Programmeervaardigheden

Programming skills

Course Code :	**1002WETPRV**
Study domain:	**Computer Science**
Academic year:	2014-2015
Semester:	Semester: 2nd semester
Contact hours:	40
Credits:	4
Study load (hours):	112
Contract restrictions:	No contract restriction
Language of instruction :	Dutch
Exam period:	exam in the 2nd semester
Lecturer(s)	Hans Vangheluwe

2de Semester 2013-2014

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

Universiteit Antwerpen

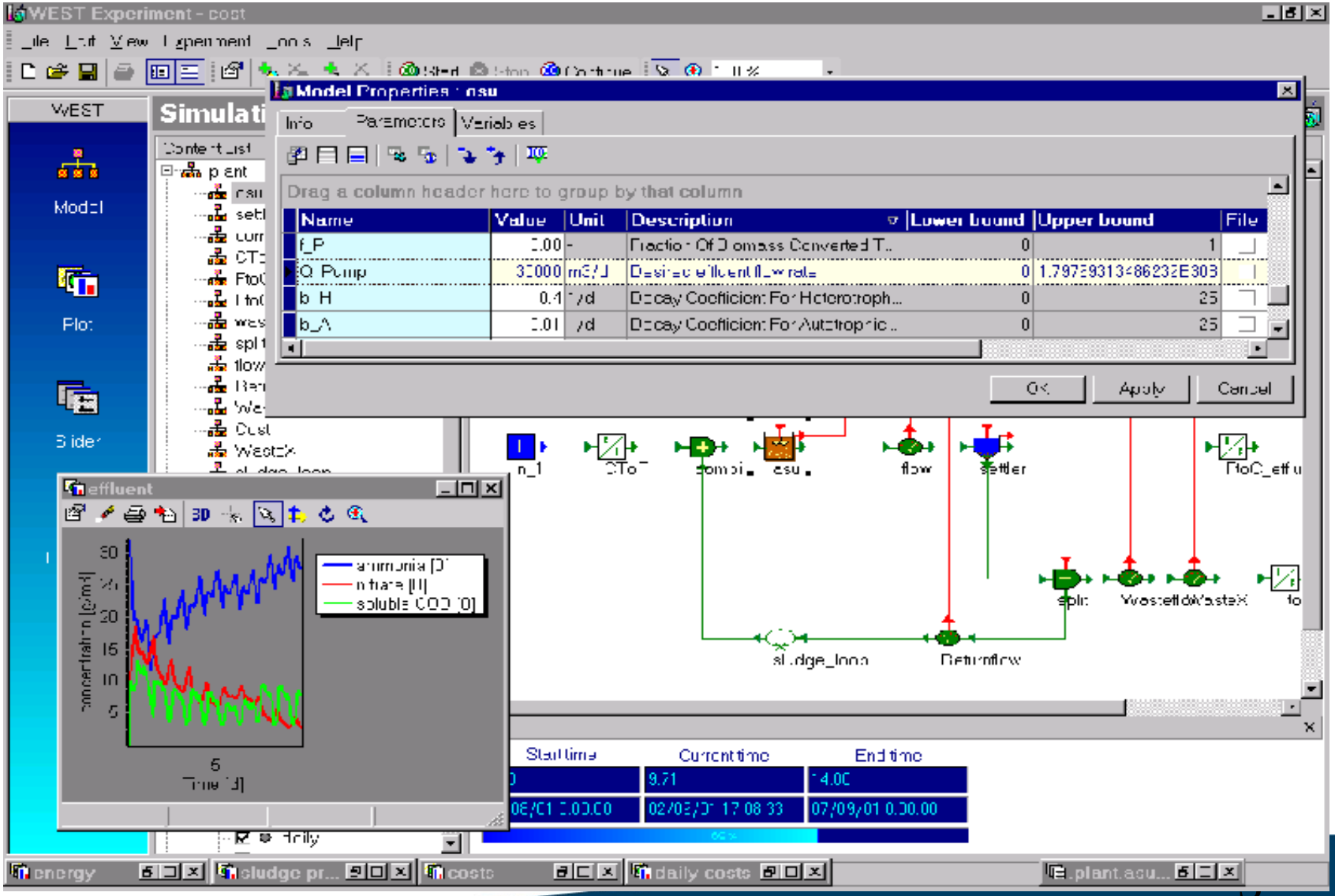
Hans Vangheluwe (Prof.)
M.G.116
Hans.Vangheluwe@uantwerpen.be

Bart Meyers (Praktijkassistent)
M.G.317
Bart.Meyers@uantwerpen.be

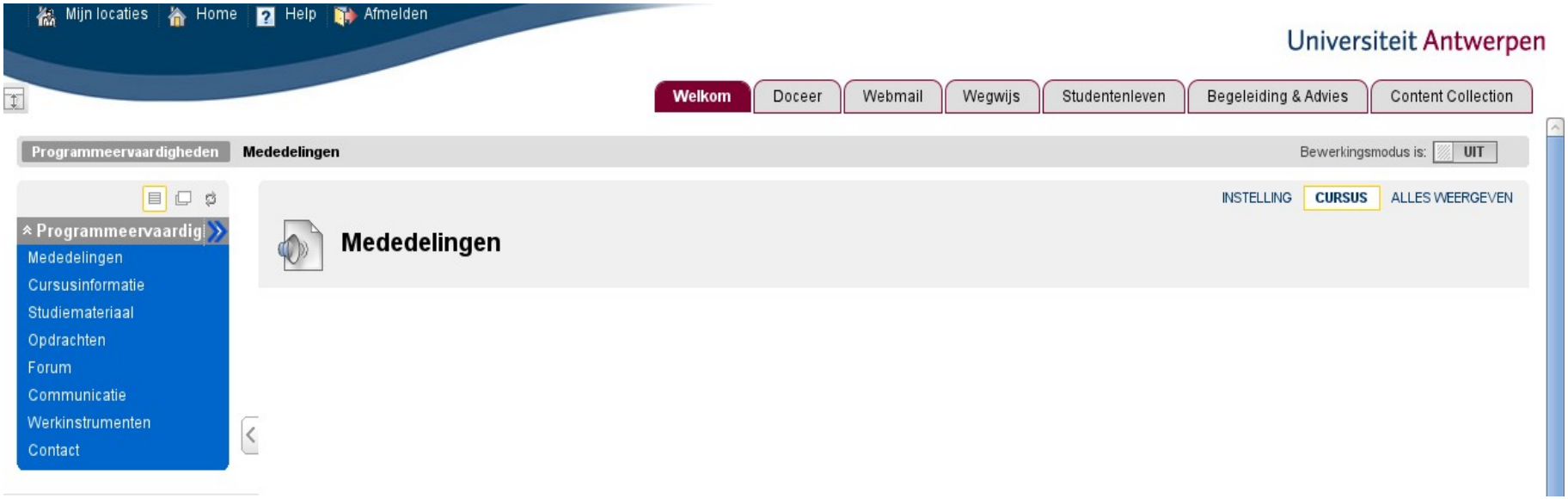
UNIVERSITEIT GENT

FBE

Department of
Mathematical Modelling,
Statistics and Bioinformatics



Online



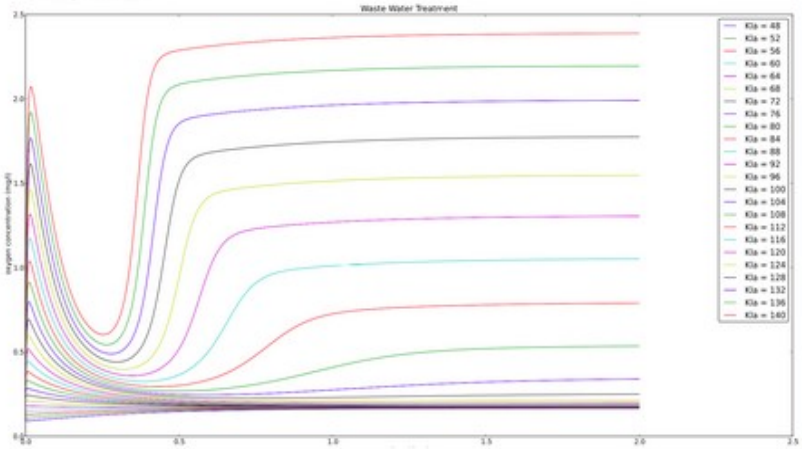
http://blackboard.uantwerpen.be

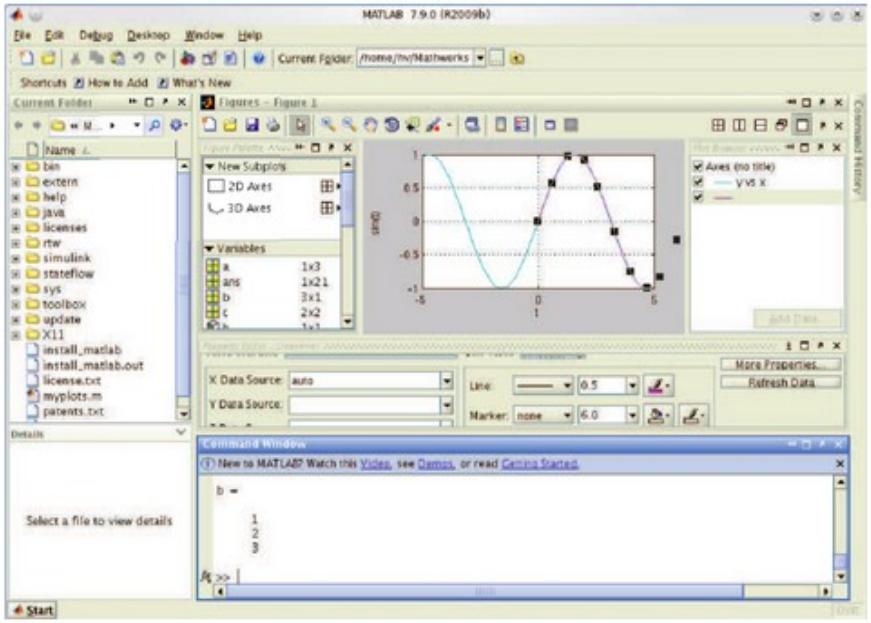
Programming Skills

On this page you will find information about the course "Programming Skills" (1002WETPRV) for the second semester of the 2013-2014 academic year at the University of Antwerp.

This page is written in English for the benefit of foreign Erasmus students.

$$\begin{cases} \frac{dS_o}{dt} = Kla(S_o^* - S_o) + \frac{Q}{V}(S_o^{in} - S_o) - r_o \\ \frac{dS_s}{dt} = \frac{Q}{V}(S_s^{in} - S_s) - \frac{r_{growth}}{Y_H} \\ \frac{dX_{BH}}{dt} = \frac{Q}{V}(X_{BH}^{in} - X_{BH}) + r_{growth} - r_{decay} \\ r_o = \frac{1-Y_H}{Y_H} r_{growth} + (1 - f_P) r_{decay} \\ r_{growth} = \mu_{max} \frac{S_s}{K_s + S_s} \frac{S_o}{K_o + S_o} X_{BH} \\ r_{decay} = b_H X_{BH} \end{cases}$$





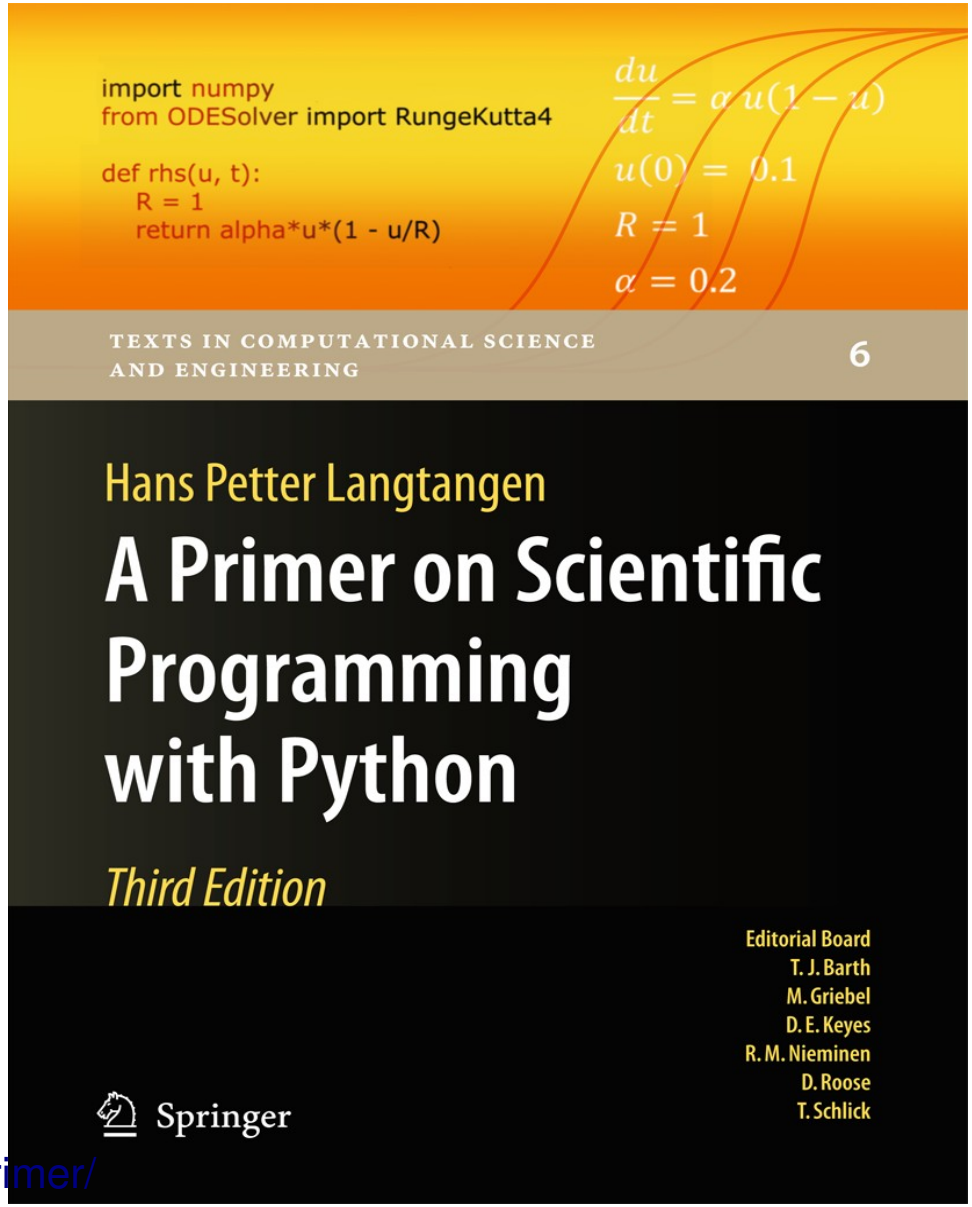
http://msdl.cs.mcgill.ca/people/hv/teaching/ProgrammingSkills/

Ultieme referentie is de les!

- Voor materiaal (ook **op bord**)
- Voor aankondigingen (korte versie op BlackBoard)

Geef **feedback**!

http://folk.uio.no/hpl/



resources: http://hplgit.github.io/scipro-primer/

Te verkrijgen op de cursusdienst (Groenenborger Campus)

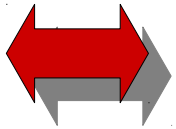
The only universal language ...

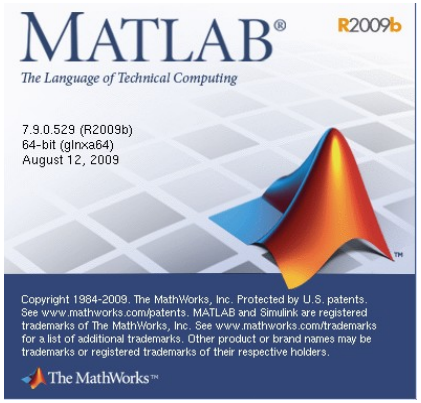
Nederlandstalig: lessen
Engelstalig: handboek, slides

Oefeningen/Examen/Vragen mogen in het Engels opgelost/beantwoord/gesteld worden

Why?

Termen vertalen is moeilijk/onmogelijk/zinloos
Continuiteit (slides, code, ...)
Wereldstandaard
Voorbereiding op arbeidsmarkt
Erasmus studenten





www.python.org

python 2.7

not (yet) python 3!

“scripting” talen
vs. C++, Java, …

www.mathworks.com

Programmeertaal

1989

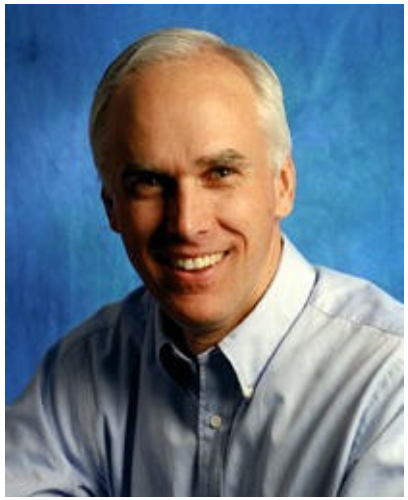
Matrix **Lab**oratory

1984

Guido Van Rossum

Cleve Moler

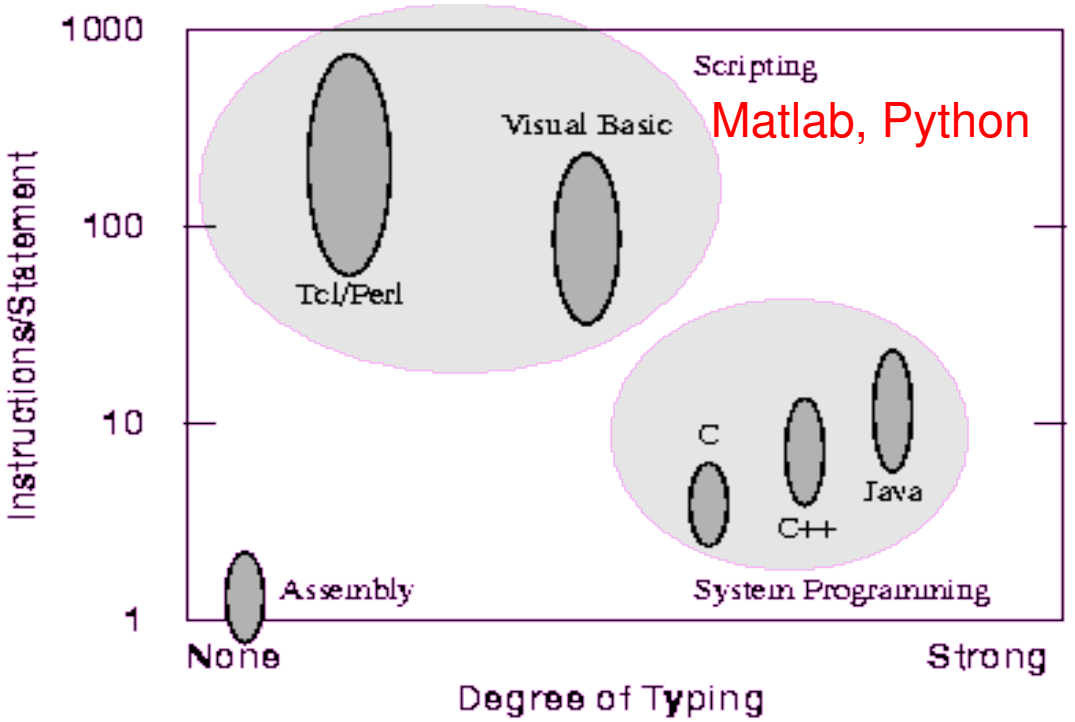
Jack Little



John Ousterhout

Scripting: Higher Level Programming for the 21st Century (IEEE Computer, March 1998)

http://www.stanford.edu/~ouster/cgi-bin/papers/scripting.pdf



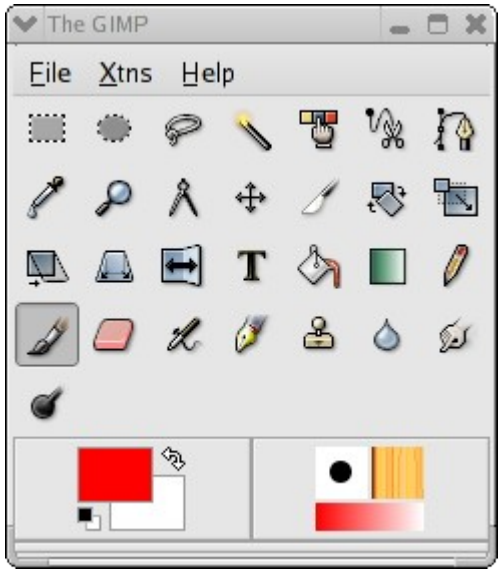
Boehm: programmer productivity LOC/day is **independent** of the language used !
Raise the **level of abstraction**: 1 LOC == many LOA.

