فرض 1 با قابلیت ذخیره معادلات نمادین جهت ذخیره ی L_i ها تعریف می کنیم.

اکنون L_i ها را می یابیم. برای این کار از دو حلقه ی تو در تو استفاده کرده و جملات سازنده ی L_i را در هر مرحله در هم ضرب می کنیم.

```
function L_out= lagrange(T)
    n=size(T,1);
    syms t;
    L_out=sym(ones(n,1));
    for i=(1:n)
        for j=(1:n)
            if i~=j
                L_out(i) = L_out(i) * (t - T(j)) / (T(i) - T(j));
            end
        end
    end
end
```

در متن برنامه نیز مقادیر L_i ها را توسط تابع lagrange در ماتریس L ذخیره می کنیم.

```
L=lagrange(T);
```

حالا L_i ها را در اختیار داریم. بنابراین می توانیم چند جمله ای $P_n(x)$ را تولید کنیم. بنابر این تابعی به نام approximate_polynominal را با ورودی ماتریس های L و V و خروجی چندجمله ای نهایی (p_out) تعریف میکنیم. و در آن تعداد متغیر ها را بدست می آوریم. حال ابتدا حاصل ضرب L_1 در V_1 را بدست آورده و وارد p_out می کنیم. سپس توسط یک حلقه برای i های بزرگتر از 2، L_i ها را در V_i های متناظر با آنها ضرب کرده و در P ذخیره می کنیم.

```
function p_out= approximate_polynomianl(L,V)
    n=size(V,1);
    p_out = V(1)*L(1);
    for j=(2:n)
        p_out = p_out + V(j)*L(j);
    end
end
```

در متن برنامه نیز حاصل نهایی چند جمله ای درونیابی شده به روش لاگرانژ را توسط تابع قبل، در p ذخیره می کنیم.

```
p=approximate_polynomianl(L,V);
```

از آنجا که a ضریب t^2 ، b ضریب t و c عدد ثابت در چند جمله ای P می باشد، باید این ضرایب را از این چندجمله ای استخراج کرده و نمایش دهیم. بنابراین با استفاده از تابع coeffs ضرایبب t را در چند جمله ای p بدست آورده و با استفاده از تابع disp چاپ می کنیم.

```
6    disp(double(coeffs(p,t)));
```

اما باید توجه شود که اعداد چاپ شده به ترتیب، ضرایب توان های صفرم ، اول و دوم t می باشند. یعنی این اعداد از چپ به راست، ضرایب c و b و a می باشند.

```
1.0857   19.6905    0.2905
```

د)

توضیح مبحث تئوری:

اگر x_n ها نقاط داده شده و f(x) تابع متناظر آنها باشد، میتوان f(x) را به روش تفاضلات تقسیم شده نیوتن، به روش زیر تقریب زد:

$$f(x) \approx p(x) = f_0 + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] + \cdots + (x - x_0)(x - x_1) \dots (x - x_{n-1})f[x_0, x_1, \dots, x_n]$$

که در آن داریم:

$$f[x_i, x_{i+1}] = \frac{f_{i+1} - f_i}{x_{i+1} - x_i}$$

و

$$f[x_0, \dots, x_n] = \frac{f[x_1, \dots, x_n] - f[x_0, \dots, x_{n-1}]}{x_n - x_0}$$

همچنین می توان از فرم ماتریسی این روش نیز استفاده کرد. برای مثال برای سه داده داریم:

x	f(x)	ت ت م 1	ت ت م 2
x1	f(x1)	-	-
x2	f(x2)	f[x2,x1]	-
x3	f(x3)	f[x3,x2]	f[x1,x2,x3]

در حل این سوال نیز از این روش استفاده شده است.

حل سوال:

در 2 سطر ابتدایی داده های مسئله را در دو ماتریس به نام هایT برای زمان ها و V برای سرعت های متناظر آنها ذخیره می کنیم. (از آنجا که توابع نوشته شده برای این سوال، برای n داده دلخواه نوشته شده اند، برای افزودن متغیر های جدید کافیست تا مقادیر جدید زمان را به ترتیب در ماتریس T و مقادیر سرعت متناظر آنها را به ترتیب در ماتریس V اضافه کنیم.)

```
1    T=[5;8;12];
2    V=[106.8;177.2;279.2];
```

حال تعداد مقادیر ورودی را توسط تابع size در متغیری به نام n ذخیره می کنیم. سپس برای ایجاد کردن ماتریس، ماتریس های T و V و ماتریس صفر با n سطر و n-1 ستون را با تابع horzcat به ترتیب به صورت افقی کنار هم قرار می دهیم. و نام آن را matrix میگذاریم.

```
4    matrix=horzcat(T,V,zeros(n,n-1));
```

حال برای آنکه مقادیر f[x1,...xn] را بیابیم، تابعی به نام matrix_maker تعریف می کنیم که ماتریس ساخته شده در مرحله ی قبل (matrix) و تعداد داده ها (n) را به عنوان ورودی گرفته و ماتریس نهایی the_matrix خروجی این تابع خواهد بود. با توجه به اینکه 2 ستون اول این ماتریس را داده های مسئله اشغال کرده اند، در یک حلقه از ستون سوم تا آخر (روی هر سطر)حرکت میکنیم. همچنین چون با حرکت روی ستون ها، مقادیر لازم برای محاسبه در هر ستون یکی یکی کاهش می یابد، در یک حلقه ی دیگر از یکی کمتر از شماره ستون تا آخر (روی هر ستون) حرکت می کنیم. حالا با توجه به فرمولهای صفحه قبل، در هر مرحله f[xi,xi+1] را محاسبه کرده و در ماتریس جاگذاری می کنیم. باید توجه داشت که برای محاسبه این تفضلات تقسیم شده، صورت که مشخص و برابر با f(xj)-f(xj-1) است ولی برای یافتن مخرج با نوشتن چند جمله اولیه متوجه می شویم عنصر اول همان مقدار xj در همان سطر است همچنین اندیس عنصر دوم را با بررسی چند نمونه بدست می آوریم که 2 تا بیشتر از اختلاف i و j می باشد.

```
11   function the_matrix= matrix_maker(matrix,n)
12       for i=(3:n+1)
13           for j=(i-1:n)
14               matrix(j,i)=(matrix(j,i-1)-matrix(j-1,i-1))/(matrix(j,1)-matrix(j-i+2,1));
15           end
16       end
17       the_matrix=matrix;
18   end
```

