Programming: Numpy

1.1 Learning outcomes

Computing and Python

- Create vector and matrices.
- Basic matrix operations.
- Matrices to improve program speed.

1.2 Introduction

The aim of these notes is to introduce the python `numpy` library. The library allows programmers to use arrays, which is a generalisation of matrices and vectors. The library provides functions to perform most common matrix operations including linear algebra operations such as inverting or diagonalising matrices.

The `numpy` library is also widely used because is allows one to perform the same arithmetic operations on a large set of numbers much faster than when using loops operating on lists.

The notes cover a large amount of material. The aim is not for you to remember all the details but rather to remember what can be done with numpy. You can then go back to these notes or use a web search whenever you want to perform a particular task.

In the first section we will see how to create arrays. We will then cover basic arithmetic operations and illustrate how numpy can make programs much much faster. We will then describe a few numpy linear algebra functions as well as a few other ones.

We will finish by describing some of the python pitfalls as they are a major source of problems.

Each time we describe a new python function, we write its name in green in the margin. The aim is to make it easier for you to find the description of the function when you need it later.

Every now and then we also provide a few programming tips as separate paragraphs coloured in blue.

1.3 Arrays are matrices

A numpy array is in many ways similar to python lists and they are used to represent vectors (arrays with 1 index), matrices (arrays with 2 indices) or tensors (arrays with several indices). We will stick to arrays of `int` and `float` types.

While list can contains a mix of numbers and sub lists, arrays are more structured. For example the list `[[1,2,3],[4,5,6]]` is made out of 2 list of 3 element each and corresponds to a matrix with 2 rows and 3 columns.

1.4 Creating Arrays

Before one can use `numpy` one must import it as a module and it is conventional to alias its name to `np`. In the example below, we create 2 vectors, known as 1 dimensional arrays, and add them together. The numpy function `array` allows one to create an array from a list or from another array.

file: matrix_1.py

```
import numpy as np

V1 = np.array([1, 2, 3])
V2 = np.array([10, 20, 30])

print("V1=", V1)
print("V2=", V2)
print("V1+V2=", V1+V2)
```

The program above, like all programs in these problem sheets, are available in the code folder. Load it in spyder, or any other python editor, and run it to see what it outputs. Run matrix_1.py to check that it does add the 2 vectors together.

To create matrices, one can also use the array function and give it a list of lists corresponding the the different rows as an argument.

file: matrix_2.py

```
import numpy as np

M1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
M2 = np.array([[10,20,30],[40,50,60],[70,80,90]])

print("M1=", M1)
print("M2=", M2)
print("M1+M2=",M1+M2)
```

Run matrix_2.py to check that it does add the 2 arrays together.

We can also access the elements of an array using indices, exactly as for lists, as illustrated in the code below. In this example, we use the numpy function random.rand which generates an array of uniformly distributed number in the range [0,1). The dimension of the array is specified by its arguments, a 2 by 3 array in this example.

file: rand_matrix.py

```
import numpy as np

dim1 = 2;
dim2 = 3
M = np.random.rand(dim1, dim2)

print("M=", M)
for i in range(dim1):
  for j in range(dim2):
    print("M[",i,",",j,"]=",M[i, j])
```

Run rand_matrix.py, look at its output and then try to understand how it works.

Programming tip: notice that in the example above the dimension of the array are used in 2 different places: to create the array and to display their values. This is why we use the two variables dim1 and dim2. We could have entered the numbers 2 and 3 explicitly (referred to as magic number in programming jargon) in the 2 places where we use them, but this is bad programming practice as if we decide to change these values later we have to remember to change them in all the places they appear, with a risk of forgetting some.

