MA10265 Semester 1 Coursework
Supplementary assessment

Worth 25 points or 25% of the total MA10265 mark (over S1 & S2)

Very Important Warning:
You should not discuss your work for this assignment with anyone else. The work which you hand in must be your own. Under no circumstances should you exchange solutions with other students. After you hand your work in, you could be asked by an examiner to explain it verbally as part of the marking process. Any evidence of cheating will entail disciplinary procedures and the risk of losing all points on the assignment for all parties involved. See `http://www.bath.ac.uk/quality/documents/QA53.pdf` **for further details.**

1 Background - Horner's rule

1.1 Simple and sparse representation of polynomials

In the lecture you came across Horner's rule for evaluating a polynomial $p(x)$ of degree n

$$p(x) = \sum_{j=0}^{n} A_j \cdot x^j \tag{1}$$

as

$$p(x) = A_0 + (A_1 + (A_2 + \cdots + (A_{n-2} + (A_{n-1} + A_n \cdot x) \cdot x) \cdot x \cdots) \cdot x) \cdot x. \tag{2}$$

Here and in the following we always assume that both x and the coefficients A_j are either integer numbers or square matrices of size $N \times N$ with integer entries. Note that in general, matrix multiplication is non-commutative, $x^j \cdot A_j \neq A_j \cdot x^j$ for $N > 1$.

Although Eq. (2) is the most general way of evaluating the polynomial $p(x)$, it is very inefficient if most of the coefficients are zero. A polynomial with this property is said to be *sparse*. Instead of the list of coefficients a_j, any polynomial of the form in Eq. (1) can also be specified by two arrays:

- The ordered list $i = [i_1, i_2, \ldots, i_k]$ of the powers of x which have a non-vanishing coefficient. You can assume that $0 \leq i_1 < i_2 < \cdots < i_k$.
- The list $B = [B_1, B_2, \ldots, B_k]$ of non-zero coefficients for those powers. Note that $B_j = A_{i_j}$.

Given those two lists i and B, the polynomial can be written as $p(x) = \sum_{j=1}^{k} B_j \cdot x^{i_j}$ or

$$p(x) = \left(B_1 + \left(B_2 + \cdots + \left(B_{k-2} + \left(B_{k-1} + B_k \cdot x^{i_k - i_{k-1}}\right) \cdot x^{i_{k-1} - i_{k-2}}\right) \cdot x^{i_{k-2} - i_{k-3}} \cdots \right) \cdot x^{i_2 - i_1}\right) \cdot x^{i_1}. \tag{3}$$

If the polynomial is sparse, i.e. $k \ll n$, evaluating it via the sparse Horner's rule in Eq. (3) is more efficient than using the standard form in Eq. (2) since the sparse Horner's rule requires less storage and fewer operations. More specifically, evaluating Eq. (3) requires k multiplications and $k-1$ additions, plus the multiplications that are necessary to compute the matrix powers x^{i_1}, $x^{i_2 - i_1}$, $x^{i_3 - i_2}$,..., $x^{i_k - i_{k-1}}$.

Example. To illustrate the evaluation of a polynomial with both Horner's rules, consider the polynomial

$$q(x) = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} + \begin{pmatrix} 8 & -2 \\ 5 & 7 \end{pmatrix} \cdot x^2 + \begin{pmatrix} -9 & 0 \\ 3 & 1 \end{pmatrix} \cdot x^4, \tag{4}$$

where x is either an integer or a 2×2 matrix with integer entries. This can be written either in the form in Eq. (2) with

$$A = \left[\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 8 & -2 \\ 5 & 7 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} -9 & 0 \\ 3 & 1 \end{pmatrix} \right]$$

as

$$q(x) = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} + \left(\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} + \left(\begin{pmatrix} 8 & -2 \\ 5 & 7 \end{pmatrix} + \left(\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} -9 & 0 \\ 3 & 1 \end{pmatrix} \cdot x \right) \cdot x \right) \cdot x \right) \cdot x$$