Extending

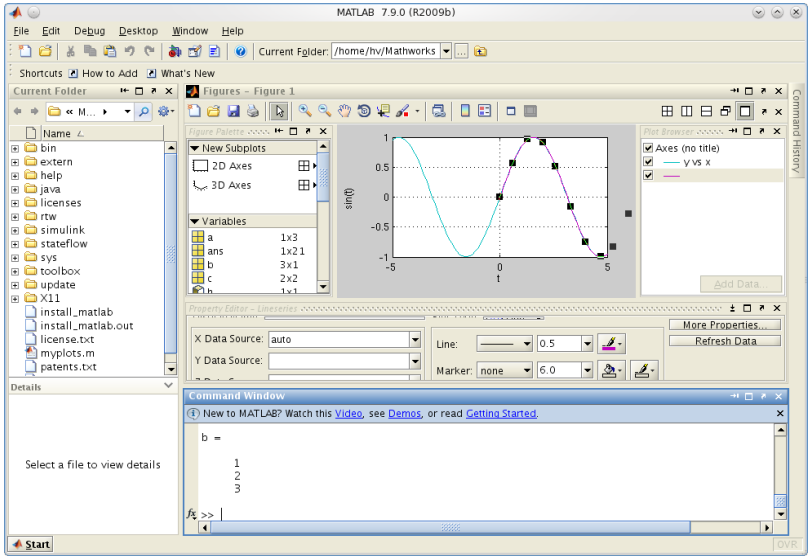
```
from Tkinter import *
root=Tk()
notice=Label(root, text="hello!")
notice.pack()
```

Embedding

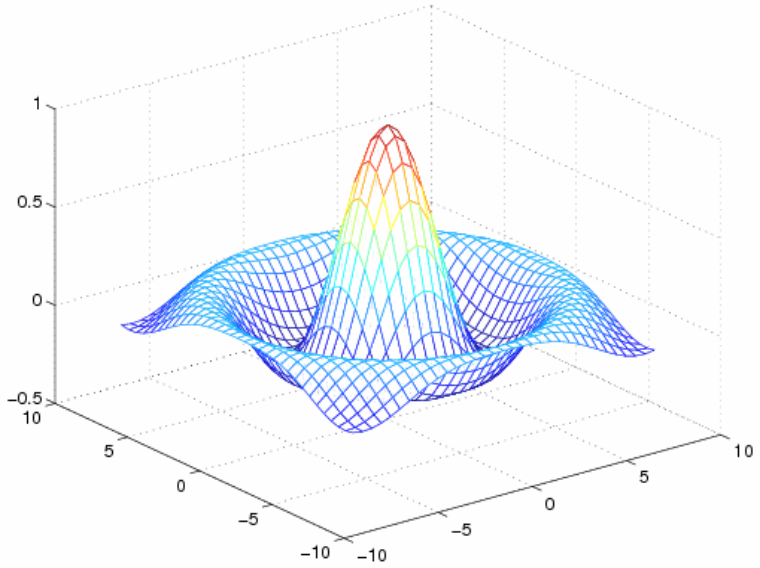
Gimp application, Python-Fu




```
dir()
dir(gimp)
gimp.get_foreground()
gimp.set_foreground(1.0,0.0,0.0,1.0)
```



MATLAB interactieve omgeving

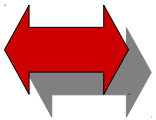


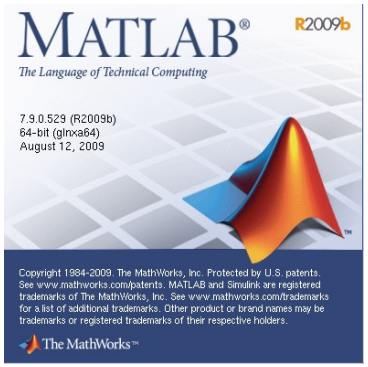
Grafisch (plotten, user interface)

```
classdef BankAccount < handle
   properties (Hidden)
      AccountStatus = 'open';
   end
% The following properties can be set only by class methods
   properties (SetAccess = private)
      AccountNumber
      AccountBalance = 0;
   end
% Define an event called InsufficientFunds
   events
      InsufficientFunds
   end
   methods
      function BA = BankAccount(AccountNumber,InitialBalance)
         BA.AccountNumber = AccountNumber;
         BA.AccountBalance = InitialBalance;
```

MATLAB programmeertaal
(matrix + if/loop/... + OO)

www.python.org





www.mathworks.com

Programmeertaal

public domain

geinterpreteerd

“glue” bibliotheken:
numeriek, plotten, ...

programmeren

Matrix **Lab**oratory

commercieel

geinterpreteerd

“glue” bibliotheken:
numeriek, plotten, ...

programmeren mogelijk

domein-specifieke toolboxes

Tentative Planning

> from course website
> gets updated during the term

Planning

Week	Date	Type	Room	Subject
1	Monday 10 February 13:45 - 15:45	Theory		No Class
1	Tuesday 11 February 10:45 - 12:45	Lab session		No Class
2	Monday 17 February 13:45 - 15:45	Theory	G.U.241	Course introduction
2	Tuesday 18 February 10:45 - 12:45	Lab session	G.US.103	Introduction to Python and MATLAB
3	Monday 24 February 13:45 - 15:45	Theory	G.U.241	Chapter 1: computing with formulas (types, variables, comments, formulas and formats)
3	Tuesday 25 February 10:45 - 12:45	Lab session	G.US.103	Debugging, Assignments 1
4	Monday 3 March 13:45 - 15:45	Theory	G.U.241	Chapter 2: loops and lists (while, for, lists)
4	Tuesday 4 March 10:45 - 12:45	Lab session	G.US.103	Assignments 2
5	Monday 10 March 13:45 - 15:45	Theory	G.U.241	Chapter 2: loops and lists (nested lists, slices, tuples)
5	Tuesday 11 March 10:45 - 12:45	Lab session	G.US.103	Assignments 3
6	Monday 17 March 13:45 - 15:45	Theory	G.U.241	Chapter 3: basic constructs (if, functions)
6	Tuesday 18 March 10:45 - 12:45	Lab session	G.US.103	Testing, Assignments 4, **Project 1**
7	Monday 24 March 13:45 - 15:45	Theory	G.U.241	Chapter 4: inputdata and modules (reading input, making modules)
7	Tuesday 25 March 10:45 - 12:45	Lab session	G.US.103	Assignments 5
8	Monday 31 March 13:45 - 15:45	Theory	G.U.241	Chapter 5: array computing and curve plotting (arrays, plotting)
8	Tuesday 1 April 10:45 - 12:45	Lab session	G.U.241	Assignments 6, **Project 2**
8	Sunday 6 April 23:55	Deadline		Project 1
	easter holiday			
9	Monday 21 April 13:45 - 15:45	Theory	G.U.241	Chapter 5: array computing and curve plotting (higher dimensional arrays)
9	Tuesday 22 April 10:45 - 12:45	Lab session	G.US.103	Assignments 7
9	To be determined	Defense	To be determined	Project 1
9	Sunday 27 April 23:55	Deadline		Project 2
10	Monday 28 April 13:45 - 15:45	Theory	G.U.241	Appendix A: sequences and difference equations
10	Tuesday 29 April 10:45 - 12:45	Lab session	G.US.103	Assignments 8, **Project 3**
10	To be determined	Defense	To be determined	Project 2
11	Monday 12 May 13:45 - 15:45	Theory		No Class
11	Tuesday 13 May 10:45 - 12:45	Lab session		No Class
12	Monday 12 May 13:45 - 15:45	Theory	G.U.241	Chapter 6: files, strings and dictionaries
12	Tuesday 13 May 10:45 - 12:45	Lab session		No Class, moved to Wednesday
12	Wednesday 14 May 10:45 - 12:45	Lab session	G.US.103	Assignments 9
13	Monday 19 May 13:45 - 15:45	Theory	G.U.241	Course conclusion
13	Tuesday 20 May 10:45 - 12:45	Lab session	G.US.103	Questions
13	Sunday 25 May 23:55	Deadline		Project 3
exam period	To be determined	Defense	To be determined	Project 3

Course Goals:

Learn basic programming using Python

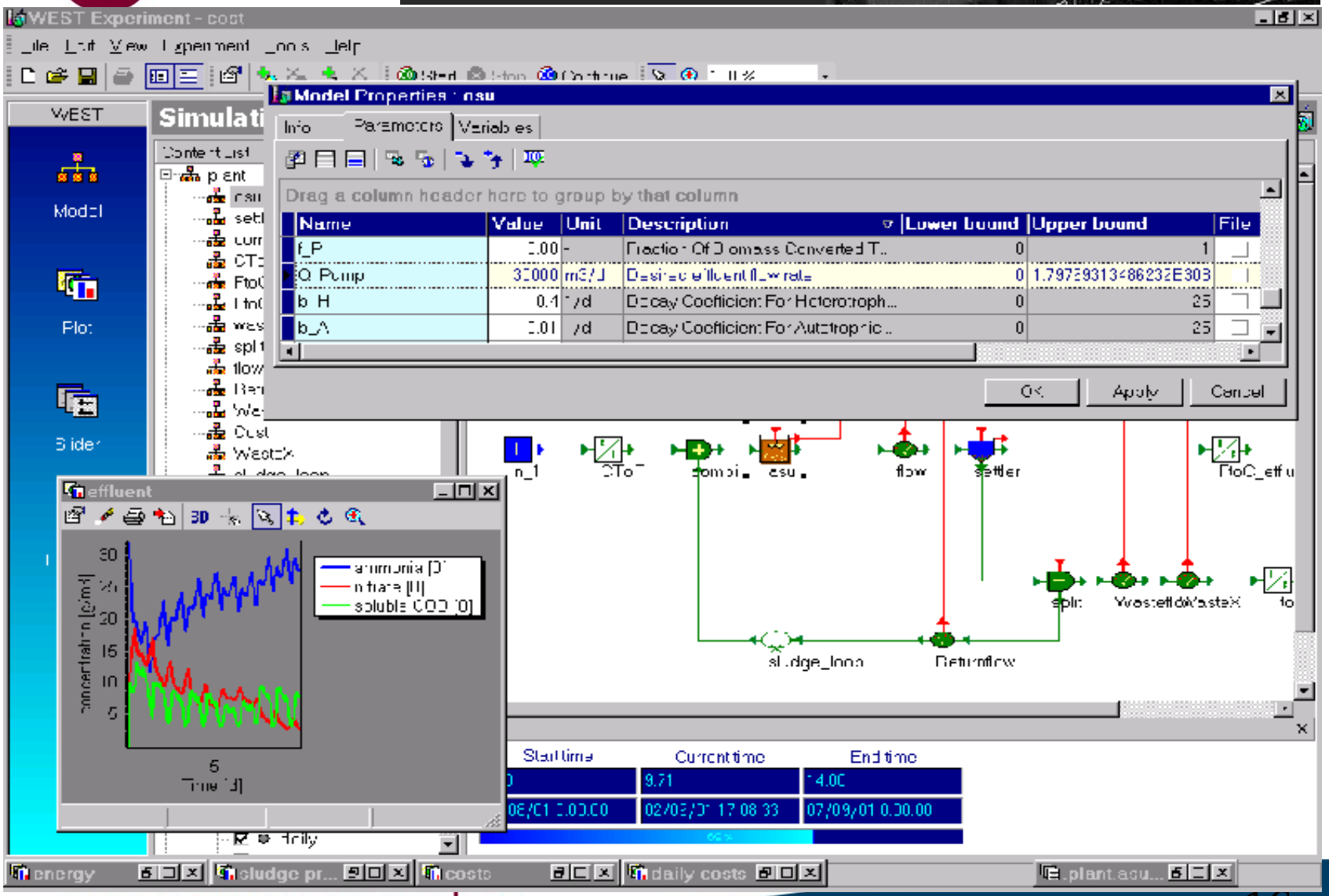
Learn to use (Python) libraries to solve problems
(mostly applied mathematics)

Become familiar with Matlab

Evaluation:

Assignments (individual).

Projects: programming, libraries, Matlab, report
In groups of (maximum) 2, individual evaluation

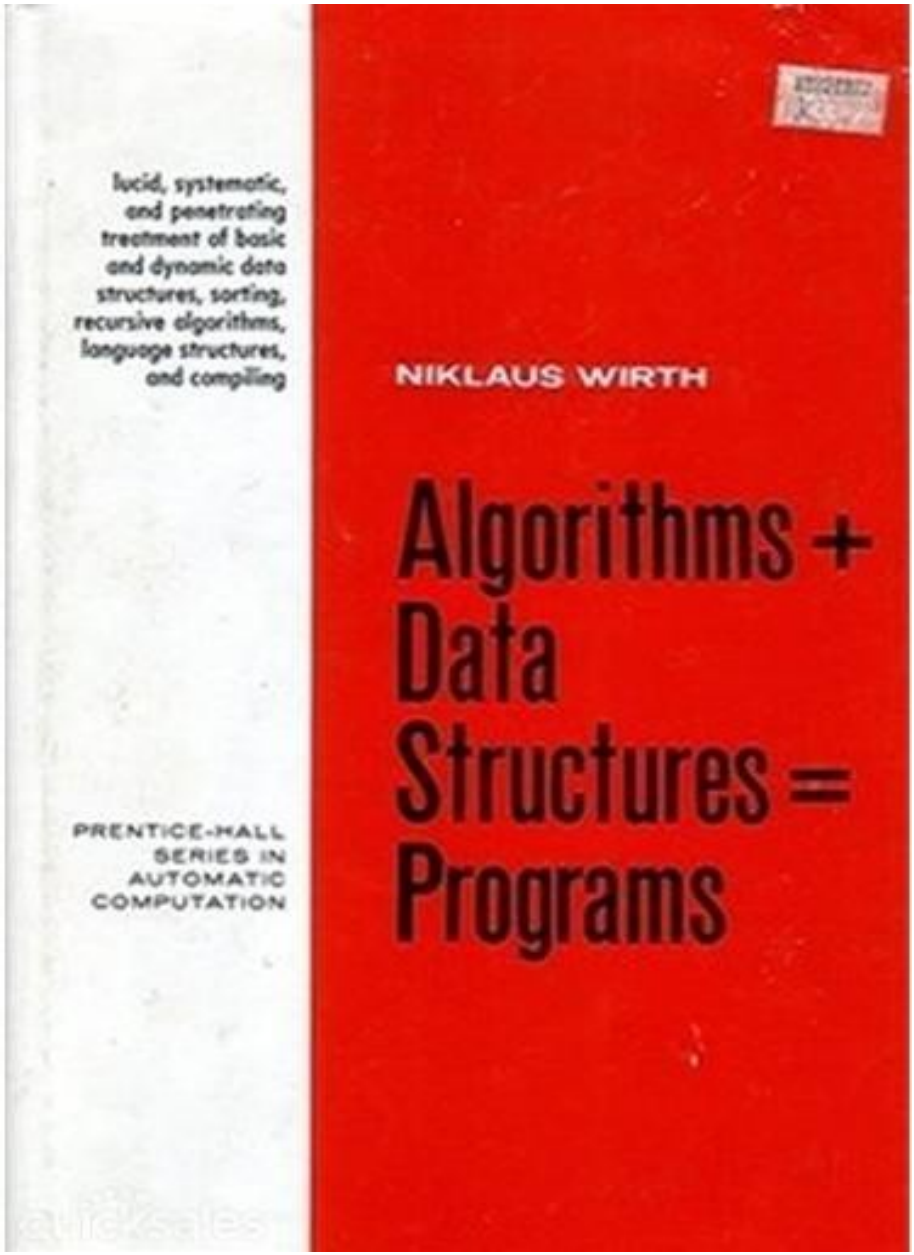

WEST Experiment - cost
WEST
Model
Plot
Model Properties
Info
Parameters
Variables
Drag a column header here to group by that column
Name
Value
Unit
Description
Lower bound
Upper bound
File
f_P
Q_Pump
b_H
b_A
OK
Apply
Cancel
effluent
Start time
Current time
End time
energy
sludge pr...
costs
daily costs
plant.asu...

problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

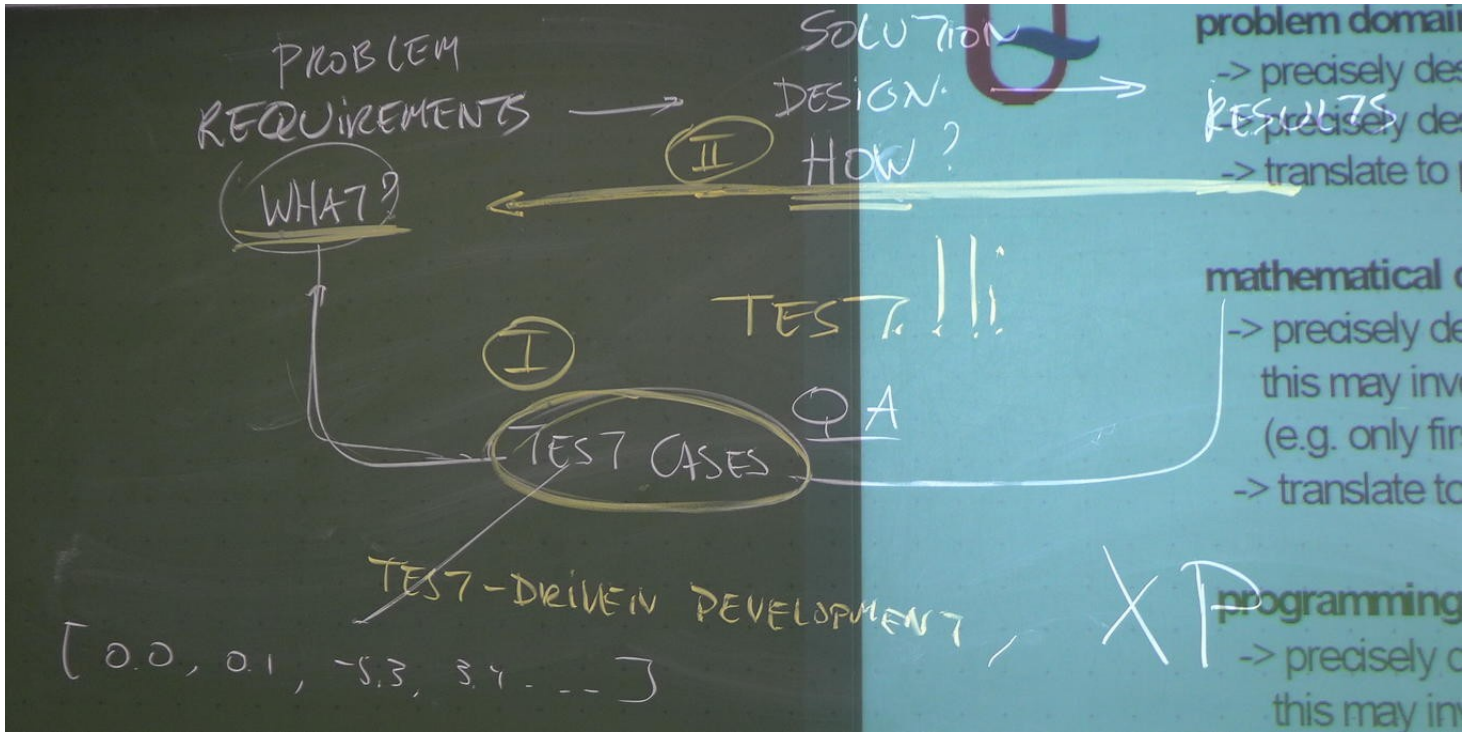
mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers)
-> design and implement program
-> test program

Niklaus Wirth's 1976 book →



test-driven development



Project (programming, libraries, Matlab, report)

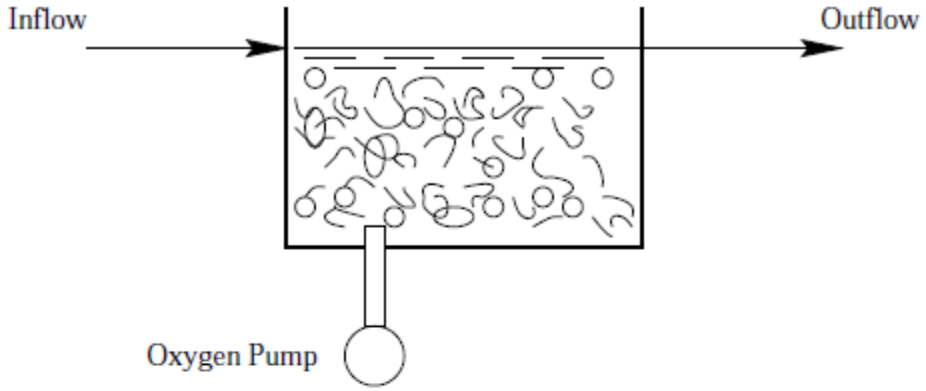


Figure 1: Aeration tank

The plant model we will consider is a simplified version of the IAWQ1 model standardized by the International association on Water Quality. The processes incorporated in the model are: growth of heterotrophic biomass and decay of heterotrophic biomass. The following three variables are crucial to the model: the oxygen concentration, the substrate concentration and the biomass concentration. In the context of modelling the effluent quality, the following model will be used:

$$\begin{cases} \frac{dS_o}{dt} = Kla(S_o^* - S_o) + \frac{Q}{V}(S_o^{in} - S_o) - r_o \\ \frac{dS_s}{dt} = \frac{Q}{V}(S_s^{in} - S_s) - \frac{r_{growth}}{Y_H} \\ \frac{dX_{BH}}{dt} = \frac{Q}{V}(X_{BH}^{in} - X_{BH}) + r_{growth} - r_{decay} \\ r_o = \frac{1-Y_H}{Y_H} r_{growth} + (1 - f_P) r_{decay} \\ r_{growth} = \mu_{max} \frac{S_o}{K_{so}+S_o} \frac{S_s}{K_{ss}+S_s} X_{BH} \\ r_{decay} = b_H X_{BH} \end{cases}$$

The University of Antwerp values **intellectual integrity**.

Students must understand the meaning and consequences of plagiarism, cheating and other academic offences.

You are encouraged to help each other formulate the ideas behind assignment/project problems, but each student is required to submit his or her own original work. Handing in work, include progam code, that is not your own, original work as if it is your own is plagiarism.

All re-use, collaboration, inspiration must be explicitly mentioned in the assignment/project.

Programmeervaardigheden

CH1: Computing with Formulas

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers)
-> design and implement program
-> test program

Programming a mathematical formula

- Learn programming through examples
- Example of a mathematical formula:

Position of a ball in vertical motion, starting at y = 0 at time t = 0

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

v_0: initial velocity

g: acceleration due to gravity

v_0, g are “parameters”

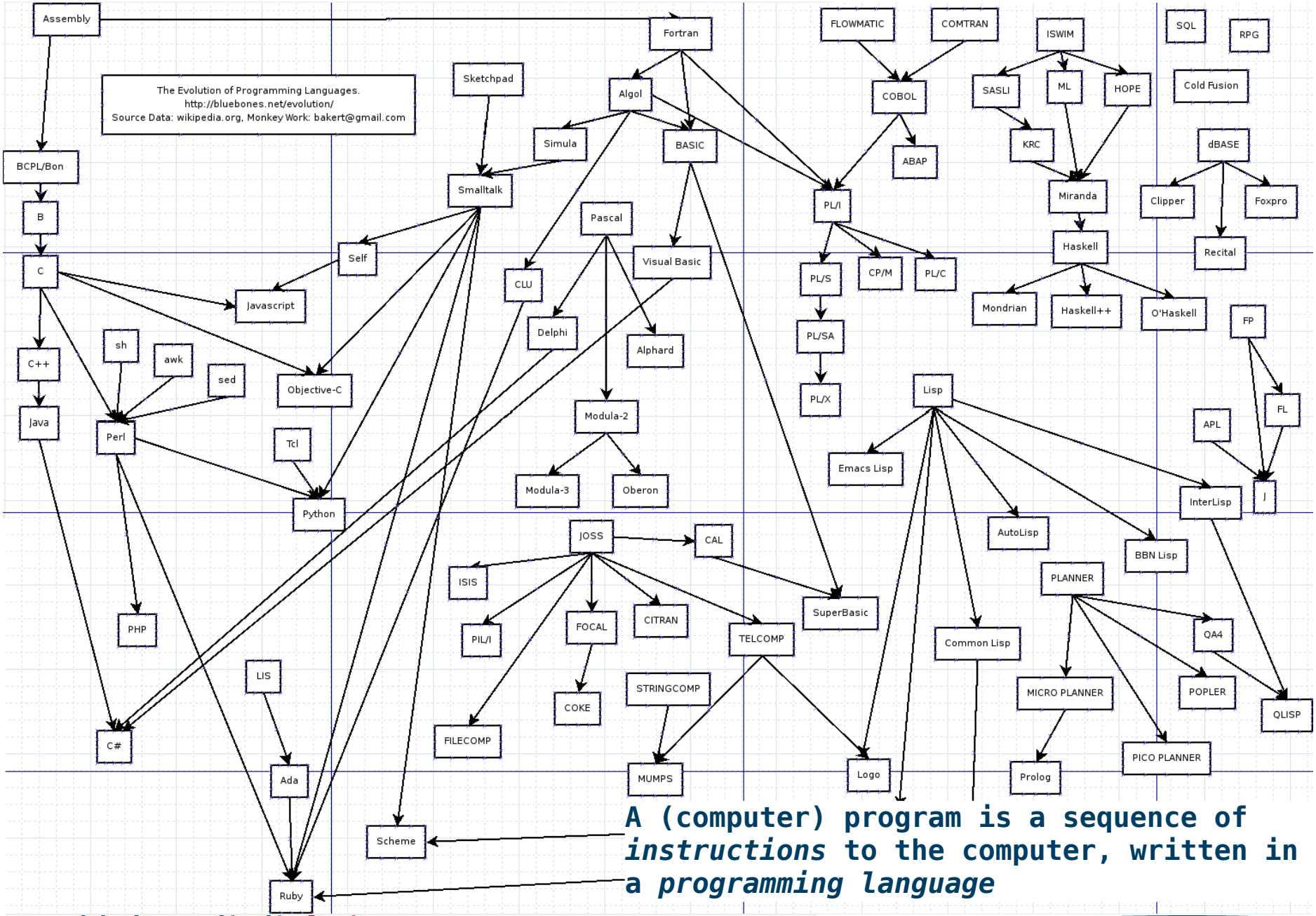
t is the “independent” (time) variable

y is the “state” variable

- Task: given v_0, g and t, compute y

parameters ...

