Problem set 2a: M-file Programming

Handed out: Friday, October 4, 2024
Due: 23:55pm, Thursday, October 10, 2024

As you did with Unit 1, you must hand in two files:

- One file, named `your_name_E2a.pdf`, with the solutions to the exercises in this set, following the template available in the virtual campus.
- The second file, named `your_name_E2a.zip`, must contain all the MATLAB code you used to solve the exercises.

Please, make sure that the code in the pdf file is **exactly the same** as in the zip file. Remember that all the MATLAB code is supposed to be **entirely yours**. Read the Course Information document for more details on this subject.
In those exercises where a function is implemented, a "help" section must be included in the code, so that when the Matlab command **`help`** is called with the name of that function proper information is provided.

Comment: In this unit it is assumed that you already know how to operate vector and matrix variables. The goal in this exercises is to implement the requested task as efficiently as possible, considering that efficiency is mostly related to execution time. In general, Matlab programs should be written avoiding as much as possible the use of loops, resorting to vectorization. As a rule of thumb, the shorter the Matlab program file is the faster it will execute.

Exercise 1. Generation of random points.

Produce an m-file named **`randpoints.m`** containing a function **`randpoints`** that returns a matrix with some random points. More precisely, the function must support, at least, the following functionalities:

```
P = randpoints(N);
P = randpoints(N,'uniform');
P = randpoints(N,'uniform',[A1,B1],[A2,B2]);
P = randpoints(N,'gaussian');
P = randpoints(N,'gaussian',[C1,C2],[Std1,Std2]);
```

In all calls **`P`** is a $2 \times N$ matrix. The first row contains the x-coordinates of the **`N`** random points, and the second row contains the corresponding y-coordinates. In the first form, with only one input argument, the random points are generated with **`rand(2,N)`**, that is, all coordinates are uniformly distributed numbers in the interval $[0, 1]$. The second input argument, if provided, must be a character string with the type of random distribution used to generate the coordinates of the points. The remaining input arguments, if provided, specify the parameters to be used in the generation of the coordinates.

The second form does the same as the first one: the coordinates are generated with **`rand(2,N)`**.

The third form produces x-coordinates uniformly distributed in the interval $[A1, B1]$, and y-coordinates in the interval $[A2, B2]$. You can use **`rand(1,N)*(B1-A1)+A1`** to produce the right distribution of the x-coordinates, and **`rand(1,N)*(B2-A2)+A2`** for the y-coordinates. Wrong or incomplete specifications (e.g., $A1 \geq B1$ or $A2 \geq B2$ or $[A2, B2]$ not specified) must cause an error. Errors can be handled with the **`error`** function, as in

```
error('A1 < B1 required in uniform mode');
```

Do not use non-informative error messages like in the following examples (typically produced by lazy programmers)

```
error('An error occurred');
error('Wrong parameters');
```

Give the most precise and helpful information you can about the error (as you would want to see when you are the user of the function).

Similarly, the last two forms use the gaussian random number generator **`randn`** instead of **`rand`**. This produces a normal distribution with zero mean and a standard deviation of 1. If the extra input arguents are provided, then **`C1`** and **`Std1`** are respectively used as the mean and standard deviation for the x-coordinates, and **`C2`** and **`Std2`** for the y-coordinates. The generating code is now **`randn(1,N)*Std1+C1`** for the x-coordinates, and a similar expression will be used for the y-coordinates. You must also deal with the possible errors (e.g., incomplete specifications or negative deviations).

The syntax of the function declaration **should** be

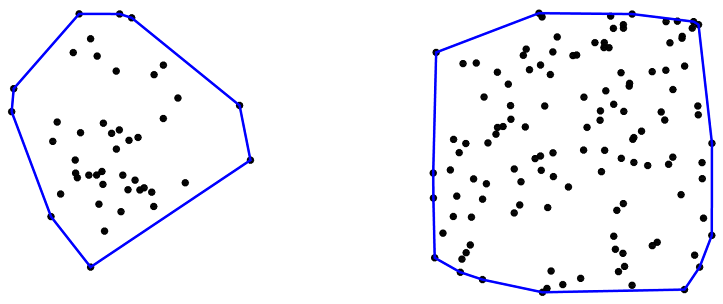
```
function P=randpoints(N,Distr,U,V)
```

You can use the value of the predefined variable **`nargin`** to know the actual number of input arguments provided in the function call, and then decide the way to generate the points. You can also use the **`switch`** / **`case`** construction to deal with the different values of the second argument **`Distr`**.

There is an alternative way to implement a function with different number and meaning of the input arguments, with the **`varargs`** cell-array, but we do not include the explanation here.

Your code will be submitted to automatic testing for help information, error handling and output consistency for the different functionalities described above. Use the **`help`** command with well-known MATLAB commands to see typical examples of how help information is given. Again, give helpful help information. It is very useful both for users wanting to use your code, or even for yourself when trying to reuse functions that you have written a long time ago.