Problem 1.1:

rand

Make a copy of the program rand_matrix.py, call it rand_vector.py and modify it so that it generates a vector of 5 random numbers and then displays each element one at a time as in the example above. To generate a random vector, random.rand only needs 1 argument (the size of the vector). As one can easily guess, element i of vector V is given by V[i].

numpy has other useful functions to create arrays:

- `zeros` : creates an array with all entries set to 0 (as float). The dimension of the array is specified by an array of integers. Example `A=np.zeros([2,3])`. zeros
- `one` : same as `zeros` but set all entries to 1 (as float). Example `A=np.ones([2,3])`. ones
- `eye` : create a square unit matrix: all diagonal elements set to `1` (as a float) and the others to `0` (as a float). Takes one argument: the size of the square matrix. Example `A=np.eye(3)`. eye
- `diag` : create a square diagonal array. Takes as argument a list of values for the diagonal elements. The size of the list determines the size of the array. Example `A=np.diag([1.0,2,3])`. diag
- `full` : create an array of the specified dimension and set all the values to a specified value such as: `A=np.full([2,3],10)`. The type of array is determined by the 2nd parameter, an `int` in this example. full
- `copy` : make a copy of an array. `B = A.copy()`. (We will explain later the difference between `B=A` and `B = A.copy()`) copy

These numpy functions are illustrated in the program below:

file: `create_array.py`

```
1  import numpy as np
2
3  dim1 = 2;
4  dim2 = 3
5  Mzero = np.zeros([dim1, dim2])
6  Mone = np.ones([dim1, dim2])
7  Munit = np.eye(dim2)
8  Mdiag = np.diag([1., 2, 3])          # 1. -> ensure this is an array of float
9  Mfull = np.full([dim1, dim2], np.pi) # np.pi is 3.1415...
10 Mcopy = Mfull.copy()
11
12 print("Mzero=", Mzero)
13 print("Mone=", Mone)
14 print("Munit=", Munit)
15 print("Mdiag=", Mdiag)
16 print("Mfull=", Mfull)
17 print("Mcopy=", Mcopy)
```

Run `create_array.py` an try to understand how each numpy function works.

Programming tip: notice that when we create the diagonal array, we have placed a *dot* just after the first item to force it to be a float instead of an integer. As a results, all the other numbers also become floats. Had we not done so, the matrix would have been made out of integer values which, if not intended, could cause some programming errors. (This is one of the weakness of Python).

It is sometime useful to know the dimension of an array, especially when passed as the argument of a function. This can be easily done using the array attribute `shape` which is a tuple where each element is the size in the corresponding dimension. shape

Notice that `shape` is not a function but a class attribute/variable. This means that if `M` is an array, one then uses `M.shape` without any parentheses. We explain why it is so in another document where we cover classes.

One can also resize a matrix easily using the function `resize`. For example, a vector can be converted to a matrix or vice versa. If a 4×5 matrix is stored as an array `M` of shape `(4,5)`, then `M.reshape([5,4])` will convert it to an array of shape `(5,4)` and `M.reshape([20])` to a vector of length `(20)`. Notice that converting an array from shape `(4,5)` to shape `(5,4)` is not the same as computing the transpose of the matrix; the shape of the arrays are the same, but the elements are not in the same order. resize

file: resize_array.py

```
import numpy as np

M = np.array([[1,2,3], [4,5,6]])
print("M=", M)
print("M.shape=", M.shape) # M: array of shape (2,3)

M.resize([6])
print("M=", M)
print("M.shape=", M.shape) # M now vector of shape (6)

M.resize([3,2])
print("M=", M)
print("M.shape=", M.shape) # M now array of shape (3,2)
```

Finally one can access ranges of elements using the sub-ranges of elements, called slices, as with lists. This can be used to extract part of arrays or to modify part of them. This is more fiddly, but very useful.