Het woord parameter wordt op verschillende manieren uitgesproken, namelijk met de klemtoon op de eerste, tweede of derde lettergreep. Correct is de klemtoon op de tweede lettergreep (de tweede 'a') met een nevenklemtoon op de vierde lettergreep. Dit omdat het een woord is dat is samengesteld uit het Latijn (en Grieks). De regels van het Latijn bepalen dat de klemtoon valt op de derde lettergreep van achteren af. Dus: parámeter.



A (computer) program is a sequence of *instructions* to the computer, written in a *programming language*

The computer program

- Evaluate our formula for some specific values:

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

$v_0 = 5$
$g = 9.81$
$t = 0.6$

$$y = 5 \cdot 0.6 - \frac{1}{2} \cdot 9.81 \cdot 0.6^2$$

- The program in Python:

```
print(5*0.6 - 0.5*9.81*0.6**2)
```

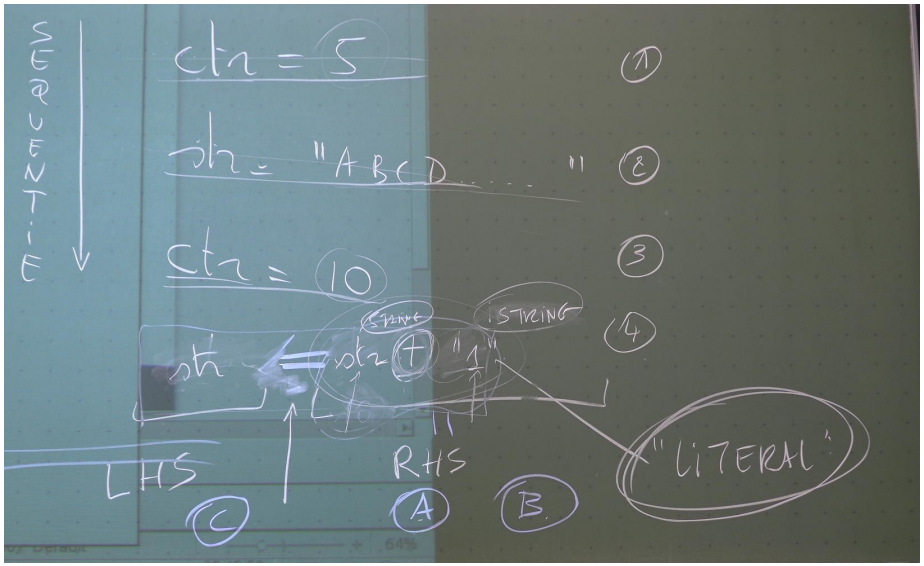
Variables

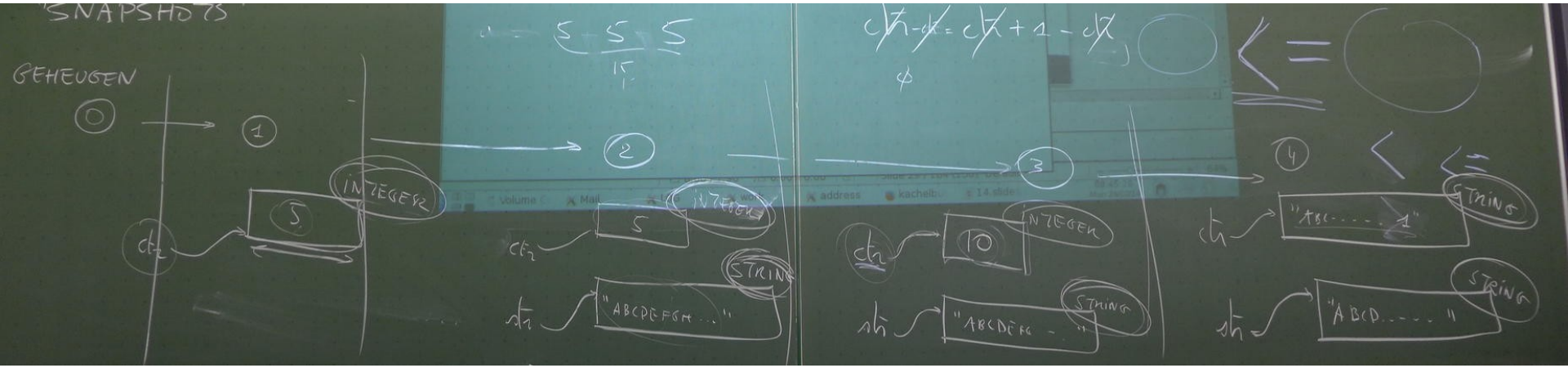
- We can use *variables* in computer programs:

```
v0 = 5
g  = 9.81
t  = 0.6
y  = v0*t – 0.5*g*t**2
print(y)
```

- A variable has a *name* associated with a *value* stored in computer *memory*
- Variable names are *case sensitive*, can contain a-z, A-Z, _ and 0-9 and may not start with a digit

a **program** and **snapshots** of its execution





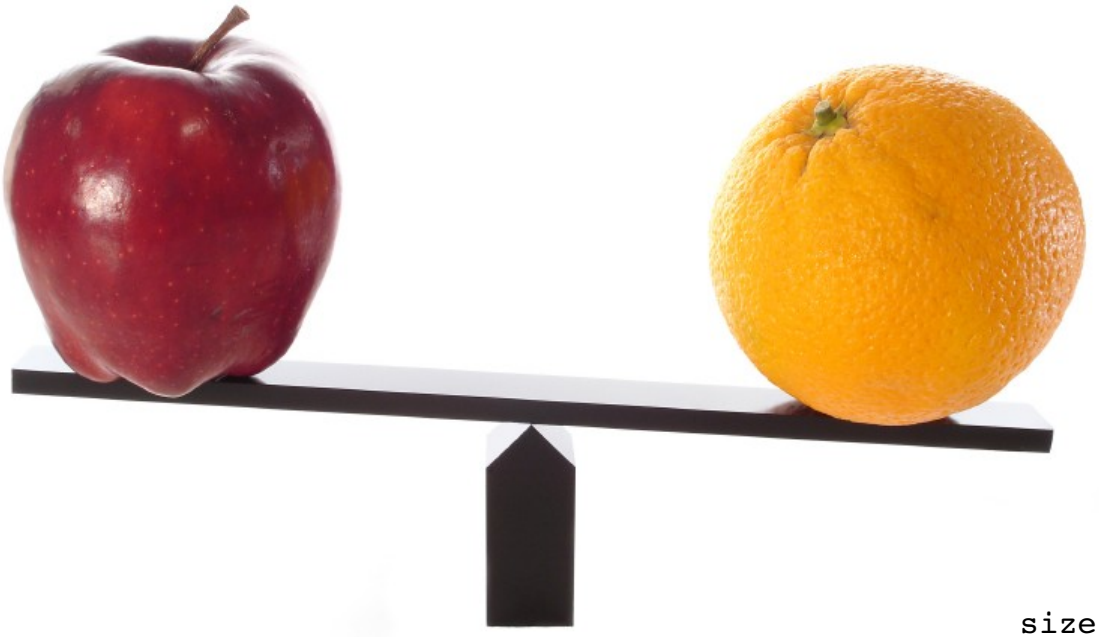
problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers)
-> design and implement program
-> test program

types

literals and *variables* have a ***type***
which in Python is not explicitly declared
and may change over time



```
size = “Large”
size = 123


# NOT explicitly declared!
# int size = 123
```

NumberOfTrees = 4

$$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$$

Python integers:

```
>>> import sys

>>> sys.maxint
 9223372036854775807

>>> -sys.maxint-1
-9223372036854775808

>>> 10**100
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000L
```

$\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$

1/49 * 49 = 1/51 * 51 = 1 (for elements of $\mathbb{R}$)

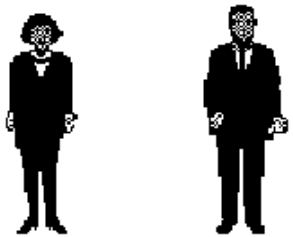
```
v1 = 1/49.0 * 49
v2 = 1/51.0 * 51
print('%.16f %.16f' % (v1, v2))
```

Output:

```
0.9999999999999999 1.0000000000000000
```

- Most real numbers are *not represented exactly* on a computer
- Real numbers and results of mathematical computations are only approximations!!!
- Some numbers can **not** be represented
- Beware when comparing: `if number == 0.0: ...`

types




```
gender = “Female”

# or

FEMALE = 1
MALE   = 2

gender = FEMALE
```

```
firstName = “Lisa”
lastName = “Simpson”
streetName = “Middelheimlaan”
streetNumber = 1
PostalCode = 2020
```

```
date =
“Mon Feb 24 01:05:08 CET 2014”

date_s = 1393200528
# seconds since 1 Jan 1970
```

types (of literals and variables)

```
>>> type(5)
<type 'int'>

>>> type(5.0)
<type 'float'>

>>> type(True)
<type 'bool'>

>>> type("hello")
<type 'str'>
```

```
>>> a = 5
>>> type(a)
<type 'int'>

>>> a = 5.0
>>> type(a)
<type 'float'>
```

types (of literals and variables)

```
>>> type(1+1j)
<type 'complex'>
```

```
>>> res = 1j*1j
>>> type(res)
<type 'complex'>
```

```
>>> type([1,2,3])
<type 'list'>
```

```
>>> from math import sin
>>> type(sin)
<type 'builtin_function_or_method'>
```

Reserved Words

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, with, while, yield

File Edit Format Run Options Windows Help

```
if a <= 5.0:
    for j in range(10):
        print(a+j)
```

Comments

- Comments are ignored by the computer and help to clarify the program to others:

```
# program to compute the height of a ball
# in vertical motion as a function of time
v0 = 5     # initial velocity
g = 9.81   # acceleration due to gravity
t = 0.6    # time
y = v0*t - 0.5*g*t**2 # vertical position
print(y)
```

- Explain design/code in comments!
- Choose meaningful variable names

```
vertical_position = v0*t - 0.5*g*t**2
print(vertical_position)
```

Capitalization Styles

```
vertical_position

verticalPosition

VerticalPosition


fidoDog = Dog()


CONSTANT = 12.34
```

camelCase

Output formatting

- `print()` *function* sends output to the screen:

```
print('this is just random text') # mind the quotes!
print 'this is just random text'  # no longer in python 3
print('a double quote "')
print("a single quote '")
print("'escaping' a special character \"")
print("newline \n and tabulating \t ...")
print(79864) # constant number
print(some_variable) # some_variable must be defined!
print(""" multi-line
New line, but not here\
output with triple quotes... """)
```

- Output for numbers can be formatted:

```
print('value of x = %.3f' % x)# 3 decimals
print('value of x = %.2e' % x)# 2 decimals, sci.not.
print('value of x = %f and escaped %%' % x)
```

Formats:

%s	a string
%d	an integer
%0xd	an integer padded with x leading zeros
%f	decimal notation with six decimals
%e	compact scientific notation, e in the exponent
%E	compact scientific notation, E in the exponent
%g	compact decimal or scientific notation (with e)
%G	compact decimal or scientific notation (with E)
%xz	format z right-adjusted in a field of width x
%-xz	format z left-adjusted in a field of width x
%.yz	format z with y decimals
%x.yz	format z with y decimals in a field of width x
%%	the percentage sign (%) itself

Statements

- *Programs* consist of *sequences* of *statements*, which instruct the computer to do something:

```
a = 1      # 1st statement (assignment)
b = 2; c = a + b # 2nd and 3rd statement (assignment)
print(c)   # 4th statement (print)
```

- Assignment statements: *evaluate* Right-Hand Side (RHS) first, THEN assign the result to the Left-Hand Side (LHS) *variable*:

```
myVar = 10; x = 7
x = myVar + x*3
```

Example: temperature conversion

- Compute Fahrenheit from Celsius temperature:

$$F = \frac{9}{5}C + 32$$

```
C = 21
F = (9/5)*C + 32  # use of parenthesis: cf. math
print(F)
```

- F should be 69.8, but output is 53 ?? (in python 2.7)
- Integer division!!

```
C = 21; F = (9.0/5)*C + 32; print(F)
```

Standard mathematical functions

- What if we need sin(x), cos(x), ln(x), … ?
- Available in Python `math` *module (library)*
- In general, modules contain useful functionality (*e.g.*, mathematical functions) and can be *imported* in other programs:

```
import math
r = math.sqrt(2)

# alternative:
from math import *
dir()
r = sqrt(2)

# selective:
from math import sin, cos, log
x = 1.2
print sin(x)*cos(x) + 4*log(x)  # log = ln (base e)
```

```
['__builtins__', '__doc__', '__name__',
'__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

"name-space pollution"

Dealing with Complex Numbers

```
>>> u = 2.5 + 3j # create a complex number
>>> v = 2 # this is an int
>>> w = u + v # complex + int
>>> w
(4.5+3j)
>>> a = -2
>>> b = 0.5
>>> s = a + b*1j # create a complex number from two floats
>>> s = complex(a, b) # alternative creation
>>> s
(-2+0.5j)
>>> s*w # complex*complex
(-10.5-3.75j)
>>> s/w # complex/complex
(-0.25641025641025639+0.28205128205128205j)
```

Dealing with Complex Numbers

```
>>> s.real
-2.0
>>> s.imag
0.5
```

```
>>> from cmath import sin, sinh
>>> r1 = sin(8j)
>>> r1
1490.4788257895502j
>>> r2 = 1j*sinh(8)
>>> r2
1490.4788257895502j
```

Unified dealing with Complex Numbers

```
>>> from numpy.lib.scimath import *
>>> sqrt(4) # float
2.0
>>> sqrt(-1) # complex
1j

# roots of a quadratic function f(x) = ax2 + bx + c:
>>> a = 1; b = 2; c = 100 # polynomial coefficients
>>> from numpy.lib.scimath import sqrt
>>> r1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
>>> r2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
>>> r1
(-1+9.94987437107j)
>>> r2
(-1-9.94987437107j)
```

Representation and round-off errors

1/49 * 49 = 1/51 * 51 = 1 (for elements of $\mathbb{R}$)

```
v1 = 1/49.0 * 49
v2 = 1/51.0 * 51
print('%.16f %.16f' % (v1, v2))
```

Output:

```
0.9999999999999999 1.0000000000000000
```

- Most real numbers are *not represented exactly* on a computer
- Real numbers and results of mathematical computations are only approximations!!!
- Some numbers can not be represented
- Beware when comparing: `if number == 0.0: ...`

Example: throwing a ball

- Throw a ball from point $(0, y_0)$ at angle θ with velocity v_0. The trajectory is a parabola:

$$y = f(x) = x\,tan\,\theta - \frac{1}{2\,v_0}\,\frac{g\,x^2}{cos^2\,\theta} + y_0$$

- We give x, y and y_0 in m, g = 9.81m/s^2, v_0 in km/h and θ in degrees
- We should take care of conversions to common units!

problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers)
-> design and implement program
-> test program

Throwing a ball, the program

```
from math import pi, tan, cos
g = 9.81      #  m/s**2
v0 = 15       #  km/h
theta = 60    #  degrees
x = 0.5       #  m
y0 = 1        #  m
print("""\
v0     = %.1f km/h
theta  = %d degrees
y0     = %.1f m
x      = %.1f m\
""" %  (v0, theta, y0, x))
# convert v0 to m/s and theta to radians:
v0 = v0/3.6
theta = theta*pi/180
y = x*tan(theta) - 1/(2*v0)*g*x**2/((cos(theta))**2) + y0
print('y       = %.1f m' % y)
```

Programmeervaardigheden

CH2: Basic Constructs

(data structures and algorithms)

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

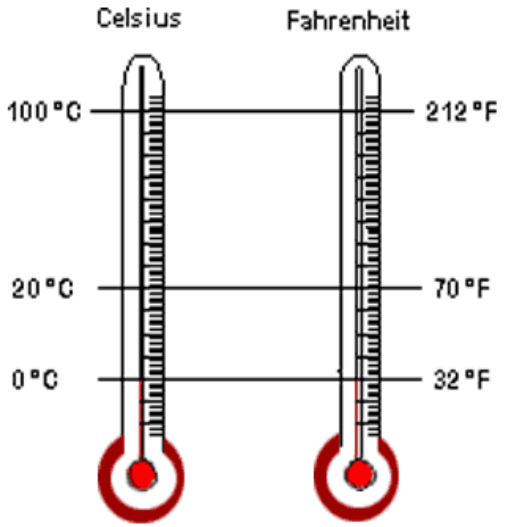
2nd Semester 2013-2014

Universiteit Antwerpen

Problem: making a table

- We want to make a table of temperatures in Celsius and their corresponding values in Fahrenheit:

```
---------------
-20.00    -4.00
-15.00     5.00
-10.00    14.00
 -5.00    23.00
  0.00    32.00
  5.00    41.00
 10.00    50.00
 15.00    59.00
 20.00    68.00
 25.00    77.00
 30.00    86.00
 35.00    95.00
 40.00   104.00
---------------
```



Fahrenheit to Celsius formula

$$F = C \cdot \frac{9}{5} + 32$$

Celsius to Fahrenheit formula

$$C = (F - 32) \cdot \frac{5}{9}$$

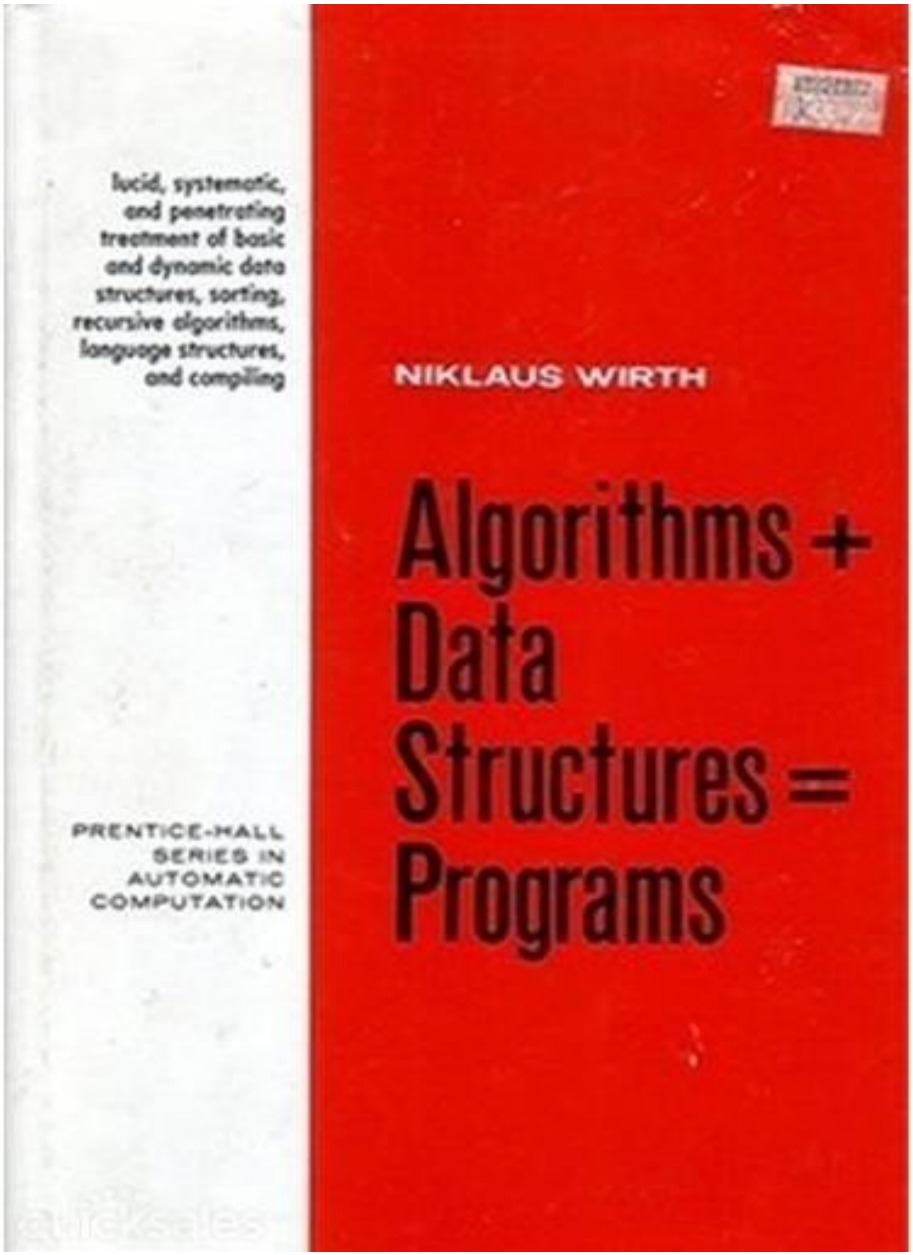
Simple solution

- Make one line in the table and repeat:

```
C = -20; F = 9.0/5*C + 32; print("%6.2f \t %6.2f" % (C, F))
C = -15; F = 9.0/5*C + 32; print("%6.2f \t %6.2f" % (C, F))
C =  35; F = 9.0/5*C + 32; print("%6.2f \t %6.2f" % (C, F))
C =  40; F = 9.0/5*C + 32; print("%6.2f \t %6.2f" % (C, F))
...
```

- Boring and error prone ... (code "cloning")
- Use a *loop* "control structure"

Niklaus Wirth's 1976 book →



The `while` loop

- Execute some statements repeatedly *as long as* a certain condition “holds” (evaluates to True):

