Summary of chapters 1-5 (part 1)

Ole Christian Lingjærde, Dept of Informatics, UiO

4 October 2017

Today's agenda

- Exercise A.4
- Lists versus arrays: which should I use?
- Vectorization: when does it work?
- Plotting: simple recipes

Exercise A.4

Compute the development of a loan

Solve (A.16)-(A.17) in a Python function:

$$
\begin{aligned}
y_n &= \frac{p}{12\cdot 100} x_{n-1} + \frac{L}{N} \\
x_n &= x_{n-1} + \frac{p}{12\cdot 100} x_{n-1} - y_n
\end{aligned}
$$

Questions (should always be asked in such problems):

- In what order should we update the equations?
- What initial conditions are required?
- What range of n-values should we compute the equations for?

Filename: loan.

In what order should we update the equations?

$$
\begin{aligned}
y_n &= \frac{p}{12 \cdot 100} x_{n-1} + \frac{L}{N} \\
x_n &= x_{n-1} + \frac{p}{12 \cdot 100} x_{n-1} - y_n
\end{aligned}
$$

The order is not always important. But here it is, since one equation requires the output from the other.

- We should assume that we already know $x_{n-1}, x_{n-2}, \ldots$ and also $y_{n-1}, y_{n-2}, \ldots$ (from previous updates).
- We can then calculate y_n (need only x_{n-1})
- We can then calculate x_n (need both x_{n-1} and y_n)

What initial conditions are required?

$$
\begin{aligned}
y_n &= \frac{p}{12\cdot 100}x_{n-1} + \frac{L}{N} \\
x_n &= x_{n-1} + \frac{p}{12\cdot 100}x_{n-1} - y_n
\end{aligned}
$$

We can often (and here) assume that the sequences x_n and y_n start at $n = 0$. This means we should give values to x_0 and y_0 before we start computing the equations for $n = 1, 2, \ldots$.

- Closer inspection reveals that x_0 and not y_0 is used to compute the equations for $n = 1$.
- To choose initial values x_0 and y_0, recall that x_0 is the initial size L of the loan and y_0 is the amount paid back during the first month (which is usually 0)

Range of n-values to use

$$
\begin{aligned}
y_n &= \tfrac{p}{12\cdot 100} x_{n-1} + \tfrac{L}{N} \\
x_n &= x_{n-1} + \tfrac{p}{12\cdot 100} x_{n-1} - y_n
\end{aligned}
$$

- We should start calculating equations for $n = 1$.
- We thus need a loop over $n = 1, 2, \ldots, N$ for some fixed number N.
- The choice of N can be left to the user of the method.

Lists and arrays

Lists and arrays can both be used to store a vector of values. Key differences:

Lists

Very flexible data structures (can add or delete elements, can contain different data types, etc), but you have to do all mathematical operations on them one element at a time.

Arrays

Less flexible (can not add or delete elements, contains only one data type at a time) but you have a whole battery of mathematical operations (numpy) that can be applied on whole arrays, which makes programming easier and faster, and program execution faster.

So what should I use?

- **Arrays** are useful for handling numerical vectors (or matrices) and are required for vectorized array computations and access to the vast library of functions in the numpy package.
- **Lists** are always an option unless you need the functionality above (or are asked to use arrays).
- **Remember:** you can always switch from array to list (`l = list(a)`) and from list to array (`a = np.array(l)`). Not very efficient for very long lists/arrays (often important in real applications).