If we have a 1 dimensional array `V` with 10 elements, then, as for lists, `V[3:6]` corresponds to the 3 elements sub array made out of `V[3]`, `V[4]` and `V[5]`. This also works for multidimensional arrays. If `M` is a 4 by 4 array, `M[1]` corresponds to the second row of `M` (remember the first row has index 0). `M[:,1]` on the other hand corresponds to the 2nd column of `M`. One can even take subsets of the columns and rows at the same time as illustrated in the program below:

file: sub_array.py

```
import numpy as np

print("Selecting sub-array")
M = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])
print("M=", M)                    # M a 4x4 array
print("M=[0]", M[0])              # The first row of M
print("M=[:,1]", M[:,1])          # The 2nd column of M
print("M=[1:3,1:3]", M[1:3,1:3]) # The 4 elements (1,2)x(1,2) at the center
    of M

print("####################################")
print("Modifying the array")
# Matrix assignment
M[1] = np.array([50, 60, 70, 80])    # Modify the 2nd row
print("M=",M)

M[:,2] = np.array([30, 7, 110, 150]) # Modify the 3rd column
print("M=",M)

M[1:3,1:3] = np.array([[-6, -7], [-10, -11]]) # Modify the center of the
    array
print("M=", M)
```

Run `sub_array.py`, look at its output and then try to understand how it works.

Problem 1.2:

Write a program called `set_sub_array.py` which uses the numpy functions `zeros` and `one` as well as slices, to generate the following array

```
[[0. 0. 0. 0.]
 [0. 1. 1. 0.]
 [0. 1. 1. 0.]
 [0. 0. 0. 0.]]
```

and prints it on the screen.

Check Point 1.1

1.5 Arithmetic Operations

Arrays would be of little use if it wasn't possible to perform mathematical operations with them. As a matter of fact numpy offers a wide range of operations and functions.

As illustrated above, one can add arrays, and we can also subtract them as easily. We should also be able to perform multiplications as in

file: array_mult.py

```
1 import numpy as np
2
3 M1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
4 M2 = np.diag([10,20,30])
5
6 print("M1=",M1)
7 print("M2=",M2)
8 print("M1*M2=",M1*M2)
```

Run array_mult.py and look carefully at the output. Something is not quite right as the results of the multiplication is not the expected one.

The reason is that all the usual arithmetic operations with arrays are performed element by element, which is very useful to perform the same operation on a large set of numbers. While for addition and subtraction this corresponds to the usual arithmetic operations, for multiplication, *division* and even powers it does not.

It is nevertheless possible to *mathematically* multiply matrices. This can be done in 2 ways: use a numpy function called dot or use the operator @ instead of *:

dot, @

file: matrix_mult.py

```
1 import numpy as np
2
3 M1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
4 M2 = np.diag([1,2,3])
5
6 print("M1=", M1)
7 print("M2=", M2)
8 print("M1@M2=", M1@M2)
9 print("M1.dot(M2)=", M1.dot(M2))
10 print("np.dot(M1,M2)=", np.dot(M1,M2))
```

Run matrix_mult.py, and convince yourself that this does perform a correct matrix multiplication.

The division operator also works elements by element. There is a function to invert a matrix, but his will be covered in the next section.

Aside from the arithmetic operations, one can also apply the most common functions such as trigonometric functions, power, logarithm to array elements by elements:

sin, cos, ...

file: matrix_trig.py

```
1 import numpy as np
2
3 M = np.array([[1,0.5],[0.25,0.125]])
4
5 print("M**2=", M**2)
6 print("sin(M)=", np.sin(M*np.pi))
7 print("cos(M)=", np.cos(M*np.pi))
8 print("tan(M)=", np.tan(M*np.pi))
9 print("arcsin(M)=", np.arcsin(M))
10 print("exp(M)=", np.exp(M))
11 print("log(M)=", np.log(M))
12 print("np.sqrt(M)=", np.sqrt(M))
```

1.6 Speeding Code

One of the great feature of numpy is that it is very fast. All the module functions have been written in a much faster programming language to make optimal use of the computer computing power.

