MATH40006: An Introduction To Computation
Module notes, Section 9

These notes, together with all the other resources you'll need, are available on Blackboard, at

https://bb.imperial.ac.uk/

9 Programming using NumPy

9.1 Indexing, slicing, appending and changing

1D NumPy arrays use the same indexing and slicing conventions as lists and tuples:

```
arr3 = np.array([3, 5, 2, 1, 9, 4])
print(arr3[3])
print(arr3[2:5])
```

```
1
[2 1 9]
```

Multi-dimensional arrays can be indexed using a comma notation:

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
print(mat1[0,1])
print(mat1[2,0:2])
print(mat1[:,0])
```

```
3
[-2  1]
[ 2  1 -2]
```

(Note the last example: the zeroth column of `mat1` is returned, as a 1D vector.)

Arrays are **mutable**, like lists, as opposed to being **immutable**, like tuples. You can change individual elements:

```
arr3[2] = 100
print(arr3)
```

```
[  3   5 100   1   9   4]
```

It's also possible to append elements to arrays, though this works differently from how it works with lists:

```
print(np.append(arr3, 77))
print(arr3)
```

```
[  3   5 100   1   9   4  77]
[  3   5 100   1   9   4]
```

Notice those differences. We use a *function* called `append` from the NumPy module, rather than the *method* called append from core Python; it's not
  `arr3.append(77)`
This function *returns as output* the list with the additional element appended; it doesn't append "in place", changing the value of `arr3` as a side-effect. Indeed, it doesn't change the value of `arr3` at all; If we wanted to do that, we'd simply have to reassign the new value to the variable `arr3`, as follows:

```
arr3 = np.append(arr3, 77)
print(arr3)
```

```
[  3   5 100   1   9   4  77]
```

9.2 Iterating over arrays

Just like lists and tuples, arrays can be iterated over, in `for` loops . . .

```
arr3 = np.array([3, 5, 2, 1, 9, 4])
for n in arr3:
    print('n = {}; 2n+1 = {}'.format(n,2*n+1))
```

```
n = 3; 2n+1 = 7
n = 5; 2n+1 = 11
n = 2; 2n+1 = 5
n = 1; 2n+1 = 3
n = 9; 2n+1 = 19
n = 4; 2n+1 = 9
```

. . . and in comprehensions . . .

```
arr3 = np.array([3, 5, 2, 1, 9, 4])
print([(n, 2*n+1) for n in arr3])
```

```
[(3, 7), (5, 11), (2, 5), (1, 3), (9, 19), (4, 9)]
```

We can even iterate across multidimensional arrays, using a NumPy function called `nditer`:

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
for n in np.nditer(mat1):
    print(f'n = {n}; 2n+1 = {2*n+1}')
```

```
n = 2; 2n+1 = 5
n = 3; 2n+1 = 7
n = -2; 2n+1 = -3
n = 1; 2n+1 = 3
n = -5; 2n+1 = -9
n = 0; 2n+1 = 1
n = -2; 2n+1 = -3
n = 1; 2n+1 = 3
n = 2; 2n+1 = 5
```

Note that in this case, Python has iterated along the rows one by one. In fact, that's not absolutely reliable; and in any case, you might want to iterate column by column. There's a way of taking charge of the iteration order, and we invite you to explore that in the exercises.

Finally, its even possible to use `nditer` to iterate across two arrays at the same time:

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
mat3 = np.array([[6, 6, 6], [11, -4, 7], [7, 5, 9]])
for m, n in np.nditer((mat3, mat1)):
    if m % (n+1) == 0:
        print(f'{m} is a multiple of ({n}+1)')
    else:
        print(f'{m} is not a multiple of ({n}+1)')
```

```
6 is a multiple of 2+1
6 is not a multiple of 3+1
6 is a multiple of -2+1
11 is not a multiple of 1+1
-4 is a multiple of -5+1
7 is a multiple of 0+1
7 is a multiple of -2+1
5 is not a multiple of 1+1
9 is a multiple of 2+1
```

9.3 Programmatic generation of arrays

There's a NumPy function called `fromfunction` that allows you to generate an n-dimensional array of floats using a function of n variables. The function will take as its arguments the row and column indexes. In the following example, for instance, each entry of the matrix is equal to twice the row index plus the column index (as a float):