در متن برنامه نیز با استفاده از تابع matrix_maker، ماتریس بدست آمده را در ماتریسی به نام the_matrix ذخیره می کنیم.

```
6    the_matrix=matrix_maker(matrix,n);
```

حال باید ضرایبی را که نیاز داریم، از ماتریسمان استخراج کنیم.(عناصری که در بخش توضیح مبحث تئوری هایلایت شده اند.) بنابراین تابعی به نام f_extractor با ورودی ماتریسِ ساخته شده در مرحله قبل (the_matrix) و تعداد داده ها (n) و خروجی ماتریسی به نام f_out که ضرایب مورد نیاز به ترتیب در آن ذخیره شده اند، تعریف می کنیم. در این تابع ابتدا ماتریس f_out با n سطر و یک ستون ستون و درایه های صفر را ایجاد می کنیم. سپس توسط یک حلقه این عناصر را (که عنصر اول در سطر اول و ستون دوم، و هر یک از عناصر دیگر در خانه ی پائین سمت راست عنصر قبلی قرار دارد. به طور کلی ما نیاز به عناصر درایه ی سطر i ام و ستون i+1 ام داریم.) بدست آورده و در ماتریس f_out ذخیره می کنیم.

```
19    function f_out=f_extractor(the_matrix,n)
20        f_out=zeros(n,1);
21        for k=(1:n)
22            f_out(k,1)=the_matrix(k,1+k);
23        end
24    end
```

در متن برنامه نیز با استفاده از این تابع، ضرایب را در ماتریسی به نام f_out ذخیره می کنیم.

```
7    f_out=f_extractor(the_matrix,n);
```

اکنون برای بدست آوردن چند جمله ای های $(x-x_0)(x-x_1)\dots(x-x_{n-1})$ تابعی با نام powers_cal ایجاد می کنیم که ورودی آن، ماتریس ما (the_matrix) و تعداد ورودی ها (n) و خروجی آن ماتریسی به نام a_out است که چند جمله ای های هر مرحله در یکی از درایه های آن به ترتیب ذخیره شده اند. ابتدا متغیری نمادین به نام t را تعریف کرده و سپس ماتریس a_out را با n سطر و 1 ستون و با مقادیر پیش فرض 1 با قابلیت ذخیره معادلات نمادین جهت ذخیره ی چند جمله ای ها تعریف می کنیم. از آنجا که این چند جمله ای ها از جمله ی دوم به بعد در تفاضلات تقسیم شده ضرب می شوند، در یک حلقه از 2 شروع کرده و تا n پیش میرویم. سپس توسط حلقه ای دیگر، چند جمله ای هارا برای جملات دیگر پیدا می کنیم.

```
25    function a_out=powers_cal(the_matrix,n)
26        syms t;
27        a_out=sym(ones(n,1));
28        for i=(2:n)
29            for j=(1:i-1)
30                a_out(i,1)=a_out(i,1)*(t-the_matrix(j,1));
31            end
32        end
33    end
```

حال در متن برنامه نیز توسط این تابع، چندجمله ها را در ماتریسی به نام a_out ذخیره می کنیم.

```
8     a_out=powers_cal(the_matrix,n);
```

اکنون که هم تفاضلات تقسیم شده و هم چند جمله ای هارا در اختیار داریم، برای بدست آوردن چند جمله ای تقریب تابع اصلی، کافیست تا تفاضلات تقسیم شده را در چند جمله ای های متناسب آنها ضرب کنیم. بنابراین از تابعی به نام polynominal_cal که ورودی های آن ماتریس های تفاضلات و چند جمله ای ها (f_out و a_out) و تعداد مقادیر (n) و خروجی آن چندجمله ای تقریبی نهایی می باشد. در این تابع ابتدا حاصل ضرب f_1 در a_1 را بدست آورده و وارد p_out می کنیم. سپس توسط یک حلقه برای i های بزرگتر از 2، f_i ها را در a_i های متناظر با آنها ضرب کرده و با P جمع کرده و در آن ذخیره می کنیم.

```
34    function p_out=polynominal_cal(a_out,f_out,n)
35        p_out=f_out(1,1)*a_out(1,1);
36        for j=(2:n)
37            p_out = p_out + f_out(j,1)*a_out(j,1);
38        end
39    end
```

در متن برنامه نیز حاصل نهایی چند جمله ای را توسط تابع قبل، در p_out ذخیره می کنیم.

```
9       p_out=polynominal_cal(a_out,f_out,n);
```

از آنجا که a ضریب t^2 ، b ضریب t و c عدد ثابت در چند جمله ای P می باشد، باید این ضرایب را از این چندجمله ای استخراج کرده و نمایش دهیم. بنابراین با استفاده از تابع coeffs ضرایبب t را در چند جمله ای p_out بدست آورده و با استفاده از تابع disp چاپ می کنیم.

```
10      disp(double(coeffs(p_out,t)));
```

اما باید توجه شود که اعداد چاپ شده به ترتیب، ضرایب توان های صفرم ، اول و دوم t می باشند. یعنی این اعداد از چپ به راست، ضرایب c و b و a می باشند.

```
1.0857   19.6905    0.2905
```

۵)

ابتدا مقادیر ورودی را در دو ماتریس جداگانه وارد کرده و سپس آنها را با استفاده از تابع horzcat کنار هم قرار می دهیم. همچنین تعداد ورودی ها را در متغیر n ذخیره می کنیم. (ضمنا تمام توابعی را که در قسمت های قبلی تعریف کردیم، در اینجا نیز از آنها استفاده می کنیم.)

```
T=[5;8;12];
V=[106.8;177.2;279.2];
variables=horzcat(T,V);
n=size(T,1);
syms t;
```

حال ابتدا دوباره ضرایب را توسط روش حذفی گاوس، دقیقا مانند قسمت الف بدست آورده و آنها را در ماتریسی به نام x_gauss ذخیره می کنیم. سپس در بازه زمانی 0 تا 40، چند جمله ای v را طبق ضرایب بدست آمده، توسط تابع polyval ایجاد می کنیم. سپس نمودار v بر حسب t را رسم کرده و با استفاده از hold on ، نمودار را حفظ کرده و نقاط داده های مسئله را نیز روی آمچن مشخص کرده و عنوان مناسبی را برای نمودار قرار می دهیم. در نهایت نیز با استفاده از hold off ، رسم را به پایان می رسانیم.

```
%gaussian_elimination.....................
[sortedvars]=sortrows(variables,-1);
[A,B]=AB_finder(sortedvars);
[A_new,B_new]=upper_triangularer(A,B);
X_gauss = gaussian_elimination(A_new,B_new);
t = linspace(0, 40);
v=polyval([X_gauss(1,1),X_gauss(2,1),X_gauss(3,1)],t);
plot (t,v);
hold on;
plot(5, 106.8, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(8, 177.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(12, 279.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
title('gaussian-elimination');
hold off;
```