Comparing lists and arrays

List	Array
`x = [1,2,3,4]`	`x = np.array([1,2,3,4])`
`x = [0]*n`	`x = np.zeros(n)`
`x = [1]*n`	`x = np.ones(n)`
`x = range(n)`	`x = np.arange(n)`
`xnew = x`	`xnew = x`
`xnew = x[:]`	`xnew = x.copy()`
`xnew = x+x`	`xnew = np.append(x,x)`
`h = float(b-a)/(n-1)` `x = [a+i*h for i in range(n)]`	`x = np.linspace(a,b,n)`
`for elem in x:` `   print(elem)`	`for elem in x:` `   print(elem)`

`xnew = [0]*len(x)`
`for i in range(len(x)):`
`   xnew[i] = math.sin(x[i])`
	`xnew = np.sin(x)`

`xnew = [0]*len(x)`
`for i in range(len(x)):`
`   xnew[i] = x[i] + 2*x[i]**2`
	`xnew = x + 2*x**2`

Challenge

There are often *many ways* of doing things in Python:

- Python 2 or Python 3? (small differences in syntax)
- Lists or numpy-arrays? (large differences in syntax)
- Plot with `matplotlib` or `scitools`?
- Write `np.linspace(..)` and `plt.plot(..)` or just `linspace(..)` and `plot(..)`?
- Use `from numpy import *` etc?
- Initiate lists with `a = [0]*n` or use `a.append(..)`?

Advice

- Be consistent, it saves you time (less choices to make).
- Lists are more versatile than arrays and can very often be used.
- But you have to know numpy-arrays as well.
- Don't automatically include `from numpy import *`, etc.

How to refer to numpy-functions

Avoid mixing explicit and implicit package references in a program (e.g. `np.linspace(..)` and `linspace(..)`). When using the numpy package, it is recommended to follow this practice:

- **General rule:** Use `import numpy as np` and refer to numpy functions as `np.linspace(..)`, `np.zeros(..)`, etc.
- **Exception:** For mathematical functions (sin, cos, log, ...) you may use `from numpy import sin, cos` and refer to as `sin(..)`, `cos(..)`, etc.

For more details, see the text book (5th ed.), page 235 and 243.

Vectorization

A key feature of the numpy package is that it allows *vectorized computations*. For example, the following (non-vectorized) code:

```
def f_list(N):
   import math
   x = [0]*N; y = [0]*N; z = [0]*N
   for i in range(N):
       x[i] = 1 + i**2
   for i in range(N):
       y[i] = 1 + i * x[i] - math.tanh(x[i])
   for i in range(N):
       z[i] = abs(y[i])
   return z
```

becomes the following vectorized code:

```
def f_array(N):
    import numpy as np
    x = 1 + np.arange(N)**2
    y = 1 + np.arange(N) * x - np.tanh(x)
    z = np.abs(y)
    return z
```

How much faster is the vectorized code?

Comparing CPU time

```
import time
N = 10**7

t0 = time.clock()
f_list(N)
t1 = time.clock() - t0
print('Nonvectorized: %4.2f seconds' % t1)

t0 = time.clock()
f_array(N)
t1 = time.clock() - t0
print('Vectorized: %4.2f seconds' % t1)
```

```
Terminal> python compare_time.py
Nonvectorized: 6.67 seconds
Vectorized: 0.29 seconds
```

In this example, the vectorized method is 23 times faster!

Vectorization is not always possible

Many array computations can in principle be performed in parallel on all elements in the array; such computations are well suited for vectorization. Other array computations have to be performed in sequence (example: x[1] requires x[0], x[2] requires x[1], etc). Such computations are usually less suitable for vectorization.

Vectorization is not always possible

- Most examples in Appendix A (Difference Equations) are *not* well suited for vectorization.
- The reason is that difference equations express x_n in terms of one or more of the terms $x_{n-1}, x_{n-2}, \ldots$. Thus we need a loop to calculate $x_1, x_2, \ldots$ one by one.
- The choice between list and array is then a matter of taste and what other computations we want to do in the program.

Example: generating all rational numbers

It is easy to print all positive rational numbers (up until a certain size) using a double for-loop:

```
for i in range(1,n):
  for j in range(1,n):
    print('%d / %d' % (i,j))
```

- However, the same number will occur many times, since $1/2 = 2/4$ etc.
- Question: is there a way to avoid this?

