Sequences

"Sequences" is a central topic in mathematics:

\[ x_0, x_1, x_2, \ldots, x_n, \ldots \]

Example: all odd numbers

1, 3, 5, 7, \ldots, 2n+1, \ldots

For this sequence we have a formula for the \( n \)-th term:

\[ x_n = 2n + 1 \]

and we can write the sequence more compactly as

\[ (x_n)_{n=0}^\infty \quad x_n = 2n + 1 \]

Other examples of sequences

1, 4, 9, 16, 25, \ldots \quad (x_n)_{n=0}^\infty \quad x_n = n^2

1, 1, 1, 1, \ldots \quad (x_n)_{n=0}^\infty \quad x_n = \frac{1}{n+1}

1, 1, 2, 6, 24, \ldots \quad (x_n)_{n=0}^\infty \quad x_n = n!

1, 1 + x, 1 + x + \frac{1}{2}x^2, 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3, \ldots \quad (x_n)_{n=0}^\infty \quad x_n = \sum_{j=0}^n \frac{x^j}{j!}

Finite and infinite sequences

Infinite sequences have an infinite number of terms \( (n \to \infty) \)

In mathematics, infinite sequences are widely used

In real-life applications, sequences are usually finite \((x_n)_{n=0}^N\)

Example: number of approved exercises every week in INF1100

\[ x_0, x_1, x_2, \ldots, x_15 \]

Example: the annual value of a loan

\[ x_0, x_1, \ldots, x_24 \]

Difference equations

For sequences occurring in modeling of real-world phenomena, there is seldom a formula for the \( n \)-th term

However, we can often set up one or more equations governing the sequence

Such equations are called difference equations

With a computer it is then very easy to generate the sequence by solving the difference equations

Difference equations have lots of applications and are very easy
to solve on a computer, but often complicated or impossible to solve for \( x_n \) (as a formula) by pen and paper!

The programs require only loops and arrays

Modeling interest rates

Put \( x_0 \) money in a bank at year 0. What is the value after \( N \) years

if the interest rate is \( p \) percent per year?

The fundamental information relates the value at year \( n \), \( x_n \), to
the value of the previous year, \( x_{n-1} \):

\[ x_n = x_{n-1} + \frac{p}{100}x_{n-1} \]

Solution by simulation:

\[ x_{n+1} = x_n + \frac{p}{100}x_n \]

The programs require only loops and arrays

Simulating the difference equation for interest rates

Simulate = solve math equations by repeating a simple procedure
(relation) many times (boring, but well suited for a computer)

Let us make a program for

\[ x_n = x_{n-1} + \frac{p}{100}x_{n-1} \]

from scitools.std import *

\# initial amount

x0 = 100

\# interest rate

p = 5

\# number of years

N = 4

\# solve the equation

x = zeros(len(index_set))

\# solution:

x[0] = x0

for n in index_set[1:]:

x[n] = x[n-1] + (p/100.0)*x[n-1]

print x

plot(index_set, x, 'ro', xlabel='years', ylabel='amount')

Note about the use of arrays

We store the \( x_n \) values in a NumPy array.

To compute \( x_n \), we only need one previous value, \( x_{n-1} \).

Thus, for the computations we do not need to store all
the previous values, i.e., we do not need any array, just two numbers:

\[ x_{\text{new}} = x_{\text{old}} \]

\[ x_{\text{new}} = x_{\text{old}} + \left(\frac{p}{100}\right)x_{\text{old}} \]

\[ x_{\text{new}} = x_{\text{new}} \]

\[ x_{\text{old}} = x_{\text{old}} \]

However, programming with an array \([x]\) is simpler, safer, and enabes plotting the sequence, so we will continue to use arrays in the examples
Daily interest rate

A more relevant model is to add the interest every day.
The interest rate per day is \( r = p/D \) if \( p \) is the annual interest rate and \( D \) is the number of days in a year.

A common model in business applies \( D = 360, \) but \( n \) counts exactly (all days).