حالا برای روش ژاکوبی نیز دقیقا همین اعمال را انجام داده و خواهیم داشت:

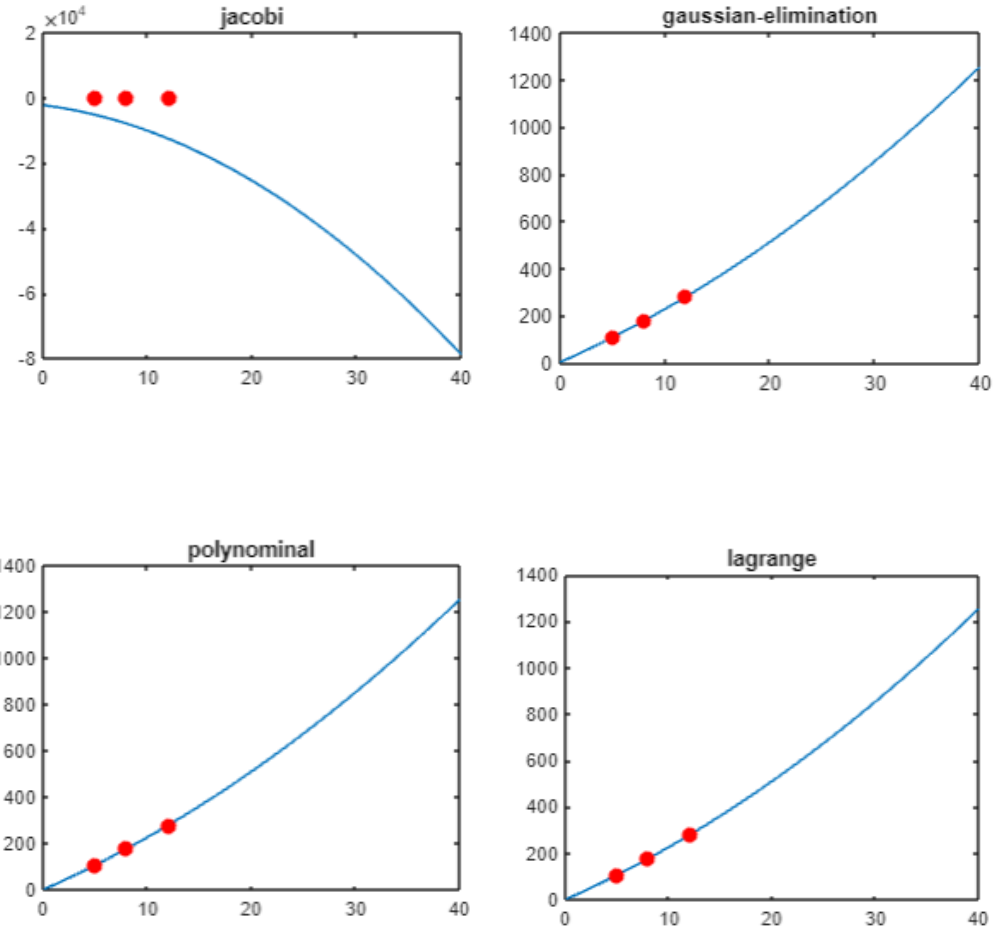
```
%jacobi.................................
X_pre_jacobi=[0;20;1];
X_jacobi=jacobi(n, A,B,X_pre_jacobi);
t = linspace(0, 40);
v=polyval([X_jacobi(1,1),X_jacobi(2,1),X_jacobi(3,1)],t);
plot (t,v);
hold on;
plot(5, 106.8, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(8, 177.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(12, 279.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
title('jacobi');
hold off;
```

همچنین برای روش های لاگرانژ و چندجمله ای درونیاب به همان صورت عمل می کنیم. ولی برای بدست آوردن ماتریس شامل ضرایب نهایی، ابتدا ضرایب چند جمله ای p (که به صورت کامل در قسمت های ج و د توضیح داده شده است) را توسط تابع coeffs بدست آورده و سپس آنها را به double تبدیل کرده و در نهایت نیز چون این ضرایب از توان صفرم تا توان دوم مرتب شده اند، آنها را توسط تابع fliplr به صورت دلخواهمان یعنی از توان بزرگتر به کوچکتر مر تب می کنیم.

```
%lagrange.............................
L=lagrange(T);
p=approximate_polynomianl(L,V);
X_lagrange=fliplr(double(coeffs(p)));
t = linspace(0, 40);
v=polyval([X_lagrange(1),X_lagrange(2),X_lagrange(3)],t);
plot (t,v);
hold on;
plot(5, 106.8, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(8, 177.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(12, 279.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
title('lagrange');
hold off;
```

```
%polynominal..........................
matrix=horzcat(T,V,zeros(n,n-1));
the_matrix=matrix_maker(matrix,n);
f_out=f_extractor(the_matrix,n);
a_out=powers_cal(the_matrix,n);
p_out=polynominal_cal(a_out,f_out,n);
X_poly=fliplr((double(coeffs(p_out))));
t = linspace(0, 40);
v=polyval([X_poly(1),X_poly(2),X_poly(3)],t);
plot (t,v);
hold on;
plot(5, 106.8, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(8, 177.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
plot(12, 279.2, 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r','linewidth',1.5);
title('polynominal');
hold off;
```

در نهایت، نتایج به صورت زیر خواهد بود:



از روی نمودار ها مشخص است روش ژاکوبی از آنجا که در ابتدا قطری غالب نشده است، کمترین دقت را دارد.

و)

در این قسمت نیز دقیقا همان کد های بخش قبلی را داریم با این تفاوت که به جای یافتن چند جمله ای v و رسم نمودار ، ابتدا متغیر the_point را که همان نقطه خواسته سوال است، تعریف کرده و مقدار آن را برابر با 10 قرار می دهیم. سپس متغیر result را که قرار است در هر مرحله پاسخ نهایی در آن ذخیره شود، تعریف کرده و مقدار اولیه آن را برابر با صفر قرار می دهیم. سپس در یک حلقه که متغیر آن از 0 تا n-1 (به تعداد ورودی ها) حرکت می کند، در هر مرحله، حاصل ضرب مقدار ضریب بدست آمده از آن روش را در توان متناظر از عدد the_point محاسبه کرده و با مقدار قبلی result جمع می کنیم. در نتیجه در نهایت مقدار چند جمله ای به ازای نقطه ی 10 بدست خواهد آمد. در پایان نیز نتیجه را چاپ می کنیم.