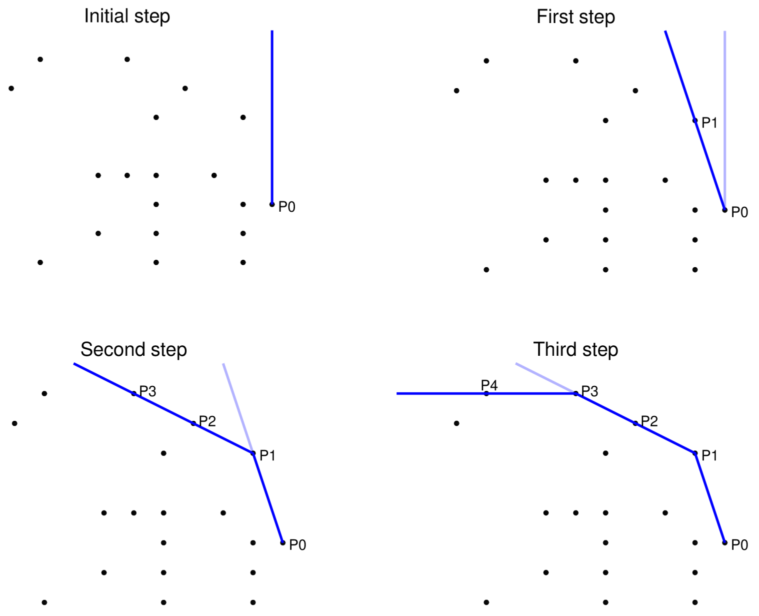
Exercise 2. Convex Hull. With the N random points generated in the previous exercise (e.g., with **`P = randpoints(N)`**) you have to compute the so-called Convex Hull, that is the smallest convex polygon that includes all the provided points in its interior. Here, 'convex' means that the line segment between any two points of the polygon is completely contained in it.



The vertices of the convex hull are always some of the points provided in the input. Hence, a natural way to give the result is as a vector of indices of the corresponding points. For instance, if the convex hull is the polygon defined by the first 4 points and the last one in the set, then the result will be the vector `H=[1,2,3,4,N,1]`. Observe that we "close" the polygon by repeating the first vertex in the last position. To plot the result, you can just recover the x-cordinates and the y-coordinates of the vertices and call the function **`plot`**, like in

```
    plot(P(1,H),P(2,H),'b');
```

A practical way to show the convex hull of a set of points in the plane is to put a pole at every point in the set and try to enclose them with a tight rubber band. Another way to think on the convex hull is the result of winding a straight wire around the poles, and this way of thinking inspires one of the most intuitive algorithms (but not the most efficient one). Observe that in this last view, the wire always leaves the set of points on the same side (e.g., on its left, if we wind the wire counterclockwise).



The algorithm: We start form a point that necessarily is in the convex hull. For instance, we can take the point with the maximal x-coordinate (i.e., the rightmost point). In case there are more than one such points, we take from those points the one with the maximal y-coordinate. Let us call it P_0. You could use the function **`max`** with the syntax **`[Vmax,Imax] = max(V)`** to retrieve the index **`Imax`** at which a vector **`V`** reaches its maximum value **`Vmax`**, but this only gives you one of the possibly many indexes where the value is reached (there could be more than one point with a maximal x-coordinate). In order to retrieve the indexes of all points in which the maximum value in **`V`** is achieved you would need to use **`Imax = find(V==max(V))`**. If you apply this to the vector of the x-coordinates **`X = P(1,:)`**, then you find the indexes **`Imax`** of the rightmost points in **`P`**. Among this indexes you can now find the one corresponding to the maximal y-coordinate with **`[~,J] = max(P(2,Imax))`**. Notice that **`J`** contains the index to the vector of indexes **`Imax`**, and therefore the index of the point P_0 in **`P`** is **`I = Imax(J)`**. You can now initialize the convex hull with the computed index **`H = I`**.

In the second step of the algorithm, the next point in the convex hull is the point P_1 in the provided set with a minimal angle between the vectors $P_0P_1 = P_1 - P_0$ and $(0,1)$, measured counterclockwise. Observe that this angle is in the interval $(0,\pi]$, because the set of points is always on the left of the “wire”, and the wire starts facing towards the direction given by the vector $(0,1)$ (the positive y-axis).

You can use the function **`atan(Y,X)`** that returns the polar angle of the vector (X,Y) in the range $[-\pi,\pi]$, where the signs of X and Y are used to recover the right quadrant in which the vector is located. The polar angle is measured counterclockwise starting from the positive x-axis. In the case of the second step of the algorithm you would want to compute the difference of the polar angles of the two vectors $P_1 - P_0$ and $(0,1)$ (this second polar angle is clearly $\pi/2$). The result of this difference between the two angles can be negative. This means that you would need to normalize the differences to the interval $[0,2\pi)$, that can be done with **`DA = mod(DA,2*pi)`**, where **`DA`** is the difference of the angles.