```
while <condition>:
    <statement 1>
    <statement 2>
    ...
```

- Mind **indentation**! (denoting a “block” of statements)

```
print('---------------')# table heading
C = -20                 # start value for C
dC = 5                  # increment of C in loop
while C <= 40:          # loop heading with condition
    F = 9.0/5*C + 32                    # 1st statement inside loop
    print(“%6.2f \t %6.2f” % (C, F))# 2nd statement inside loop
    C = C + dC                          # last statement inside loop
print('---------------')# end of table line
```

The while loop

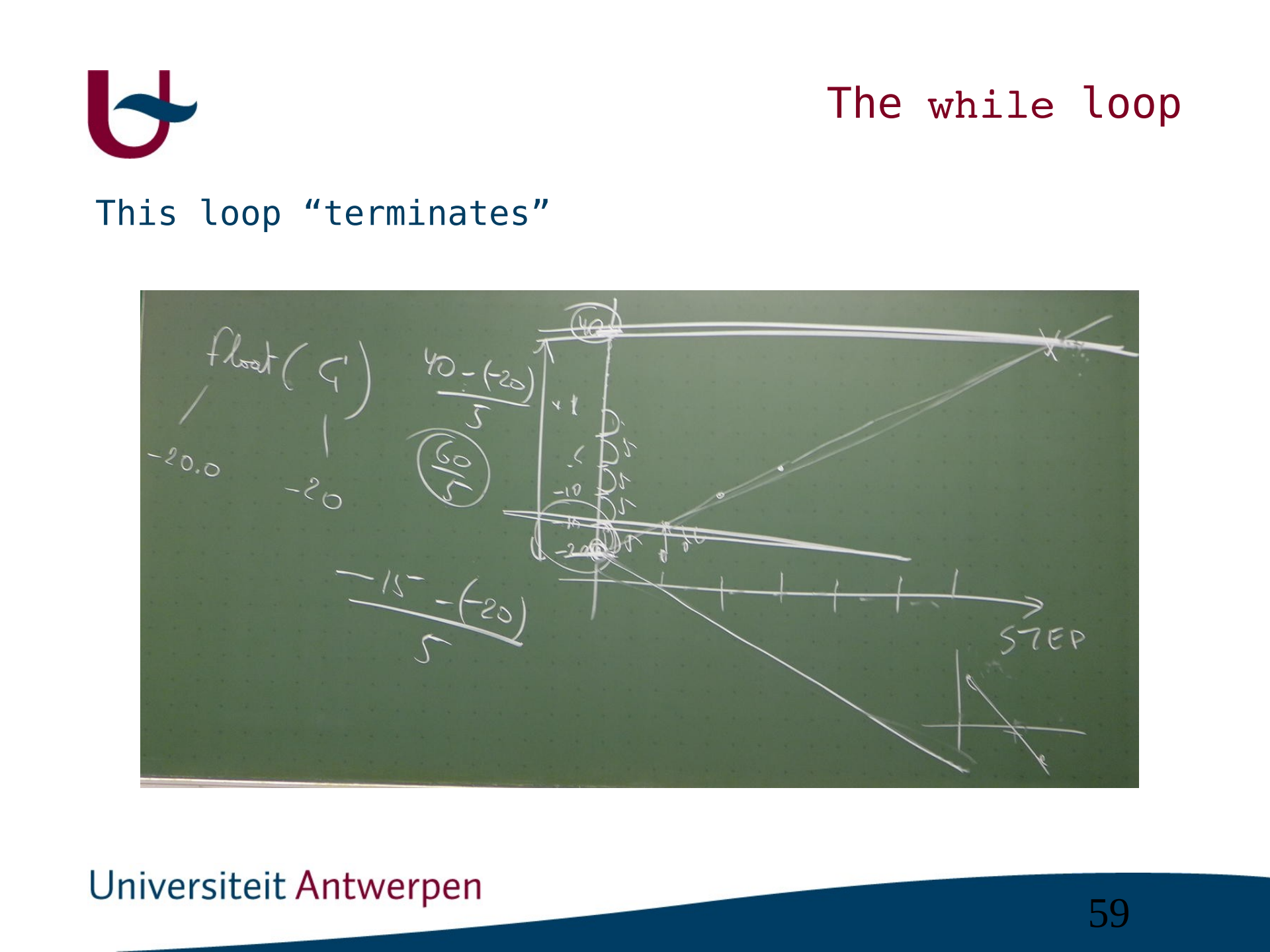
- This loop “terminates”

```
print('---------------')# table heading
C = -20                 # start value for C
dC = 5                  # increment of C in loop
while C <= 40:          # loop heading with condition
    F = 9.0/5*C + 32                    # 1st statement inside loop
    print(“%6.2f \t %6.2f” % (C, F))# 2nd statement inside loop
    C = C + dC                          # last statement inside loop
print('---------------')# end of table line
```

- Why?

- How many “iterations”? (~ complexity of algorithm)

- Can a loop be executed 0 times?



The while loop

This loop “terminates”

The while loop

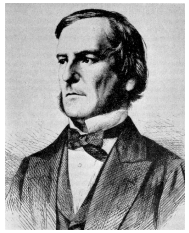
- Beware of “infinite loop”

```
C = -20                     # start value for C
dC = 5                      # increment of C in loop
while C <= 40:              # loop heading with condition
    F = 9.0/5*C + 32                    # 1st statement inside loop
    print(“%6.2f \t %6.2f” % (C, F))# 2nd statement inside loop
    C = C - dC                          # last statement inside loop
```

- Check variables used in termination condition
- Make sure variables are updated inside the loop
- Make sure the subsquent updates will lead to the condition becoming False

- Infinite loops have their uses ...

```
while True:                 # never terminates
    acceptRequest()
    processRequest()
```



Boolean expressions

Georges Boole (1815-1864)

- A Boolean expression is a part of a statement which evaluates to either *True* or *False*:

```
C == 40   # note the double ==, C=40 is an assignment!
C != 40
C >= 40
C <= 40
C > 40
C < 40
```

- Boolean expressions are often used to *test* values, for example in the termination condition of loops.
- Can be combined with *and*, *or*, *not*:

```
x >= 0 and y < 1
x >= 0 or y < 1
x > 0 and (not y > 1)
not (x > 0 or y > 0)
```

- Beware! *operator precedence* (when in doubt, use ())

(Taylor) series expansion

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Mathematics, **exact**,
but **infinite** series

$$sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + \frac{x^N}{N!}$$

Mathematics, **approximate**,
but **finite** series
stop at term with order N(=2i+1)

$$sin(x) \approx \sum_{i=0}^{\frac{N-1}{2}} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

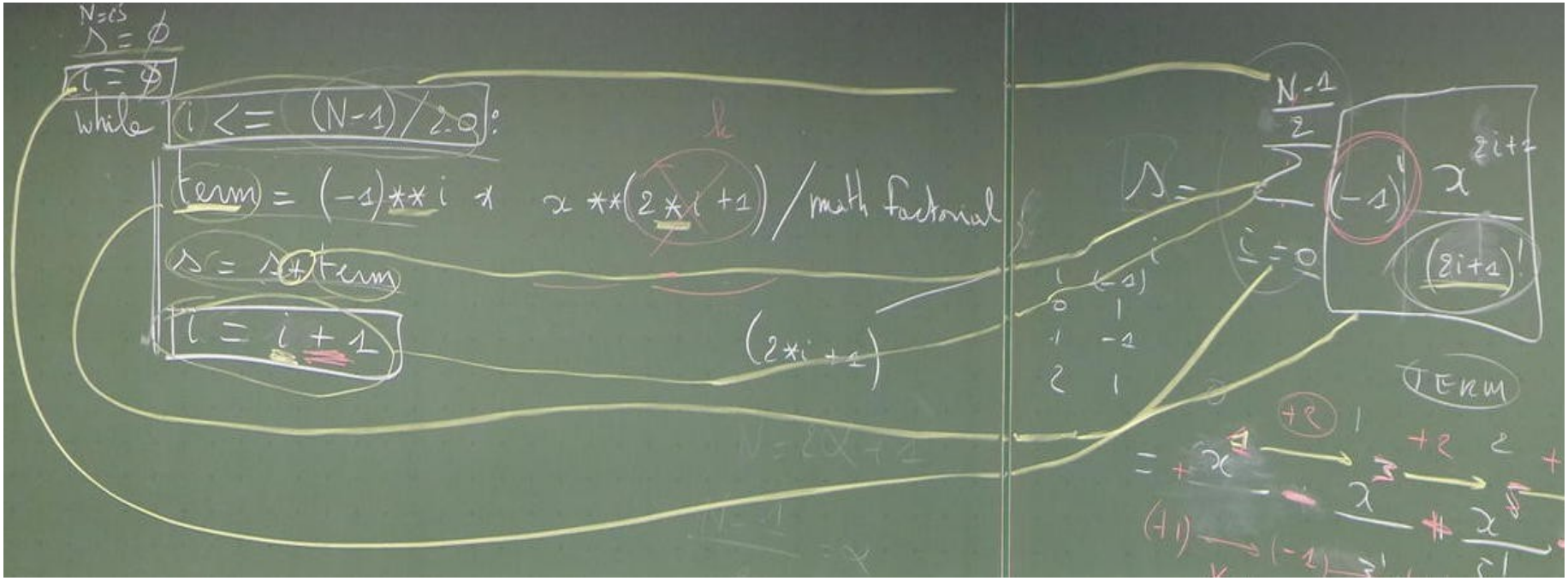
Same as above

Individual summed terms explicit

Now, we are ready to encode this
as a **program**

(Taylor) series expansion

$$sin(x) \approx \sum_{i=0}^{\frac{N-1}{2}} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

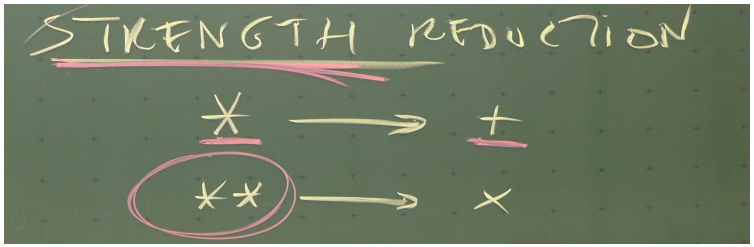


(Taylor) series expansion

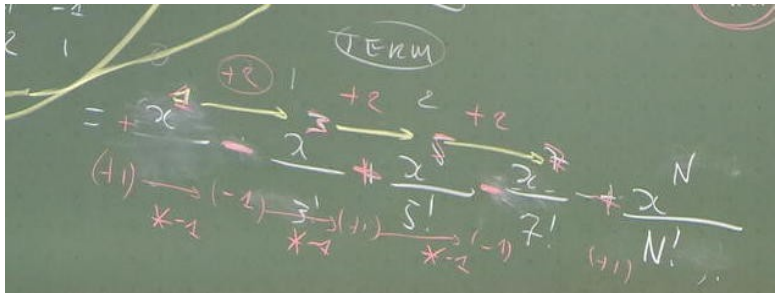
$$sin(x) \approx \sum_{i=0}^{\frac{N-1}{2}} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

```
x = 1.2 # the value whose sin() we want to compute
N = 25  # maximum power in sum (order of approximation)

i = 0   # initial value of index
s = 0   # initial value of result (sum of terms)

import math
while i <= (N -1)/2 :
  sign = (-1)**i
  term = sign*x**(2*i+1)/math.factorial(2*i+1)
  s = s + term
  i = i + 1

Print('sin(%g) = %g (approximation with %d terms)' % (x, s, N))
```

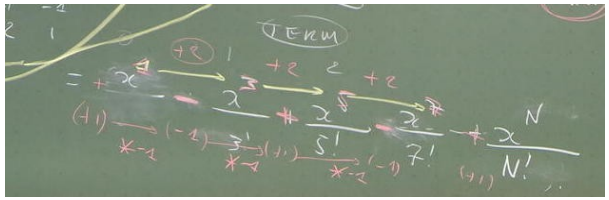
(Taylor) series expansion




```
x = 1.2 # the value whose sin() we want to compute
N = 25  # maximum power in sum (order of approximation)

k = 1      # initial value of index (increase by adding 2)
sign = 1   # will alternate 1, -1, 1, -1, ...
s = 0      # initial value of result (sum of terms)

import math
while k <= N :
  term = sign*x**k/math.factorial(k)
  s = s + term
  k = k + 2
  sign = -sign

Print('sin(%g) = %g (approximation order %d)' % (x, s, N))
```



(Taylor) series expansion

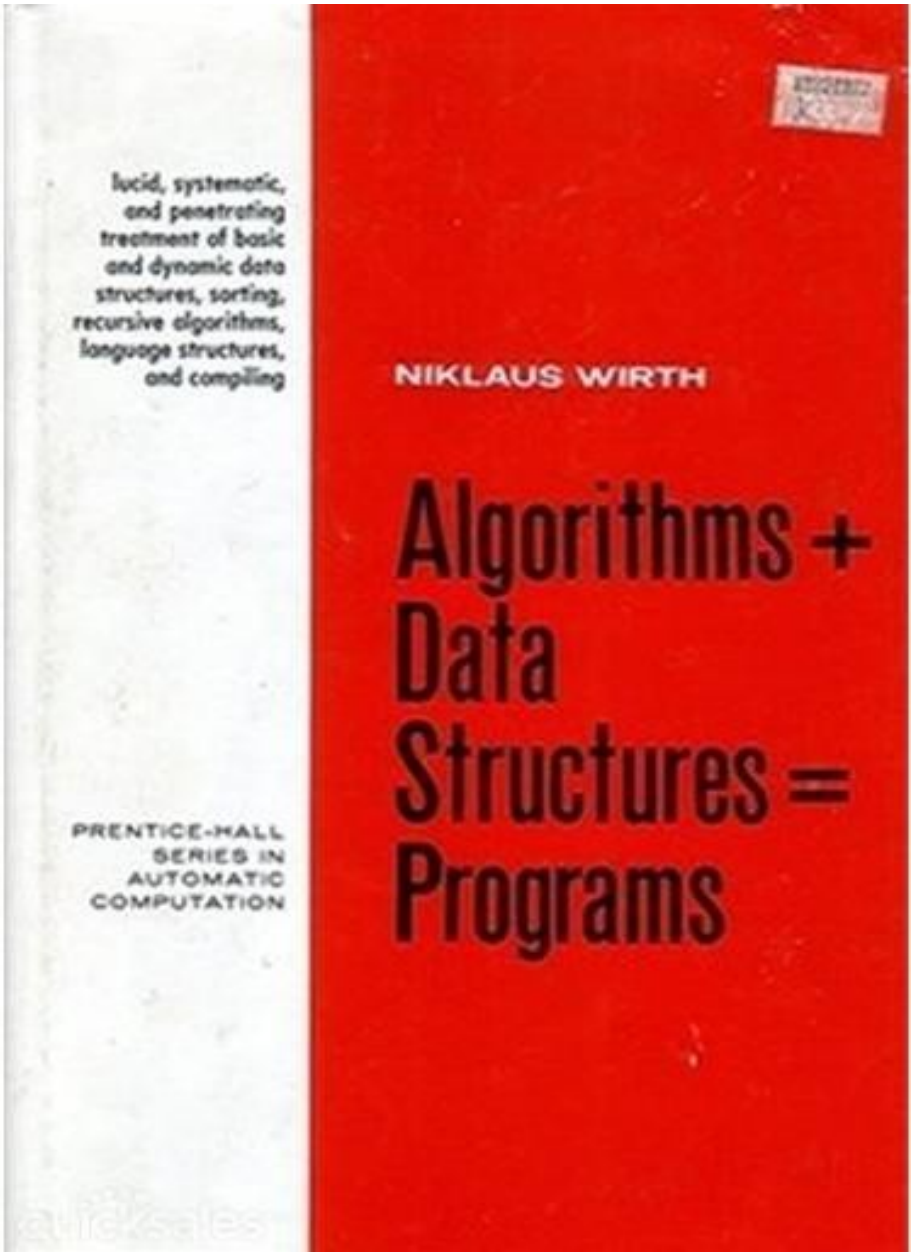


```
x = 1.2 # the value whose sin() we want to compute
xSquare = x**2
N = 25  # maximum power in sum (order of approximation)

k = 1     # initial value of index (increase by adding 2)
xPowerK = x
sign = 1  # will alternate 1, -1, 1, -1, ...
s = 0     # initial value of result (sum of terms)

import math
while k <= N :
  term = sign*xPowerK/math.factorial(k)
  s = s + term
  k = k + 2
  xPowerK = xPowerK * xSquare
  sign = -sign

print('sin(%g) = %g (approximation order %d)' % (x, s, N))
```

common operation ⇒ own notatation

```
C += dC # equivalent to C = C + dC
C -= dC # equivalent to C = C - dC
C *= dC # equivalent to C = C*dC
C /= dC # equivalent to C = C/dC
```

Niklaus Wirth's 1976 book →



Lists


MINI CLIP
Sale items
Rice cakes
Bread, bakery
Vegetables
Nuts, Seeds
Tomatoes
Cucumbers
Cold produce
Lettuce, Herbs
Mushrooms
Fruit
Bananas
Apples
Dried fruit
Cleaning
Laundry
Squeegees
peaches
melon
Feminine
Baby food
Materna
Diapers
Bath & face
Paper/plastic
Candles
Aluminum foil
Napkins
Sponges
Dish soap
Cleansers
Toilet cleaner
Air fresheners
Gum
Batteries
Chips/Bamba
Soda seltzer
Wine
Grape juice
Mashka, beer
Nectars, water
Bug sprays
Eggs
Crackers
Coffee, tea
Cocoa, shoko mix
Chocolate bars
Candy
Cookies/cakes
Choc. spread
Peanut butter
Jelly/jam
Deli cheese
Baking chocolate
Baking powder
Gelatin powder
Sesame seeds
Yeast, sprinkles
Cake, Pancake mix
Frozen pastry
Ice cream
Pasta, rice
Beans, popcorn
Soupnuts, Insta soups
Malawach
Frozen veggies
Toothpicks
Grilling
Dips/salads
Fresh chicken
Packaged deli
Salad dressing
Tomato sauce
Tuna, sardines
Frozen meats
Canned fruit
Canned olives
Tomato prdcts.
Mayonnaise
Tehina
Ketchup
Mustard
Oils
pitted
28°
Spices, salt
Flour, Sugar
Asian
Oatmeal
Soymilk
Granola
Cereal, bars
Milk, Shoko
Margarine
Yogurt, Cottage
Gvina levana

Lists

- One variable for each value is sometimes *inconvenient* (too many variables) or *impossible* (number of variables not known beforehand) and we often wish to go over the variables in a particular *order*
- We can use an *ordered collection* of values: a *list*

```
myNumbers = [-20, -15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40]
anotherList = [587, -5, 0.5, 7]
myList = ['some text', 0, -89]    # “polymorphic” list
```

- List elements are accessed via an *index*, starting at 0

```
myList = [4, 6, -3.5]
print(myList[0])  # 4
print(myList[2])  # -3.5
print(myList[-1]) # -3.5  (-1: last element)
print(myList[3])  # error: list index out of range
```

- Lists can be *unpacked* into individual variables:

```
myList = [4, 'some text']
a, b = myList     # (a, b) = myList
                  # a single, structured LHS object
print(a, b)       # 4 some text
```

Lists

- Index -1 indicates first element (“wraps around”)

```
myList = [4, 6, -3.5]
print(myList[-1]) # -3.5  (-1: last element, “wraps around”)
```

- Is “wrapping around” possible in the other direction?

```
myList = [4, 6, -3.5]
index = 4
indexModuloList = index % len(myList)
                  # % means remainder after integer division
print(myList[indexModuloList]) # 6
```

Calling Functions and Operations

- A *call* to a *function* or *operation* is a request for a particular computation defined elsewhere
- The result of the computation may depend on variables, for which concrete values can be provided as *parameters* or *arguments*

$$f(x) = 2x + 1$$

```
y = f(7)      # y == 15
q = f(1)      # q == 3
z = f(q)      # z == 7
```

- An *operation* or *method* is a function which "belongs" to one of its parameters, known as the *target*
- Instead of computing a result, an operation often *manipulates* (modifies) the target

```
myList = [4, 6, -3.5]
listLength = len(myList) # function call, parameter myList
print(listLength)        # 3
myList.append(35)        # operation or method call,
                         # target myList
print(myList)            # [4, 6, -3.5, 35]
```

Functions and operations for lists

range(start, stop[, step])

This function creates lists containing arithmetic progressions.
It is most often used in `for` loops. The arguments must be plain **integers**.
If the `step` argument is omitted, it defaults to **1**.
If the `start` argument is omitted, it defaults to **0**.
The full form returns a list of plain integers `[start, start + step, start + 2 * step, ...]`.
If `step` is positive, the last element is the largest `start + i * step` less than `stop`;
if `step` is negative, the last element is the smallest `start + i * step` greater than `stop`.
step must not be zero (or else ValueError is raised).