We will illustrate this with a small example. We will create a random matrix `R` with 100 rows and 100000 columns and create a vector of size 100000 which is the product of the corresponding rows of `R`. Mathematically:

$$M_j = \prod_{i=0}^{99} R_{i,j}, \qquad j = 0 \ldots 99999. \tag{1.1}$$

Graphically, this can be represented as follows:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \ldots & a_{0,99999} \\ a_{1,0} & a_{1,1} & \ldots & a_{1,99999} \\ \vdots & \vdots & \vdots & \vdots \\ a_{99,0} & a_{99,1} & \ldots & a_{99,99999} \end{pmatrix} \rightarrow \begin{pmatrix} a_{0,0}\, a_{1,0} \ldots a_{99,0} \\ a_{0,1}\, a_{1,1} \ldots a_{99,1} \\ \vdots \\ a_{0,99999}\, a_{1,99999} \ldots a_{99,99999} \end{pmatrix}^t . \tag{1.2}$$

The program `fast_numpy.py` performs the calculations using 2 nested loops over i and j.

file: `fast_numpy.py`

```
1  import numpy as np
2  import time
3
4  dim1 = 100
5  dim2 = 100000
6  R = np.random.rand(dim1, dim2)
7
8  Ms = np.full((dim2), 1.0) # Creates a long vector initialised to 1
9
10 # SLOW
11 t_start = time.time()     # Time in second since 1 Jan 1970
12
13 for i in range(dim1):
14   for j in range(dim2):
15     Ms[j] *= R[i, j]      # M is the product of the R[i]
16 t_end = time.time()       # Time in second since 1 Jan 1970
17
18 print("Loop time: ", t_end-t_start)
19
20 # FAST
21 Mf = np.full((dim2), 1.0) # Creates a long vector initialised to 1
22 t_start = time.time()     # Time in second since 1 Jan 1970
23
24 for i in range(dim1):
25 #  TO COMPLETE
26   pass                    # To remove when you complete the line above
27
28 t_end = time.time()       # Time in second since 1 Jan 1970
29 print("Numpy time: ", t_end-t_start)
30 print("(M2-M)**2=", (Mf-Ms)@(Mf-Ms))
```

At the end, the program computes the time taken to perform the calculation in seconds. (You do not need to understand how this is done at this stage).

Problem 1.3:

Modify the program `fast_numpy.py` and replace the line where it says `TO COMPLETE` to perform the multiplication of the vectors using numpy (*i.e.* replacing the `j` loop by a numpy operation). Instead of multiplying each element of the vector `Mf` by the corresponding element of row `i` of `R`, as on line 15, use numpy to perform all these multiplications at once, keeping only the loop over the rows.

The last line of the code computes the difference between your answer and the one obtained with a double loop. It should be `0.0`.

Run the program and check the difference in speed. Depending on the system you are using, Numpy should be about 100 to 1000 times faster!

Check Point 1.2

1.7 Linear Algebra

As mathematician, we are particularly interested in using arrays for linear algebra. More specifically we want to invert matrices, diagonalise them, compute their eigen values or solve system of linear algebraic equations.

The numpy function `transpose` does what is says: transpose the array.

transpose

To invert a matrix, or compute its eigen vectors, one must use the module `numpy.linalg`.

- `inv`: computes the inverse of a square matrix. So if `M` is a $N \times N$ array, `np.linalg.inv(M)` computes its inverse.

inv

- `eig`: computes the eigen value and the eigen vector of a square matrix. If M is an $N \times N$ square matrix, the N eigen values λ_i and the corresponding eigen vectors V_i are defined by

eig

$$MV_i = \lambda_i V_i. \tag{1.3}$$

`np.linal.eig(M)` returns a vector and an array as a tuple. The vector elements are the eigen values (λ_i) and the corresponding columns of the array are the eigen vectors (V_i). As they are defined up to an overall constant, numpy normalises the eigen vectors to 1. This is illustrated in `lin_alg.py`:

