PRA1004 Scientific Computing 2013

Frans Oliehoek
<frans.oliehoek@maastrichtuniversity.nl>

Lab Assignments Week 4

Overview Lab 4

In this week's lab, we will cover the following topics:

- Some more useful Matlab functions.
- Inner products and matrix products.
- Numerical (and symbolical) methods for differentiation and integration.

The topics are grouped in 5 sections. The time I expect a section to take is also indicated. I would appreciate it if you can let me know if my estimate is way off.

Finally, this week there will be no hand-in assignment.

1 Some more useful Matlab functions: min, max, etc (30mins)

Here we will introduce a number of functions, such as min, max, etc. that you will end up using very frequently when working with Matlab.

We start with min and max. Try to execute the following command:

- `X = [7, 8, 6, 9]`
- `min(X)`
- `A = [11,5;4,51;12,7]`
- `min(A)`

Clearly, the first command returns the minimum (also try `max`), for a matrix, however, you see that the behavior is a bit different.

- what does it do exactly?
- how can you get the single maximum element from a matrix?

In many cases, it is also useful to be able to find the index of the minimal/maximal element, this can be done using

- `[m,index] = min(X)`
- `[m,index] = max(X)`
- `A = [11,5;4,51;12,7]`
- `[m,index]=min(A)`

Another function that you will definitely need to know about in order to find the row and column indices of some elements is `find`. In particular, `find` will return to you the indices of all the elements of a vector or matrix that are non-zero. This is very useful in combination with 'boolean operators' such as < (less than), == (equals), and >= (greater or equal than). For instance try:

- `inches = [0:5:50]`
  `cm = inches * 2.54`
  `table = [inches', cm']find(table(:,2)<50)`
  (to understand what is going on, also look what `table(:,2)<50` gives you!)

By default, the indices returned by `find` are *linear indices*. That is, the indices you get when first counting all the elements of the first column, then continuing counting in the second column etc. To understand what this means, try:

- `table<20`
  `lin_indices = find(table<20)`

Linear indices can be very convenient, since you can use them to easily extract the corresponding elements from your matrix:

- `table(lin_indices)`

In many cases, however, you want to know the row and column indices of the entries. that can be done by asking for two output arguments:

- `[row, col] = find(table<20)`

Now try to combine some of these commands:

- Generate a random 70x70 matrix, find the maximum element as well as its row and column index.

Some other useful functions that you should know about are `sum` and `cumsum` (and similarly `prod` and `cumprod`)

- `X = [1:4]`
- `sum(X)`
- `cumsum(X)`

and `mean` (and also `median`)

- `A = reshape(1:6, 2, 3)`
- `R = rand(10,5)`
- `m = mean(R)`

Another convenient function is `round`, which rounds numbers to the nearest integer. E.g.:

- `X = [7.2, 8.5, 6.1, 9.9]`
- `round(X)`

Try using round:

- Compute a 3,4 matrix with random integers between 0, 42. Use `round`.

Practice more:

- Ch.3 ex. 2,4,5, 8–10

	calories	fat(g)	Calcium(mg)	Vitamin C(mg)
banana	200	0.7	11.3	19.6
slice of bread	68.9	1.1	26.8	0
cup of milk	149	8.4	246	2.2
apple	65	0.2	7.5	5.7

Table 1: Nutrition in food items.

2 Inner Products (30 mins)

The 'inner product', also called 'dot product' is a particular way of multiplying the elements of two vectors, and then adding them up. In particular, if we have two vector $\langle 1, 2, 3\rangle$ and $\langle 4, 5, 6\rangle$, their inner product is:

$$1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$$

Expressed in formulas, the inner product of two vectors $a = \langle a_1, a_2, \ldots, a_n\rangle$, $b = \langle b_1, b_2, \ldots, b_n\rangle$ is

$$dot(a, b) = \sum_{i=1}^{n} a_i b_i.$$

This simply means multiply the corresponding elements of the vectors, and then add everything up.

While inner products seem strange at first, they are in fact very convenient. For instance, we can use them to easily compute the amount of calories in our lunch. The number of calories in certain lunch foods are shown in Table 1.

- Store the number of calories for the various items in a row vector called `calories`. Use the same ordering as in the table.
- Create another row vector, `lunch`, that specifies the amount of each item that you think is appropriate for a big lunch. E.g., 1 banana, 4 slices of bread, 1 cup of milk and 2 apples.
- Now we can easily compute the number of calories using `lunch_cals = dot(calories, lunch)`.