As you need to compute the angles for all possible points in the set **`P`**, **you must use vectorization instead of a ‘for’ loop**. To do this, you can use the vector **`X = P(1,:)`** that contains all the x-coordinates of the points, and the vector **`Y = P(2,:)`** with the y-coordinates. Then, you can subtract to the vectors the coordinates of P_0 and use a single call to the function **`atan2`** to retrieve all the polar angles at once. Now, you can subtract the polar angle of the vector $(0,1)$ and use the function **`mod`** to normalize the result to the interval $[0,2\pi)$.

Notice that the point P_0 is also included in the vector, and this results in a call to **`atan(0,0)=0`**. Actually, the angle computed for P_0 is nonsense, and you must patch it with some dummy value, that has no effect in further computations. For instance, you can set it to 2π with **`DA(I) = 2*pi`**, where **`I`** is the index of the point P_0 computed in the previous step. Now, you can directly call **`DAmin = min(DA)`** and **`Imin = find(DA==DAmin)`** to retrieve all the points with minimal angle (again, it could be more than one point with the same minimal angle).

We will add all the points with the minimal angle to the convex hull, but they must be sorted before from the nearest to the farthest one. To do that, you can use the **`sort`** function with two output arguments. The call **`[S,Perm] = sort(V);`** produces the sorted vector **`S`** with the same entries of **`V`** sorted in ascending order, and the permutation **`Perm`** used in the sorting process, that is such that **`V(Perm)`** gives **`S`**. You can apply this to sort the vector of the squared distances from the points P with minimal angle to the point P_0.

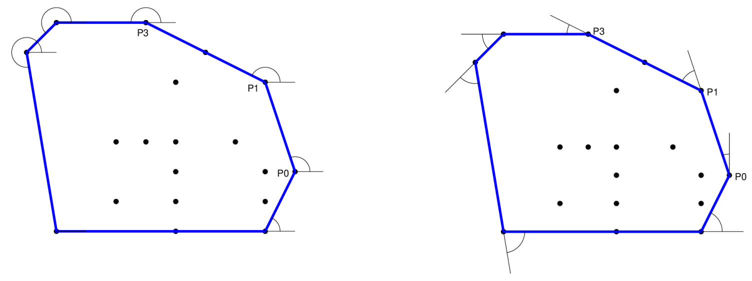
```
Xmin = P(1,Imin); Ymin = P(2,Imin);
D2 = (Xmin-X0).^2+(Ymin-Y0).^2;
[~,Perm] = sort(D2);
Imin = Imin(Perm);
```

The last line uses the permutation given by **`sort`** to put the indexes of the points with minimal angle in the right order to be added to the convex hull.

The following steps are identical to the second one, described above, but now P_0 is replaced by the last point added to the convex hull and the vector $(0, 1)$ is replaced by the last segment added to the convex hull, or better, the reference angle ($\pi/2$ in the first step) is incremented by the angle **`DAmin`** computed in the previous step.

The algorithm ends when the convex hull is closed, that is, when the last point added to the convex hull is the starting point P_0.

The following figure shows the different angles involved in the algorithm (left: polar angles, right: **`DAmin`** angles).



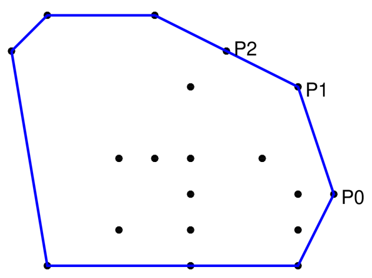
What to do: You must implement the described algorithm in a m-file named **`convexhull.m`** and function declaration

```
function H = convexhull(P)
```

returning the (row) vector of indices of the points defining the convex hull of the points in **`P`**. We assume that no point is repeated in set defined by **`P`**, and it contains at least three points.

Test your function with several sets of random points (generated for instance the function of the previous exercise), and test also the following special set, where the algorithm will find multiple points with the same minimal angle in some of the iterations.

```
P = [1,3,7,5,5,6,8,3,8,4,8,0,5,4,1,9,8,5,5;
     7,1,3,5,0,6,2,3,0,3,1,6,2,7,0,2,5,3,1]
```



Your code must output the vector

```
H = [16,17,6,14,1,12,15,5,9,16]
```

MATLAB has a function **`convhull`** implementing a faster algorithm, and you can only use it for testing your own program.

Do not forget providing help information in the m-file, and consider also generating error messages when the input is badly formed.

Again, your code will be submitted to automatic testing tools, including tests with cleverly selected inputs, random inputs, inputs with repeated points, missing input parameters, etc.