5mm.

Sequences and Difference Equations (Appendix A)

Hans Petter Langtangen

Simula Research Laboratory

University of Oslo, Dept. of Informatics

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

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

$$1,\ \frac{1}{2},\ \frac{1}{3},\ \frac{1}{4},\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \frac{1}{n+1}$$

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

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

Finite and infinite sequences

- Infinite sequences have an infinite number of terms ($n \rightarrow \infty$)
- 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

$$x_0, x_1, \ldots, x_{20}$$

Difference equations

- For sequences occuring 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:
  - start with x_0 (known)
  - compute $x_1 = x_0 + 0.01px_0 = (1+0.01p)x_0$
  - compute $x_2 = x_1 + 0.01px_1 = (1+0.01p)^2x_0$
  - ...continue this boring procedure...
  - find that $x_n = (1+0.01p)^n x_0$
- ...which is what you learned in high school
- That is: we can solve this difference equation by hand

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 *
x0 = 100                      # initial amount
p = 5                         # interest rate
N = 4                         # number of years
index_set = range(N+1)
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_old = x0
for n in index_set[1:]:
    x_new = x_old + (p/100.)*x_old
    x_old = x_new   # x_new becomes x_old at next step
```

- However, programming with an array `x[n]` is simpler, safer, and enables 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 exact (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?

```
>>> 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
367
```


Program for daily interest rate

```
from scitools.std import *
x0 = 100                         # initial amount
p = 5                            # annual interest rate
r = p/360.0                      # daily interest rate
import datetime
date1 = datetime.date(2007, 8, 3)
date2 = datetime.date(2011, 8, 3)
diff = date2 - date1
N = diff.days
index_set = range(N+1)
x = zeros(len(index_set))

# solution:
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (r/100.0)*x[n-1]
print x
plot(index_set, x, 'ro', xlabel='days', ylabel='amount')
```


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,...,N` (this can be a bit challenging)
- Modified program:

```
p = zeros(len(index_set))
# fill p[n] for n in index_set

r = p/360.0                      # daily interest rate
x = zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (r[n-1]/100.0)*x[n-1]
```


Payback of a loan

- A loan L is paid back with a fixed amount L/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 month to the other:

$$x_n = x_{n-1} + \frac{p}{12 \cdot 100} x_{n-1} - \left(\frac{p}{12 \cdot 100} x_{n-1} + \frac{L}{N}\right)$$

  which simplifies to

$$x_n = x_{n-1} - \frac{L}{N}$$

- (The constant term L/N makes the equation *nonhomogeneous*, while the previous interest rate equation was *homogeneous* (all terms contain x_n or x_{n-1}))
- The program is left as an exercise

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 (n 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 I percent inflation per year and that c_0 is q 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:

$$\begin{aligned} x_n &= x_{n-1} + \frac{p}{100} x_{n-1} - c_{n-1}, \quad x_0 = F,\ c_0 = \frac{pq}{10^4} F \\ c_n &= c_{n-1} + \frac{I}{100} c_{n-1} \end{aligned}$$

- 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

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

- 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

- Program:

```
N = int(sys.argv[1])
from numpy import zeros
x = zeros(N+1, int)
x[0] = 1
x[1] = 1
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
    print n, x[n]
```

Fibonacci numbers and overflow (part 1)

- Run the program with $N = 50$:

```
2 2
3 3
4 5
5 8
6 13
...
45 1836311903
Warning: overflow encountered in long_scalars
46 -1323752223
```

- 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

Fibonacci numbers and overflow (part 2)

- Best: use Python scalars of type `int` – these automatically changes 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):

```
import sys
N = int(sys.argv[1])
xnm1 = 1                          # "x_n minus 1"
xnm2 = 1                          # "x_n minus 2"
n = 2
while n <= N:
    xn = xnm1 + xnm2
    print 'x_%d = %d' % (n, xn)
    xnm2 = xnm1
    xnm1 = xn
    n += 1
```

- Run with $N = 200$:

```
x_2 = 2
x_3 = 3
...
x_198 = 173402521172797813159685037284371942044301
x_199 = 280571172992510140037611932413038677189525
x_200 = 453973694165307953197296969697410619233826
```

- Can simulate until the computer's memory becomes too small to store all the digits...

Exponential growth with limited resources

- The model for growth of money in a bank has a solution of the type

$$x_n = x_0 C^n \quad (= x_0 e^{n \ln C})$$

This is exponential growth in time (n)
- 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:

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

- The growth rate r must decay to zero as x_n approaches M
- A very simple $r(n)$ function with this behavior is

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

- Observe that $r(n) \approx \varrho$ for small n when $x_n << M$, and $r(n) \rightarrow 0$ as $x_n \rightarrow M$ and n is big
- The model for limited growth, called logistic growth, is then

$$x_n = x_{n-1} + \frac{\varrho}{100} x_{n-1} \left(1 - \frac{x_{n-1}}{M}\right)$$

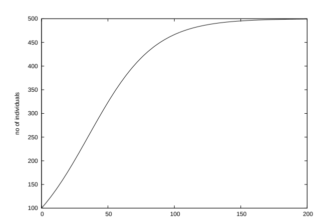
The evolution of logistic growth

- In a program it is easy to introduce logistic instead of exponential growth, just replace

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

by

```
x[n] = x[n-1] + (rho/100.0)*x[n-1]*(1 - x[n-1]/float(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 $x_n = n!$ 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^x reads