برای مثال برای روش حذفی گاوس خواهیم داشت:

```
6   %gaussian_elimination.....................
7   [sortedvars]=sortrows(variables,-1);
8   [A,B]=AB_finder(sortedvars);
9   [A_new,B_new]=upper_triangularer(A,B);
10  X_gauss = gaussian_elimination(A_new,B_new);
11  result=0;
12  the_point=10;
13  for i=(0:n-1)
14      result=result+(X_gauss(n-i)*(the_point^i));
15  end
16  fprintf('gaussian_elimination : t=10 so v=%d',result);
```

در نهایت نتایج بدست آمده به صورت زیر خواهند بود:

```
gaussian_elimination : t=10 so v=2.270381e+02

      jacobi : t=10 so v=-1.013432e+04

     lagrange : t=10 so v=2.270381e+02

  polynominal : t=10 so v=2.270381e+02
```

ر)

از آنجا که توابع نوشته شده در قسمت های قبل، برای n متغیر بودند، بنابراین در اینجا نیز از همان توابع استفاده می کنیم. با این 3 تفاوت:

1- در ماتریس های ورودی، مقادیر نقطه ی جدید را اضافه می کنیم.

```
T=[5;8;12;6];
V=[106.8;177.2;279.2;129.6];
```

2- در ماتریس ژاکوبی، مقادیر اولیه را با توجه به نتایج روش های دیگر، به صورت زیر تعیین می کنیم:

```
X_pre_jacobi=[0;0;18;4];
```

3- در هر قسمت، یک حلقه به صورت زیر برای چاپ ضرایب نیز قرار می دهیم:

```
for i=(1:n)
    disp(['the coefficient ', num2str(i), ' is : ',num2str(X_gauss(i,1)) ]);
end
```

نتایج بدست آمده در نهایت به شرح زیر خواهند بود:

روش گاوس:

```
                                        the coefficient 1 is : -0.0071429
                                        the coefficient 2 is : 0.46905
                                        the coefficient 3 is : 18.2905
gaussian_elimination : t=10 so v=2.271810e+02  the coefficient 4 is : 4.5143
```

روش ژاکوبی:

```
                                    the coefficient 1 is : -14907.3921
                                    the coefficient 2 is : -198182.2359
                                    the coefficient 3 is : -1410845.5206
     jacobi : t=10 so v=-5.566661e+07 the coefficient 4 is : -6832542.0919
```

روش لاگرانژ:

```
                                     the coefficient 1 is : -0.0071429
                                     the coefficient 2 is : 0.46905
                                     the coefficient 3 is : 18.2905
   lagrange : t=10 so v=2.271810e+02 the coefficient 4 is : 4.5143
```

روش چندجمله ای درونیاب:

```
                                      the coefficient 1 is : -0.0071429
                                      the coefficient 2 is : 0.46905
                                      the coefficient 3 is : 18.2905
polynominal : t=10 so v=2.271810e+02  the coefficient 4 is : 4.5143
```

سوال سوم:

الف)

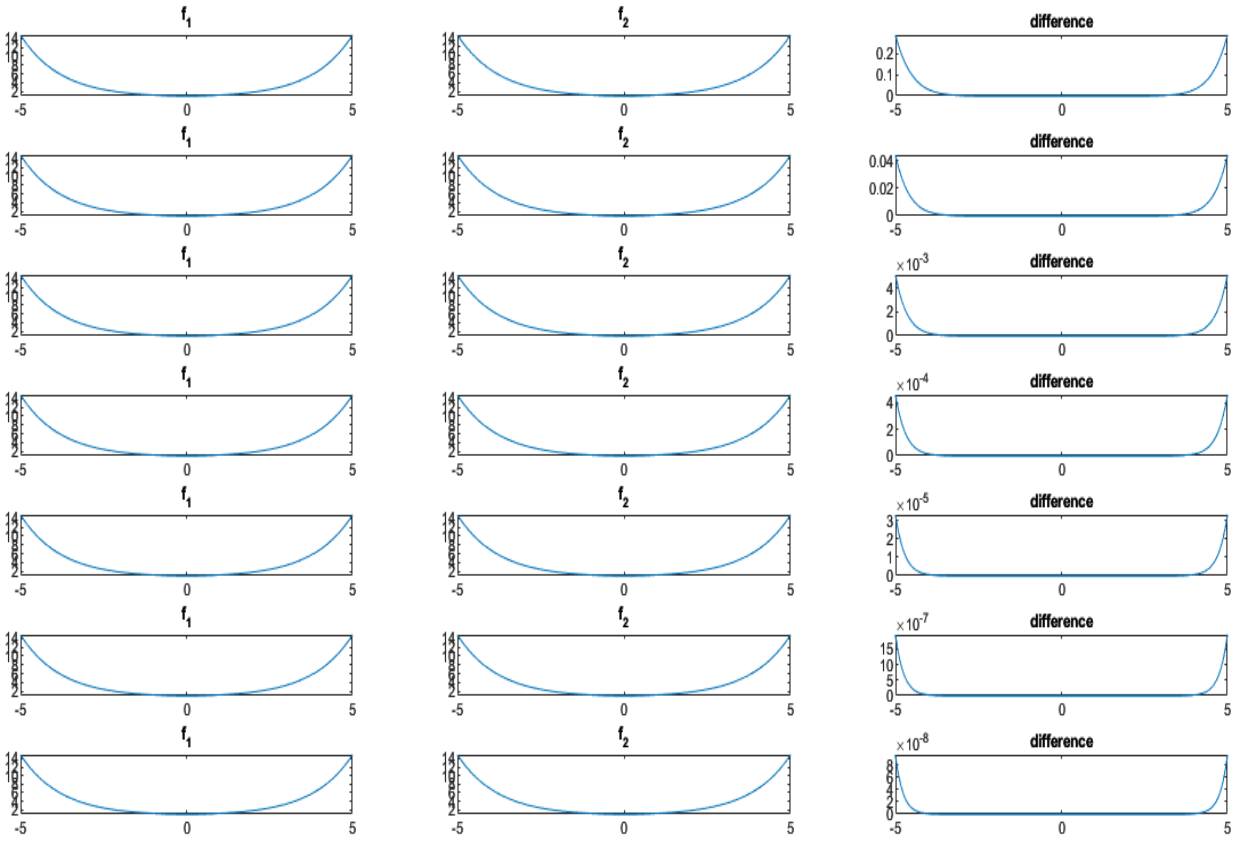
ابتدا متغیری نمادین به نام x را تعریف می کنیم. سپس دو تابع با نام های f_1 و f_2 را تعریف می کنیم که همان توابع داده شده در صورت سوال هستند. توجه کنید که f_1 ورودی ندارد ولی از آنجا که برای تابع دوم نیاز به n داریم، در هر مرحله آنرا از ورودی می گیریم. همچنین در تابع دوم، ابتدا مقدار نتیجه را برابر با صفر قرار داده و در یک حلقه که n بار تکرار می شود، مقدار زیگمای سوال را به مقدار نتیجه اضافه می کنیم.

```
function result_1=f_1
    syms x;
    result_1=(sinh(x))/x;
end
function result_2=f_2(n)
    syms x;
    result_2=0;
    for i=(0:n)
        result_2=result_2+((x^(2*i))/factorial((2*i)+1));
    end
end
```

