Problem set 1a: MATLAB Fundamentals and Graphics

Handed out: Thursday, September 12, 2024
Due: 23:55pm, Thursday, September 19, 2024

You must hand in two files:

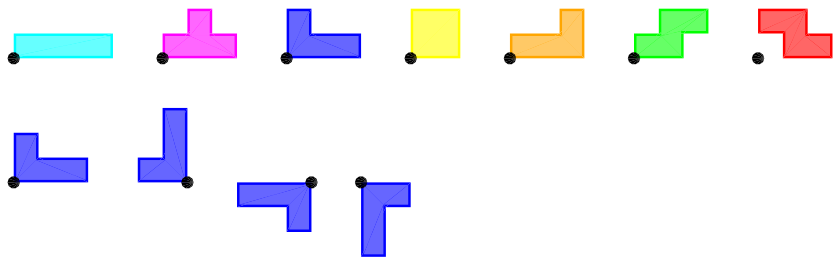
- One file, named `your_name_E1a.pdf`, with the solutions to the exercises in this set, following the template available in the virtual campus. For every exercise, you must explain how you solved it, the necessary MATLAB code, the results obtained (run transcripts or plots), and some final comments (if any).
- The second file, named `your_name_E1a.zip`, must contain all the MATLAB code you used to solve the exercises, organized in one or more text (`.m`) files, to allow the teacher to check your solution.

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**. 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, available in the virtual campus, for more details on this subject.

Exercise 1. Polygons and colours.

Generate the game board shown in the figure below. To do that, first draw the empty board. You can draw a rectangle of the same size of the whole board, and then draw on top of it the small squares, both using **`fill`**. This means creating two vectors **`Xb`** and **`Yb`** with the coordinates of the vertices of each rectangle, and then calling the **`fill`** function, specifying also the desired filling color. It is possible (but not easy) to draw all the small squares in the board with a single call to **`fill`**. Indeed, **`Xb`** and **`Yb`** must now be matrices instead of vectors. Each row in the matrix is interpreted as a separate polygon to be filled. This method is faster because it avoids the use of **`for`** loops, which typically slow down the program.

Once the board is drawn, you can position the pieces on it. To do it, first create the coordinate vectors **`X1`**, **`Y1`** ...**`X7`**, **`Y7`** defining the seven shapes shown in the figure below. Notice that the pieces need to be properly rotated and positioned before drawing them on the board. You can call the function **`fill`** with the desired color specification to draw every piece. For instance, a 90 degree counterclockwise rotation is obtained with **`fill(-Y3,X3,color3)`**, and a you can add some constants to the coordinate vectors to apply a translation to the piece, as in **`fill(X1+3,Y1+10,color1)`**. In the figure you can see the selected piece centers for the rotations, and also an example of the different rotations applied to a particular (blue) piece.



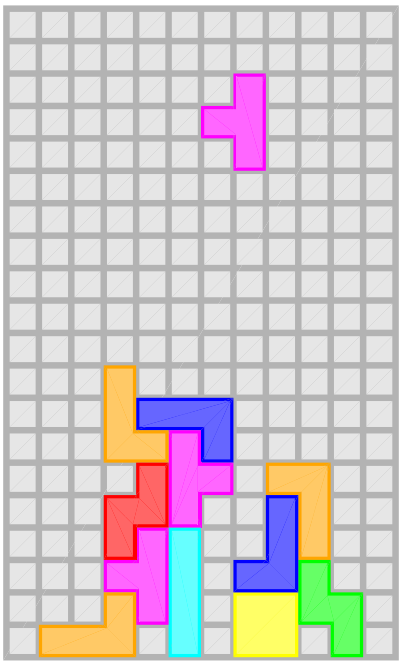
Using cell-arrays to store the seven pieces together into a single object would improve the code compactness and readability: you can write **`X{I}`** for any value of **`I`** instead of explicitly using the seven variable names **`X1`** ...**`X7`**. Using 2-dimensional arrays is also possible, but notice that the pieces have different number of corners, and then the coordinate vectors have different lengths, and you would need to add some padding to the shorter. You can also pack the seven colors into a single object (vector or cell-array).