Similar computations can be used for many things. For instance, to perform similar operations on the other nutrition columns, but also to compute the weight of a collection of items (e.g., to compute the mass of a spacecraft – Example 6.1 in the book Etter [2011]).

- Do Ch. 6, ex. 1,2.

As said, similar computation can be used for the other listed nutrients. In fact, by using **vector-matrix multiplication**, we can compute all of them at the same time. If we have a $k \times n$ matrix A, and a $n \times 1$ (i.e., column) vector b, their product is written as

$$Ab = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{k1} & & & a_{kn} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix} \tag{1}$$

the result, a column vector c, has as its entries different inner products. Let us write r_i^A for the i-th row of matrix A. Then

$$c_i = dot(r_i^A, b),$$

that is, the i-th entry of c is the inner product between the i-th row of A and b. This means that we can compute several inner products at the same time, as we will now demonstrate.

- Create a matrix, say `nuts`, containing the all the (data) columns from Table 1.
- Create a variable, say `nuts_T`, that contains the transpose of `nuts`.
- Transform `lunch` to a column vector.
- Compute the total amount of nutritions in `lunch` using: `lunch_nuts = nuts_T*lunch`.

The result should be a (column) vector of which the first entry are the number of calories in the lunch, the second entry denotes the amount of fat, etc.

- Notice what is going on: the first entry is the inner product between `lunch`, and what part of matrix `lunch_T`?

The inner product of two vectors is frequently denoted as a special case of matrix. E.g., when a, b are column vectors, we have that

$$dot(a,b) = [a_1, a_2, \ldots, a_n] * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = a^T * b \tag{2}$$

(where T denotes the the transpose). This is also simply written as $a^T b$. Note that this is just a special case of equation (2).

- Define a row vector `a` and column vector `b` and multiply them: `a*b`. Does it match the inner product?
- What happens when you reverse the arguments? (I.e., when you type `b*a`)? In order to fully understand what is going on, you will need to understand matrix-matrix products...

3 Matrix Products (1h)

This section introduces the general matrix product. Let A be a $m \times k$ matrix, and B a $k \times n$ matrix. That is, the number of columns of A is the number of rows of B is k. The result of matrix multiplication is a $m \times n$ matrix C:

$$AB = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1k} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{m1} & & & a_{mk} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ b_{k1} & & & b_{kn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ c_{m1} & & & c_{mn} \end{bmatrix} \tag{3}$$

where $c_{ij} = dot(row_i^A, col_j^B)$.

As you can tell, matrix multiplication requires quite a few multiplications ($m \cdot k \cdot n$ of them, although more efficient methods do exist). Fortunately, we don't have to do them by hand; they are easy to do in Matlab. For instance try the following example:

- ```
  A=reshape(1:6, 2,3)
  B=A'*10
  C=A*B
  ```
- Make sure to verify the result to see that it makes sense.

While matrix multiplication may seem like a very abstract and strange operation at first, it is actually used very frequently in all areas of science. To illustrate this, we will consider an example dealing with populations of trees, adopted from [Allman and Rhodes, 2003].