حال در متن برنامه ابتدا متغیری به نام i را برای جابجایی روی خانه های subplot تعریف کرده و مقدار آن را برابر با 1 قرار می دهیم. حال در یک حلقه که از 4 تا 10 تکرار می شود، در یک subplot که 7 ردیف و 3 ستون دارد، در ستون اول تابع اول و در ستون دوم، تابع دوم به ازای مقادیر متفاوت n و در ستون سوم هر ردیف، تابع اختلاف آنها را نمایش می دهیم. در نهایت نیز i را 3 واحد شیفت می دهیم تا وارد سطر بعدی شود.

```
syms x;
i=1;
for n=(4:10)
    subplot(7,3,i);
    fplot(f_1);
    title('f_1');
    subplot(7,3,i+1);
    fplot(f_2(n));
    title('f_2');
    subplot(7,3,i+2);
    fplot(f_1 - f_2(n));
    title('difference');
    i=i+3;
end
```

خروجی نیز به صورت زیر خواهد بود:



ب)

در این بخش نیز دقیقا مانند بخش قبلی، توابع f_1 و f_2 را تعریف می کنیم. سپس ابتدا مقدار حد تابع اول را در x=0 توسط تابع limit محاسبه کرده و چاپ میکنیم.

```
2       fprintf('limit of f_1 when x=0  = %d\n',limit(f_1,x,0));
```

سپس در یک حلقه، به ازای مقادیر مختلف n، مقدار حد تابع دوم را محاسبه کرده و چاپ می کنیم.

```
3       for n=(4:10)
4           fprintf('limit of f_2 when x=0 & n=%d  = %d\n',n,limit(f_2(n),x,0));
5       end
```

نتیجه به صورت زیر بدست می آید:

```
limit of f_1 when x=0  = 1
limit of f_2 when x=0 & n=4  = 1
limit of f_2 when x=0 & n=5  = 1
limit of f_2 when x=0 & n=6  = 1
limit of f_2 when x=0 & n=7  = 1
limit of f_2 when x=0 & n=8  = 1
limit of f_2 when x=0 & n=9  = 1
limit of f_2 when x=0 & n=10  = 1
```

ج)

با توجه به قسمت های قبل، می توان نتیجه گرفت که با افزایش n ، اختلاف بین دو تابع کاهش می یابد.

در واقع تابع دوم، به ویژه در نقطه ی x=0 ، تقریبی برای تابع اول است که وقتی مقدار n افزایش می یابد، مقدار خطا کاهش می یابد.

د)

توضیح مبحث تئوری:

می دانیم هرگاه بخواهیم ریشه (ساده) تابعی مانند f را به روش نیوتن-رافسون بدست آوریم، با داشتن یک نقطه ی شروع، می توان در هر مرحله ریشه را به صورت زیر بدست آورد:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

حل سوال:

در این سوال می خواهیم ریشه ی تابع زیر را بدست آوریم:

$$g = f_1 - f_2$$

بنابراین ابتدا توابع f_1 و f_2 را همانند قسمت های قبلی، مطابق با صورت سوال تعریف می کنیم.

```
13    function result_1=f_1
14        syms x;
15        result_1=(sinh(x))/x;
16    end
17    function result_2=f_2(n)
18        syms x;
19        result_2=0;
20        for i=(0:n)
21            result_2=result_2+((x^(2*i))/factorial((2*i)+1));
22        end
23    end
```

حال، متغیر x_n را با مقدار اولیه 10 و متغیر x_new را با مقدار اولیه صفر تعریف می کنیم. سپس در یک حلقه ی بی نهایت، ابتدا رابطه ی نیوتن رافسون را نوشته و مقدار بدست آمده از آن به ازای x_n را در x_new ذخیره سازی می کنیم. سپس با یک شرط بررسی میکنیم که اگر اختلاف بین دو ریشه قدیم و جدید، یعنی x_n و x_new، از 10^{-4} کمتر بود، از حلقه ی بی نهایت خارج می شویم. در غیر اینصورت، مقدار x_new را در x_n ریخته و دوباره روش نیوتن رافسون را تکرار می کنیم تا به مقدار خطای دلخواه برسیم.

در نهایت نیز مقدار ریشه ی بدست آمده را چاپ می کنیم.

```
syms x;
x_n=10;
x_new=0;
while true
    x_new=double(x_n - ( subs( (f_1-f_2(4) ),x_n )/subs( diff((f_1-f_2(4)),x),x_n ) ));
    if (abs(x_new-x_n)<(10^(-4)))
        break;
    end
    x_n=x_new;
end

disp(x_new);
```

مقدار ریشه نهایی چاپ شده به صورت زیر خواهد بود:

```
8.3115e-04
```

سوال چهارم:

الف)

توضیح مبحث تئوری:

هرگاه دو داده ی متوالی و مقادیر آنها در تابع را داشته باشیم، می دانیم برای بدست آوردن تقریبی نقطه ای که در آن، مقدار تابع برابر با صفر شود، به روش وتری در هر مرحله از رابطه ی زیر بدست می آید:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

و یا:

$$x_r = x_u - \frac{f(x_u)(x_u - x_l)}{f(x_u) - f(x_l)}$$

حل سوال:

ابتدا برای آنکه محاسبات ما تا 15 رقم پس از اعشار انجام شوند، در سط اول از دستور format long استفاده می کنیم.

```
1    format long
```

