Problem set 1b: MATLAB Fundamentals and Graphics

Handed out: Friday, September 20, 2024
Due: 23:55pm, Thursday, October 3, 2024

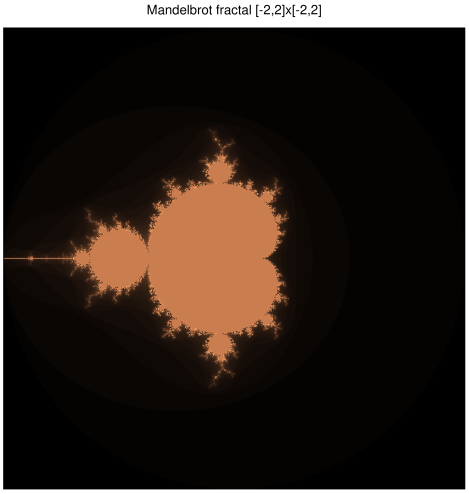
You must hand in two files:

- One file, named `your_name_E1b.pdf`, with the solutions to the exercises in this set, following the template available in the virtual campus. You must explain at least how you solved every exercise, the MATLAB code you generated and the results obtained (run transcripts or plots).
- The second file, named `your_name_E1b.zip`, must contain all your MATLAB code, organized in one or more text (`.m`) files.

Please, make sure that the code referred to in the pdf file **exactly** matches the code provided in the zip file. Remember that all the MATLAB code is supposed to be **entirely yours**. Otherwise, you must clearly specify which parts are not, and properly attribute them to the source (names of collaborators, links to webpages, book references, ...). Read the Course Information document for more details.

Exercise 4. Drawing the Mandelbrot fractal set.

The goal of this exercise is to build a short MATLAB program capable to draw the celebrated Mandelbrot set, with some (limited) user interaction features to select the desired region and resolution. The exercise is written in a tutorial style, and it introduces some MATLAB techniques like complex numbers, vectorization, boolean indexing, basic flow control and graphics. **The pieces of code given below are not intended to be copied and pasted into your program**, but they are provided just to make the explanation more clear.



The formal definition. Mandelbrot set is a region of the complex plane, related to the sequence $(z_0 = 0, z_1 = c, z_2 = c^2 + c, \ldots)$ defined by the recurrence

$$z_{n+1} = z_n^2 + c$$

and its convergence depending on the complex parameter $c \in \mathbb{C}$. Indeed, the Mandelbrot set is the locus of all $c \in \mathbb{C}$ such that the sequence does not diverge (not meaning that it converges, because it still can be an 'oscillating' sequence).

The approximations. The shape of the Mandelbrot set (or in particular the shape of its boundary) is really intrincate. Hence, we need a computer program to build an approximation to it, and this is the goal of the exercise.

A simple fact about the sequence $(z_0, z_1, z_2, \ldots)$ is that if for some n, $|z_n| > 2$, then the sequence diverges (and the corresponding value of c does not belong to the Mandelbrot set).

The usual way to approximate the set is the following:

1. Choose the desired rectangle in the complex plane, and discretize it with a $m_x \times m_y$ mesh of points, $c_{i,j}$, for $i = 1, \ldots, m_x$, $j = 1, \ldots, m_y$.
2. For every point $c_{i,j}$ in the mesh, compute the first k terms of the sequence, $(0, z_1, z_2, \ldots, z_k)$.
3. Assign to this mesh point the value $v_{i,j}$ of the first index n (if any) such that $|z_n| > 2$. If no such index value exists, then set $v_{i,j} = k + 1$.
4. Draw a color image such that pixel (i, j) is colored according to its associated value $v_{i,j}$.

Observe that the points in the Mandelbrot set never produce sequence elements such that $|z_n| > 2$, and therefore they always have $v_{i,j} = k + 1$.

The program. You can find many programs on the internet that compute and draw approximations of the Mandelbrot set. However, the aim of the exercise is doing it yourself in MATLAB, obtaining a simple but decent functionality, while you learn some useful MATLAB techniques.

A typical Mandelbrot set program allows you to choose the desired region (to see the details), the number of points $m_x \times m_y$ (or image resolution) and the number of iterations to be performed k. In the following, we describe step by step the main ideas to build your program in MATLAB.