In this example, there are two types of trees type 1 (a) and type 2 (b). T1 trees live longer than T2 trees: every year 1% of the T1 trees dies, while 5% of the T2 trees die. To make up for that, the T2 trees
  ```

grow faster. This also means that they are more likely to occupy a vacant spot left by a dead tree: 75% of the spots that become available go to T2 trees. Therefore, the number of trees in a next year can be expressed as

$$vacant\_spots_t = 0.01 \cdot a_t + 0.05 \cdot b_t,$$

$$a_{t+1} = 0.99 \cdot a_t + 0.25 \cdot vacant\_spots_t, \tag{4}$$

$$b_{t+1} = 0.95 \cdot b_t + 0.75 \cdot vacant\_spots_t. \tag{5}$$

- Simplify the above expressions to the following form:

$$a_{t+1} = c_{11} \cdot a_t + c_{12} \cdot b_t \tag{6}$$

$$b_{t+1} = c_{21} \cdot a_t + c_{22} \cdot b_t \tag{7}$$

  (so find what are the numbers c_{11}–c_{22}. To do this, first substitute in $vacant\_spots_t$ in equations (4) and (5)).

Now note that the numbers c_{11}–c_{22} can be put in a matrix, such that we can describe the next year's population $x_{t+1} = [a_{t+1}, b_{t+1}]^T$ using a matrix-vector product:

$$x_{t+1} = \begin{bmatrix} a_{t+1} \\ b_{t+1} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} a_t \\ b_t \end{bmatrix} = Cx_t \tag{8}$$

- Define the matrix `C` in Matlab.
- Define the vector of initial populations: `x0=[10;990]`.
- Simulate the evolution for a number of years using (8). (Note, you can 'simulate' what happens by repeatedly typing: `C * ans`, where `ans` contains the value of the previous prediction.)
- For what year will we get the predicted result if we do $C \cdot C \cdot C \cdot x_0$?

While we can investigate the growth for a few iterations in this way, it is difficult to predict what will happen over 100s of years. To be able to still do this, we will exploit the 'associative' property of matrix multiplication: $A(BC) = (AB)C$.

- Verify associativity by checking whether `C(C*x0)` is the same as `(C*C)*x0`.

In Matlab we can use the power symbol (`^`) to take the power of a matrix:

- Compute $C \cdot C$ in Matlab by raising the matrix to the power 2.
- Also compute $C \cdot C \cdot C \cdot C \cdot C$. Verify your result.

Given these results, you can easily compute the population after, say, 150 years by first raising C to the power 150 and then multiplying by x_0.

- Compute the populations at 1, 5, 25, 100, 200, and 1000 years.
- Plot the graphs.

4 Numerical Differentiation (1.5h)

Here we investigate methods for numerical differentiation in Matlab. First we consider on a more abstract level some different approaches to numerical differentiation, and cover how this is done in Matlab. Next, we will use these to determine the acceleration of a rocket, making use of just altitude data.

4.1 Forward, Backward, and Centered Finite Difference

As you know, there are three simple approaches to numerical differentiation: using the formula for Forward, Backward, and Centered Finite Difference:

$$\text{forward:} \qquad f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

$$\text{backward:} \qquad f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

$$\text{centered:} \qquad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Here we will explore these methods. In particular, Matlab offers the function `diff` to compute differences between adjacent vector entries. E.g., try

- `diff([1:5].^2)`

Essentially this function provided the basis for all the above approximations of the derivative: the only thing that it does not take care of is normalizing for the step length. We will demonstrate this for the function $f(x) = x^3 + 2x^2$.

- Specify `Xrange=[-2:0.025:0.5]` and plot the function f over this range.

Now we will try and approximate the derivative of f.

- Open `numdiff.m`.

The file consists of 3 main parts: the first computes the differences using `diff`, the second plots the estimated derivatives, and the third plots the estimated derivative together with the original function. Various parts in the script are not completed (marked by 'TODO'), you will need to complete these parts to make the script work.

Let's start by looking at the first part:

- Notice how the forward and backward differences are given by the same formula! (But also notice what is the difference!)
- Compute `cent_diffs`. Hint: the centered difference is the average of the forward and backwards difference.
- Note that, in order to convert to actual estimations of the derivative, we first need to divide by the step size (`dX`).

Now you should be able to plot the estimates of the derivatives.

- Additionally plot the true derivative of f. Use a thicker line to discriminate from the estimated values.
- Which of the estimates is the better one?

Finally, we want to plot the estimated derivatives together with the original function. For this, we use the function quiver, which can be used to plot vector fields. It takes four arguments: the locations of the arrows (X and Y) and the sizes of them (again X and Y).

- Look at the generated plot: the forward estimation seems to coincide with the function. Why is this?
- Fix the plot such that it better reflects the reality by plotting the function f with a higher resolution (as in the beginning of this assignment.)

4.2 Acceleration of a Rocket

In this assignment, you will apply your knowledge about computing derivatives numerically on data of a rocket flight. (Adopted from Etter [2011], ch. 8. ex. 12–14)

- load the rocket data: `rocketData.m`, and investigate it.

The first column contains the time (in seconds since launch), while the second column contains the altitude.

- What is Δt the difference between time points?

To goal is to create a script that plots the altitude, velocity and acceleration of the rocket in three 'sub plots' by using the `subplot` function.

- Have a look at the documentation of subplot. Also, you can use the following script (in the lab kit as `rocket.m`) as a template:

```
load rocket;
Data deltaT = TODO;
figure(1)
clf;

%plot the altitude
subplot(3,1,1);
Ts = TODO
rocketAlt = TODO
plot(Ts, rocketAlt);