حال دو ماتریس تهی به نام های sigma_s_matrix و d_vatari_matrix به ترتیب برای ذخیره مقادیر دانسیته ذرات و قطر دانه ها که از محاسبات به دست آمده است ایجاد می کنیم.

```
2    sigma_s_matrix=[];
3    d_vatari_matrix=[];
```

برای حل این سوال نیاز به یک تابع داریم تا مقداریری را که تابع در آن ها صفر می شود را بیابیم. از آنجا که همه مقادیر به جز قطر ذرات و دانسیته ذرات ثابت و موجود هستند، هدف در این سوال یافتن ریشه های تابع زیر است:

$$f = \sqrt{\frac{3gd(\sigma_s - \sigma_f)}{\sigma_f}} - v = 0$$

بنابراین یک تابع ایجاد کرده و تابع قبل را همراه با مقادیر پیش فرض سوال، در آن تعریف می کنیم. نام تابع f بوده و مقادیر ورودی آن به ترتیب قطر ذرات و دانسیته ذرات می باشند. همچنین در هر مرحله مقدار تابع در متغیری به نام zero ذخیره شده و آن خروجی تابع ما خواهد بود.

```
24      function zero=f(d,sigma_s)
25          g=9.81;
26          sigma_f=1.5;
27          v=21.69;
28          zero=sqrt((3*g*d*(sigma_s-sigma_f))/sigma_f)-v;
29      end
```

حال باید مقادیر d را به ازای دانسیته های متفاوت و در هر مرحله با 30 بار تکرار با روش وتری بدست آوریم، برای اینکار ابتدا در یک حلقه ی for دانسیته های مختلف را تولید کرده و برای هر مرحله درون آن نقاط اولیه را تعریف کرده و در سطر 7 ام در هر مرحله، دانسیته ی تولید شده را به ماتریس sigma_s_matrix اضافه می کنیم.

حال در یک حلقه دیگر، تعداد 30 تکرار را پیاده سازی می کنیم. و در هر مرحله بررسی می کنیم در صورتی که تابع ما صفر نشده باشد، ابتدا مقدار نقطه ی جدید را با جایگذاری نقاط اولیه در فرمول روش وتری و با استفاده از تابع تعریف شده در مرحله ی قبل یافته و سپس مقادیر نقاط اولیه را عوض می کنیم. در پایان این حلقه و قبل از اتمام حلقه ی اولیه، نقطه ی حاصل از روش وتری را به انتهای ماتریس d_vatari_matrix اضافه می کنیم.

```
4       for sigma_s=(5:0.05:9.5)
5           d_u=0;
6           d_l=15;
7           sigma_s_matrix=horzcat(sigma_s_matrix,sigma_s);
8           for n=(1:30)
9               if f(d_u,sigma_s)~=0
10                  d_n=d_u-((f(d_u,sigma_s)*(d_u-d_l))/(f(d_u,sigma_s)-f(d_l,sigma_s)));
11                  d_l=d_u;
12                  d_u=d_n;
13              end
14          end
15          d_vatari_matrix=horzcat(d_vatari_matrix,d_n);
16      end
```

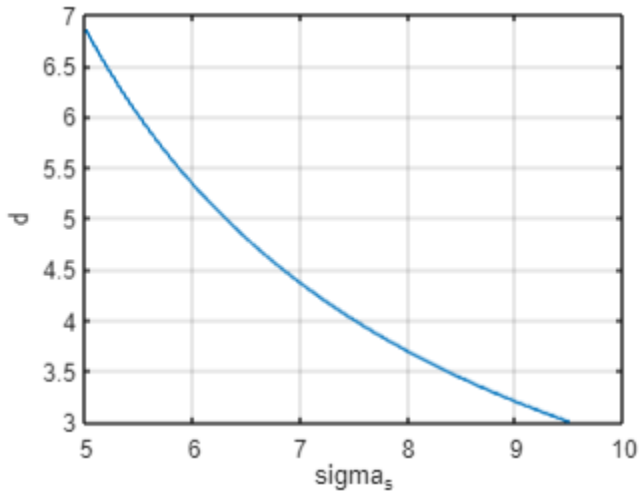
اکنون ماتریس مقادیر قطر ذرات را به سطر دوم مقادیر دانسیته اضافه کرده و کل آنرا ماتریسی به نام values می نامیم و در نهایت آنرا نمایش می دهیم. به صورتی که در ماتریس نمایش داده شده، سطر اول برابر با مقادیر دانسیته ذرات و سطر دوم شامل مقدار قطر ذرات متناسب با هر یک از آنها می باشد.

```
values= vertcat(sigma_s_matrix,d_vatari_matrix);
disp(values);
```

حال برای نمایش مقادیر d به ازای مقادیر دانسیته ذرات، از تابع plot استفاده کرده و محور ها را نامگذاری می کنیم.

```
plot(sigma_s_matrix, d_vatari_matrix);
xlabel('sigma_s');
ylabel('d');
grid on;
```

در نهایت، منحنی خروجی به شکل زیر خواهد بود:



ب)

توضیح مبحث تئوری:

برای حل معادلات به روش دوبخشی یا تصنیف، با استفاده از قضیه ی بولتزانو می دانیم اگر حاصل ضرب مقادیر تابع در دو نقطه متفاوت منفی باشد، ریشه ، بین آن دو نقطه قرار خواهد داشت. بنابراین در صورتی که a و b دو نقطه ی اولیه باشند، الگوریتم کلی برای یافتن ریشه های یک تابع در هر مرحله به صورت زیر خواهد بود:

$$x_m = \frac{a+b}{2}$$

If $f(x_m)f(a) < 0 : b = x_m,$

elif $f(x_m)f(b) < 0 : a = x_m,$

else $x_r = x_m$.