```
def twoxplusy(x, y):
    return 2*x+y
np.fromfunction(twoxplusy, (4, 3))
```

```
array([[0., 1., 2.],
       [2., 3., 4.],
       [4., 5., 6.],
       [6., 7., 8.]])
```

Actually, the easiest way to do that kind of thing is probably to use a **lambda-expression**, which (as you may recall) is a way of referring to a function not via its name but via a description of what it does:

```
np.fromfunction(lambda x, y: 2*x+y, (4, 3))
```

```
array([[0., 1., 2.],
       [2., 3., 4.],
       [4., 5., 6.],
       [6., 7., 8.]])
```

In this, you read

```
lambda x, y: 2*x+y
```

as “the function whose arguments are x and y and whose output is $2\,x+y$”.

9.4 Vectorizing your algorithms

NumPy arrays are nice things: you can use them to represent banks of data, or vectors and matrices, or polynomials. But the biggest impact of NumPy on your life as a Python user may well be the way it allows you to write much more efficient code. This makes use of the fact that we can do things in one go using NumPy arrays that in core Python would require a comprehension or a loop.

Challenge 1: create a one-dimensional data structure containing all the squares of the integers between 0 and 9.

If we wanted to do this using only core Python data structures and functions, we'd either use a loop . . .

```
squares = []
for n in range(10):
    squares.append(n**2)
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

. . . or a comprehension . . .

```
print([n**2 for n in range(10)])
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here's how we could do it using NumPy arrays:

```
print(np.arange(10)**2)
```

```
[ 0  1  4  9 16 25 36 49 64 81]
```

If we want to square every element of a NumPy array, we can simply square the array. This is arguably neater (though this may be a matter of preference); what's undeniable is that for long programming tasks this **vectorized** approach tends to be much, much more efficient.

Challenge 2: write and test a function called `pi_sum`, which takes as its argument a number `n`, assumed to be a non-negative integer, and returns the value of

$$\sum_{r=0}^{n} \frac{4 \times (-1)^r}{2\,r+1}.$$

Here's how we did that using a loop:

```
def pi_sum1(n):
    total = 0.0
    for r in range(n+1):
        total += (4*(-1)**r)/(2*r+1)
    return total
```

Here's how we might do it using NumPy arrays:

```
def pi_sum2(n):
    from numpy import arange
    r_array = arange(n+1)
    return sum((4*(-1)**r_array)/(2*r_array+1))
```

The latter executes quite a lot more quickly for large n, despite the fact that it creates a transparent data structure whereas the first version uses an opaque one. In the following tests, with n equal to ten million, we use the `time` function from the `time` module to time the operations; the time in seconds is the second printed value.

```
from time import time
start = time()
print(pi_sum1(10000000))
print(time()-start)
```

```
3.1415927535897814
6.411463699904994
```

```
from time import time
start = time()
print(pi_sum2(10000000))
print(time()-start)
```

```
3.1415927535897814
1.8668125593228129
```

Challenge 3: write and test a function called `trapezium_rule`, which takes as its arguments a function `func`, floats (or ints) `xmin` and `xmax` and a positive int `n`, and returns the trapezium rule estimate for the integral of `func` with respect to x between `xmin` and `xmax`, using `n` subintervals.

The formula will be familiar to you all:

$$\int_a^b y\,dx \approx \frac{h}{2}\left(y_0 + 2\,y_1 + 2\,y_2 + \cdots + 2\,y_{n-1} + y_n\right),$$

where h is the interval width.

Here's an implementation based on loops:

```
def trapezium_rule1(func, xmin, xmax, n):
    """
    Returns the trapezium rule approximation to the integral of func
    between xmin and xmax; uses n intervals
    """

    h = (xmax - xmin)/n

    total = func(xmin) + func(xmax)

    for r in range(1, n):
        total += 2*func(xmin + r*h)

    return h/2 * total
```

Here's a vectorized one based on NumPy arrays; the key difference is that if `func` is the right kind of function, we can apply it to all the elements in an array in one go:

```
def trapezium_rule2(func, xmin, xmax, n):
    """
    Returns the trapezium rule approximation to the integral of func
    between xmin and xmax; uses n intervals
    """
    from numpy import array, arange

    h = (xmax - xmin)/n

    # end and middle values of x
    ends = array([xmin, xmax])
    mids = arange(xmin+h,xmax,h)

    return h/2 * (sum(func(ends)) + 2*sum(func(mids)))
```

Then the following both work:

```
print(trapezium_rule1(lambda x: x**5 - x**4, 0, 1, 10))
print(trapezium_rule2(lambda x: x**5 - x**4, 0, 1, 10))
```

```
-0.032505
-0.032505
```

The following also both work

```
import numpy as np
print(trapezium_rule1(np.sin, 0, np.pi/2, 10))
print(trapezium_rule2(np.sin, 0, np.pi/2, 10))
```

```
0.9979429863543572
0.9979429863543572
```

However, only `trapezium_rule1` works in the following case:

```
import math
print(trapezium_rule1(math.sin, 0, math.pi/2, 10))
print(trapezium_rule2(math.sin, 0, math.pi/2, 10))
```

```
0.9979429863543572

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-d1d64aa73f60> in <module>()
      2 import math
      3 print(trapezium_rule1(math.sin, 0, math.pi/2, 10))
```

```
----> 4 print(trapezium_rule2(math.sin, 0, math.pi/2, 10))

<ipython-input-42-21a208578962> in trapezium_rule2(func, xmin, xmax, n)
     14
     15     # return
---> 16     return h/2 * (sum(func(ends)) + 2*sum(func(mids)))

TypeError: only size-1 arrays can be converted to Python scalars
```

The error arises because the second implementation tries to apply `func` to and array, and if `func` is a `math` function that won't work.

The second implementation, though, works much more quickly for large values of n; try it!

So, what kinds of tasks can be vectorized in this way? Certainly not everything. There's no vectorized implementation of "Iterate e^{-x} ten times starting at $x = 0$", for example, because that's a task that is in its very nature iterative: each value of x depends on the previous one. Vectorization only works when you want to do the same thing to every element of a data structure: square it, or substitute it into the expression

$$\frac{4 \times (-1)^n}{2n+1},$$

or use it as the argument of some `func`.

Each stage in the task, in other words, can be done independently of the others, in any order; it's a little like the set of circumstances in which you can **parallelize** an algorithm. That's not a bad way to think about vectorization, in fact; as a kind of quasi-parallelization. In fact, vectorized algorithms don't make much use of parallelization, but they're the same sort of thing conceptually; the subtasks can happen "all in one go" instead of having to be thought of sequentially.

9.5 Plotting using arrays

The various plotting functions in `matplotlib` all work just as well with NumPy arrays as with lists, tuples or `range` objects. The ease and efficiency with which arrays can be created using vectorized approaches can make this substantially preferable. Let's illustrate this using a fairly complicated plotting task.

Challenge 4: write a function `plot_hypotrochoid`, which takes as its arguments the numbers `outer_radius`, `inner_radius` and `rho`, with default values 11, 5 and 5, and the keyword-only arguments `npoints` and `show_circles`, with default values 200 and

False; it should then plot the hypotrochoid curve with parametric equations

$$\begin{aligned} x &= (R-r)\cos\theta + \rho\cos\left(\frac{R-r}{r}\,\theta\right), \\ y &= (R-r)\sin\theta - \rho\sin\left(\frac{R-r}{r}\,\theta\right), \end{aligned}$$

for $0 \le \theta \le 2\pi r$, where R, r and ρ correspond to `outer_radius`, `inner_radius` and `rho`, together with the outer and inner circles if specified. Do this using core Python lists with comprehensions, and also using NumPy arrays.

Here's a listing of a "Core Python" implementation of that function:

```
def plot_hypotrochoid(outer_radius=11, inner_radius=5, rho=5, *,
                  npoints=200, show_circles=False):
    """
    Draws the hypotrochoid curve corresponding to a certain set of
    parameters
    """
    from math import sin, cos, pi
    import matplotlib.pyplot as plt