```
>>>>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
```

Functions and operations for lists

```
l = range(-10,31,5) # range function computes list of integers
print(l)            # [-10, -5, 0, 5, 10, 15, 20, 25, 30]
l.append(35)        # add new element 35 at the end
print(l)            # [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
l = l + [40, 45] # extend C at the end ('+' does not change l, = does!!)
print(l)            # [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
l.insert(0, -15) # insert -15 at index 0
print(l)            # [-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
l.pop(2)            # delete 3rd element (position number 2)
print(l)            # [-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
l.pop(2)            # delete what is now 3rd element
print(l)            # [-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
print(len(l))       # 11 ('len': function)
print(l.__len__()) # 11 ('__len__': operation)
```

- Some functions/operations return a result
- Some functions/operations manipulate parameters
- Mostly a matter of syntax and convention ...

The for loop

- for loop: visit each element in a list and do something with it, one by one:

```
for element in somelist:
    # process element
```

- use a variable which holds each element once:

```
degrees = [0, 10, 20, 40, 100]
print('The degrees list has %d elements' % len(degrees))
for C in degrees:
    print('list element: %d' % C)
```

- *for* loop can always be rewritten as a *while* loop:

```
index = 0
while index < len(somelist):
    element = somelist[index]
    # process element
    index += 1
```

(Taylor) series expansion

```
x = 1.2 # the value whose sin() we want to compute
xSquare = x**2
N = 25  # maximum power in sum (order of approximation)

k = 1      # initial value of index (increase by adding 2)
xPowerK = x
sign = 1  # will alternate 1, -1, 1, -1, ...
s = 0      # initial value of result (sum of terms)

import math
for k in range(1,N,2):
  term = sign*xPowerK/math.factorial(k)
  s = s + term
  k = k + 2
  xPowerK = xPowerK * xSquare
  sign = -sign

print('sin(%g) = %g (approximation order %d)' % (x, s, N))
```

Another example

- Start out with a list of Celsius values, compute a list of Fahrenheit values:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10,
            15, 20, 25, 30, 35, 40]
Fdegrees = []                 # start with empty list
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)    # add new element to Fdegrees
```

- After the `for` loop terminates, the value of `Fdegrees` is:

```
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0,
 68.0, 77.0, 86.0, 95.0, 104.0]
```

for loops over lists of indices

- Sometimes we want access to the indices of list elements, so we loop over a list of integers:

```
myList = [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for i in range(2, len(myList), 1):
    myList[i] = myList[i-1] + myList[i-2]
```

- Alternative:

```
myList = [1, 1]
for i in range(0, 9, 1):
    newVal = myList[-1] + myList[-2]
    myList.append(newVal)
```

Modifying list elements

- Using element values does **not** work, as the values are simply *copied* to the loop variable:

```
values = [-1, 1, 10]
for e in values:
    e = e + 2
print values    # [-1, 1, 10]
```

- We must modify the value *at the list position* itself, using its *index*:

```
values = [-1, 1, 10]
for index in range(len(values)): # range(0, len(values), 1)
    v[index] = v[index] + 2
print values    # [1, 3, 12]
```

List comprehensions

- List comprehensions are essentially *for* loops written in an even more compact form, and can be used to compute a new list based on an existing list:

```
[<expression> for <element> in <somelist>]
```

$$\{2x+1 | x \in 1,2,\ldots,5\}$$

```
myList = [2*x+1 for x in range(1,6)]
```

List comprehensions

```
[<expression> for <element> in <somelist>]
```

- Compute a list of Celsius degrees based on a list of integers, and then a Fahrenheit list based on the Celsius one:

```
Cdegrees = [-5 + i*0.5 for i in range(16)]
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
```

- Equivalent to:

```
Cdegrees = []    # empty list
for i in range(16):
    Cdegrees.append(-5 + i*0.5)
Fdegrees = []
for i in range(16):
    Fdegrees.append((9.0/5)*Cdegrees[i] + 32)
```

Nested lists

- Lists can contain any type of element, including lists

```
myTable = []
l1 = [1,2,3]
l2 = [4,5,6]
myTable.append(l1)
myTable.append(l2)
```

- We end up with a table (matrix), i.e. a two-dimensional structure
  Note: need not be a matrix (e.g., `[[1,2], "hello", [3,4,5]]`)

```
print myTable[0]        # [1,2,3]
print myTable[1][2]     # 6
print myTable           # [[1,2,3],[4,5,6]]
```

- Make it look like a matrix:

```
for row in myTable:
    print(row)
```

- Can be generalized to n-dimensional...

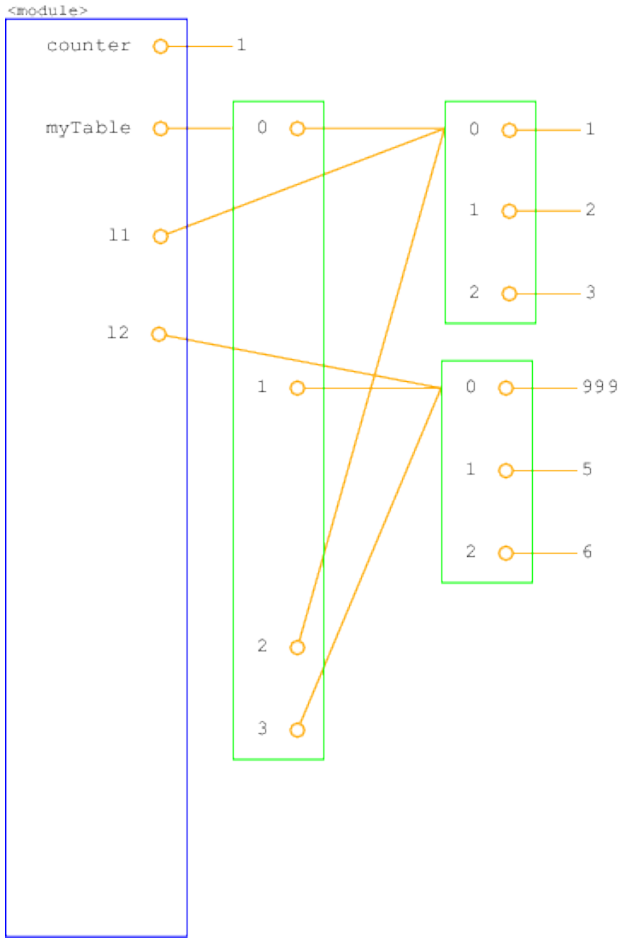
Nested lists

What happens when we overwrite elements of a list?

```
myTable = []

l1 = [1,2,3]
l2 = [4,5,6]

for counter in range(2):
 myTable.append(l1)
 myTable.append(l2)

myTable[1][0] = 999
```

Nested lists

```
from scitools.Lumpy import Lumpy
lumpy = Lumpy()
lumpy.make_reference()

myTable = []

l1 = [1,2,3]
l2 = [4,5,6]

for counter in range(2):
 myTable.append(l1)
 myTable.append(l2)

myTable[1][0] = 999

lumpy.object_diagram()
```



Appendix H.3 of "A Primer on Scientific Programming with Python" (3rd Ed.)

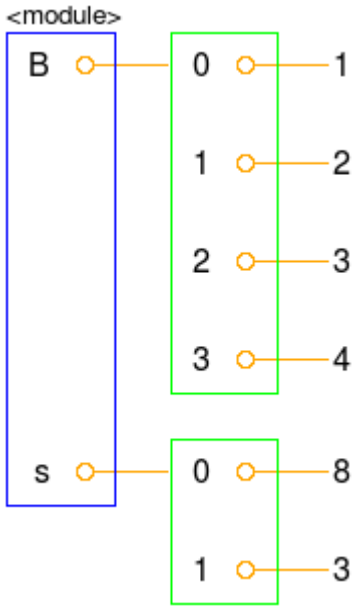
Slices

- A *slice* is a sublist, i.e. a part of a list

```
A = [2, 3.5, 8, 10]
A[2:]   # from index 2 to end of list; [8, 10]
A[1:3]  # from 1 to 3, excl.; [3.5, 8]
A[:3]   # from start to 3, excl.; [2, 3.5, 8]
A[1:-1] # from 1 to last, excl.; [3.5, 8]
A[:]    # the whole list; [2, 3.5, 8, 10]
```

- Slices are *copies* of the original, so modifying them does not affect the original list.

```
B = [1,2,3,4]
s = B[1:3]          # [2,3]
s[0] = 8            # s == [8,3]
print B             # [1,2,3,4]
```



Tuples

- Tuples are lists which are “immutable”, i.e., *can't be modified*

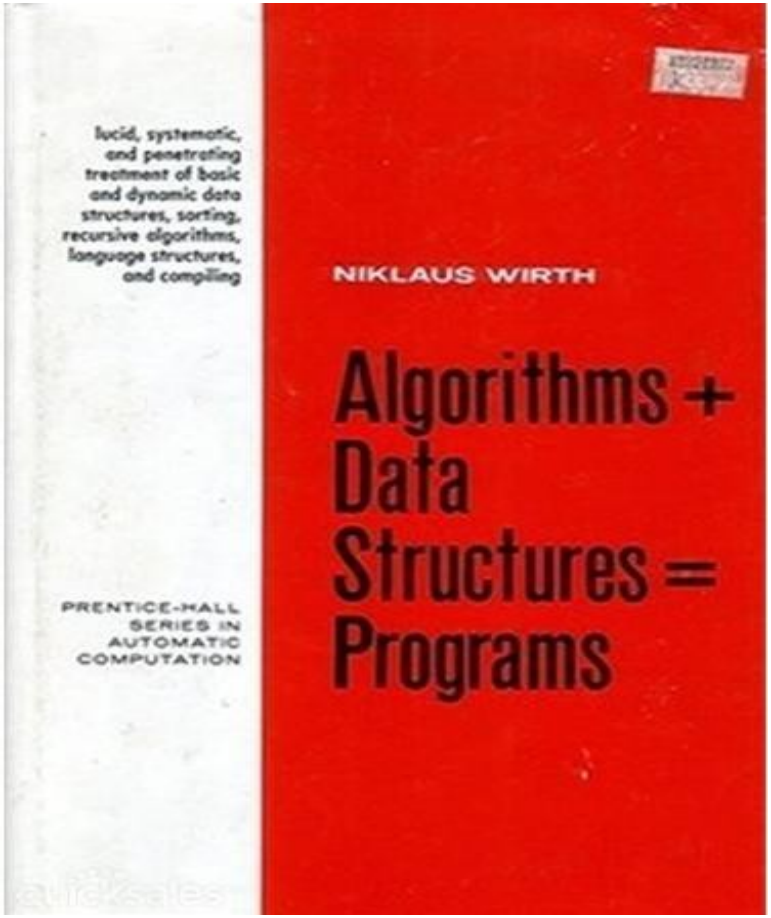
```
t = (2, 4, 6, 'temp.pdf')    # define a tuple
t[1] = -1    # error!
t.append(0)  # error!
t.pop(1)     # error!
```

- Otherwise, tuples behave like lists:

```
t = t + (-1.0, -2.0)  # add two tuples (concat!)
print(t)              # (2, 4, 6, 'temp.pdf', -1.0, -2.0)
e = t[1]              # indexing
print(e)              # 4
s = t[2:]             # subtuple/slice
print(s)              # (6, 'temp.pdf', -1.0, -2.0)
6 in t                # membership; True
```

- Tuples are faster and 'safer' than lists

Visualizing Program Execution



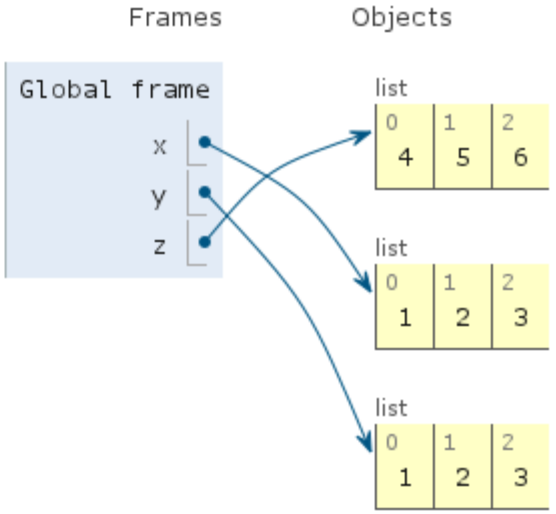
Visualizing Program Execution

```
a = 10
while a < 13:
  print a
  print "entering inner loop"
  b = 20
  while b < 30:
    print a, b
    b = b + 3
  print "end of inner loop"
  print b
  a = a + 1
```

http://pythontutor.com/visualize.html

Visualizing Program Execution

```
1  x = [1, 2, 3]
2  y = [4, 5, 6]
3  z = y
4  y = x
5  x = z
6
7  x = [1, 2, 3] # a different [1, 2, 3] list!
8  y = x
9  x.append(4)
10 y.append(5)
11 z = [1, 2, 3, 4, 5] # a different list!
12 x.append(6)
13 y.append(7)
14 y = "hello"
```



http://pythontutor.com/visualize.html

Programmeervaardigheden

CH3: Functions and Branching

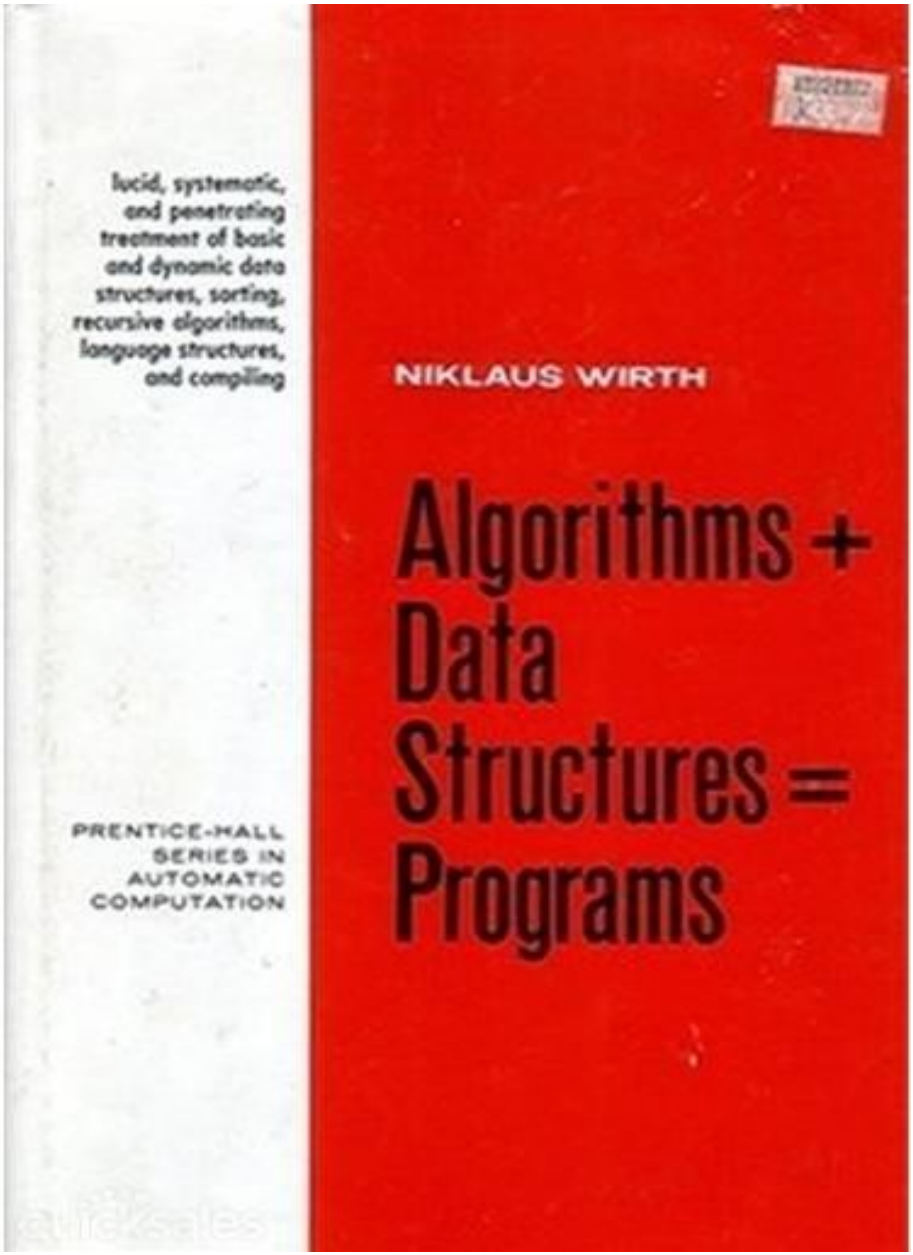
Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

Niklaus Wirth's 1976 book →



problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers)
-> design and implement program
-> test program

functions to tackle complexity ...

We have used several kinds of functions so far:

```
from math import *
y = sin(x)*log(x)       # mathematical functions
n = len(somelist)       # function working on list
ints = range(5, n, 2)   # function producing a list
C = [5, 10, 40, 45]
i = C.index(10)         # list operation; dot syntax
C.append(50)            # list operation modifying “target” C
C.insert(2, 20)         # list operation modifying “target” C
                        # insert 20 before position with index 2
```

Functions

- Functions must be *defined* once,
  and can then be *called* (re-*used*) multiple times
  in different locations in a program
- Functions *group* and *encapsulate* instructions that belong together (i.e., serve some common purpose) and give them a *name*
- Functions allow “inside” and “outside” to *vary independently*
- Functions can take *parameters*
- Functions *compute* and/or *modify* values

Calling Functions and Operations

- A *call* to a function or operation is a request for a particular computation defined elsewhere
- The result of the computation may depend on variables, for which concrete values can be provided as *parameters*

$$f(x) = 2x + 1$$

```
y = f(7)      # y == 15
q = f(1)      # q == 3
z = f(q)      # z == 7
```

- An *operation* or *method* is a function which "belongs" to one of its parameters, known as the *target*
- Instead of (or in addition to) *computing a result*, an operation often *manipulates* the target

```
myList = [4, 6, -3.5]
a = len(myList)          # function call, parameter myList
print(a)                 # 3
myList.append(35)        # operation call, target myList
print(myList)            # [4, 6, -3.5, 35]
```

Defining functions

- A function definition consists of several parts:

header

```
def <function name> ( <parameter names> ):
```

body

```
    <statement 1>
    <statement 2>
    …
    return <value>      # optional
                        # returns None by default
```

- Example of a mathematical function:

$$F(C)=\frac{9}{5}C+32$$

```
def f(c):
    val = (9.0/5)*c + 32
    return val
```

- A function *definition* does not execute anything!
- When a function is *called* (mathematics: evaluated), its body is *executed*.

Example

- Recall the formula for throwing a ball in the air:

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

```
def yfunc(t, v0):
    g = 9.81
    return v0*t – 0.5*g*t**2

# sample calls:
y = yfunc(0.1, 6)          # t is assigned 0.1, v0 is assigned 6
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)
```

- Functions can take an arbitrary number of parameters
- When a function call is *evaluated*,
  it is *replaced* by the *value* it *returns*

(Taylor) series expansion

```
def sinApprox(x, N):
# x: the value whose sin() we want to compute
# N: the maximum power (=2i+1) in the sum (order of approximation)
  xSquare = x**2
  k = 1     # initial value of index (increase by adding 2)
  xPowerK = x
  sign = 1  # will alternate 1, -1, 1, -1, ...
  s = 0     # initial value of result (sum of terms)
  import math
  while k <= N :
    term = sign*xPowerK/math.factorial(k)
    s = s + term
    k = k + 2
    xPowerK = xPowerK * xSquare
    sign = -sign
  return s

y = 1.2
pow = 25
print('sin(%g) = %g (approximation order %d)' % (y, sinApprox(y,pow), pow))
```

return values

- A function may do several things:
  - compute a value (like in mathematics)
  - cause “side-effects” (output, invoke operations, …)
  - a combination of these
- If a function computes a value, that value must be explicitly *returned* in order to replace function calls

```
def mySum(a,b):
    result = a + b
    return result
```

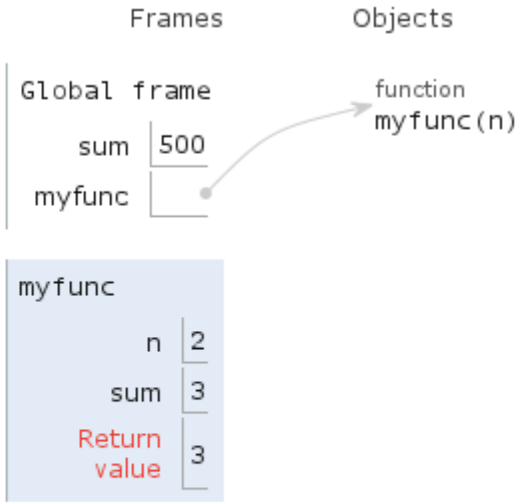
```
def myAppender(l,e):      # no return value
    print “modify list”
    l.append(e)           # side-effect on l
```

```
def niceSumAndProduct(c,d):
    print “calculating...”   # side-effect
    s = c + d
    p = c * d
    return s,p   # tuple!! - same as   return (s,p)

t = niceSumAndProduct(3,4)   # replace call: t = (7,12)
sum,product = t              # sum = 7; product = 12
```

Variables in functions

- Names of variables to which a value is assigned inside a function are not visible outside that function (*local* variables)
- Names of variables created outside any function are visible everywhere (*global* variables)
- Global variables can be *hidden* by a local version in functions.

```
1  print(sum) # sum is a builtin Python function
2  sum = 500  # re-"bind" sum to the int with value 500
3  print(sum) # sum is a global integer value
4
5  def myfunc(n):
6    # n behaves like a local variable
7    # (type is same as given argument)
8    sum = n + 1 # a new local integer variable is created
9    print(sum)  # the value of the local variable
10   return sum
11
12 sum = myfunc(2) + 1 # new value in global variable sum
13 print(sum)
```



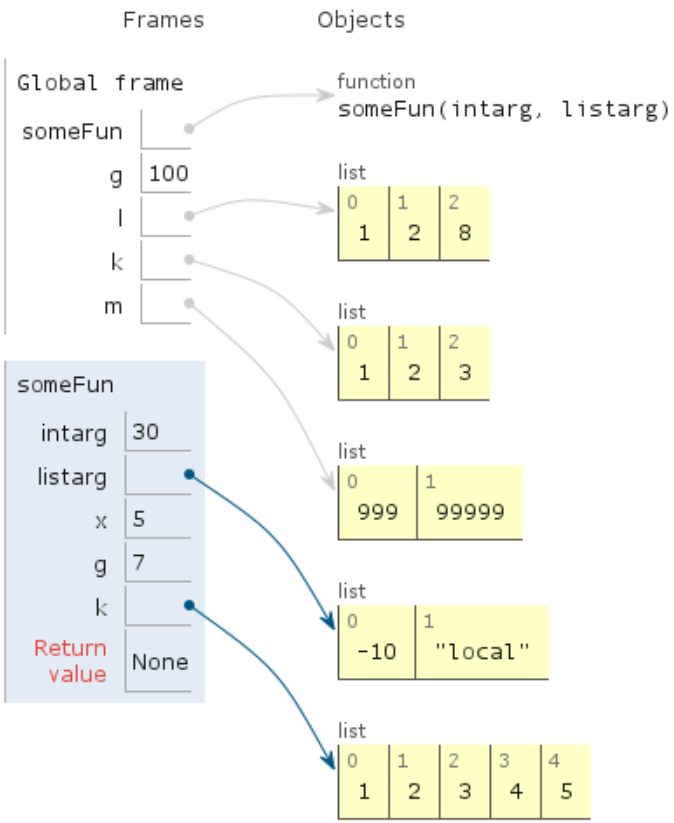
Variables in functions

```
1  def someFun(intarg, listarg):
2    intarg = 30 # local variable (copy !!!)
3    listarg = [-10,"local"]
4    x = 5 # local variable x
5    g = 7 # local variable g, hides the global variable g
6    l.append(8) # operation on global variable l
7    k = [1,2,3,4] # local variable k, hides global variable k
8    k.append(5)   # no effect on global variable k
9
10 # global variables
11 g = 100
12 l = [1,2]
13 k = [1,2,3]
14 m = [999, 99999]
15
16 someFun(1, m)
```

Edit code

<< First | < Back | Step 15 of 15 | Forward > | Last >>

line that has just executed
next line to execute



Variables in functions

- Values of global variables can be accessed in functions.
- Values of global variables can not be overwritten in functions (assignment creates new local variable!) except `global` (to be avoided)

```
def f():
    global x    # global variable, for assignment!
                # breaks RE-USE !
    x = 10
    y = 20 + z  # local variable

x = 100
y = 200
z = 300
f()

print(x)
print(y)
```

Variables in functions

- Values of global variables can be accessed in functions.
- Values of global variables can not be overwritten in functions

  (assignment creates new local variable!) except `global`
- Careful: different (list) variables can hold the same value!

```
k = [1,2,3]
def someFun():
    x = 5    # local variable x
    y = k    # y and k hold the same list value now!
    z = y    # z too!
    k.append(6) # effect on value of z and y!
    y.append(8) # effect on value of k and z!
    z.append(9) # effect on value of k and y!
    # k = [4,5,6] # new local k (uncomment => error at 'y=k')

someFun()
print k      # [1,2,3,6,8,9]
print y      # error!
```

Parameters

- A function may want to behave differently depending on external (non-global) information
- Solution: Parameters are local variables (copies!) which are assigned a value exactly when the function is called

```
def myFun(par1, par2):
    res = par1 + par2
    return res

A = 100
B = 20
C = myFun(A+B, A-B) + 10
```

- Values passed as parameters can be modified through operations, but **not overwritten (they are local copies)**. Beware of **lists**!