file: `lin_alg.py`

```
1  import numpy as np
2  import numpy.linalg as la
3
4  # Transpose of a matrix
5  M = np.array([[1,2], [1,1]], dtype=np.float64)
6  Mt= np.transpose(M)
7  print("M=", M, "\nMt=", Mt)
8
9  # Inverse of a matrix
10 print("M=",M)
11 Minv = la.inv(M)
12 print("inv(M)=", Minv)
13 print("M*inv(M)=", M@Minv)
14
15 # Eigen values and eigen vectors of a matrix
16 lam, V = la.eig(M)
17 print("Eigen values", lam)
18 print("Right Eigen vectors array", V)
19
20 dim = M.shape[0] # The size of the square array
21 for i in range(dim):
22     print("Eigen value:", lam[i])
23     v = V[:,i]                    # Selects column i from V
24     print("Eigen vector:", v)
25     print("M*V-lam*V=", M@v-lam[i]*v)
```

Run `lin_alg.py`, and convince yourself that it does what it says. Our advice is not to memorise this, but remember it can be done and look it up (Google, these notes ...) when you need to invert or diagonalise matrices.

Problem 1.4:

The problem below is more complex than the previous ones. Write your program one step (bullet point) at a time and check it after each step.

Programming tip: when writing programs, subdivide the tasks in small steps (small section of the code) that you can test individually. This is more effective than writing the full program and then trying to find the bugs in it at the end.

- Write a program called `sym_matrix.py` which generates a random square array `M` of dimension 3 (remember not to use magic numbers).
- Generate a symmetric matrix `S` obtained by adding M and its transposed: $S = M + M^t$.
- Compute the eigen values and eigen vectors of `S`. Print the eigen vectors and their eigen values.
- Compute and print the scalar product between each pair of eigen vectors (including themselves) to verify that they are orthonormal.

Write your code so that it works in any dimension. Increase the dimension of the arrays to 4, 5 or more and check that your program still works.

Check Point 1.3

One can also use numpy to solve system of linear algebraic equations of the type

$$Mx = b \tag{1.4}$$

where M and b are respectively known $n \times n$ matrix and n vector while x is an unknown n vector.

file: `matrix_solve.py`

```
1  import numpy as np
2  import numpy.linalg as la
3
4  # Transpose of a matrix
5  M = np.array([[1,2,3],[6,5,4],[8,7,9]],dtype=np.float64)
6  b = [3, 2, 1]
7  print("M=\n", M, "\nb=", b)
8
9  x = la.solve(M, b)
10 print("Mx=b -> x=\n", x)
11
12 # We test that is is correct
13 b2 = M@x
14
15 print("b2=M.x=\n", b2)
```

1.8 Other Matrix Functions

Numpy has several very useful functions and we will now describe a few of them.

arange

The first one is `arange` which is similar to the Python `range` except that it generates an array and works with float values as well as integers. It takes the same arguments as `range`: the lower bound, the upper bound and the step. The values are returned as an array of integers, excluding the upper bound.

linspace

Similar but more useful is `linspace` . It also takes 3 arguments: the lower bound, the upper bound and the total number of values (*i.e.* the size of the array). `linspace` does include the upper bound in the values returned.

This is illustrated in the code below and shows how `linspace` can be used to plot a function.

file: `matrix_ranges.py`

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 1, 0.1)           # A simple example
print("np.arange(0, 1, 0.1) ->",x) # 1 is not included


x = np.linspace(0, 1, 11)          # A simple example. 1 is included
print("linspace(0, 1, 11) ->",x)

x = np.linspace(0, 2*np.pi,200)    # 200 points for the figure
y = np.sin(x+0.1*x*2)/(1+x**3)     # A function