    # theta from 0 to 2*pi*inner_radius
    theta_values = [inner_radius*2*r*pi/(npoints-1)
                    for r in range(npoints)]

    rdiff = (outer_radius-inner_radius)

    x_values = [rdiff*cos(theta) + rho*cos(rdiff*theta/inner_radius)
                for theta in theta_values]
    y_values = [rdiff*sin(theta) - rho*sin(rdiff*theta/inner_radius)
                for theta in theta_values]

    plt.figure(figsize = (7,7))
    plt.axes().set_aspect(aspect='equal')

    plt.plot(x_values, y_values, 'r')

    if show_circles:
        # outer circle
        theta_values2 = [2*r*pi/(192) for r in range(193)]
        x_values2 = [outer_radius*cos(theta)
                     for theta in theta_values2]
        y_values2 = [outer_radius*sin(theta)
                     for theta in theta_values2]
        plt.plot(x_values2, y_values2, 'k')
        # inner circle
        x_values3 = [rdiff+inner_radius*cos(theta)
                     for theta in theta_values2]
        y_values3 = [inner_radius*sin(theta)
                     for theta in theta_values2]
        plt.plot(x_values3, y_values3, 'b')
        # pen position
        plt.plot([rdiff+rho],[0],'g.',markersize=30)
```

Here's the implementation based on arrays:

```
def plot_hypotrochoid2(outer_radius=11, inner_radius=5, rho=5, *,
                  npoints=200, show_circles=False):
    """
    Draws the hypotrochoid curve corresponding to a certain set of
    parameters
    """
    from numpy import sin, cos, pi, linspace
    import matplotlib.pyplot as plt

    # theta from 0 to 2*pi*inner_radius
    theta_values = linspace(0, 2*pi*inner_radius, npoints)

    rdiff = (outer_radius-inner_radius)

    x_values = (rdiff*cos(theta_values) +
                rho*cos(rdiff*theta_values/inner_radius))
    y_values = (rdiff*sin(theta_values) -
                rho*sin(rdiff*theta_values/inner_radius))

    plt.figure(figsize = (7,7))
    plt.axes().set_aspect(aspect='equal')

    plt.plot(x_values, y_values, 'r')

    if show_circles:
        # outer circle
        theta_values2 = linspace(0, 2*pi, 192)
        x_values2 = outer_radius*cos(theta_values2)
        y_values2 = outer_radius*sin(theta_values2)
```

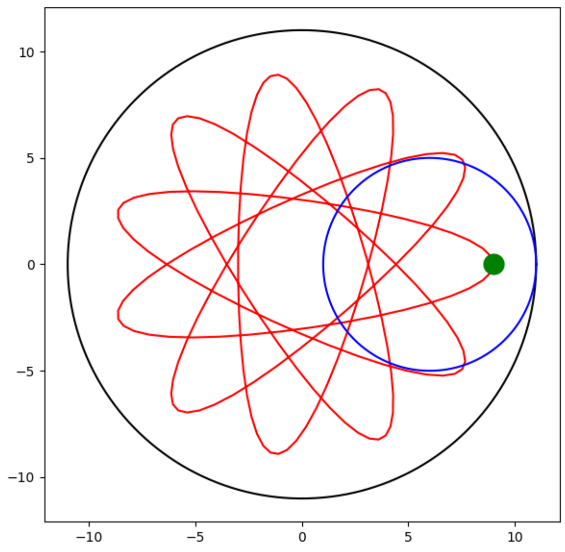

Figure 1: A hypotrochoid curve

```
        plt.plot(x_values2, y_values2, 'k')
        # inner circle
        x_values3 = rdiff+inner_radius*cos(theta_values2)
        y_values3 = inner_radius*sin(theta_values2)
        plt.plot(x_values3, y_values3, 'b')
        # pen position
        plt.plot([rdiff+rho],[0],'g.',markersize=30)
```

Both produce the same kinds of diagram; for example