New model:

\[
x_n = x_{n-1} + \frac{r}{100} x_{n-1}
\]

How can we find the number of days between two dates?

```python
>>> import datetime
>>> date1 = datetime.date(2007, 8, 3) # Aug 3, 2007
>>> date2 = datetime.date(2008, 8, 4) # Aug 4, 2008
>>> diff = date2 - date1
>>> print diff.days
365
```

Payback of a loan

A loan \( I \) is paid back with a fixed amount \( I/N \) every month over \( N \) months + the interest rate of the loan.

Let \( p \) be the annual interest rate and \( p/12 \) the monthly rate.

Let \( x_n \) be the value of the loan at the end of month \( n \).

The fundamental relation from one year to the other:

\[
x_n = x_{n-1} + \frac{p}{12 \times 100} x_{n-1} - \left( \frac{p}{12 \times 100} \right) I
\]

which simplifies to

\[
x_n = x_{n-1} - \frac{I}{N}
\]

(The constant term \( I/N \) makes the equation nonhomogeneous, while the previous interest rate equation was homogeneous (all terms contain \( x_n \) or \( x_{n-1} \)).)

The programming is left as an exercise.

But the annual interest rate may change quite often...

This is problematic when computing by hand.

In the program, a varying \( p \) is easy to deal with.

Just fill an array \( p \) with correct annual interest rate for day no. \( n \), \( n=0, \ldots, N \) (this can be a bit challenging).

Modified program:

```python
p = zeros(len(index_set)) # fill p[n] for n in index_set
r = p/360.0 # daily interest rate
x = zeros(N+1, int) # x = zeros(len(index_set))

for n in index_set[1:]:
    x[n] = x[n-1] + (r[n]/100.0) * x[n-1]

x0 = 100 # initial amount
x = x + x
```

How can we find the number of days between two dates?

```python
>>> import datetime
>>> date1 = datetime.date(2007, 8, 3) # Aug 3, 2007
>>> date2 = datetime.date(2008, 8, 4) # Aug 4, 2008
>>> diff = date2 - date1
>>> print diff.days
365
```

How to make a living from a fortune (part 1)

We have a fortune \( F \) invested with an annual interest rate of \( p \) percent.

Every year we plan to consume an amount \( c_n \) (in counts years)

Let \( x_n \) be the development of our fortune.

A fundamental relation from one year to the other is

\[
x_n = x_{n-1} + \frac{p}{100} x_{n-1} - c_n
\]

Simplest possibility: keep \( c_n \) constant

Drawback: inflation demands \( c_n \) to increase...

How to make a living from a fortune (part 2)

Assume \( p \) percent inflation per year and that \( c_n \) is \( g \) percent of the interest the first year.

\( c_n \) then develops as money with interest rate \( I \), and \( x_n \) develops with rate \( p \) but with a loss \( c_n \) every year:

\[
x_n = x_{n-1} + \frac{p}{100} x_{n-1} - c_{n-1}, \quad x_0 = F, \quad c_n = \frac{g}{100} F
\]

This is a coupled system of two difference equations.

The programming is still simple: we update two arrays \( x[n], c[n] \) inside the loop.

Fibonacci numbers; mathematics

No programming or math course is complete without an example on Fibonacci numbers!

Fibonacci derived the sequence by modeling rat populations, but the sequence of numbers has a range of peculiar mathematical properties and has therefore attracted much attention from mathematicians.

The difference equation reads

\[
x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1, \quad x_1 = 1
\]

This is a homogeneous difference equation of second order (three levels: \( n, n-1, n-2 \)) -- this classification is important for mathematical solution technique, but not for simulation in a program.