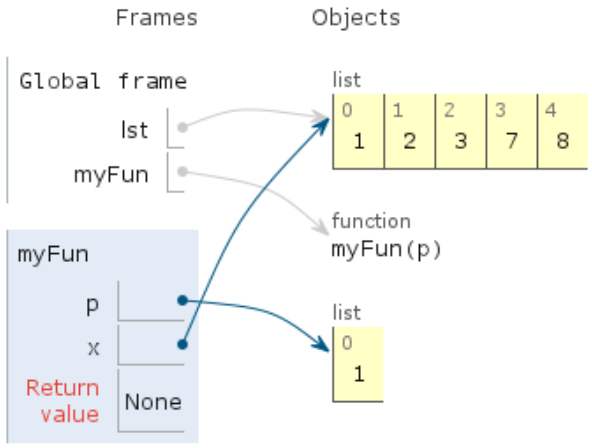
```
lst = [1,2,3]
def myFun(p):
    p.append(7)    # modify list through copied reference!
    x = p          # local variable x now also holds value of p
    p = [1]        # no effect on original passed argument!
    x.append(8)    # still has effect on original passed argument!

myFun(lst)
print lst          # [1,2,3,7,8]
```

Parameters

```
1  lst = [1,2,3]
2  def myFun(p):
3      p.append(7)    # modify list through copied reference!
4      x = p          # local variable x now also holds value of p
5      p = [1]        # no effect on original passed argument!
6      x.append(8)    # still has effect on original passed argument!
7
8  myFun(lst)
9  print lst          # [1,2,3,7,8]
```

Edit code

<< First | < Back | Step 9 of 10 | Forward > | Last >>



Parameter position and default value

- When a function is called, “formal” parameters are associated with their values *by position:*

```
def myFun(par1, par2):    # function definition
    ...
x = 7
myFun(x+1, 8*2)           # function call; par1 = 8, par2 = 16
```

- Parameters with a *default* value can be skipped in the call
- *Keyword* syntax can be used in general, in which case parameter position is unimportant

```
def myFun(par1, par2 = 5):  # par2 is 5 by default
    ...
x = 7
myFun(x)                  # par1 = 7, par2 = 5
myFun(8, x)               # par1 = 8, par2 = 7
myFun(par1=8, par2=x)     # par1 = 8, par2 = 7
myFun(par2=x, par1=8)     # par1 = 8, par2 = 7, position unimportant!
```

Example

- This mathematical function is an approximation to ln(1+x) for finite n and x >= 1

$$L(x;n)=\sum_{i=1}^{n}\frac{1}{i}\left(\frac{x}{1+x}\right)^{i}$$

- We can program and use such function as follows:

```
def L(x, n=10):
    x = float(x) # ensure float division below
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i  # a += b == a = a + b
    return s

q = 5
from math import log as ln # imported only when needed
print L(q), L(x=q, n=100), ln(1+q)
```

Example (cont.)

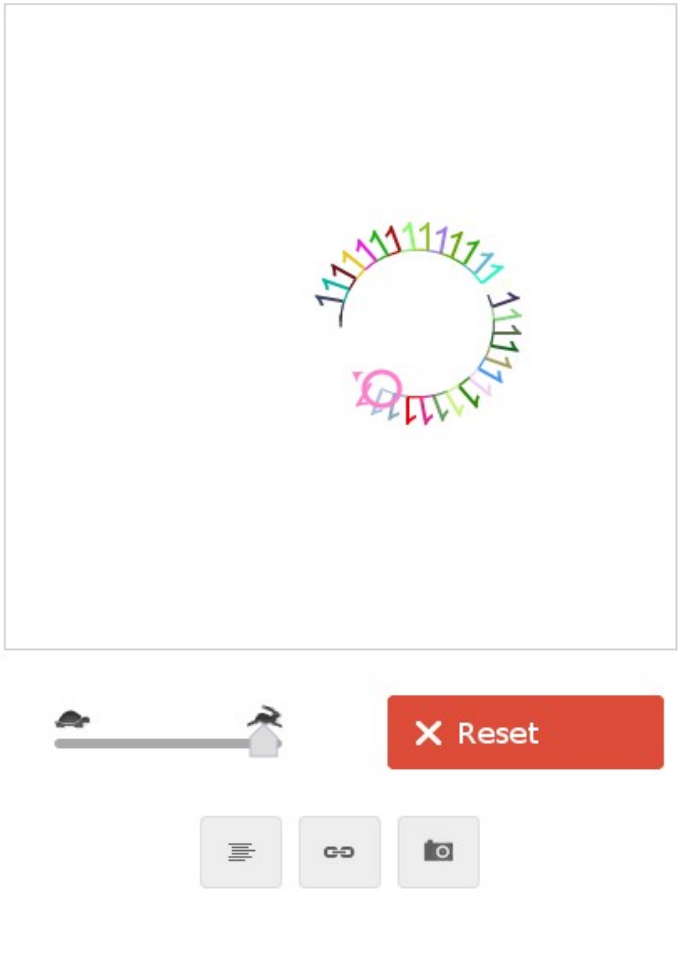
- We could also return the first neglected term in the sum and the error (ln(1+x)-L(x;n)) which it approximates (first step towards “adaptive” algorithms)

```
def L2(x, n):
    x = float(x)
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
```

turtle graphics in blockly

Blockly : Turtle Graphics

http://code.google.com/p/blockly/



Functions as function parameters

- Some functions may depend on other *functions* (which can be *passed as parameters)*
- Examples:

$$\int_a^b f(x)\,dx$$ numerical integration

$$f'(x)$$ numerical differentiation

$$f(x)=0$$ numerical root finding

- Example:

$$f''(x) \approx \frac{f(x-h)-2f(x)+f(x+h)}{h^2}$$

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

- Straightforward in Python, not in some other languages!

Example

- We apply the *diff2* function and generate a table for decreasing values of *h*
- Roundoff errors!! always check!
- *decimal* module allows for higher precision...

```
def g(t):
    return t**(-6)

for k in range(1,15):
    h = 10**(-k)
    print 'h=%.0e: %.5f' % (h, diff2(g, 1, h))
```

```
# Exact g''(1)
#  == -6*-7*1**-8
#  == 42
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
h=1e-13: 66613381477.50939
h=1e-14: 0.00000
```

Limited Precision!

```
>>> A = 500.0
>>> B = A + 1e-15
>>> C = B - 500.0

>>> print C
0.0

>>> A = 500.0
>>> B = A – 500.0
>>> C = B + 1e-15

>>> print C
1e-15
```

The main program

- A program contains functions and ordinary statements outside function definitions (use `dir()`). The latter constitute the *main program*

```
from math import *          # in main

def f(x):                   # in main
    e = exp(-0.1*x)         # NOT in main
    s = sin(6*pi*x)         # NOT in main
    return e*s              # NOT in main

x = 2                       # in main
y = f(x)                    # in main
print 'f(%g)=%g' % (x, y) # in main
```

- Execution starts with the first statement in the main program and proceeds line by line, top to bottom ("sequential" vs. parallel)
- `def` statements *define* a function, but the statements inside the function are not *executed* before the function is called

Summarizing example

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

- Compute (t,y(t)) values for the function for t values for which y >= 0 generating exactly the following output (position until hit ground):

```
0.00  0.00
0.25  0.94
0.50  1.27
0.75  0.99
1.00  0.09
max y(t) = 1.27375
```

- Requirements: function y(t), function *table* to create a nested list, a loop for printing the list, a loop for finding the maximum *y* value

Summarizing example (cont.)

```
g  = 9.81
v0 = 5
dt = 0.25

def y(t):
    return v0*t - 0.5*g*t**2

def table():
    data = [] # store [t, y] pairs in a “nested” list
    t = 0
    while y(t) >= 0:
        data.append([t, y(t)])
        t += dt
    return data

data = table()

for t, y in data:
    print '%5.2f %5.2f' % (t, y)
```

Summarizing example (cont.)

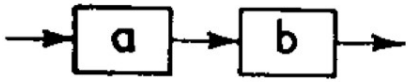
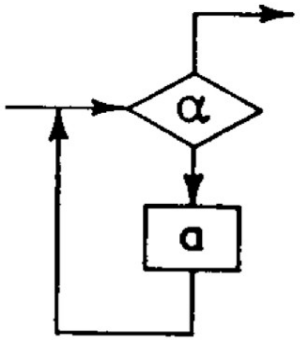
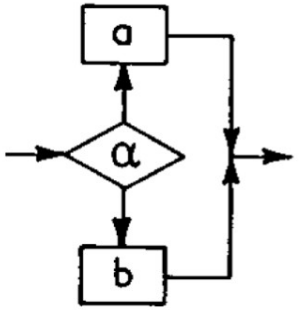
```
# extract all y values from data:
y = [y for t, y in data]      # y for (t,y) in data


# find maximum y value (knowing that all y >= 0):
ymax = 0 # more general: use -float('inf')
for yi in y:
    if yi > ymax:
        ymax = yi
print 'max y(t) =', ymax
```

Functions

- Functions must be *defined* once,
  and can then be *called* (re-*used*) multiple times
  in different locations in a program
- Functions *group* and *encapsulate* instructions that belong together (i.e., serve some common purpose) and give them a *name*
- Functions allow “inside” and “outside” to *vary independently*
- Functions can take *parameters*
- Functions *compute* and/or *modify* values

Böhm-Jacopini theorem

(Corrado Böhm and Guissepe Jacopini)

Δ	Sequence: Executing a subprogram in order.	
Ω	Iteration: Executing a subprogram until a condition is reached.	
Π	Selection: Executing a subprogram based on a condition.	

problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers)
-> design and implement program
-> test program

if tests

- Behaviour may be different depending on a condition
- An `if` test allows for this functionality:

```
if <condition1>:
    <statements, executed if condition1 is True>
elif <condition2>:          # optional
    <statements, executed if condition2 is True>
...
else:                       # optional
    <statements, executed if condition1 and condition2 are False>
```

```
def f(x):
  if 0 <= x <= pi:
    value = sin(x)
  else:
    value = 0
  return value
```

```
if C < -273.15:
  print '%g degrees Celsius is non-physical!' % C
  print 'The Fahrenheit temp will not be computed.'
else:
  F = 9.0/5*C + 32
print F
```

Example, split a list of numbers into positives and negatives:

```
l = [1,-1,5,7,-9.2,0,-7,45]
pos = []; neg = []
for e in l:
    if e >= 0:
        pos.append(e)
    else:
        neg.append(e)
print pos       # [1,5,7,0,45]
print neg       # [-1,-9.2,-7]
```

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases}$$

```
def N(x):
  if x < 0:
    return 0.0
  elif 0 <= x < 1:
    return x
  elif 1 <= x < 2:
    return 2 - x
  elif x >= 2:
    return 0.0
```

breaking out of loops

```
listOfNumbers = range(10)

# check if the value 6 occurs in listOfNumbers

found = False # initially not found

for element in listOfNumbers:
    if element == 6:
      found = True
      break # break out of loop

print(found)
```

Programmeervaardigheden

CH4: Input Data and Modules

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

Universiteit Antwerpen

Getting input from the user

- So far, input was hardcoded in the program:

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

```
C = 21
F = (9.0/5)*C + 32
print F
```

- To change these inputs, we have to modify the program
- Modification is error-prone and sometimes impossible
- Solution: Make the program ask the user for input:

```
C_str = raw_input('C=? ')    # C becomes a string
C = float(C_str)             # convert to numeric format
F = (9./5)*C + 32
print F
```

- Input is always interpreted as *string*; conversion needed!

Example: print *n* first even numbers

- Read *n* from keyboard and print *n* first even numbers:

```
n = int(raw_input('n=? '))    # convert to int

for i in range(2, 2*n+1, 2):
    print(i)

# or:
print(range(2, 2*n+1, 2))

# or:
for i in range(1, n+1):
    print(2*i)
```

Modules

- We have used modules before:

```
from math import log
r = log(6)                  # call log function in math module
```

- A module is a collection of useful data and functions (and classes) that “belong together”
- Modules are supposed to be reused in other programs
- In order to create a module, the required content simply needs to be stored in a separate file
- Using a function *f* from a module stored in file *mymodule.py*:

```
import mymodule
result = mymodule.f(arg1, arg2, arg3)
```

Example: Computing with interest

- Here are formulas for computing with interest rates:

$$A = A_0\left(1+\frac{p}{360\cdot 100}\right)^{n}$$

$$A_0 = A\left(1+\frac{p}{360\cdot 100}\right)^{-n}$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln\left(1+\frac{p}{360\cdot 100}\right)}$$

$$p = 360\cdot 100\left(\left(\frac{A}{A_0}\right)^{\frac{1}{n}}-1\right)$$

- A_0: initial amount; p: pct; n: days; A: final amount

Example (cont.)

- Collect the 4 corresponding *related* Python functions in a single module, filename *interest.py*:

```
from math import log as ln # overlapping names in libs

def present_amount(A0, p, n):
    return A0*(1 + p/(360.0*100))**n

def initial_amount(A, p, n):
    return A*(1 + p/(360.0*100))**(-n)

def days(A0, A, p):
    return ln(A/A0)/ln(1 + p/(360.0*100))

def annual_rate(A0, A, n):
    return 360*100*((A/A0)**(1.0/n) - 1)
```

Example (cont.)

- Now we can use the module from a separate program:

```
# How long does it take to double an amount of money at 5%?

from interest import days

A0 = 1; A = 2; p = 5
n = days(A0, 2, p)
years = n/365.0

print('Money has doubled after %.1f years' % years)
```

Testing a module

- It is a good idea to include a *test block of code* at the end of a module, to allow for testing (in particular, regression testing)
- This block is only executed when the module file is executed separately, NOT when it is imported

```
if __name__ == '__main__':          # don't execute when imported
    A = 2.2133983053266699
    A0 = 2.0
    p = 5
    n = 730
    print 'A=%g (%g) A0=%g (%.1f) n=%d (%d) p=%g (%.1f)' % \
          (present_amount(A0, p, n), A,
           initial_amount(A, p, n), A0,
           days(A0, A, p), n,
           annual_rate(A0, A, n), p)
```

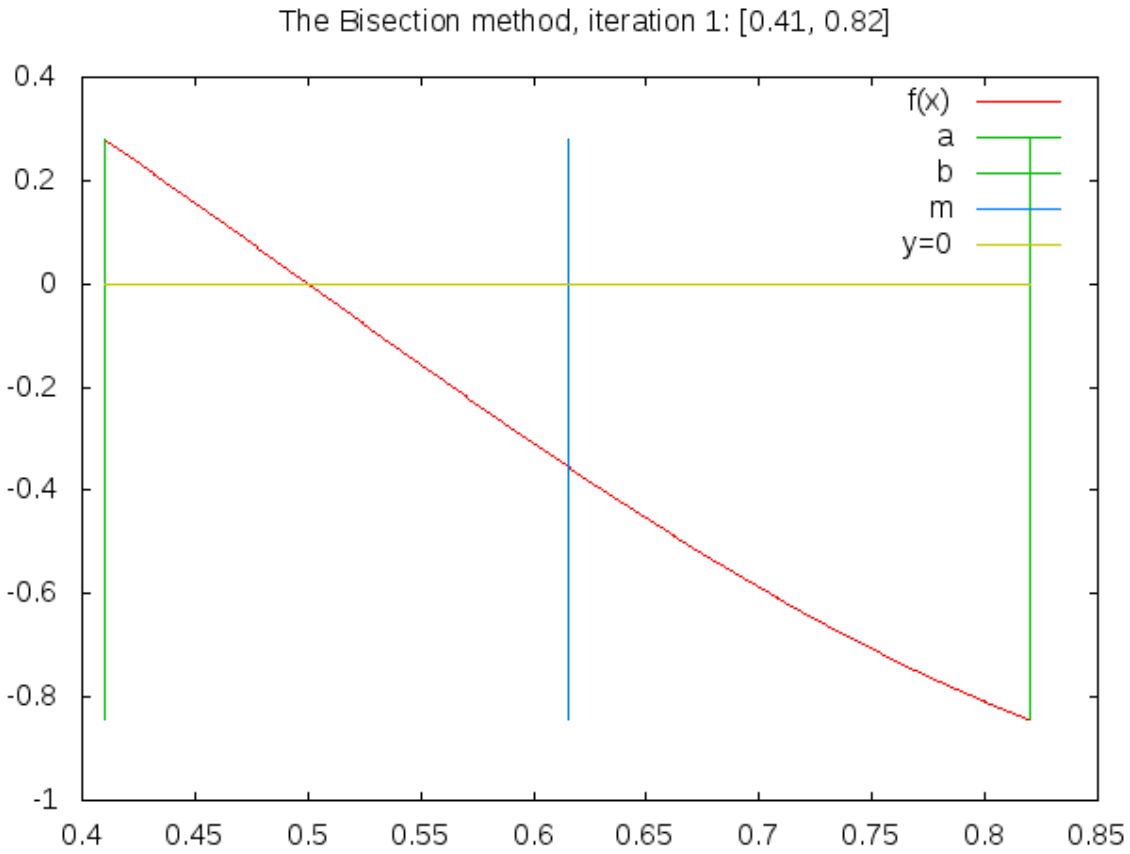
```
# main.py

print “also executed when imported”

if __name__ == '__main__':
 print “not executed when imported”
```

Example: bisection method

- General numerical method for root finding: solving f(x) = 0 (for arbitrary f)
- Can be impossible analytically ("transcendental")
  x = 1 + sin x (f(x) = x – 1 – sin(x))

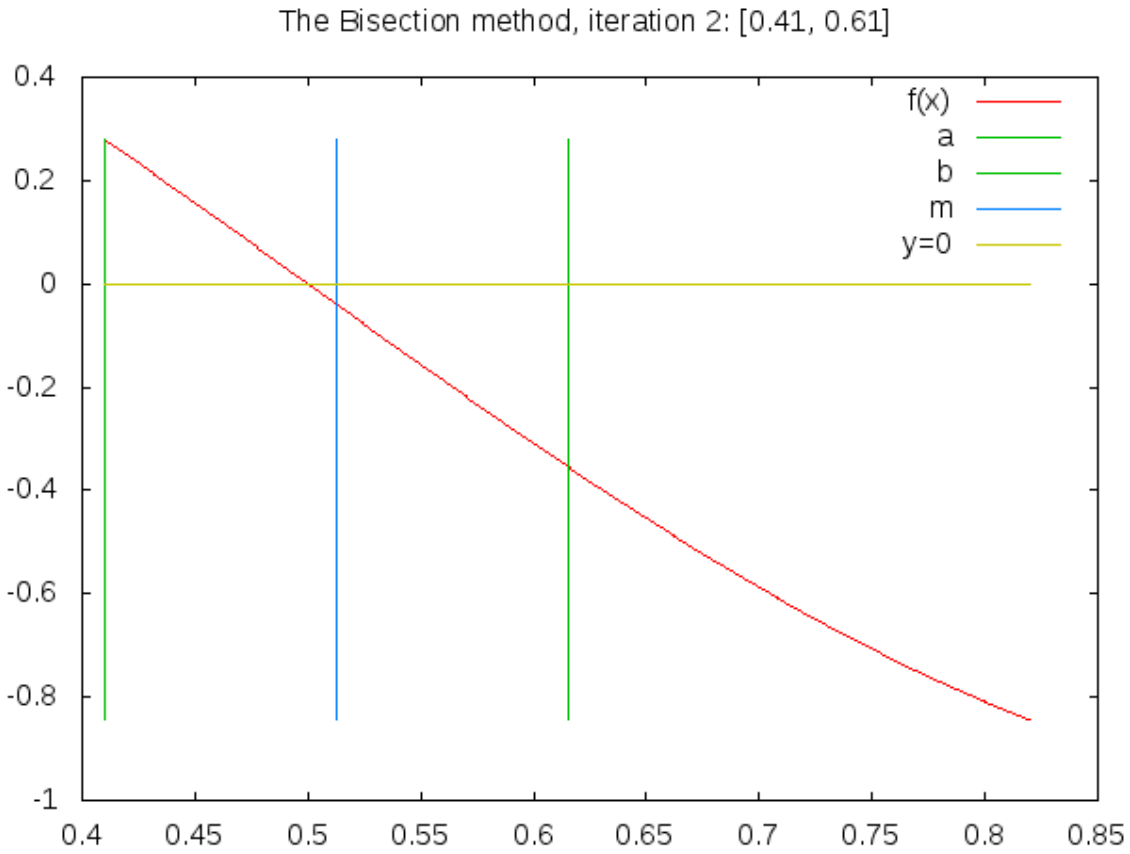
```
# bisection algorithm pseudocode

# Note that more sophisticated (faster) techniques exist,
# for example, when the derivative of f is known

# start with interval [a,b] where f(x) changes sign
# while interval not yet too small:
#       halve the interval; m=(a+b)/2
#       if f(x) changes sign in [a,m]:
#          continue with [a,m]   (set b = m)
#       else:
#          continue with [m,b]   (set a = m)
```

Example: bisection method (cont.)