Example (cont'd)

A more elegant solution to the above problem involves the *Stern sequence* defined by the following difference equations:

$$\begin{aligned} x_{2n} &= x_n \\ x_{2n+1} &= x_n + x_{n+1} \end{aligned}$$

and with $x_0 = 0$ and $x_1 = 1$.
Amazingly, the sequence $y_n = x_n / x_{n+1}$ contains every positive rational number exactly once. So if we solve the difference equations the unique rationals are just

$$x_1/x_2,\ x_2/x_3,\ x_3/x_4, \ldots$$

Example (cont'd)

Below is Python code to print the first rational numbers generated from the Stern sequence introduced on the previous slide:

```
def stern(N):
    x = [0]*(2*N)
    x[0] = 0; x[1] = 1
    for n in range(1,N):
        x[2*n] = x[n]
        x[2*n+1] = x[n] + x[n+1]
    return x

def printRationals(N):
    x = stern(N)
    for n in range(2*N-1):
        print('%d / %d' % (x[n], x[n+1]))

# We test the method
import sys
printRationals(eval(sys.argv[1]))
```

Testing the method

```
Terminal> python stern.py 10
0 / 1
1 / 1
1 / 2
2 / 1
1 / 3
3 / 2
2 / 3
3 / 1
1 / 4
4 / 3
3 / 5
5 / 2
2 / 5
5 / 3
3 / 4
4 / 1
1 / 5
5 / 4
4 / 7
```

Curve plotting

- The book mentions various options for plotting curves, including `matplotlib.pyplot`, `scitools.std`, EasyViz, Mayavi. Only the first one in required for this course.
- The recommended way to use plot functions is to `import matplotlib.pyplot as plt` and then use `plt.plot(..)` etc to use plot functions (see p. 243 in the book).
- When you use `plot(x,y)` the variables `x` and `y` can be either lists or numpy-arrays.

Plotting a single curve

Suppose x and y are numerical lists or arrays of the same length.

Curve only

```
import matplotlib.pyplot as plt
plt.plot(x, y)                  # Create plot
plt.savefig('Figure1.pdf')      # Save plot as pdf
plt.show()                      # Show plot on screen
```

Curve with decoration

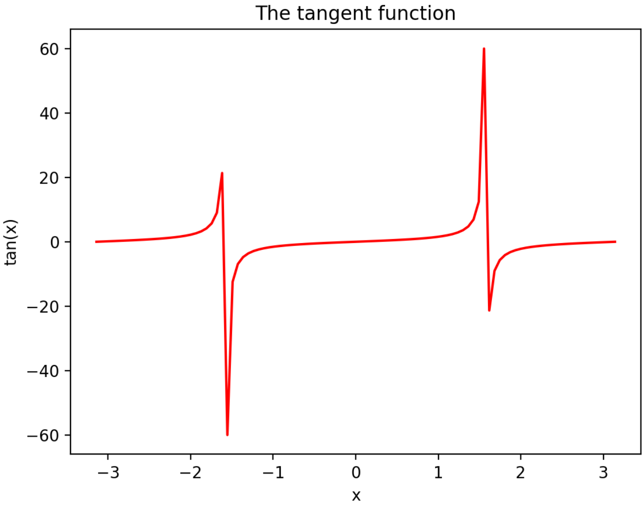
```
import matplotlib.pyplot as plt
plt.plot(x, y, 'r-') # Red line (use 'ro' for red circle)
plt.xlabel('x')      # Label on x-axis
plt.ylabel('y')      # Label on y-axis
plt.title('My plot') # Title on top of plot
plt.axis([0,5,0,1])  # Range on x-axis [0,5] and y-axis [0,1]
plt.show()
```