Your MATLAB code will receive as input a matrix **`M`** describing the pieces to be drawn. Namely, each row contains four numbers: the shape identifier (e.g., a number between 1 and 7), the x-displacement, the y-displacement of the piece and the number of 90 degree counterclockwise rotations needed (e.g., between 0 and 3). The code first draws the board, and then for each row of **`M`** it takes the **`Xi`**, **`Yi`** vectors corresponding to the specified shape, applies the required rotations and translations, and draws the piece on the board.

Do not worry too much with the decorations, but beware with the axis properties (the cells must be squares and not rectangles), and also the thickness of the board cells separations. The main goal of the exercise is the use of **`fill`**, the use of vectors, matrices and cell-arrays and the organization of the code. In some cases **`switch`** constructions are preferred to long chained **`if`**/**`elseif`** ones.

The example figure below corresponds to the function call

```
tetris([1,6,0,1;2,5,1,1;4,7,0,0;3,7,8,2;2,5,7,3;5,10,3,1;
  5,3,9,3;2,8,15,1;7,5,3,1;6,11,0,1;5,1,0,0;3,9,2,1]);
```



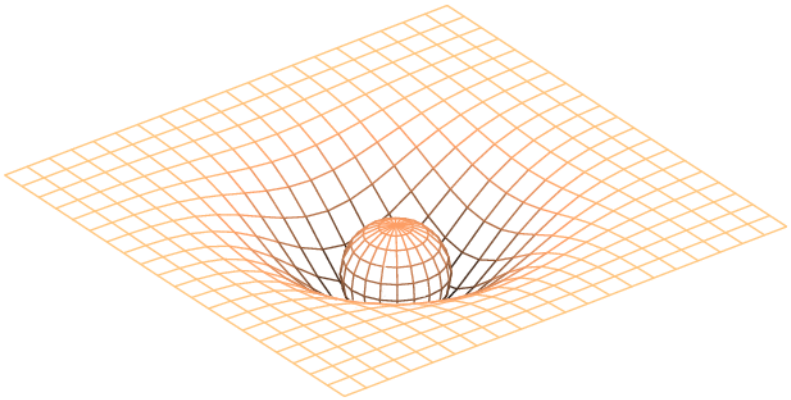
Exercise 2. Distorted plane.

The parametric equations of a sphere with radius R, centered at the origin, are:

$$\begin{cases} x = R\sin\psi\cos\theta \\ y = R\sin\psi\sin\theta \\ z = R\cos\psi \end{cases}$$

where $0 \leq \psi \leq \pi$ and $0 \leq \theta \leq 2\pi$.

Consider an elastic horizontal plane at $z = 0$ that is distorted because of the weight of a small sphere lying on it, as shown in the figure below:



Use **meshgrid**, **mesh** and **hold** to build the figure, assuming (unrealistically) that the distorted plane can be modelled by the function $f(x, y) = -Ke^{-(x^2+y^2)/\alpha}$, where $\alpha = 5R^2$, and K is the depth of the distortion (which depends on the weight of the sphere).

Choose adequate values for R and K, and notice that the sphere and the distorted plane are tangent in $(x, y) = (0, 0)$. Thus you need to adjust the z component of the sphere. Your code must work for any values of R and $K \leq \frac{5R}{2}$ (for larger values of K part of the sphere will be below the distorted plane).

Firstly plot the distorted plane with **mesh** using the given distortion model, in the usual way a two-variable function is plotted. Then add the plot of the sphere using **hold on** to preserve the previous plot.

In order to plot the sphere you need to create two matrices **Theta** and **Psi** with **meshgrid**, and then compute the matrices **X**, **Y** and **Z**, using the equations given above, needed in the call to **mesh(X,Y,Z)**. You are not allowed to use specialized functions like **sphere**, since the goal of the exercise is learning the use of **meshgrid**, **mesh** and the use of parametric equations.