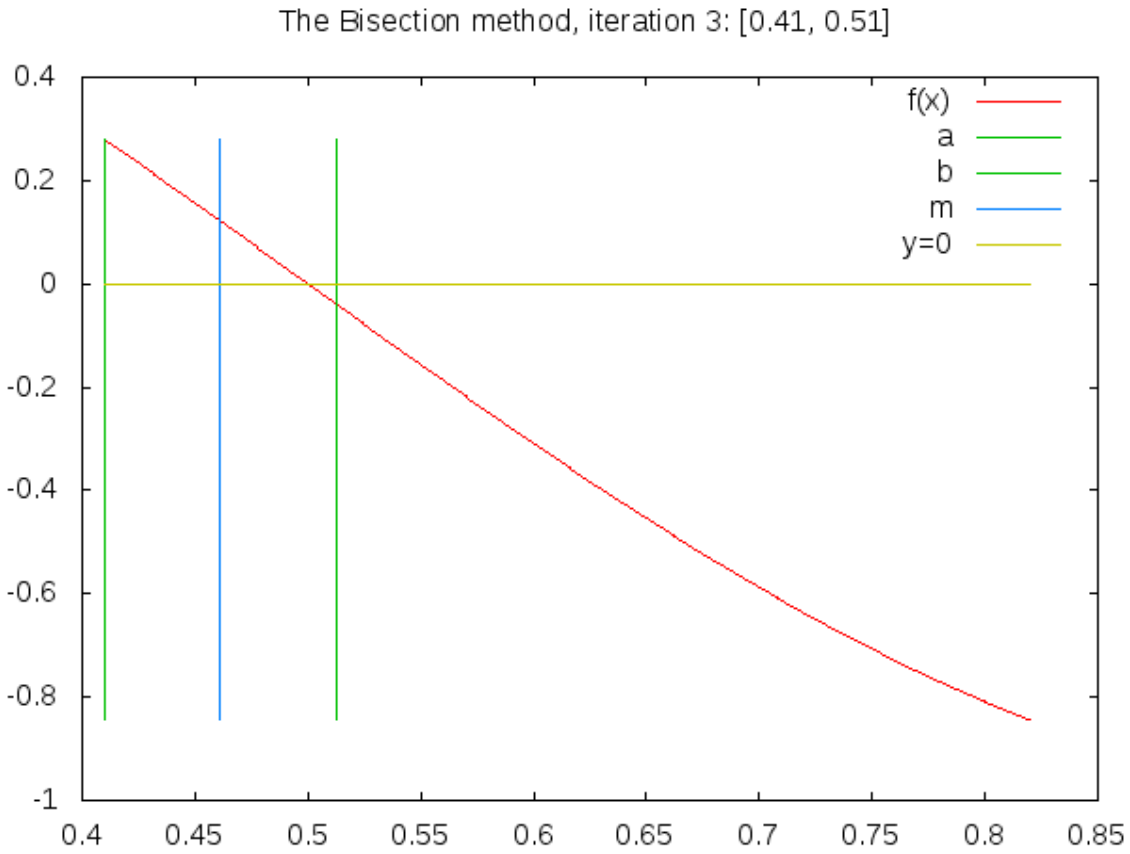
$$f(x) = \cos(\pi * x)$$

Example: bisection method (cont.)



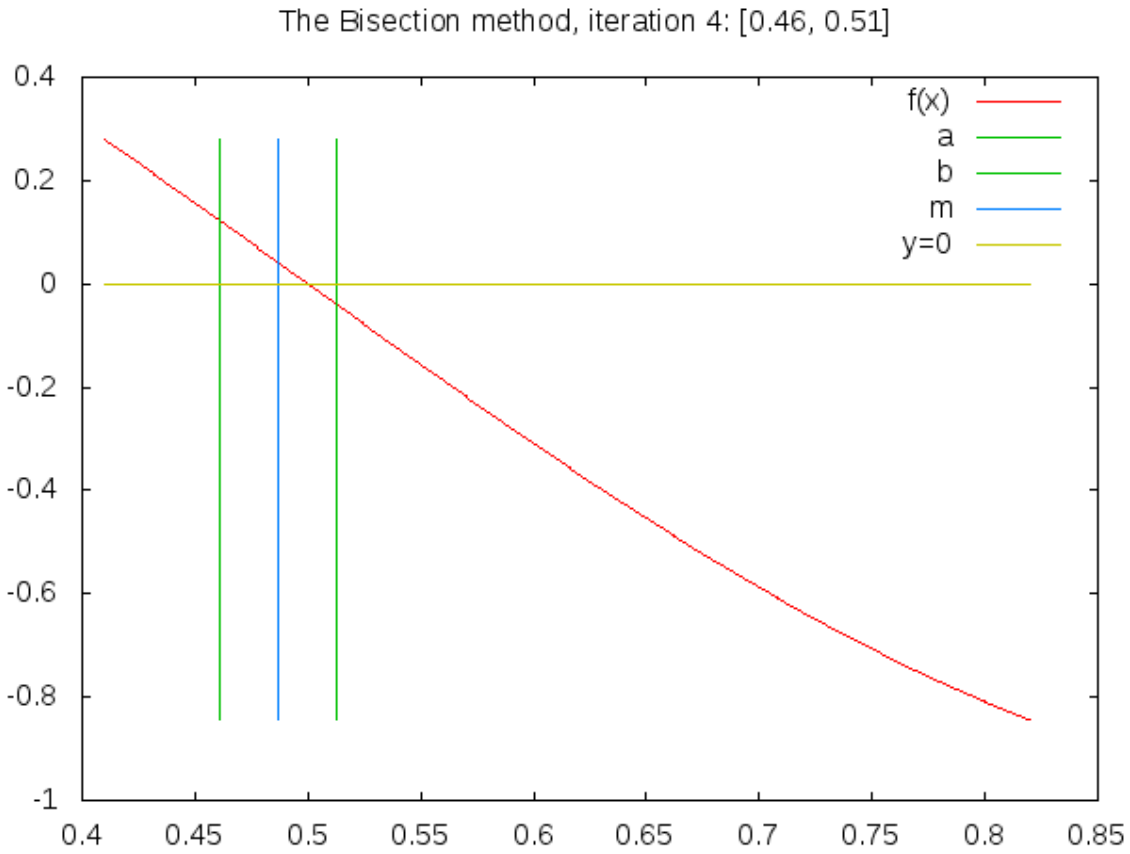
$$f(x) = \cos(\pi * x)$$

Example: bisection method (cont.)



$$f(x) = \cos(\pi * x)$$

Example: bisection method (cont.)



$$f(x) = \cos(\pi * x)$$

bisection algorithm implementation

```
def f(x):
    return 2*x - 3    # one root at x=1.5

eps = 1E-5     # tolerance, determines when to stop
a, b = 0, 10   # initial interval
fa = f(a)
if fa*f(b) > 0:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
    sys.exit(1)
i = 0          # iteration counter
while b-a > eps: # no need for abs()
                 # cannot use for loop if False at start
    i += 1
    m = (a + b)/2.0
    fm = f(m)
    if fa*fm <= 0:
        b = m  # root is in left half of [a,b]
    else:
        a = m  # root is in right half of [a,b]
        fa = fm
x = m          # this is the approximate root
```

bisection implementation in a function (mod.)

```
def bisection(f, a, b, eps=1E-15):
    fa = f(a)
    if fa*f(b) > 0:
        return None, 0

    m = 0; i = 0      # i counts iterations
    while b-a > eps:
       i += 1
       # may want to put a limit
       # on the number of iterations (how?)
       m = (a + b)/2.0
       fm = f(m)
       if fa*fm <= 0:
           b = m # root is in left half of [a,b]
       else:
           a = m # root is in right half of [a,b]
           fa = fm
    return m, i
```

(module: bisectn.py)

bisection function use

```
import bisection

def my_f(x):
    return 2*x - 3    # one root at x=1.5

root, num_iterations =
  bisection.bisection(f=my_f, -1, 20, eps=1E-10)
```

Programmeervaardigheden

CH5: Array Computing and Curve Plotting

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

Overview

- We want to plot function curves
- Store points on curves in `arrays` (≈ `lists`)
- A plot is created by connecting the consecutive points (often by a straight line)
- `arrays` are similar to `lists`, less flexible, but computationally more efficient
- It is possible to perform computations with whole arrays at once (“vectorization”)

Arrays

- In general, an n-dimensional array can be thought of as an n-dimensional matrix
- In Python, upto now, matrices were encoded as (nested) lists.
- A *vector* is a 1-dimensional array.
- In Python, when we want to perform array (or vector) computations, we use *Numerical Python* (a library) arrays for efficiency reasons.

storing (x,y) points on a curve in arrays

- Given a function y=f(x), we can easily store the x and y-values in two separate arrays by converting lists:

```
def f(x):
    return x**3          # sample function

n = 5                    # no. of points in [0,1]
dx = 1.0/(n-1)           # x spacing

xlist = [i*dx for i in range(n)]
ylist = [f(x) for x in xlist]

from numpy import *      # get access to numpy (incl. arrays)
xarr = array(xlist)      # convert list to array
yarr = array(ylist)
```

- This representation will allow for efficient plotting and vector/matrix operations

Creating arrays directly

- `numpy` comes with some functions which create arrays:

```
n = 5                          # number of points
x2 = linspace(0, 1, n)         # array with n points between 0 and 1
y2 = zeros(n)                  # array with n zeros (float data type)
for i in xrange(n):            # like 'range' but faster
    y2[i] = f(x2[i])           # unlike lists: array cannot grow
```

- List comprehensions create lists, and hence need conversion:

```
y2 = array([f(xi) for xi in x2])        # list -> array
```

Characteristics of arrays

- Arrays can only hold objects of the *same type* (as opposed to lists)
- Arrays are *more efficient* than lists, esp. with basic number types (float, int, complex)
- Computations can be performed on whole arrays at once (called *vectorization*):

```
# x is NumPy array; traditional code:
y = zeros(len(x))
for i in xrange(len(x)):
    y[i] = sin(x[i])

# equivalent, vectorized code: (does NOT work with lists!)
y = sin(x)
```

Vectorization

- `numpy` supports arithmetic operations on arrays, which correspond to the *element-wise* equivalent
- Mathematical functions in Python without if-tests automatically work for both scalar and array parameters

```
from numpy import *
x = 5
y = sin(x)  # numpy.sin delegates to math.sin

x = array([1,2,3,4,5])
y = sin(x)  # numpy.sin
```

- Behind the scenes, very efficient C (i.e., non-Python) functions are executed which incorporate for-loops

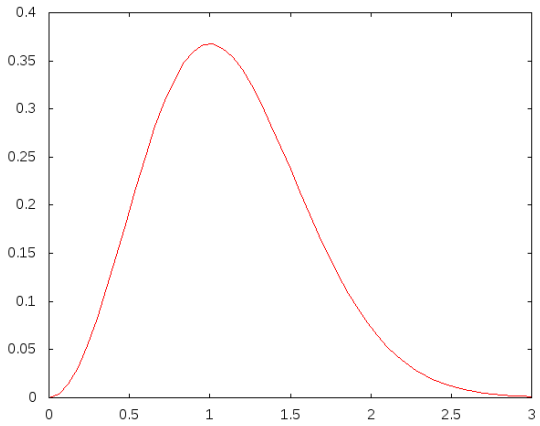
Summarizing Array Example

```
from numpy import linspace, exp, sin, pi
def f(x):
    return exp(-x)*sin(omega*x)  # numpy sin: arrays and scalars

omega = 2*pi
x = linspace(0, 5, 51)
y = f(x)

# without numpy, the same is much more elaborate

from math import exp, sin, pi
def f(x):
    return exp(-x)*sin(omega*x)

omega = 2*pi
dx = (5-0)/float(51)
x = [i*dx for i in range(51)]
y = [f(xi) for xi in x]
```

Plotting: Basics

If we store (x,y) points along a curve y = f(x) in two one-dimensional arrays x and y, simply calling plot(x,y) will draw a plot on the screen

```
from scitools.std import * # import numpy and plotting
t = linspace(0, 3, 51)     # 51 points between 0 and 3
y = t**2*exp(-t**2)        # vectorized expression
plot(t, y)                 # plot function
hardcopy('tmp1.eps') # make PostScript image for reports
hardcopy('tmp1.png') # make PNG image for web pages
```



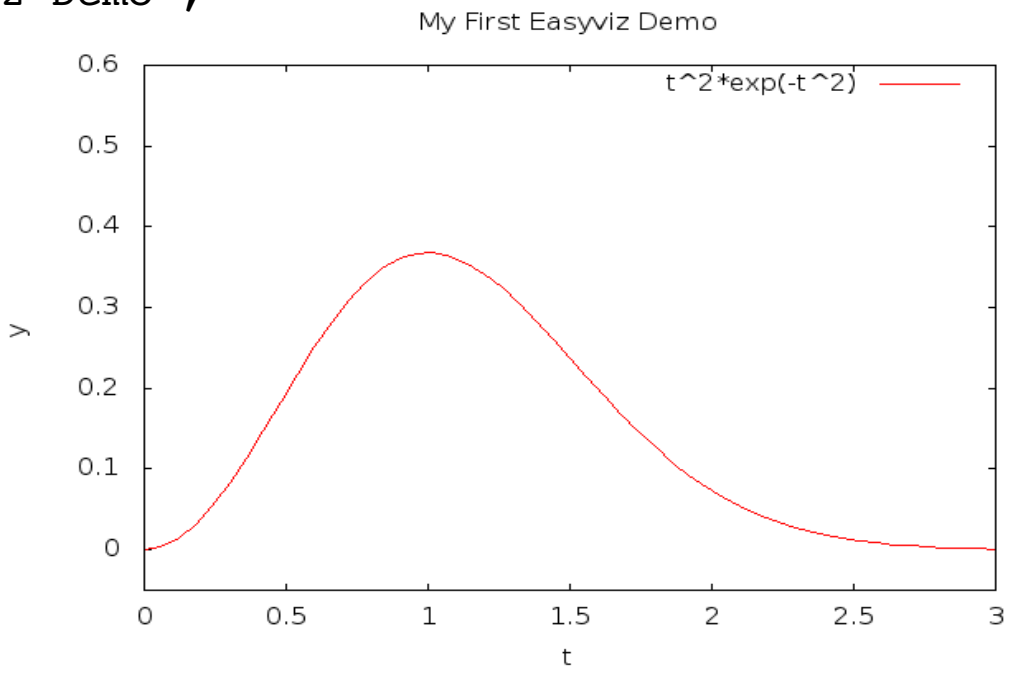
Plotting: Decorating the plot

Additional information (title, axis labels, axis extent, …) through additional function calls or parameters (which have default values)

```
def f(t):
    return t**2*exp(-t**2)
t = linspace(0, 3, 51)      # 51 points between 0 and 3
y = f(t)
plot(t, y)
xlabel('t')                 # set label for X axis
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6])    # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')
hardcopy('tmp1.eps')        # store as file on disk
```

Plotting: Decorating the plot

```
# OR: parameters instead of function calls

plot(t, y,
     xlabel='t', ylabel='y',
     legend='t^2*exp(-t^2)',
     axis=[0, 3, -0.05, 0.6],
     title='My First Easyviz Demo',
     hardcopy='tmp1.eps',
     show=True)
     # also display on
     # the screen (default)
```



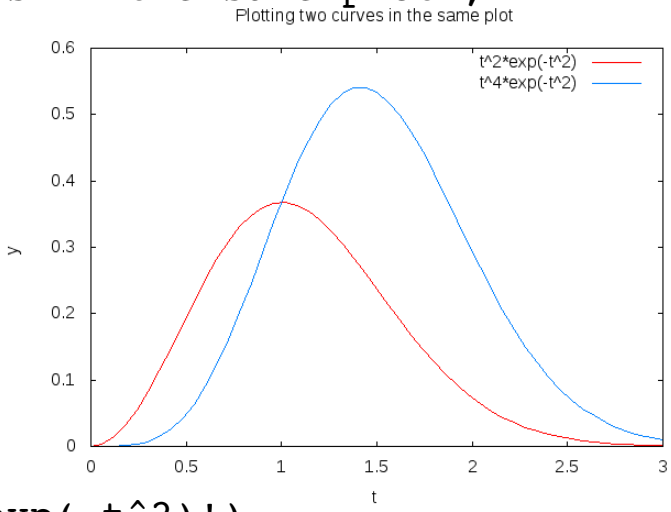
Plotting: Decorating the plot

```
# programmatically construct arguments ...

counter = 5
base_label_string = 'plot%05d_' % counter
xlabel_string = base_label_string + 't'
ylabel_string = base_label_string + 'y'
...
plot(t, y, xlabel = xlabel_string, ylabel = ylabel_string, ...)

# even the filename is a string and can hence be
# program generated

filenameBase = 'tmp'
filenameExtension = 'eps'
counter = 1
hardcopy = '%s%02d.%s' % (filenameBase, counter, filenameExtension)
```

Plotting several curves in one plot

In order to plot multiple curves, either pass additional arguments or “hold” the plot window

```
plot(t, y1, t, y2,
     xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.eps')
```




```
# equivalent to
plot(t, y1)
hold('on')
plot(t, y2)
xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')
```

Controlling line styles

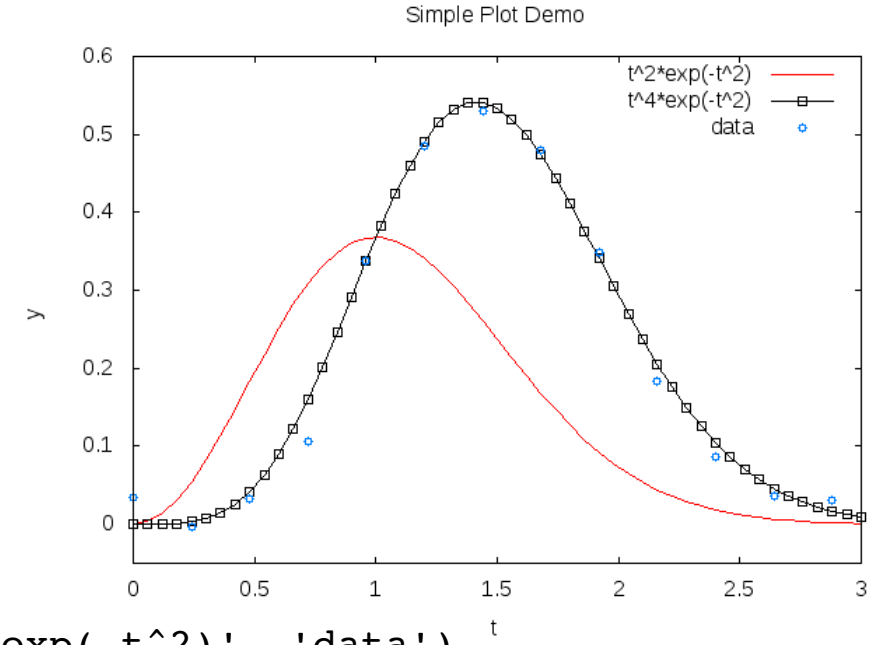
We can control the line type and color for each plot, again through additional parameters

```
plot(t, y1, 'r-')   # red (r) line (-)
hold('on')
plot(t, y2, 'bo')   # blue (b) circles (o)

# or
plot(t, y1, 'r-', t, y2, 'bo')
```

Example: two curves + random noise

Plot two functions, plus some random points around f2, every four points

```
plot(t, y1, 'r-'); hold('on'); plot(t, y2, 'ks-')
# pick out each 4 points and add random noise:
t3 = t[::4]     # slice, stride 4
random.seed(11) # fix random sequence (repeatability of experiments!)
noise = random.normal(loc=0, scale=0.02, size=len(t3))
y3 = y2[::4] + noise
```

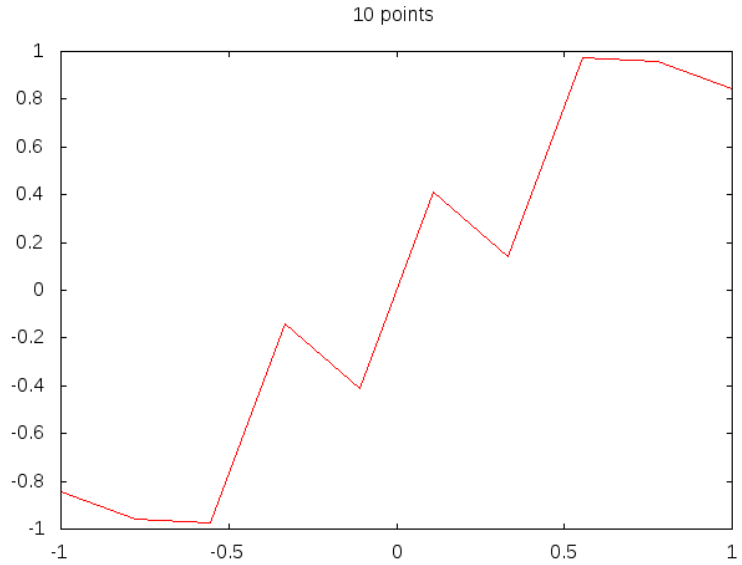


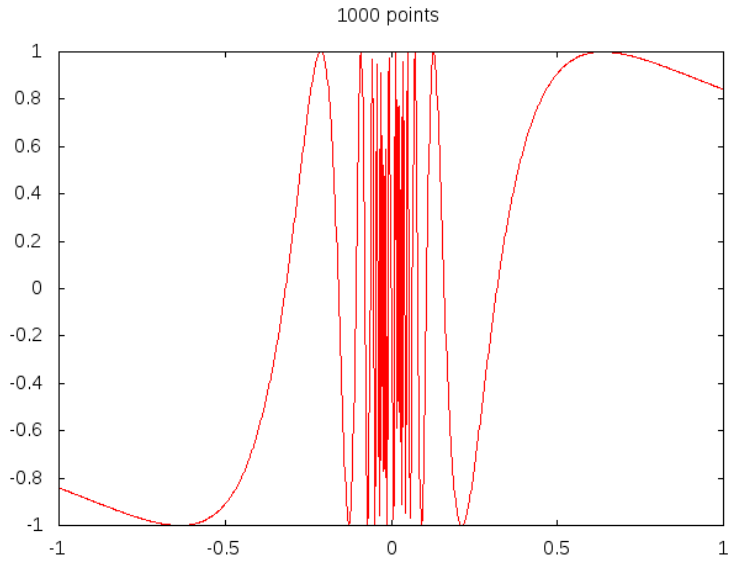

```
plot(t3, y3, 'bo')
title('Simple Plot Demo')
axis([0, 3, -0.05, 0.6])
xlabel('t'); ylabel('y')
hardcopy('tmp3.png')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data')
```

Example: Rapidly varying function

- The number of points used for a plot can be important!

```
def f(x):
    return sin(1.0/x)

x1 = linspace(-1, 1, 10)    # use 10 points
x2 = linspace(-1, 1, 1000)  # use 1000 points
plot(x1, f(x1), title='%d points' % len(x1), hardcopy='tmp4.png')
figure()           # open a second plot window
plot(x2, f(x2), title='%d points' % len(x2), hardcopy='tmp5.png')
```