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

- 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!):

$$\begin{aligned} a_n &= \frac{x}{n} a_{n-1}, \quad a_0 = 1 \\ e_n &= e_{n-1} + a_n, \quad e_0 = 1 \end{aligned}$$

- See the book for how to solve the difference equations by hand and show that the solution is the Taylor series for e^x

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 $x_n \to x_s$, where x_s solves $f(x_s) = 0$
- 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_s as we want
- The program is therefore a bit different: we simulate the difference equation as long as $f(x) > \epsilon$, where ϵ is small
- However, Newton's method may (easily) diverge, so to avoid simulating forever, we stop when $n > N$

A program for Newton's method

- A simple program can be

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100):
    n = 0
    while abs(f(x)) > epsilon and n <= N:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, n, f(x)
```

- Note: $f(x)$ 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(x)/dfdx(x)` 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)

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:

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100,
           store=False):
    f_value = f(x)
    n = 0
    if store: info = [(x, f_value)]
    while abs(f_value) > epsilon and n <= N:
        x = x - float(f_value)/dfdx(x)
        n += 1
        f_value = f(x)
        if store: info.append((x, f_value))
    if store:
        return x, info
    else:
        return x, n, f_value
```


Application of Newton's method

Example: solve $e^{-0.1x^2}\sin(\frac{\pi}{2}x) = 0$

```
from math import sin, cos, exp, pi
import sys

def g(x):
    return exp(-0.1*x**2)*sin(pi/2*x)

def dg(x):
    return -2*0.1*x*exp(-0.1*x**2)*sin(pi/2*x) + \
           pi/2*exp(-0.1*x**2)*cos(pi/2*x)

x0 = float(sys.argv[1])
x, info = Newton(g, x0, dg, store=True)
print 'zero:', x
for i in range(len(info)):
    print 'Iteration %3d: f(%g)=%g' % \
          (i, info[i][0], info[i][1])
```


Results from this test problem

- Start value 1.7:

```
zero: 1.999999999768449
Iteration  0: f(1.7)=0.340044
Iteration  1: f(1.99215)=0.00828786
Iteration  2: f(1.99998)=2.53347e-05
Iteration  3: f(2)=2.43808e-10
```

  This works fine!
- Start value 3:

```
zero: 42.49723316011362
Iteration  0: f(3)=-0.40657
Iteration  1: f(4.66667)=0.0981146
Iteration  2: f(42.4972)=-2.59037e-79
```

  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
- This tone is a sine wave with frequency 440 Hz:

$$s(t) = A\sin(2\pi f t), \quad f = 440$$

- On a computer we represent $s(t)$ by a discrete set of points on the function curve (exactly as we do when we plot $s(t)$)
- How many points (samples) along the curve do we need to use?
- CD quality needs 44100 samples per second

Making a sound file with single tone (part 1)

- r: sampling rate (samples per second, default 44100)
- f: frequency of the tone
- m: duration of the tone (seconds)
- Sampled sine function for this tone:

$$s_n = A\sin\left(2\pi f\frac{n}{r}\right), \quad n = 0, 1, \ldots, m \cdot r$$

- Code (we use descriptive names: `frequency`=f, `length`=m, `amplitude`=A, `sample_rate`=r):

```
import numpy
def note(frequency, length, amplitude=1,
         sample_rate=44100):
    time_points = numpy.linspace(0, length,
                                 length*sample_rate)
    data = numpy.sin(2*numpy.pi*frequency*time_points)
    data = amplitude*data
    return data
```


Making a sound file with single tone (part 2)

- We have `data` as an array with `float` and unit amplitude
- 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)

```
data = note(440, 2)
data = data.astype(numpy.int16)
max_amplitude = 2**15 - 1
data = max_amplitude*data
import scitools.sound
scitools.sound.write(data, 'Atone.wav')
scitools.sound.play('Atone.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

```
# 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`, ...:

```
# put data1, data2, ... after each other in a new array:
data = numpy.concatenate((data1, data2, data3, ...))
```

- The start of "Nothing Else Matters" (Metallica):

```
E1 = note(164.81, .5)
G = note(392, .5)
B = note(493.88, .5)
E2 = note(659.26, .5)
intro = numpy.concatenate((E1, G, B, E2, B, G))
...
song = numpy.concatenate((intro, intro, ...))
scitools.sound.play(song)
scitools.sound.write(song, 'tmp.wav')
```


Summary of difference equations

- Sequence: $x_0, x_1, x_2, \ldots, x_n, \ldots, x_N$
- 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, ... if more levels enter the equation)
- Solution of difference equations by simulation:

```
index_set = <array of n-values: 0, 1, ..., N>
x = zeros(N+1)
x[0] = x0
for n in index_set[1:]:
    x[n] = <formula involving x[n-1]>
```

- Can have (simple) systems of difference equations:

```
for n in index_set[1:]:
    x[n] = <formula involving x[n-1]>
    y[n] = <formula involving y[n-1] and x[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

Summarizing example: music of sequences

- Given a $x_0, x_1, x_2, \ldots, x_n, \ldots, x_N$
- 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

$$x_n = x_{n-1} + qx_{n-1}(1 - x_{n-1}), \quad x = x_0$$

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)

Module file: soundeq.py

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

```
Unix/DOS> python soundseq.py oscillations 40
Unix/DOS> python soundseq.py logistic 100
```

Try to change the frequency range from 200 to 400.