Example 1

The tangent function

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-3.14, 3.14, 100)
y = np.tan(x)
plt.plot(x, y, 'r-') # Red line (use 'ro' for red circle)
plt.xlabel('x')      # Label on x-axis
plt.ylabel('tan(x)') # Label on y-axis
plt.title('The tangent function')
plt.show()
```

Result

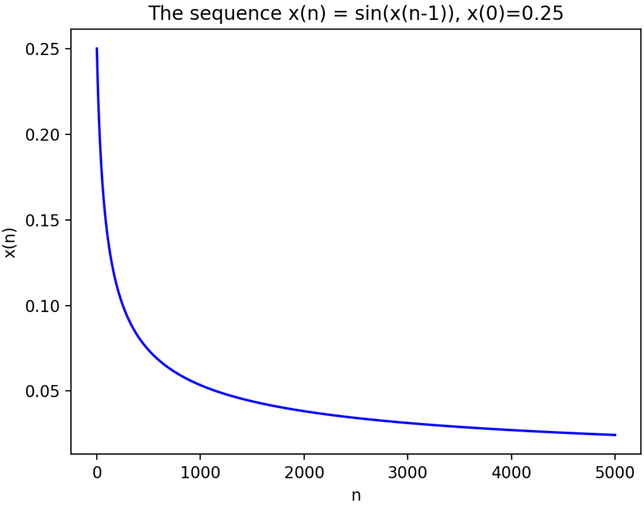


Example 2

The sequence 0.25, sin(0.25), sin(sin(0.25)),...

```
import matplotlib.pyplot as plt
import math
N = 5000
y = [0]*N
y[0] = 0.25
for i in range(1,N):
    y[i] = math.sin(y[i-1])
plt.plot(range(N), y, 'b-')  # Blue line
plt.xlabel('n')              # Label on x-axis
plt.ylabel('x(n)')         # Label on y-axis
plt.title('The sequence x(n) = sin(x(n-1)), x(0)=0.25')
plt.show()
```

Result



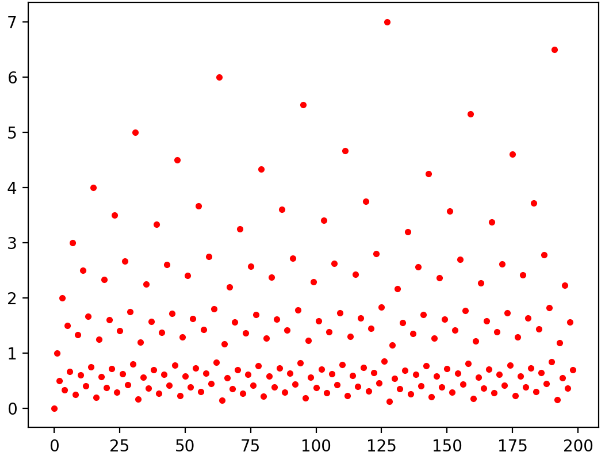
The sequence x(n) = sin(x(n-1)), x(0)=0.25
0.25
0.20
0.15
0.10
0.05
x(n)
0
1000
2000
3000
4000
5000
n

Example 3

The Stern sequence rational numbers

```
def stern(N):
    x = [0]*(2*N)
    x[0] = 0; x[1] = 1
    for n in range(1,N):
        x[2*n] = x[n]
        x[2*n+1] = x[n] + x[n+1]
    return x

import matplotlib.pyplot as plt
N = 100
x = stern(N)
y = [float(x[n])/x[n+1] for n in range(2*N-1)]
plt.plot(range(2*N-1), y, 'r.')
plt.title('Rational numbers from Stern sequence')
plt.show()
```

Result



Plotting curves on top of each other

Suppose x1 and y1 are numerical lists or arrays of the same length, and that x2 and y2 are numerical lists or arrays of the same length.

Curves only