or in the sparse form in Eq. (3) with

$$i = [0, 2, 4], \qquad B = \left[\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}, \begin{pmatrix} 8 & -2 \\ 5 & 7 \end{pmatrix}, \begin{pmatrix} -9 & 0 \\ 3 & 1 \end{pmatrix} \right]$$

as

$$q(x) = \left(\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} + \left(\begin{pmatrix} 8 & -2 \\ 5 & 7 \end{pmatrix} + \begin{pmatrix} -9 & 0 \\ 3 & 1 \end{pmatrix} \cdot x^{4-2} \right) \cdot x^{2-0} \right) \cdot x^0.$$

1.2 Further generalisation

The evaluation of sparse polynomials with Eq. (3) can be further accelerated by calculating the *greatest common divisor*[1] $g = g(i_1, i_2, \ldots, i_k)$ of the numbers $i_1, i_2, \ldots, i_k$ and pre-calculating x^g. For this write $i_j = g \cdot i'_j$ for $j = 1, \ldots, k$ and define $y = x^g$. Then the polynomial $p(x)$ in Eq. (3) can be written as

$$p(x) = \Big(B_1 + \Big(B_2 + \cdots + \Big(B_{k-2} + \Big(B_{k-1} + B_k \cdot y^{i'_k - i'_{k-1}}\Big) \cdot y^{i'_{k-1} - i'_{k-2}}\Big) \cdot y^{i'_3 - i'_2} \cdots \Big) \cdot y^{i'_2 - i'_1}\Big) \cdot y^{i'_1}. \tag{5}$$

For the example in Eq. (4) we would have, since the greatest common divisor of the set $\{0, 2, 4\}$ is 2:

$$q(x) = \left(\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} + \left(\begin{pmatrix} 8 & -2 \\ 5 & 7 \end{pmatrix} + \begin{pmatrix} -9 & 0 \\ 3 & 1 \end{pmatrix} \cdot y^{2-1} \right) \cdot y^{1-0} \right) \cdot y^0. \qquad \text{with } y = x^2.$$

1.2.1 Euclid's algorithm

The greatest common divisor $g(r, s)$ of two non-negative integer numbers r and s can be calculated with Euclid's algorithm. Pseudocode for a recursive implementation is given in Algorithm 1.

Algorithm 1 Euclid(r, s): calculates the greatest common divisor $g(r, s)$ of two integers $r, s \geq 0$.

```
1: if $s > r$ then
2:     return Euclid($s$,$r$)
3: else if $s == 0$ then
4:     return $r$
5: else
6:     return Euclid($s$,$r - s$)
7: end if
```

Based on this, the greatest common divisor $g(r_1, r_2, \ldots, r_m)$ of m non-negative integers can be calculated by repeatedly applying Euclid's algorithm as shown in Algorithm 2.

Algorithm 2 PolyEuclid($[r_1, r_2, \ldots, r_m]$): calculates the greatest common divisor $g(r_1, r_2, \ldots, r_m)$ of m integer numbers $r_1, r_2, \ldots, r_m \geq 0$.

```
1: $g \hookleftarrow r_1$
2: for $j = 2, \ldots, m$ do
3:     $g \hookleftarrow$ Euclid($g$,$r_j$)
4: end for
5: return $g$
```

[1] More specifically, the *greatest common divisor* $g = g(r_1, r_2, \ldots, r_k)$ of m non-negative integer numbers $r_1, r_2, \ldots, r_m \in \mathbb{N}_0$ is defined as the largest non-negative integer g such that $r_j = g \cdot r'_j$ with $r'_j \in \mathbb{N}_0$ for all $j = 1, \ldots, m$. In the special case that $r_1 = r_2 = \cdots = r_m = 0$ the greatest common divisor is defined to be $g(0, 0, \ldots, 0) = 0$.