```
%matplotlib inline
plot_hypotrochoid2(rho=3, show_circles=True)
```

produces a diagram as in Figure 1.

However, I think the code for the second implementation is easier to write and to read, and it will also be quicker to execute (though execution time isn't a major problem here).

Time for one last vectorization programming challenge. The **logistic map**,

$$f_k(x) = k\,x\,(1-x),$$

exhibits lots of different types of behaviour for different values of the parameter k. One way to get a picture of those behaviours, and how they vary with k, is to create a **bifurcation**

diagram. To do that, you choose a value of k, iterate the map m times, then retain the next n iterates (several hundred in each case for the best effect). This creates $(n+1)$ coordinate pairs, the horizontal coordinates being k in each case, and the vertical coordinates the iterates $x_m, x_{m+1}, x_{m+1}, \ldots, x_{m+n}$. You then do this for a different value of k, and so on.

Challenge 5: write and test a function called `bifurcation_diagram`. It should take as its arguments:

- a positive int `resolution`;
- a positive int `nskip`;
- a positive int `niterate`;
- a keyword-only argument, `markersize`, by default equal to 0.5.

It should then create a bifurcation diagram. The parameter k should range from 1 to 4, with a step size equal to the reciprocal of the resolution. The map should first be iterated `nskip` times, with the iterates discarded, and then `niterate` times, with the iterates retained. The bifurcation diagram should then take the form of a point plot, with the `markersize` option set to `markersize`. No data should be returned.

Here's an implementation based on lists, which uses comprehensions:

```
def logistic_map(k, x):
    return k * x * (1 - x)

def bifurcation_diagram1(resolution, nskip, niterate, *, markersize=0.5):
    """
    Plots a bifurcation diagram for the logistic map.
    """
    h = 1/resolution

    k_list = [h * r for r in range(resolution,4*resolution)]

    # initial values of x, 0.3 throughout
    x_list = [0.3 for r in range(3*resolution)]

    # iterate and skip
    for i in range(nskip):
        x_list = [logistic_map(k_list[j], x_list[j])
                  for j in range(3*resolution)]

    # initial lists of k- and x-values
    k_values = k_list
    x_values = x_list

    # iterate and retain
    for i in range(niterate):
        x_list = [logistic_map(k_list[j], x_list[j])
                  for j in range(3*resolution)]
        k_values = k_values + k_list
        x_values = x_values + x_list

    import matplotlib.pyplot as plt
    plt.figure(figsize=(10,7))
    plt.plot(k_values,x_values, '.k', markersize=markersize)
```

Here's one based on arrays:

```
def logistic_map(k, x):
    return k * x * (1 - x)

def bifurcation_diagram2(resolution, nskip, niterate, *, markersize=0.5):
    """
    Plots a bifurcation diagram for the logistic map.
    """
    from numpy import linspace, ones, concatenate

    k_array = linspace(1, 4, 3*resolution+1)

    # initial values of x, 0.3 throughout
    x_array = 0.3 * ones(3*resolution+1)

    # iterate and skip
    for i in range(nskip):
        x_array = logistic_map(k_array, x_array)

    # initial arrays of k- and x-values
    k_values = k_array
    x_values = x_array

    for i in range(niterate):
        x_array = logistic_map(k_array, x_array)
        k_values = concatenate((k_values, k_array))
        x_values = concatenate((x_values, x_array))
    import matplotlib.pyplot as plt
    plt.figure(figsize=(10,7))
    plt.plot(k_values,x_values, '.k', markersize=markersize)
```

Note the use of NumPy's `concatenate` function, as we can't concatenate using the + operator. Notice too how easy it is to update an array of x-values:

```
   x_array = logistic_map(k_array, x_array)
```

The contrast with the first implementation is especially clear here, I think.

What's more, if we type

```
from time import time
start = time()
bifurcation_diagram1(700, 300, 700, markersize=0.01)
print(time() - start)
```

we get

```
9.604787682899456
```

and if we type

```
from time import time
start = time()
bifurcation_diagram2(700, 300, 700, markersize=0.01)
print(time() - start)
```

we get

```
3.535992631587874
```

In both cases, the diagram looks like figure 2, but the execution time is much better in the vectorized version.

9.6 Boolean arrays

If we type