حل سوال:

ابتدا برای آنکه محاسبات ما تا 15 رقم پس از اعشار انجام شوند، در سطر اول از دستور format long استفاده می کنیم.

```
1    format long
```

حال دو ماتریس تهی به نام های sigma_s_matrix و d_tasnif_matrix به ترتیب برای ذخیره مقادیر دانسیته ذرات و قطر دانه ها که از محاسبات به دست آمده است ایجاد می کنیم.

```
2    sigma_s_matrix=[];
3    d_tasnif_matrix=[];
```

برای حل این سوال نیاز به یک تابع داریم تا مقادیری را که تابع در آن ها صفر می شود را بیابیم. از آنجا که همه مقادیر به جز قطر ذرات و دانسیته ذرات ثابت و موجود هستند، هدف در این سوال یافتن ریشه های تابع زیر است:

$$f = \sqrt{\frac{3gd(\sigma_s - \sigma_f)}{\sigma_f}} - v = 0$$

بنابراین یک تابع ایجاد کرده و تابع قبل را همراه با مقادیر پیش فرض سوال، در آن تعریف می کنیم. نام تابع f بوده و مقادیر ورودی آن به ترتیب قطر ذرات و دانسیته ذرات می باشند. همچنین در هر مرحله مقدار تابع در متغیری به نام zero ذخیره شده و آن خروجی تابع ما خواهد بود.

```
24    function zero=f(d,sigma_s)
25        g=9.81;
26        sigma_f=1.5;
27        v=21.69;
28        zero=sqrt((3*g*d*(sigma_s-sigma_f))/sigma_f)-v;
29    end
```

حال باید مقادیر d را به ازای دانسیته های متفاوت و در هر مرحله با 40 بار تکرار با روش دوبخشی(تصنیف) بدست آوریم، برای اینکار ابتدا در یک حلقه ی for دانسیته های مختلف را تولید کرده و برای هر مرحله درون آن نقاط اولیه a و b را تعریف کرده و در سطر 7 ام در هر مرحله، دانسیته ی تولید شده را به ماتریس sigma_s_matrix اضافه می کنیم.

اکنون در یک حلقه دیگر، تعداد 40 تکرار را پیاده سازی می کنیم. و در هر مرحله ابتدا متغیری به نام avg را تعریف کرده و مقدار آن را برابر با میانگین a و b قرار می دهیم. در واقع متغیر avg همان x_m است. حال بررسی می کنیم در صورتی که حاصل ضرب مقادیر تابع به ازای avg و a منفی باشد، یعنی ریشه بین این دو مقدار قرار دارد؛ بنابراین مقدار avg را در b میریزیم. در غیر اینصورت، یعنی زمانی که حاصلضرب مقادیر تابع به ازای avg و b منفی باشد، یعنی ریشه بین این دو مقدار قرار دارد؛ بنابراین مقدار avg را در a می ریزیم.

حال اگر هیچ یک از این دو حالت رخ ندهد، یعنی avg دقیقا برابر با ریشه ی تابع می باشد، بنابراین در این حالت دیگر ادامه نداده و از حلقه خارج می شویم. در انتها بعد از 40 بار تکرار، مقدار avg بدست آمده نهایی را به انتهای ماتریس d_tasnif_matrix اضافه می کنیم.

```
for sigma_s=(5:0.05:9.5)
    a=0;
    b=100;
    sigma_s_matrix=horzcat(sigma_s_matrix,sigma_s)
    for i=(1:40)
        avg=(a+b)/2;
        if f(a,sigma_s)*f(avg,sigma_s)<0
            b=avg;
        elseif f(avg,sigma_s)*f(b,sigma_s)<0
            a=avg;
        else
            break;
        end
    end
    d_tasnif_matrix=horzcat(d_tasnif_matrix,avg);
end
```

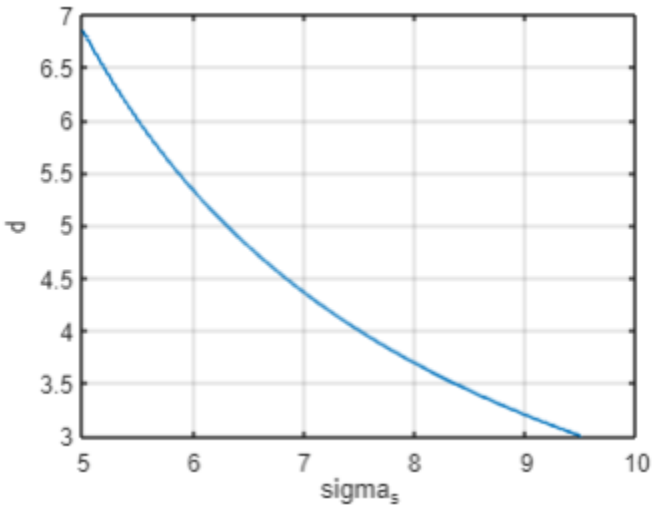
در پایان که مقادیر d به ازای دانسیته های مختلف بدست آمده است، ماتریس مقادیر قطر ذرات را به سطر دوم مقادیر دانسیته اضافه کرده و کل آنرا ماتریسی به نام values می نامیم و در نهایت آن را نمایش می دهیم. به صورتی که در ماتریس نمایش داده شده، سطر اول برابر با مقادیر دانسیته ذرات و سطر دوم شامل مقدار قطر ذرات متناسب با هر یک از آنها می باشد.

```
values=vertcat(sigma_s_matrix, d_tasnif_matrix);
disp(values);
```

حال برای نمایش مقادیر d به ازای مقادیر دانسیته ذرات، از تابع plot استفاده کرده و محور ها را نامگذاری می کنیم.

```
plot(sigma_s_matrix, d_tasnif_matrix);
xlabel('sigma_s');
ylabel('d');
grid on;
```

نمودار بدست آمده در نهایت به شکل زیر خواهد بود:



ج)

ابتدا برای آنکه محاسبات ما تا 15 رقم پس از اعشار انجام شوند، در سط اول از دستور format long استفاده می کنیم.

```
1    format long
```

حال باید رابطه ی d را برحسب دیگر متغیر ها بدست آوریم. بنابراین خواهیم داشت:

$$v = \sqrt{\frac{3gd(\sigma_s - \sigma_f)}{\sigma_f}} \Rightarrow d = \frac{v^2.\sigma_f}{3g(\sigma_s - \sigma_f)}$$

بنابراین یک تابع ایجاد کرده و تابع مقدار d را همراه با مقادیر پیش فرض سوال، در آن تعریف می کنیم. نام تابع real_d بوده و مقدار ورودی آن دانسیته ذرات می باشد. همچنین در هر مرحله مقدار تابع در متغیری به نام f ذخیره شده و آن خروجی تابع ما خواهد بود.

```
47  function f=real_d(sigma_s)
48      g=9.81;
49      sigma_f=1.5;
50      v=21.69;
51      f=((v^2)*sigma_f)/(3*g*(sigma_s-sigma_f));
52  end
```

اکنون می خواهیم مقادیر اصلی d را به ازای دانسیته های مختلف بدست آوریم. بنابراین ابتدا دو ماتریس تهی به نام های sigma_s_matrix و d_real_matrix به ترتیب برای ذخیره مقادیر دانسیته ذرات و قطر دانه ها که از محاسبات به دست آمده است ایجاد می کنیم.

```
2   d_real_matrix=[];
3   sigma_s_matrix=[];
```