2 Assignment

In this coursework you will write Matlab functions for evaluating a polynomial using Horner's scheme. Note that

- you should **not** use the built-in Matlab hat-operator "`^`" or the function `mpower()` for raising a number or a matrix to the n-th power, e.g. `x^n` to calculate x^n. You will lose some points if you use the hat-operator or `mpower()`, but you might choose to do so if you can not complete the assignment otherwise.
- you should **not** use any built-in Matlab functions for calculating the greatest common divisor. You will lose some points if you use such functions, but you might choose to do so if you can not complete the assignment otherwise.
- you can use **any** code you wrote in the Tickables, any code which has been made available as a model solution on moodle and all code which has been presented in the lectures.
- You must **not** copy code from any other sources, e.g. the internet. Doing so will be considered cheating.

2.1 Submission

Please read the following instructions very carefully; it is your responsibility to submit your code correctly on moodle.

All code has to be stored in one directory called `coursework_USERNAME` where `USERNAME` is your username, e.g. `coursework_em459` in my case. This directory must then be zipped as `coursework_USERNAME.zip` (this would be `coursework_em459.zip` in my case). Submit this single zip file on moodle (the submission point is named "Coursework submission point [supplementary assessment]" and can be found in the section "Coursework" of the MA10265_S1 moodle page). Please see the handout on "Getting Started with Matlab"' for detailled instructions.

You must use exactly the names of the functions and files given below. The same applies to the ordering of parameters.

Check that your code works by using the tests (see Appendix A). You will lose marks if your code does not pass those and similar tests, for example because you used incorrect function- or filenames.

Comment your code. You will lose marks for code that is poorly commented or badly formatted.

When you completed the entire assignment, the directory should contain the following four files, and all additional files which are necessary to run and test your code:

- HornerSimple.m
- HornerSparse.m
- Euclid.m
- PolyEuclid.m
- HornerEuclid.m

Please include **all** files which are necessary to run the code in this directory, but do **not** include any unused files.

Part I

Write a recursive function `HornerSimple` which evaluates a polynomial $p(x)$ using the Horner scheme given in Eq. (2). The function should return the value of the polynomial for a given coefficient list A and for a given x. It should be passed the following two arguments **in exactly this order**, i.e. it should be called as `HornerSimple(A,x)`:

`A` : The list $A = [A_0, A_1, \ldots, A_n]$ of coefficients (given as a row-vector if the A_j are integers or as a three-dimensional array if the A_j are matrices).

`x` : The integer number or a square matrix with integer entries at which the polynomial is evaluated.

The function `HornerSimple` should also work in the special case when the list `A=[]` is empty. In this case it should return a zero matrix of the same size as x (and the 1×1 matrix `[0]` if x is a number). `HornerSimple` should be able to handle all of the following cases:

1. both x and the coefficients A_j are integers
2. both x and the coefficients A_j are integer-valued $N \times N$ matrices
3. x is an integer, and the coefficients A_j are integer-valued $N \times N$ matrices
4. x is an integer-valued $N \times N$ matrix and the coefficients A_j are integers

Hint: Use the MATLAB function `cat(3,·)` to concatenate matrices into a list (see MATLAB documentation https://uk.mathworks.com/help/matlab/math/multidimensional-arrays.html):

- `A=cat(3,`$A_0, A_1, \ldots, A_n$`)` creates the array $A = [A_0, A_1, \ldots, A_n]$
- `A(:,:,j)` returns the j-th matrix stored in the array A (Note that Matlab starts counting at 1, so `A(:,:,1)` is A_0, `A(:,:,2)` is A_1 and so on).

[5 points + 1 style point]

Part II

Write an <u>iterative</u> function `HornerSparse` which evaluates a polynomial $p(x)$ using the Horner scheme given in Eq. (3). The function should return the value of the polynomial for given lists i, B and given x. It should be passed the following three arguments **in exactly this order**, i.e. it should be called as `HornerSparse(index,B,x)`:

`index` : The list $i = [i_1, i_2, \ldots, i_k]$ (given as a row-vector) of powers of x for all non-vanishing terms. You can assume that the list i is sorted and $0 \le i_1 < i_2 < \cdots < i_k$ (you do not have to write code to check this).