Step 1: Computing with complex numbers. MATLAB handles complex numbers in a natural way, and all functions and operators that can be defined over the complex numbers are supported. For instance, the following code produces the first 101 terms (including the initial $z_0 = 0$) of the sequence defined above for $c = 1 + 0.5i$:

```
C=1+0.5i;
Z=zeros(1,101); %generate a row vector with 101 zeros
for n=1:100
  Z(n+1)=Z(n)^2+C;
end
```

Another way to build a sequence is by appending a new element at each iteration, like in

```
C=1+0.5i;
Z=0;
for n=1:100
```

```
  Z=[Z,Z(end)^2+C];
end
```

but this latter form is less efficient due to the memory management (at every iteration a new vector has to be allocated in memory and all the old contents must be copied to the new location, causing around 5000 memory operations in the given example, compared to only 100 in the former version). However, here we do not need to store the whole sequence but only the last element, or even better, we only need to iterate until $|z_n| > 2$:

```
C=1+0.5i;
Z=0;
V=101; %just in case abs(Z)>2 never happens
for n=1:100
  Z=Z^2+C;
  if abs(Z)>2
    V=n;
    break %stop iterating
  end
end
```

Step 2: Vectorization. Vectorization is a powerful tool that often makes the inefficient loops in an interpreted language like MATLAB become as efficient as in the low-level compiled languages like C. Without vectorization, we would need two nested 'for' loops to compute the sequence of complex numbers for every $c_{i,j}$, in the form:

```
for I=1:MX
  for J=1:MY
    %perform here the above computations for C=C(I,J)
  end
end
```

To benefit from vectorization, we will compute on matrices instead of computing on the elements one-by-one. To do that, firstly fill the matrix C with all the complex numbers $c_{i,j}$, typically using

```
[CX,CY]=meshgrid(RangeX,RangeY);
C=Cx+CY*1i;
```

for suitable vectors **RangeX** and **RangeY**. Now you can iterate all values at once without any additional loop:

```
Z=zeros(size(C)); %Z is a matrix with the same shape as C
for n=1:100
  Z=Z.^2+C; %elementwise computations
end
```

By using the dot notation, the square is computed elementwise and then we are computing the sequence of complex numbers at once for all (i, j).

The problem now is how to compute the values $v_{i,j}$ at once. Indeed, we cannot use the above **`if`** and **`break`** construction, because each sequence will reach the stop condition $|z_n| > 2$ at a different iterationi n. Fortunately, there is a (very technical) way to solve the problem, as explained in next step.

Step 3: Boolean indexing. Typical MATLAB indexing of vectors and matrices is by position, as in many programming languages, like in **`V(3)`** or **`M(1,5)`**, or the multiple-indexes extension used in MATLAB, **`V(2:5)`**, **`M(3:5,:)`**. But there is a powerful alternative way to select elements from a vector or matrix. For instance, **`M(M>2)`** corresponds to the part of the matrix **`M`** containing only the elements that are greater than 2.

The way it works is the following: **`B=M>2`** creates a matrix with the same size as **`M`** of booleans, in such a way that **`B(I,J)`** is true if and only if **`M(I,J)>2`**. This is just an example, but it would work for any boolean expression.

Now the boolean matrix **`B`** can be used as a 'selector' of elements in **`M`** (or in any other matrix of the same size as **`B`**) by simply writing **`M(B)`**. Look the following examples:

```
B=M>2;
M(B)=2; %write 2 in the positions of M indicated by B
M(B)=M(B).^2; %square the positions of M indicated by B
V(B)=n; %write n in the positions of V indicated by B
```

Applying this technique to our problem, we can perform the computations in each iterations only in those elements that fulfil $|z_n| \leq 2$. That is,

```
Z=zeros(size(C)); %Z is a matrix with the same shape as C
V=ones(size(C)); %idem for the resulting matrix V
for n=1:100
  B=abs(Z)<=2;
  Z(B)=Z(B).^2+C(B); %compute only where needed
  V(B)=V(B)+1; %update the final values
end
```

Observe that the values stored in **`V`** range from 1 to 101, since each $v_{i,j}$ is incremented every time the corresponding sequence element z_n fulfils $|z_n| \leq 2$. The obtained benefit is twofold: boolean indexing allowed us to use vectorization, speeding up the program, and we only iterate when necessary.

Being honest, the solution given above is still sub-optimal, and it can be improved in a number of ways (e.g., write every **`V(I,J)`** only once, instead of incrementing it one-by-one, or do not check again whether $|z_n| \leq 2$ if the condition failed). However, any further improvement will obscure the code even more, and this is not the aim of the exercise.