```
import matplotlib.pyplot as plt
plt.plot(x1, y1, 'r-')
plt.plot(x2, y2, 'b-')
plt.show()
```

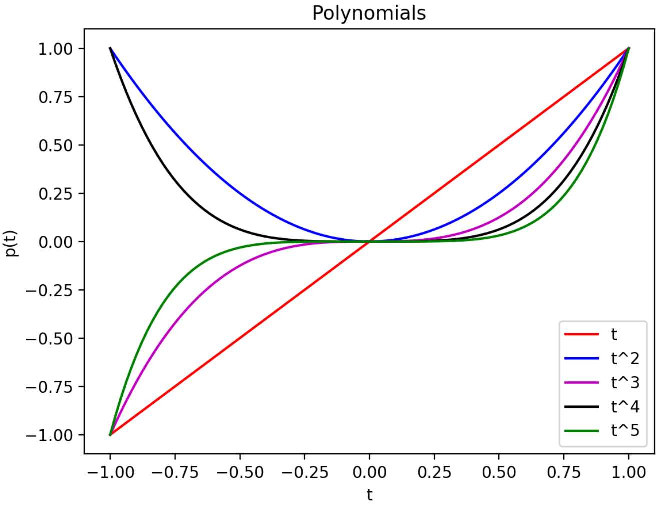
Curves with decoration

```
import matplotlib.pyplot as plt
plt.plot(x1, y1, 'r-')
plt.plot(x2, y2, 'b-')
plt.legend(['y1', 'y2'])
plt.xlabel('x')
plt.ylabel('y')
plt.title('My multiplot')
plt.axis([0,7,0,7])
plt.show()
```

Example 1: Polynomials

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

def p(t,k):
  return t**(k+1)

col = ['r-', 'b-', 'm-', 'k-', 'g-']

t = np.linspace(-1, 1, 100)
for k in range(5):
  plt.plot(t, p(t,k), col[k])

plt.xlabel('t')
plt.ylabel('p(t)')
plt.legend(['t', 't^2', 't^3', 't^4', 't^5'])
plt.title('Polynomials')
plt.show()
```

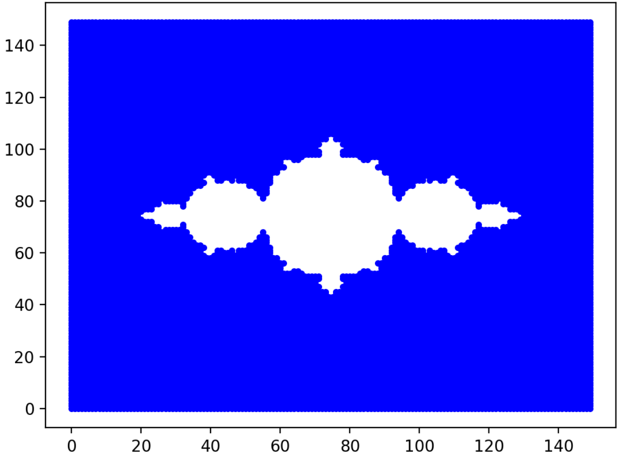
Result


Polynomials
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
p(t)
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
t
t
t^2
t^3
t^4
t^5

Example 2: Julia set

```
import matplotlib.pyplot as plt
import numpy as np
n = 150
x = np.linspace(-2, 2, n); z = [0, 0]
for i in range(n):
   for j in range(n):
      z[0] = x[i]; z[1] = x[j]; k = 0
      while abs(z[0])+abs(z[1]) < 100 and k < 100:
         z = [z[0]**2-z[1]**2-0.75, 2*z[0]*z[1]]
         k = k+1
      if k < 100:
         plt.plot(i, j, 'b.')
plt.show()
```

Result



Multipanel plots

Suppose x1 and y1 are numerical lists or arrays of the same length, and that x2 and y2 are numerical lists or arrays of the same length.

Curves with titles

```
import matplotlib.pyplot as plt
plt.subplot(1,2,1)
plt.plot(x1, y1, 'r-')
plt.title('Title for left panel')
plt.subplot(1,2,2)
plt.plot(x2, y2, 'b-')
plt.title('Title for right panel')
plt.show()
```

Example: Polynomials

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

def p(t,k):
  return t**k

t = np.linspace(-1, 1, 100)
for k in range(1,5):
  plt.subplot(2,2,k)
  plt.plot(t, p(t,k), 'r-')
  plt.xlabel('t')
  plt.ylabel('p(t)')
  plt.legend(['t^%d' % k])
plt.show()
```

Result