`B` : The list $B = [B_1, B_2, \ldots, B_k]$ of non-zero coefficients. You can assume that the list B has the same length as i (you do not have to write code to check this). `B` is given as a row-vector if the B_j are integers or as a three-dimensional array if the B_j are matrices.

`x` : An integer number or a square matrix with integer entries at which the polynomial is evaluated.

The function should also work in the special case when the lists `index=[]` and `B=[]` are both empty. In this case it should return a zero matrix of the same size as x (and the 1×1 matrix `[0]` if x is a number). `HornerSparse` should be able to handle all of the following cases:

1. both x and the coefficients B_j are integers
2. both x and the coefficients B_j are integer-valued $N \times N$ matrices
3. x is an integer, and the coefficients B_j are integer-valued $N \times N$ matrices
4. x is an integer-valued $N \times N$ matrix and the coefficients B_j are integers

[6 points + 1 style point]

Part III

(a) Write a <u>recursive</u> function `Euclid` which calculates the greatest common divisor of two integer numbers using Algorithm 1. The function should be passed two numbers `r` and `s`, i.e. it should be called as `Euclid(r,s)`. The function should return -1 if either r or s is negative, and it should return -2 if either r or s is a positive, non-integer number.

[5 points + 1 style point]

(b) Write an <u>iterative</u> function `PolyEuclid` which calculates the greatest common divisor of a list $r = [r_1, r_2, \ldots, r_m]$ of n integer numbers using Algorithm 2. The function should be passed a list $[r_1, \ldots, r_m]$, i.e. it should be called as, for example, `PolyEuclid([2,8,6])`. You can assume that all entries r_j are non-negative integers (you do not have to write code to check this). The function should also work and return 0 if the list r is empty.

[2 points + 1 style point]

(c) Use the functions `PolyEuclid` and `HornerSparse` to write a function `HornerEuclid` which evaluates a polynomial using the improved sparse Horner scheme given in Eq. (5). `HornerEuclid` should be passed the lists

i, B and the value x; it should return the value of the polynomial at x. `HornerEuclid` should include a suitable call to `HornerSparse` - do **not** reimplement the sparse Horner scheme in `HornerEuclid`.

The function `HornerEuclid` should **take exactly the same parameters `index`, `B` and `x` (in this order) as the function `HornerSparse`**, i.e. it should be called as `HornerEuclid(index,B,x)`.

[2 points + 1 style point]

Hint: Special care has to be taken in the case $i = [0]$.

3 Guidance

The first function, `HornerSimple`, should be attempted first, then move on to `HornerSparse`. Once you have `HornerSimple` working (test it on small examples, e.g. 1×1 matrices = ordinary polynomials, first!), you can use that to test that you have `HornerSparse` working. After that, work on Part III. You might find the tests in Appendix A useful, but you need to design additional tests yourself (these additional tests will not be marked, but they will help you to verify the correctness of your code). A student who has actively engaged with the course and completed all tickables will be able to complete the assignment in 15-20 hours.

3.1 Comments and style

Comment all your code in sufficient detail (see model solutions of tickables for examples) and format it correctly (e.g. indent code in if-statements and for-loops). You will receive style points for well documented and formatted code. Follow the Matlab style guides which are available at

- `https://uk.mathworks.com/matlabcentral/fileexchange/46056-matlab-style-guidelines-2-0`
- `https://uk.mathworks.com/matlabcentral/fileexchange/45047-matlab-style-guidelines-cheat-sheet`

but use **mixed case, starting with a capital letter** for long variable- and function names: `Area`, `CalculateArea()`, `SumOfSquares`.