حالا در یک حلقه به ازای مقادیر مختلف دانسیته، اولا مقدار دانسیته را در ماتریس مربوط به خود ذخیره کرده و ثانیا مقدار واقعی d را با استفاده از تابع real_d محاسبه کرده و در ماتریس مربوط به خودش ذخیره میکنیم.

```
4   for sigma_s=(5:0.05:9.5)
5       sigma_s_matrix=horzcat(sigma_s_matrix,sigma_s);
6       d_real_matrix=horzcat(d_real_matrix, real_d(sigma_s));
7   end
```

حال برای نمایش مقادیر بدست آمده ابتدا ماتریس مقادیر d را به سطر دوم مقادیر دانسیته اضافه کرده و کل آنرا ماتریسی به نام values می نامیم و در نهایت آنرا نمایش می دهیم. به صورتی که در ماتریس نمایش داده شده، سطر اول برابر با مقادیر دانسیته ذرات و سطر دوم شامل مقدار واقعی قطر ذرات متناسب با هر یک از آنها می باشد.

```
values= vertcat(sigma_s_matrix,d_real_matrix);
disp(values);
```

از آنجا که برای بخش دوم سوال، به مقادیر بدست آمده از قسمت های قبلی نیاز داریم، ابتدا همان تابع f را که در قسمت های الف و ج از آن استفاده کرده بودیم، اینجا نیز پیاده سازی می کنیم.

```
function zero=f(d,sigma_s)
    g=9.81;
    sigma_f=1.5;
    v=21.69;
    zero=sqrt((3*g*d*(sigma_s-sigma_f))/sigma_f)-v;
end
```

سپس با استفاده از برنامه نوشته شده برای قسمت الف، مقادیر d بدست آمده از روش وتری را در ماتریسی به نام d_vatari_matrix ذخیره می کنیم.

```
%VATARI.................................
d_vatari_matrix=[];
for sigma_s=(5:0.05:9.5)
    d_u=0;
    d_l=15;
    for n=(1:30)
        if f(d_u,sigma_s)~=0
            d_n=d_u-((f(d_u,sigma_s)*(d_u-d_l))/(f(d_u,sigma_s)-f(d_l,sigma_s)));
            d_l=d_u;
            d_u=d_n;
        end
    end
    d_vatari_matrix=horzcat(d_vatari_matrix,d_n);
end
```

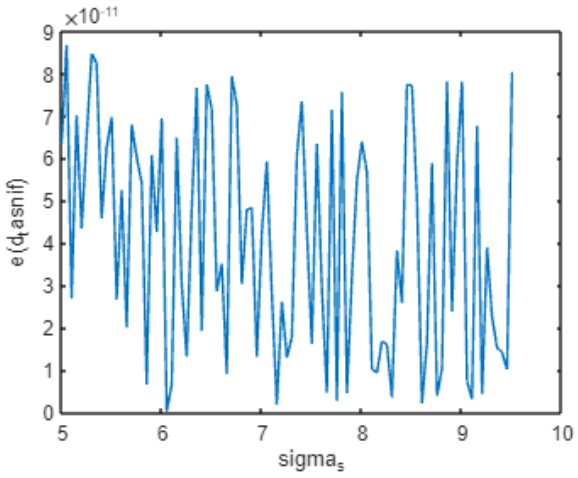
در نهایت نیز با استفاده از برنامه نوشته شده برای قسمت ب، مقادیر d بدست آمده از روش دوبخشی (تصنیف) را در ماتریسی به نام d_tasnif_matrix ذخیره می کنیم.

```
%TASNIF.....................................
d_tasnif_matrix=[];
for sigma_s=(5:0.05:9.5)
    a=0;
    b=100;
    for i=(1:40)
        avg=(a+b)/2;
        if f(a,sigma_s)*f(avg,sigma_s)<0
            b=avg;
        elseif f(avg,sigma_s)*f(b,sigma_s)<0
            a=avg;
        end
    end
    d_tasnif_matrix=horzcat(d_tasnif_matrix,avg);
end
```

حال برای رسم نمودار قدر مطلق خطای d برای هر یک از روش ها، از تابع plot استفاده کرده و در آن برای هر کدام ، ابتدا قدر مطلق اختلاف مقادیر بدست آمده از روش مورد نظر را با مقدار واقعی d محاسبه کرده و آنرا بر حسب دانسیته ذرات رسم می کنیم. البته بعد از رسم هر یک از نمودار ها، محور ها را نامگذاری می کنیم.

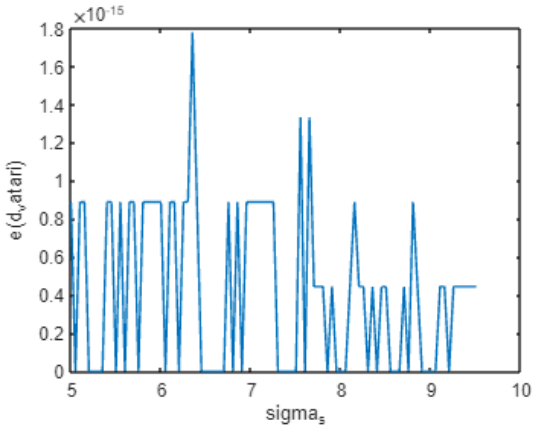
```
%ERROR TASNIF.................................................
plot(sigma_s_matrix, abs(d_tasnif_matrix-d_real_matrix));
ylabel("e(d_tasnif)");
xlabel("sigma_s");
%ERROR VATARI.................................................
plot(sigma_s_matrix, abs(d_vatari_matrix-d_real_matrix));
ylabel("e(d_vatari)");
xlabel("sigma_s");
```

نمودار قدر مطلق خطای d بر حسب دانسیته ذرات برای روش دوبخشی (تصنیف) به صورت زیر خواهد بود.



طبق نمودار مشاهده میگردد که کمترین مقدار خطا در دانسیته ی 6.05 و بیشترین مقدار آن در دانسیته ی 5.05 رخ می دهد.

نمودار قدر مطلق خطای d بر حسب دانسیته ذرات برای روش وتری نیز به صورت زیر خواهد بود.



طبق نمودار مشاهده میگردد که کمترین مقدار خطا که برابر صفر می باشد، در دانسیته های مختلف و بیشترین مقدار خطا در دانسیته ی 6.35 رخ می دهد.