plt.plot(x,y, "b-")
plt.xlabel("x")
plt.ylabel("y")
plt.title("F(x)= sin(x+0.1*x**2)/(1+x**3)")
plt.show()
```

It is also sometimes useful to move the elements of an array sideways, like shifting all the elements one position to the right (or to left). When shifting to the right, one must decide what to do with the first element of the array: set it to zero or replace it by the last element?

The first case can be done using slices, while the second is implemented by the roll function:

roll

- `roll(V,1)` : shifts all the elements of the vector `V` to the right by one position, moving the last element in the first position. `[1,2,3] -> [3,1,2]`.
- To shift the element of `V` to the right we copy all but the last element of `V` into all but the first element of `V`: `V[1:] = V[0:-1]`. We then set `V[0]=0` and this results in `[1,2,3] -> [0,1,2]`.

This is illustrated for vectors and square arrays in the following program:

file: `matrix_roll.py`

```
import numpy as np

# circular shift
print("Circular shifts on vectors")
V = np.linspace(0, 10, 6)
print("V=",V)
V2= np.roll(V,1)               # roll 1 to the right
print("roll(V,1) ->", V2)
V3= np.roll(V,-2)              # roll 2 to the left
print("roll(V,-2) ->", V3)

V = np.linspace(0, 10, 6)
print("\nV=",V)
V2= np.roll(V, 1)              # roll 1 to the right
print("roll(V, 1) ->", V2)
V3= np.roll(V, -2)             # roll 2 to the left
print("roll(V, -2) ->", V3)

print("\nCircular shifts on matrices")
M = np.array([[1,2,3],[3,4,5],[6,7,8]])
print("M=\n", M)
M2 =np.roll(M, 1, axis=0)      # roll 1 to the down
print("roll(M, 1, axis=0) ->\n", M2)

M3=np.roll(M,-1,axis=1)        # roll 1 to the leftt
print("roll(M, -1, axis=1) ->\n", M3)

print("\nLinear shifts")       # shift 1 to the right. fill in with zeros
V = np.linspace(0, 10, 6)
print("V=", V)
V[1:] = V[0:-1]                # we lose the original V here
V[0] = 0                       # V[0] is the old value -> set it to 0
print("shifted V=", V)
```

Notice that to perform the linear shift, we use the expression `W[1:] = V[0:-1]`. `W[1:]` means all the elements of `W` but the first one, while `V[0:-1]` returns all the elements of `V` but the last one.

1.9 Numpy Pitfalls

One of the major pitfall of python is that objects such as lists or arrays are not copied by default but passed "by reference". To illustrate this, we use the following example: we create an array `A` and copy it to `B`. We then modify the first element of `B` and noticed that `A` has changed as well. The reason is that `B` is effectively another name for the array that `A` refers to; they both correspond to the same object. The fix is to replace `A=B` by `A=B.copy()`.

file: `pitfalls.py`

```
1  import numpy as np
2
3  # No copy during assignment
4  A = np.arange(1, 6, 1)
5  print("A=", A)
6  B = A
7  print("B=", B)
8  B[0] = 10
9  print("Set B[0] to 10")
10 print("A=", A)
11 print("B=", B)
12 print("Whoops!\n")
13
14 # No copy as argument of a function
15 def no_diagonal(C):
16     n = C.shape[0] # The size of C
17     for i in range (0, n):
18         C[i,i] = 0
19     return(C)
20
21 print("Whith a function now")
22 A = np.array([[1,0.5], [0.5,1]])
23 print("A=", A)
24 B = no_diagonal(A)
25 print("B=", B)
26 print("A=", A)
27 print("Not necessarily intended!")
```

The same is true when passing an array as a function argument. In the example, line 24, the function `no_diagonal` is given the matrix `A` as an argument and calls it `C` (line 15). `C` is then modified, but that modifies `A` as well.

One can fix this in 2 ways: the best is to add `C = C.copy()` below line 15. This ensures that the function will never modify the array that it is given as an argument. One could also replace line 24 by `B = no_diagonal(A.copy())`, but that fixes the problem only for that function call. Try to fix the program using either method.

1.10 More About Arrays

As we mentioned in section 1, arrays can contain various types of values, but all the element of a given array are always of the same type. Numpy does offer some means to explicitly specify which type of object an array must contain. This is particularly useful to plot functions using numpy. This is illustrated in the example below:

file: `array_types.py`