Fibonacci numbers; program

```python
from scipy.linalg import solve
from numpy import zeros

N = int(sys.argv[1])
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
print x[N]
```
Fibonacci numbers and overflow (part 1)

- Run the program with $N = 50$:
  
  ```python
  N = int(sys.argv[1])
  x_0 = 1
  x_1 = 1
  while n <= N:
    x_2 = x_1 + x_0
    x_0 = x_1
    x_1 = x_2
    print 'x_%d = %d' % (n, x_2)
  ```

- The model for growth of money in a bank has a solution of the type $x_n = x_0 e^{r n}$, where $0 < r < 1$

- Exponential growth with limited resources

  The model for growth of money in a bank has a solution of the type $x_n = x_0 e^{r n}$, where $0 < r < 1$

  - Best: use Python scalars of type int – these automatically change to long when overflow in int
  - The long type in Python has arbitrarily many digits (as many as required in a computation)
  - Note: long for arrays is 64 bit integer (int64), while scalar long in Python is an integer with as "infinitely" many digits

Program with Python's long type for integers

- The program now avoids arrays and makes use of three int objects (which automatically changes to long when needed):

  ```python
  N = int(sys.argv[1])
  x_0 = 1
  x_1 = 1
  x_2 = x_1 + x_0
  print 'x_%d = %d' % (n, x_2)
  ```

- Can change int to long or int64 for array elements - now we can generate numbers up to $N = 91$ before we get overflow and garbage

- Can use float96 despite the fact that $x_n$ are integers (float gives only approximatively correct numbers) – now $N$ up to 23600 is possible

Exponential growth with limited resources

- The model for growth of money in a bank has a solution of the type $x_n = x_0 e^{r n}$

  This is exponential growth in time ($t$)

- Populations of humans, animals, and cells also exhibit the same type of growth as long as there are unlimited resources (space and food)

- The environment can only support a maximum number $M$ of individuals

- How can we model this?

- We shall introduce a logistic model

Modeling logistic growth

- Initially, when there are enough resources, the growth is exponential:

  $$ x_n = x_{n-1} \left( \frac{r}{M} \right)^{n-1} $$

- The growth rate $r$ must decay to zero as $x_n$ approaches $M$

- A very simple $r(n)$ function with this behavior is

  $$ r(n) = \frac{x_n}{M} $$

- Observe that $r(n) \approx 0$ for small $n$ when $x_n << M$, and $r(n) \to 0$ as $x_n \to M$ and $n$ is big

- The model for limited growth, called logistic growth, is then

  $$ x_n = x_{n-1} + \frac{r}{M} x_{n-1} - \frac{x_{n-1}}{M} $$

The factorial as a difference equation

- The factorial $n!$ is defined as $n(n-1)(n-2) \cdots 1 (0! = 1)$

- The following difference equation has $n!$ as solution and can be used to compute the factorial:

  $$ x_n = n x_{n-1}, \quad x_0 = 1 $$

Taylor series as difference equations

- The Taylor series for $e^r$ reads

  $$ e^r = \sum_{n=0}^{\infty} \frac{r^n}{n!} $$

- We can formulate this series as two coupled difference equations (and solving these difference equations is (probably) the most efficient way to compute the Taylor series):!

  $$ a_n = \frac{r}{n-1}, \quad a_0 = 1 $$

  $$ e_n = a_{n-1} + a_n, \quad e_0 = 1 $$

- See the book for how to solve the difference equations by hand and show that the solution is the Taylor series for $e^r$
Newton’s method for finding zeros

Newton’s method for solving \( f(x) = 0 \) reads

\[
X_n = X_{n-1} - \frac{f(X_{n-1})}{f'(X_{n-1})}, \quad x_0 \text{ given}
\]

This is a (nonlinear!) difference equation

As \( n \to \infty \), we hope that \( x_n \to x \), where \( x \) solves \( f(x) = 0 \)

Now we will not simulate \( N \) steps, because we do not know how large \( N \) must be in order to have \( x_n \) as close to the exact solution \( x \) as we want.

The program is therefore a bit different: we simulate the difference equation as long as \( f(x) > \epsilon \), where \( \epsilon \) is small.