A Testing

You should carefully test your code and make sure that it works in all cases discussed in the assignment. You can use the following tests to check your code for correctness. This list is non-exhaustive, and you have to write additional tests which you designed yourself.

You can of course simply type the tests into the Matlab command window one by one to check whether the test passes, for example:

```
>> HornerSimple([2],[1,2;3,4])==[2,0;0,2]
```

You should see

```
ans =
  logical
   1
```

if the test passes and

```
ans =
  logical
   0
```

or an error message if the test fails. Or, you could just call the function and inspect the output:

```
>> HornerSimple([2],[1,2;3,4])
ans =

     2     0
     0     2
```

For your convenience, Matlab programs for running the following tests automatically can be downloaded from the "Coursework" section on the moodle MA10265_S1 moodle page (`https://moodle.bath.ac.uk/course/view.php?id=59019#section-2`). Make sure that you use the files provided under the "test scripts for coursework [supplementary assessment]" entry.

To use those tests, unzip the downloaded .zip file and put the scripts testHornerSimple.m, testHornerSparse.m, testEuclid.m, testPolyEuclid.m and testHornerEuclid.m **into the same directory in which you implemented the functions** that you are asked to write in this coursework. You can then run the individual scripts by typing e.g. `testHornerSimple` in the command window.

Suggested tests for function `HornerSimple`

1. `HornerSimple([1,2,3],5)==86`
2. `HornerSimple([1,3],[1,2;3,4])==[4,6;9,13]`
3. `HornerSimple([1,2,3,4],eye(1024))==10*eye(1024)`
4. `HornerSimple(cat(3,[1,2;3,4]),eye(2))==[1,2;3,4]`
5. `HornerSimple(cat(3,[3,9;-1,1],[0,-2;3,4],[1,5;-2,-7]),[2,1;7,-1])==[35,52;-38,-58]`
6. `HornerSimple(cat(3,eye(2),2*eye(2)),7)==15*eye(2)`

Suggested tests for function `HornerSparse`

1. `HornerSparse([1],[3],7)==21`
2. `HornerSparse([0,1,2],[1,2,3],5)==86`
3. `HornerSparse([1],[3],[1,2;3,4])==[3,6;9,12]`
4. `HornerSparse([0,1,2],[1,2,3],[1,2;3,4])==[24,34;51,75]`
5. `HornerSparse([381,635],[2,3],eye(1024))==5*eye(1024)`
6. `HornerSparse([0],cat(3,eye(2)),eye(2))==eye(2)`
7. `HornerSparse([2,5],cat(3,[1,2;7,3],[3,-7;1,0]),[5,7;2,1])==[11160,12546;14970,16782]`

Suggested tests for function `Euclid`

1. `Euclid(9,39)==3`
2. `Euclid(8,0)==8`
3. `Euclid(0,7)==7`
4. `Euclid(48,49)==1`
5. `Euclid(0,-3)==-1`
6. `Euclid(0,3.2)==-2`

Suggested tests for function `PolyEuclid`

1. `PolyEuclid([8,6])==2`
2. `PolyEuclid([9,39,15])==3`
3. `PolyEuclid([8,0,12])==4`
4. `PolyEuclid([1,1,6])==1`
5. `PolyEuclid([0,1,0])==1`

Suggested tests for function `HornerEuclid`

1. `HornerEuclid([0],[2],7)==2`
2. `HornerEuclid([0,1,2],[1,2,3],5)==86`
3. `HornerEuclid([],[],[1,2;3,4])==zeros(2)`
4. `HornerEuclid([0,2,4],cat(3,[2,1;0,3],[8,-2;5,7],[-9,0;3,1]),[1,2;3,6])==[-3071,-6145;2240,4483]`
5. `HornerEuclid([0,2,4],[1,2,3],[1,2;3,4])==[612,890;1335,1947]`
6. `HornerEuclid([381,635],[2,3],eye(1024))==5*eye(1024)`