```
arr2d1 = np.array([[1, 0, 1], [0, 1, 0], [0, 0, 1]])
arr2d2 = np.array([[2, 1, -1], [-3, 4, 2], [1, 1, -4]])
print(arr2d1 < arr2d2)
```

we get

```
[[ True  True False]
 [False  True  True]
 [ True  True False]]
```

This is what's called a **Boolean array**, and they're very useful things.

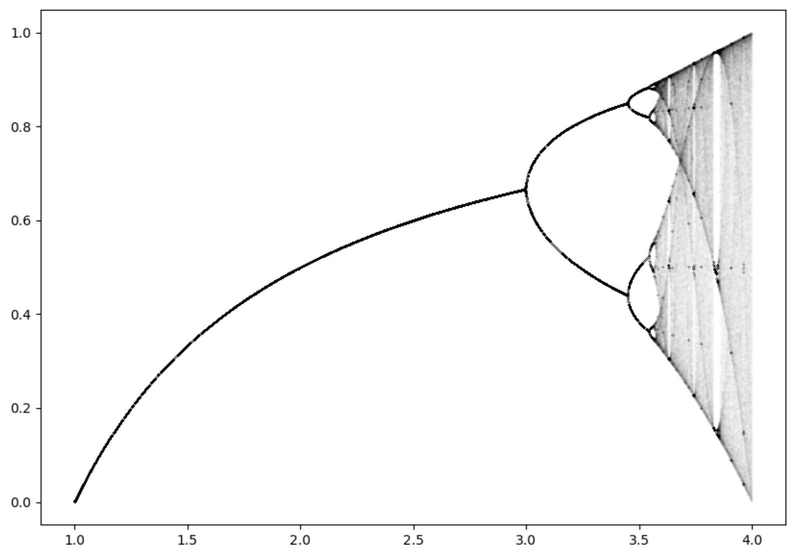

Figure 2: Bifurcation diagram for the logistic map

Challenge 6: create a Boolean array corresponding to those integers between 1 and 24 inclusive that are factors of 24.

```
ints = np.arange(1,25)
24 % ints == 0
```

```
array([ True,  True,  True,  True, False,  True, False,  True, False,
       False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False,  True])
```

The operators for Boolean *expressions* are, of course, `not`, `and` and `or`. The corresponding operators for Boolean *arrays* are `~`, `&` and `|`.

Challenge 7: create a Boolean array corresponding to those integers between 1 and 24 inclusive that are not factors of 24.

```
ints = np.arange(1,25)
~(24 % ints == 0)
```

```
array([False, False, False, False,  True, False,  True, False,  True,
        True,  True, False,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False])
```

Challenge 8: create a Boolean array corresponding to those integers between 1 and 24 inclusive that are *odd* factors of 24.

```
ints = np.arange(1,25)
(24 % ints == 0) & (ints % 2 > 0)
```

```
array([ True, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False])
```

What's really lovely is that you can use Boolean arrays in **indexing**; this offers a neat (and efficient) NumPy alternative to the use of comprehensions for filtering.

Challenge 9: create an array consisting of those integers between 1 and 24 inclusive that are factors of 24.

```
ints = np.arange(1,25)
ints[24 % ints == 0]
```

```
array([ 1,  2,  3,  4,  6,  8, 12, 24])
```

Challenge 10: create an array consisting of those integers between 1 and 24 inclusive that are odd factors of 24.

```
ints = np.arange(1,25)
ints[(24 % ints == 0) & (ints % 2 > 0)]
```

```
array([1, 3])
```

By contrast, here's a reminder of how we'd do those tasks with lists and comprehensions:

```
ints = list(range(1, 25))
[n for n in ints if 24 % n == 0]
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
```

```
ints = list(range(1, 25))
[n for n in ints if 24 % n == 0 and n % 2 > 0]
```

```
[1, 3]
```

These Boolean-array indexes look weird when you first meet them, but they're very nice to work with. In the exercises, you use this to create two NumPy implementations of the Sieve of Eratosthenes for calculating lists of primes.