However, Newton’s method may (easily) diverge, so to avoid simulating forever, we stop when \( n > N \)

A better program for Newton’s method

Only one \( f(x) \) call in each iteration, optional storage of \((x, f(x))\) values during the iterations, and float division:

\[
\text{def Newton(f, x, dfdx, epsilon=1.0E-7, N=100, store=False):}
\]

\[
f\_value = f(x)
\]

\[
\text{if store: info = [(x, f\_value)]}
\]

\[
\text{while abs(f\_value) > epsilon and n <= N:}
\]

\[
\quad x = x - \frac{f\_value}{dfdx(x)}
\]

\[
\quad f\_value = f(x)
\]

\[
\text{if store: info = info.append((x, f\_value))}
\]

\[
\text{return x, info}
\]

\[
\text{else: return x, n, f\_value}
\]

Application of Newton’s method

Example: solve \( x - 0.14^2 \sin(\frac{\pi}{2}x) = 0 \)

\[
\text{from sympy import sin, pi, exp,untos}
\]

\[
\text{def g(x): return -0.1*exp(-0.1*x**2)*sin(pi/2*x) + exp(-0.1*x**2)*cos(pi/2*x)}
\]

\[
\text{def dg(x): return -2*0.1*x*exp(-0.1*x**2)*sin(pi/2*x) + 2*0.1*x*exp(-0.1*x**2)*cos(pi/2*x)}
\]

\[
\text{def Newton(f, x, dfdx, epsilon=1.0E-7, N=100, store=False):}
\]

\[
\text{f_value = f(x)}
\]

\[
\text{if store: info = [(x, f_value)]}
\]

\[
\text{while abs(f_value) > epsilon and n <= N:}
\]

\[
\quad x = x - float(f_value)/dfdx(x)
\]

\[
\quad f_value = f(x)
\]

\[
\quad if store: info = info.append((x, f_value))
\]

\[
\text{return x, info}
\]

Results from this test problem

Start value 1.7:

Zero: 1.999999999768449

Iteration 0: f(1.7)=3.40446

Iteration 1: f(1.7)=3.40446

Iteration 2: f(1.7)=3.40446

This works fine!

Start value 3:

Zero: 42.49723316011362

Iteration 1: f(3)=0.0981146

Iteration 2: f(3)=0.0981146

WHAT???

Lesson learned: Newton’s method may work fine or give wrong results! You need to understand the method to interpret the results!

Programming with sound

Sound on a computer = sequence of numbers

Example: A 440 Hz tone

\[
s(t) = A \sin(2 \pi f t)
\]

\[
f = 440
\]

This tone is a sine wave with frequency 440 Hz:

\[
s(t) = \text{A sin}(2 \pi f t)
\]

\[
f = 440
\]

On a computer we represent \( s(t) \) as a sequence of numbers:

\[
f(t) \text{ is evaluated twice in each pass of the loop – only one evaluation is strictly necessary (can store the value in a variable and reuse it)}
\]

Note: \( f(t) / df(t) \) can give integer division

Note: it could be handy to have an option for storing the \( x \) and \( f(x) \) values in each iteration (for plotting or printing a convergence table)

Making a sound file with single tone (part 1)

\[
r: \text{ sampling rate (samples per second, default 44100)}
\]

\[
f: \text{ frequency of the tone}
\]

\[
x: \text{ duration of the tone (seconds)}
\]

\[
e: \text{ Sampled sine function for this tone:}
\]

\[
x_n = A \sin \left( \frac{2\pi fn}{r} \right), \quad n = 0, 1, \ldots, n - r
\]

\[
\text{Code (we use descriptive names: frequency=f, length=x, amplitude=A, sample_rate=r):}
\]

\[
\text{import numpy}
\]

\[
\text{def note(frequency, length, amplitude=1, sample_rate=44100):}
\]

\[
\text{time_points = numpy.linspace(0, length, length+sample\_rate)}
\]

\[
\text{data = amplitude\_data}
\]

\[
\text{return data}
\]