Step 4: Plotting the results. Given an integer matrix **`V`**, we can execute **`image(V)`** to draw the corresponding color image. However, the number of colors available in the palette is limited (normally to 64, or more generally, to **`size(colormap,1)`**). Then, we need to remap the values in **`V`** to a specific range (say, **`1:64`**). This can be easily done with **`image(mod(V,64)+1)`**, where any additional multiple of 64 is simply ignored by the modulo operation. There are also

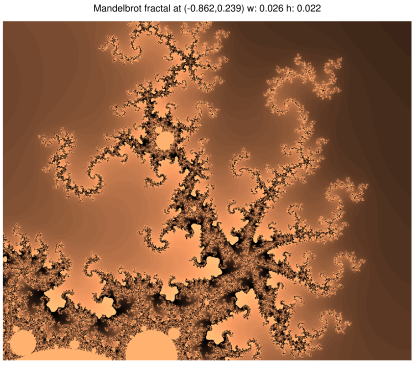
some useful options given by the command **axis** that would be useful to tailor the resulting image. Try the following:

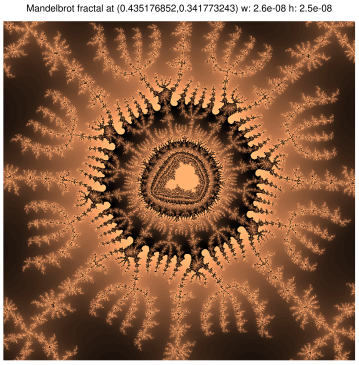
```
image([Xmin,Xmax],[Ymin,Ymax],mod(V,64)+1);
axis xy;
axis equal;
axis off;
```

You can also change the palette by either a predefined or a custom one. E.g., try **colormap copper**.

Step 5: Adding user interaction. There exist powerful tools in MATLAB to build graphic user interfaces to allow the user to interact with the program, but they are out of the scope of this unit. Alternatively, the very simple function **ginput** would be enough here. A call **[a,b]=ginput(2);** waits for the user to click the mouse twice on the figure and returns the coordinates of the clicked points. In particular, **a** is a column vector with the two x-coordinates of the clicked points, and **b** similarly contains the y-coordinates.

If you call **ginput** after plotting the image, the user can select a rectangular region (by clicking on two oposite corners of it). Then, the program can recompute everything again and plot the resulting image for the selected region. This procedure can be repeated arbitrarilly in a loop until some specific condition is detected (e.g. a key has been pressed on the keyboard).





The function **ginput** also gives information about which mouse button is pressed or if a key form the keyboard was pressed instead. This information is given in a third argument. For instance, **[a,b,c]=ginput(1)** returns the coordinates of a single mouse click (then for two clicks just call the function twice), and **c** takes values 1,2,... for the left, right, center mouse buttons, or the ASCII code of the keyboard key pressed instead. Then you can use something like

```
[a1,b1,c]=ginput(1);
if c<=3 %the mouse was clicked at point (a1,b1)
  [a2,b2]=ginput(1); %the second click point at (a2,b2)
  %recompute everything, probably in a loop
else %the keyboard was pressed instead
  switch c
  case 'q','Q'
```

```
      %terminate the program
    case '+'
      %increase the number of iterations and recompute
    case '-'
      %decrease the number of iterations
    ...
    ...
    end
  end
```

Some useful options for the user interface are:

- increase/decrease the number of iterations (it was 100 in the code samples given above),
- increase/decrease the resolution (m_x and m_y, that are the number of grid points in each dimension),
- zoom out a given factor,
- undo the last zoom operation (this requires storing all the selected regions used before into a matrix).

There is a number of optimizations that can speed up the program, but the combination of vectorization and boolean indexing achieves a reasonable computation time for m_x, m_y, k of the order of a few hundreds.

When modifying program parameters like the resolution or the number of iterations, always make sure that the new values are reasonable (e.g., the number of iterations or the number of points in any dimension must not be negative or zero). Also the undo operation must be limited in order to not trying to undo an undone zoom.

Another important point when updating the number of iterations or the resolution is that the meaningful way to increase or decrease the parameter is by multiplication, and not by adding a fixed constant. For instance, increasing the number of iterations from 100 to 150 is meaningful, whule if you change from 1000 to 1050 iterations you will almost not notice the difference. The same can be applied to the zooming-out operation.

You can also add an optional legend with the available interaction commands, and also some printable information about the region selected (in sake of reproducibility). In the last case, you would want to use a high precision numeric representation, as in

```
Mandelbrot fractal at (0.435176852,0.341773243)
Width = 2.5e-08; Height = 2.5e-08
Iterations = 1400; Points = 840x840
```

Try to understand the tutorial step-by-step, read the help information of every MATLAB function mentioned in the tutorial, and incrementally add the functionalities that you want to your program. Just as an orientation, a fully functional MATLAB program implementing all the above functionalities would take around 80 lines of code (excluding comments).