%plot the velocity
subplot(3,1,2);
TODO

%plot the acceleration
subplot(3,1,3);
TODO
```

5 Numerical Integration (1h)

The height of a population of people can be modeled as a bell-curve called *Gaussian, or normal distribution*. See, Figure 1 for an illustration. In particular it expresses the 'probability density' of height h. The formula is

$$p(h) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{h-\mu}{\sigma}\right)^2} \tag{9}$$

where

- h is the height for which we want to know the probability density.
- μ is the mean height (use 180).
- σ is the standard deviation (use 10).

Now, given this model, we can find the probability P that a randomly drawn person has height $h \in (190, 210)$ (or some other range) by taking the integral:

$$P = \int_{190}^{210} p(h)dh.$$

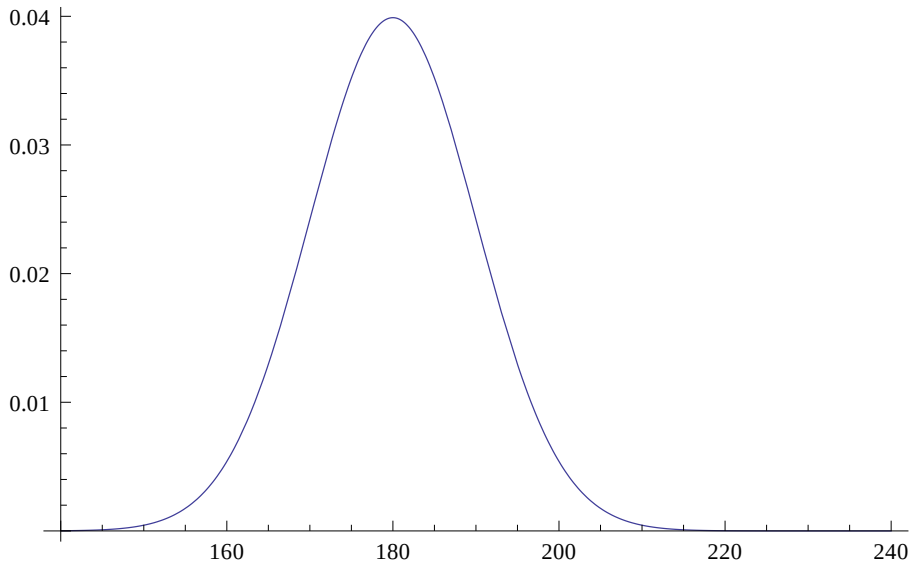

Figure 1: Used model of probability distribution of heights.

In this assignment, we will compute this probability numerically by using the `quad` functions in Matlab. In order to do this, we will need to define a user-defined function `HeightProb(h)` that will return $p(h)$ for each value of h.

- To do this, open up a new `.m` file and call it `HeightProb.m`. In it type:

```
function p = HeightProb(h)
%Helper function that returns the probability density at height h
mu = 180;
sigma = 10;
p = TODO;
return;
```

  - Note that it is important that the file name (`HeightProb.m`) and the function name (`HeightProb`) match!
  - Also note that it is important to realize that the input argument `h` should be allowed to be a vector (the function `quad` that we will use below will give vectors as arguments to this function). Therefore, make sure that you use element-wise operators to transform `h` into the corresponding densities `p`.

- Finish the implementation by replacing 'TODO' by the definition of (9).
- Verify your implementation by recreating the plot of Figure 1. To apply your own formula to some vector of heights, `heights`, you can use the command
  `probs=arrayfun(@HeightProb, heights)`
  next you can plot the `heights` versus the `probs`.

Now that we have defined the function, we can easily perform numerical integration using

- `quad('HeightProb', 190, 210)`
- What is the probability? Does is seem to be a reasonable estimate?

You should now that even when an integral seems very difficult, Mathematica is often able to find a solution for it. That is, when faced with an integral, Mathematica is probably the first thing you want to try out. Let us try to do this for this problem

- Check in Mathematica what the 'closed-form' solution is.
  Some pointers:
  - in Mathematica, all build in functions start with capital letters.
  - functions use square brackets.
  - use `p[h_] = (1/(sigma*Sqrt[2*Pi]))*Exp[...]` to define the function $p(h)$.
  - use `Integrate[p[h], h]` to find the indefinite integral.
- Also find in Mathematica the numerical approximation.
  1. use `Integrate[p[h], {h,190,210}]` to find the definite integral.
  2. use `N[...]` to find a numerical evaluation.

References

Elizabeth S. Allman and John A. Rhodes. *Mathematical Models in Biology: An Introduction*. Cambridge University Press, 2003.

Delores .M. Etter. *Introduction to Matlab*. Pearson Education, second edition, 2011. ISBN 978-0-13-217065-9.