```
1 import numpy as np
2
3 MF = np.array([1, 2, 3], dtype=np.float64) # Usually what we want
4 MC = np.array([1, 2, 3], dtype=np.complex) # Complex array
5 MI = np.array([1, 2, 3], dtype=np.int64)   # Array of integers
6
7 print("MF=",MF)
```

```
8  print("MC=",MC)
9  print("MI=",MI)
10
11 ZF = np.zeros([3, 3], dtype=np.float64) # Usualily what we want
12 ZC = np.zeros([2, 3], dtype=np.complex) # Complex array
13 ZI = np.zeros([1, 4], dtype=np.int64)   # Array of integers
14
15 print("ZF=", ZF)
16 print("ZC=", ZC)
17 print("ZI=", ZI)
```

Run the program `array_types.py` and notice how the output differ for each type of object. Notice also how `ZI` in that example is a 2 dimensional array with only one column.

1.11 Problems

Problem 1.5:

A cyclist is following an itinerary on a map. The list contains several landmarks with the total distances cycled from the start for each of them. The cyclist would like to know the distance between each successive landmark.

Write a python program called `distance.py` which must contain a function called `distance` that must take an array of total distances as an argument. The function must convert that array into an array of relative distances and return it as an array of the same size as the argument one. You are expected to use slices and no loop.

Test your program with the following values: `10, 14, 30, 45, 53`.

Problem 1.6:

A vector contains an even number of element. Write a function called `swap_odd_even` that takes an array as an argument and returns the array were the elements with an even index and those with an odd index have been swapped over.

So for example `1, 4, 7, 2, 3, 6` would become `4, 1, 2, 7, 6, 3`.

You must start by creating an array that will hold the result and then copy the different items in the correct place. You must use array slices, not loops. Slices are of the form `F:L:S` where `F` is the first index (`0` by default)), `L` is the last index plus 1 (the size of the array by default) and `S` is the step (`1` by default).

The function must make sure the array has an even number of elements and return an empty list if it does not. (If `M` is a numpy vector, `M.shape[0]` is the size of the vector.)

Check Point 1.4

Problem 1.7:

One way to define an exponential is as the following limit:

$$\exp(x) = \lim_{n\to\infty} (1 + \frac{x}{n})^n. \tag{1.5}$$

Moreover to evaluate the exponential of a number numerically we can evaluate a term of the limit for a large but finite n. We can make this particularly fast if we chose $n = 2^m$ as all we have to do is to define $y = 1 + x/n$ and then perform the operation $y \to y^2$, m times.

This can also be used to define the exponential of a square matrix.

The problem consists in computing the exponential of an antisymmetric matrix and to check one of its properties. To do this write a program called `mat_exp.py` and proceed as follows:

- Define a function called `mat_exp(M,m)` which computes $(\mathbb{1} + M/2^m)^{2^m}$ for the array `M` and returns the result. In that expression, $\mathbb{1}$ must be the unit matrix of the same size as `M`. Also, one can't use the power operator as this computes the power of each matrix elements instead of the power of the matrix. Instead, one must evaluate $A = \mathbb{1} + M/2^m$ and then repeat `m` times `A= A@A` to obtain A^{2^m} efficiently.
- Define a function called `test_exp(d)` which:
  - Generates a random $d \times d$ square matrix called `M`.
  - Generate an antisymmetric matrix $A = 0.1 \times (M - M^t)$. (The coefficient 0.1 makes the algorithm work better).
  - Use the function `mat_exp` to compute the exponential of $B = \exp(A)$ using $m = 40$.
  - Compute the matrix product $B \times B^t$ and print it on the screen.
- Call the function `test_exp(3)` a few times. Is the result expected?