You probably want to use **axis off** and **axis equal** to produce the picture.

Exercise 3. Sonine polynomials (a.k.a. generalized Laguerre polynomials).

Given a parameter α, Sonine polynomials are recursively defined by $S_0(t) = 1$, $S_1(t) = 1+\alpha-t$ and

$$S_n(t) = \frac{2n-1+\alpha-t}{n} S_{n-1}(t) - \frac{n-1+\alpha}{n} S_{n-2}(t).$$

We will take a fixed parameter value $\alpha = 1.5$.

(1) Compute $S_2(t)$, $S_3(t)$, $S_4(t)$ and $S_5(t)$ using MATLAB (i.e., do not compute them by hand or with a symbolic tool). In this exercise, you are asked to compute the vectors of coefficients of the polynomials (not only their plots!). Use **conv** for polynomial multiplication, and normal vector operations for addition and substraction of polynomials. In the last case,

you need first to make sure that all the vectors have the same length. In particular, you need to append some zeros to the left of the polynomial with less degree.

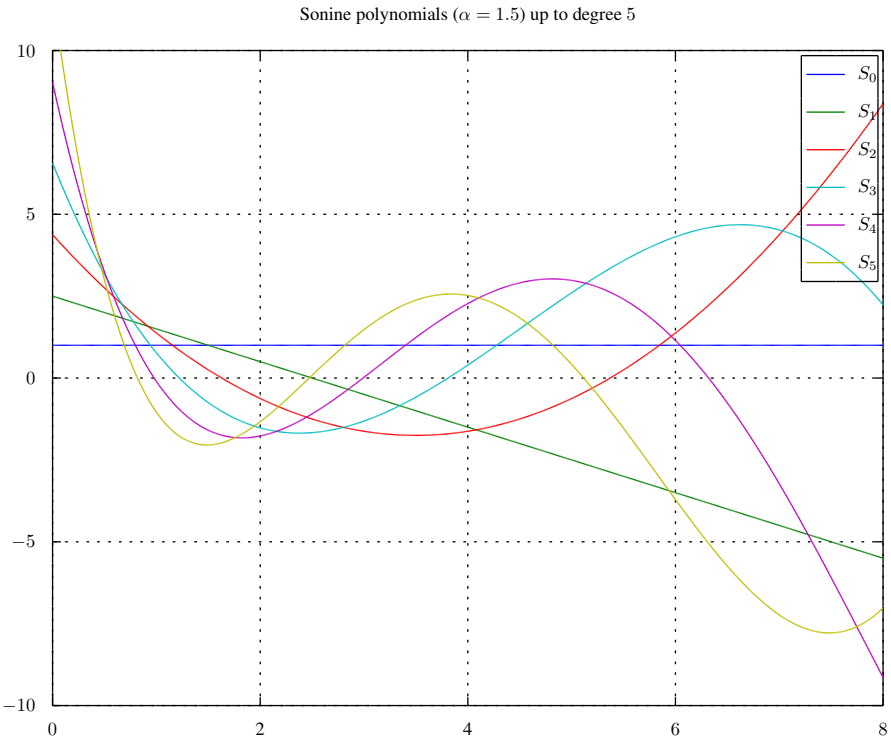
Multiplication by t can simply be implemented by appending a zero on the right of the vector of polynomial coefficients.

Try to generalize the code for arbitrary values of n and α.

Print the computed coefficients. For instance, for S_2 and S_3 you must obtain:

```
S2:   0.50000 -3.50000 4.37500
S3:   -0.16667 2.25000 -7.87500 6.56250
```

(2) Plot all six polynomials ($S_i(t)$, $i = 0, \ldots, 5$) in the same axis in the interval $[0, 8]$ using **`polyval`** and **`plot`**, as shown in the figure. You probably want to use **`axis`** to limit the vertical axis to the interval $[-10, 10]$, and **`grid on`** can also help to achieve the desired effects.



Generalizing the graphics for an arbitrary value of n is not so simple, because of the legend and the title.