Making a sound file with single tone (part 2)

We have data as an array with float and unit amplitude

\[
\text{Sound data in a file should have 2-byte integers (int16) as data elements and amplitudes up to 2^{15} – 1 (max value for int16 data)}
\]

\[
\text{data = note(440, 2)}
\]

\[
\text{data = data.astype(int16)}
\]

\[
\text{max\_amplitude = 2^{15} - 1}
\]

\[
\text{data = max\_amplitude\_data}
\]

\[
\text{import scitools.sound}
\]

\[
\text{scitools.sound.write(data, 'Atom.wav')}
\]

\[
\text{scitools.sound.play('Atom.wav')}
\]
Reading sound from file

Let us read a sound file and add echo

Sound = array s[n]

Echo means to add a delay of the sound

\[ \text{Echo: } e[n] = \beta s[n] + (1-\beta)s[n-b] \]

def add_echo(data, beta=0.8, delay=0.002, sample_rate=44100):
    newdata = data.copy()
    shift = int(delay*sample_rate) # b (math symbol)
    for i in xrange(shift, len(data)):
        newdata[i] = beta*data[i] + (1-beta)*data[i-shift]
    return newdata

Load data, add echo and play:

data = scitools.sound.read(filename)
data = data.astype(float)
data = add_echo(data, beta=0.6)
data = data.astype(int16)
scitools.sound.play(data)

Playing many notes

Each note is an array of samples from a sine with a frequency corresponding to the note

Assume we have several note arrays data1, data2, ...

The start of “Nothing Else Matters” (Metallica):

\[
\begin{align*}
    E1 &= \text{note}(164.81, .5) \\
    G &= \text{note}(392, .5) \\
    B &= \text{note}(493.88, .5) \\
    E2 &= \text{note}(659.26, .5) \\
    \text{intro} &= \text{numpy.concatenate}(E1, G, B, E2, G) \\
    \text{song} &= \text{numpy.concatenate(intro, intro, ...)} \\
    \text{scitools.sound.play(song)} \\
    \text{scitools.sound.write(song, 'tmp.wav')}
\end{align*}
\]

Summary of difference equations

Sequence: \( x_0, x_1, x_2, \ldots, x_n, \ldots \)

Difference equation: relation between \( x_n, x_{n-1} \), and maybe \( x_{n-2} \)
(or more terms in the "past") + known start value \( x_0 \) (and more values \( x_1, \ldots \) if more levels enter the equation)

Solution of difference equations by simulation:

For \( n \) in index_set[1:]:
    \( x[n] = \text{formula involving } x[n-1] \)

Can have (simple) systems of difference equations:

For \( n \) in index_set[1:]:
    \( x[n] = \text{formula involving } y[n-1] \)
    \( y[n] = \text{formula involving } x[n-1] \) and \( a[n] \)

Taylor series and numerical methods such as Newton’s method can be formulated as difference equations, often resulting in a good way of programming the formulas

Module file: soundeq.py

Look at files/soundeq.py for complete code. Try it out in these examples:

Unix/DOS> python soundeq.py oscillations 40
Unix/DOS> python soundeq.py logistic 100

Try to change the frequency range from 200 to 400.

Summarizing example: music of sequences

Given a \( x_0, x_1, x_2, \ldots, x_n, \ldots \)

Can we listen to this sequence as "music"?

Yes, we just transform the \( x_n \) values to suitable frequencies and use the functions in scitools.sound to generate tones

We will study two sequences:

\[
x_n = e^{-4n/N} \sin(8\pi n/N)
\]

and

\[
x_n = x_{n-1} + qx_{n-1} \left( 1 - x_{n-1} \right)
\]

The first has values in \([-1, 1]\), the other from \( x_0 = 0.01 \) up to around 1

Transformation from "unit" \( x_n \) to frequencies:

\[
y_n = 440 + 200x_n
\]

(first sequence then gives tones between 240 Hz and 640 Hz)