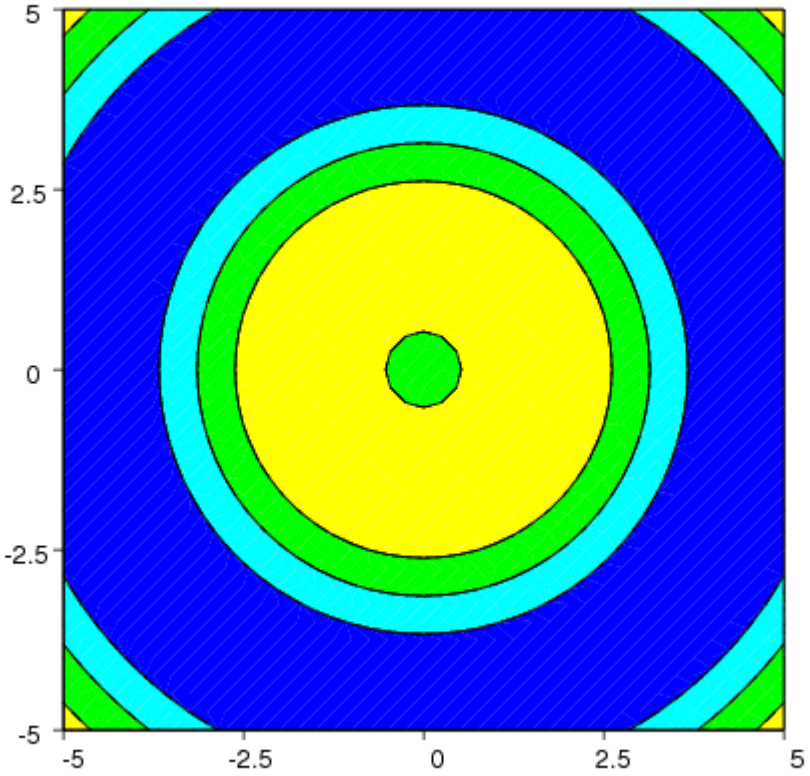


Copying arrays

- Simply assigning an array variable to another makes them aliases and **does not copy** the elements (it does, lazily, in Matlab)
- The copy() operation can be used instead
- The same holds for slices (unlike for list slices which do copy!)

```
a = x
a[-1] = q  # x[-1] is now also q!

a = x.copy()  # solution...

a = x[:]
a[-1] = q     # still changes x[-1]!
a = x[:].copy()
a[-1] = q     # does not change x[-1]
```

All you ever wanted to know about plotting



http://code.google.com/p/scitools/wiki/EasyvizDocumentation

Two-dimensional arrays (matrices)

- We often need two-dimensional arrays (matrices), e.g. for representing tables
- Indexing with arrays is more convenient than with lists

```
Cdegrees = [-30 + i*10 for i in range(3)]
Fdegrees = [9./5*C + 32 for C in Cdegrees]
table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]

print(table[2][1])
table[4][0] = 45

table2 = array(table)
print(table2[2,1])
table2[4,0] = 45
```

Higher-dimensional arrays (matrices)

```
>>> from numpy import *

>>> r = zeros((3,3,4), float)
>>> print r

array([[[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]],

       [[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]],

       [[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]]])

>>> r[0,2,3] = 5
```

Slicing two-dimensional arrays

slices of the form `start:stop:inc` can be used for each index

```
table2[:, 1]              # extract 2nd column (index 1)

table2[0:, 1]             # same

# shape is a tuple containing the array dimensions
table2[0:table2.shape[0], 1] # same

# extract 2x2 matrix from a bigger one
table2[0:2,0:2]

# extract different 2x2
table2[0:2,1:3]
```

Array functionality

```
array(ld)                copy list data ld to a numpy array
asarray(d)               make array of data d (copy if necessary)
zeros(n)                 make a vector/array of length n, with zeros (float)
zeros(n, int)            make a vector/array of length n, with int zeros
zeros((m,n), float)      make a two-dimensional array with shape (m,n)
zeros(x.shape, x.dtype)  make array with same shape and element type as x
linspace(a,b,m)          uniform sequence of m numbers between a and b
a.shape                  tuple containing a's shape
a.size                   total no of elements in a
len(a)                   length of a one-dim. array a (same as a.shape[0])
a.reshape(3,2)           return array a reshaped as 2 × 3 array
a[i]                     vector indexing
a[i,j]                   two-dim. array indexing
a[1:k]                   slice: reference data with indices 1,. . . ,k-1
a[1:8:3]                 slice: reference data with indices 1, 4,. . . ,7
b = a.copy()             copy an array
sin(a), exp(a), ...      numpy functions element-wise applicable to arrays
c = concatenate(a, b)    c contains a with b appended
c = where(cond, a1, a2)  c[i] = a1[i] if cond[i], else c[i] = a2[i]
isinstance(a, ndarray)   is True if a is an array
# scitools.numpyutils
seq(a,b,h)               uniform sequence of numbers from a to b with step h
iseq(a,b,h)              uniform sequence of integers from a to b with step h
```

Programmeervaardigheden

Appendix A: Sequences and Difference Equations

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

Sequences and Recurrence Relations

- Sequences are common in mathematics, e.g., all odd numbers, all prime numbers, …
- Sequences can be finite or infinite
- For the sequence of odd numbers 1,3,5,7,... we have a direct formula for the n-th term: $x_n = 2n+1$
- In general, such general n-th term formulas are rare
- However, we can often define the sequence through one or more “difference equations” (= recurrence relations)
- With the help of loops and arrays it is very easy in Python to generate a sequence from a recurrence relation
- Recurrence relations often appear as the result of (Taylor) series expansions or discretization (of continuous functions, derivatives, integrals and differential equations).

Example: Modeling interest rates

- Recurrence relation:

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

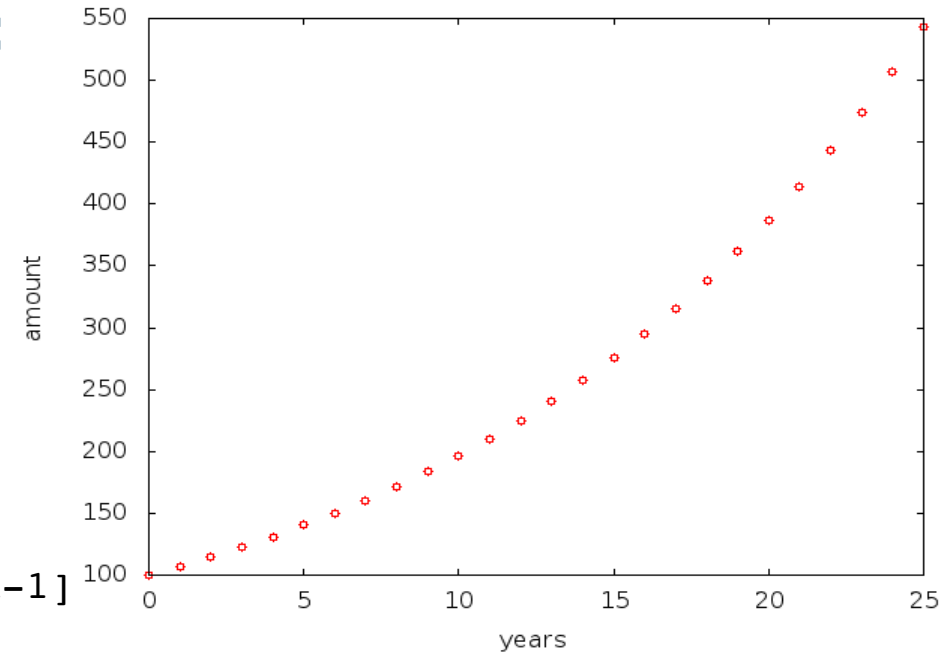
x_n : capital at year n

x_{n-1}: capital at year n-1

p: interest rate

- Solve by simulation:

```
from scitools.std import *
x0 = 100          # initial amount
p = 7             # interest rate
N = 25            # 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')
```



Example: Varying daily interest rates

- Start out with an array `p` containing annual interest rates for every day
- Divide all elements by 360 to get one model of daily interest rates, and store in another array `r`

```
r = p/360.0    # vectorized operation: daily interest rate
N = 412        # number of days
index_set = range(N+1)
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]
```

Example: Fibonacci numbers

- Recurrence relation: $x_n = x_{n-1} + x_{n-2}$

```
N = int(raw_input('N=?' ))
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]
```

- Run with N=100:

```
...
89 2880067194370816120
90 4660046610375530309
91 7540113804746346429
__main__:2: RuntimeWarning: overflow encountered in long_scalars
92 -6246583658587674878
93 1293530146158671551
94 -4953053512429003327
...
```

Example: Logistic Growth

- The model for growth of money in a bank is exponential: $x_n = x_0 C^n \ \ (= x_0 e^{n \ln C})$
- Populations also exhibit this type of growth in case of unlimited resources
- A *logistic model* takes into account a maximum of M individuals supported by the environment:

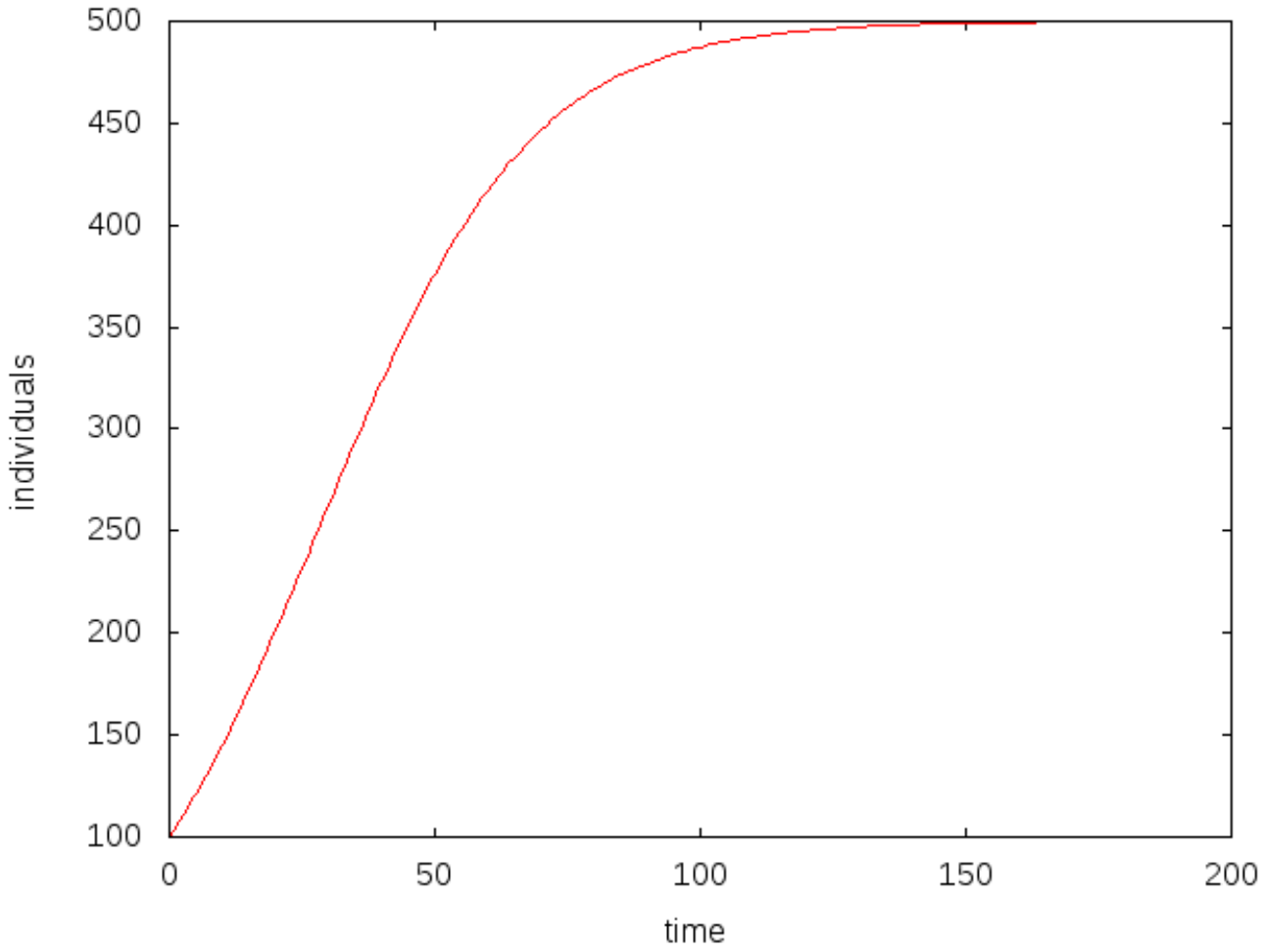
$$x_n = x_{n-1} + \frac{\rho}{100} x_{n-1} \left(1 - \frac{x_{n-1}}{M}\right)$$

ρ: growth rate
x_n: individuals at time n
M: maximum number of individuals

- In a Python program:

```
x[n] = x[n-1] + (rho/100.0)*x[n-1]*(1 - x[n-1]/float(M))
```

Logistic Growth: Plot



M: 500
x_0: 100

Example: Newton's root finding method

- Newton's method as a recurrence relation:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

- as n increases, we hope x_n approaches a root of f
- We will now not stop after N steps, but when $f(x_n)$ is close enough to zero (or after N steps...)

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100):
    f_value = f(x)
    n = 0
    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)
        info.append((x, f_value))
    return x, info
```

Programmeervaardigheden

Appendix B: Discrete Calculus

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

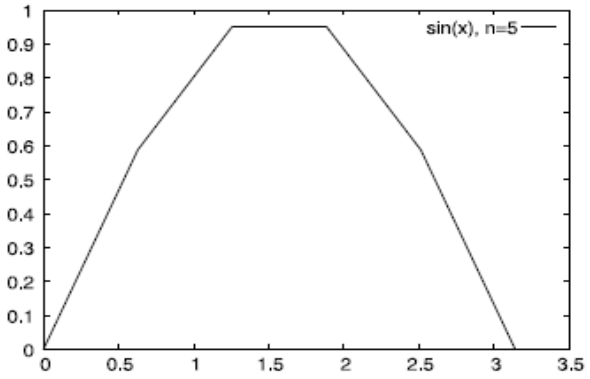
problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

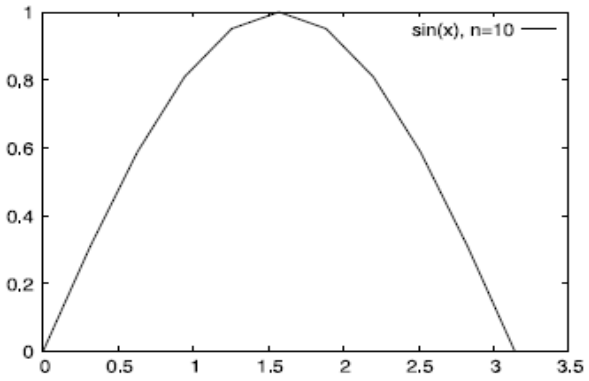
programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers;
lists or arrays to store sequences of numbers)
-> design and implement program
-> test program

Discretising

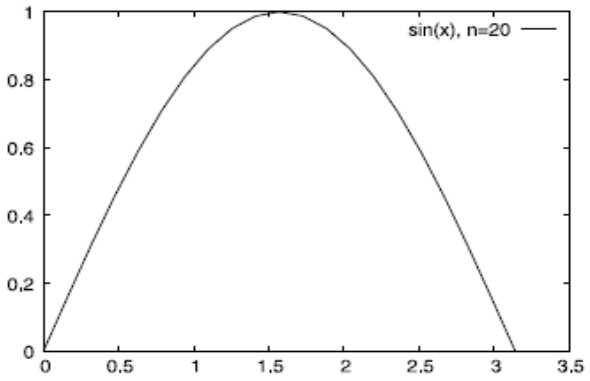
```
from scitools.std import *

n = int(sys.argv[1])

x = linspace(0, pi, n+1)
s = sin(x)
plot(x, s, legend='sin(x), n=%d' % n, hardcopy='tmp.eps')
```



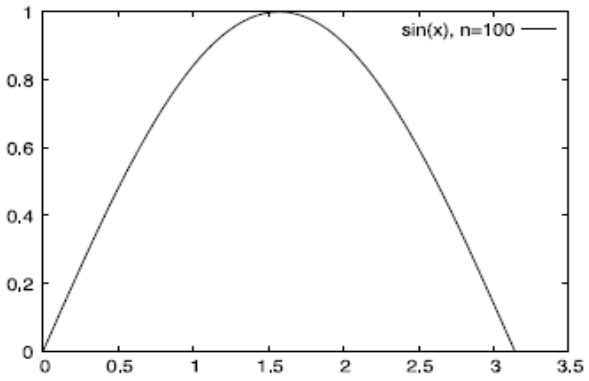
(a) $n = 5$



(b) $n = 10$



(c) $n = 20$



(d) $n = 100$

Exceptions

```
#!/usr/bin/env python
from scitools.std import *

try:
    n = int(sys.argv[1])
except:
    print "usage: %s n" %sys.argv[0]
    sys.exit(1)

x = linspace(0, pi, n+1)
s = sin(x)
plot(x, s, legend='sin(x), n=%d' % n, hardcopy='tmp.eps')
```

```
from numpy import *
import sys

xp = eval(sys.argv[1])
n = int(sys.argv[2])

def S_k(k):
    return s[k] + \
           ((s[k+1] - s[k])/(x[k+1] - x[k]))*(xp - x[k])
h = pi/n
x = linspace(0, pi, n+1)
s = sin(x)
k = int(xp/h)

print 'Approximation of sin(%s):  ' % xp, S_k(k)
print 'Exact value of sin(%s):    ' % xp, sin(xp)
print 'Eror in approximation:     ', sin(xp) - S_k(k)
```

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k)$$

$$k = \lfloor x/h \rfloor$$

Terminal

```
eval_sine.py'sqrt(2)' 5
Approximation of sin(1.41421356237):   0.951056516295
Exact value of sin(1.41421356237):     0.987765945993
Eror in approximation:      0.0367094296976
```

Terminal

```
eval_sine.py'sqrt(2)' 10
Approximation of sin(1.41421356237):   0.975605666221
Exact value of sin(1.41421356237):     0.987765945993
Eror in approximation:      0.0121602797718
```

Terminal

```
eval_sine.py'sqrt(2)' 20
Approximation of sin(1.41421356237):   0.987727284363
Exact value of sin(1.41421356237):     0.987765945993
Eror in approximation:      3.86616296923e-05
```

Discretizing a Function

$$h = \frac{b-a}{n} \qquad x_i = a + ih \text{ for } i = 0, 1, \ldots, n$$
$$y_i = f(x_i) \text{ for } i = 0, 1, \ldots, n$$

```
def discrete_func(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    for i in xrange(len(x)):
        y[i] = func(x[i])
    return x, y

from scitools.std import *

f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])
f = StringFunction(f_formula)

x, y = discrete_func(f, a, b, n)
plot(x, y)
```

Differentiating a Function

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Differentiating a Function

```
def diff(f, x, h):
    return (f(x+h) - f(x))/float(h)

from math import *
import sys

x = eval(sys.argv[1])
h = eval(sys.argv[2])

approx_deriv = diff(sin, x, h)
exact = cos(x)
print 'The approximated value is: ', approx_deriv
print 'The correct value is:      ', exact
print 'The error is:              ', exact - approx_deriv
```

Running the program for $x = 1$ and $h = 1/1000$ gives

Terminal

```
forward_diff.py 1 0.001
The approximated value is:  0.53988148036
The correct value is:       0.540302305868
The error is:               0.000420825507813
```

Differentiating a Function

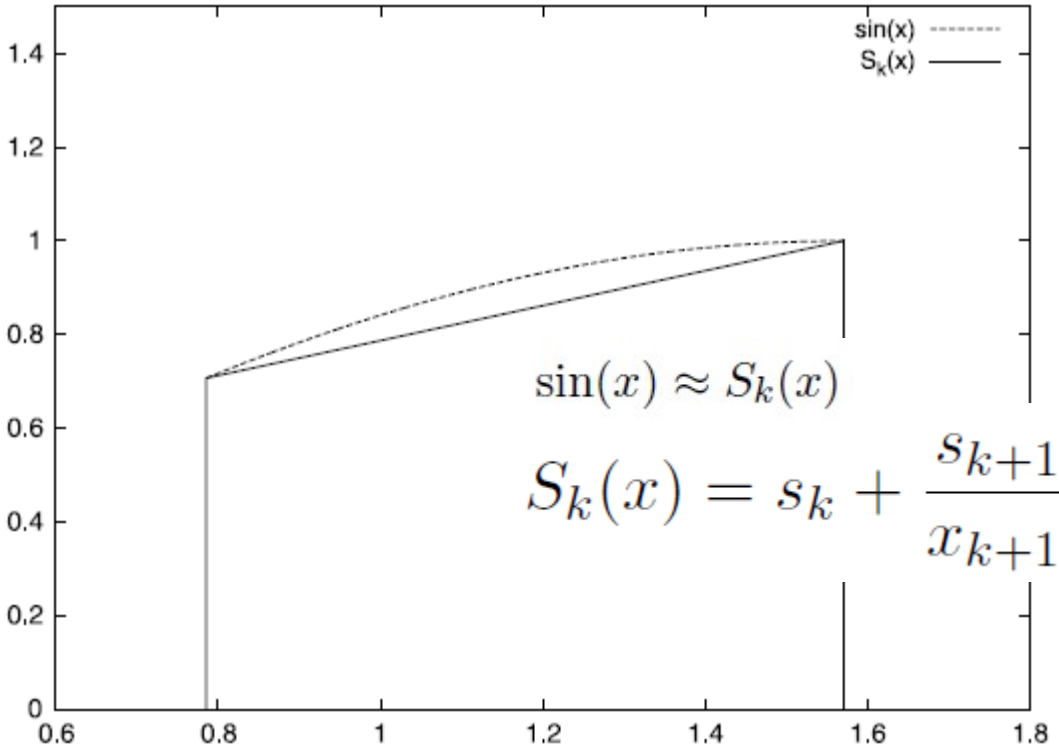
```
def diff(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    z = zeros(len(x))
    h = (b-a)/float(n)
    for i in xrange(len(x)):
        y[i] = func(x[i])
    for i in xrange(len(x)-1):
        z[i] = (y[i+1] - y[i])/h
    z[n] = (y[n] - y[n-1])/h
    return y, z

from scitools.std import *
f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])

f = StringFunction(f_formula)
y, z = diff(f, a, b, n)
plot(x, y, 'r-', x, z, 'b-',
     legend=('function', 'derivative'))
```

Integrating a Function

$$\int_0^{\pi} \sin(x)dx$$

$$\int_0^{\pi} \sin(x)dx = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x)dx$$

Integrating a Function



$$\sin(x) \approx S_k(x)$$

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k)$$

$$\int_{x_k}^{x_{k+1}} \sin(x)dx \approx \int_{x_k}^{x_{k+1}} S_k(x)dx$$

Integrating a Function

$$\int_{x_k}^{x_{k+1}} \sin(x)dx \approx \int_{x_k}^{x_{k+1}} S_k(x)dx$$

$$\int_{x_k}^{x_{k+1}} \sin(x)dx \approx \frac{1}{2}(s_{k+1} + s_k)(x_{k+1} - x_k)$$

$$= \frac{h}{2}(s_{k+1} + s_k).$$

Integrating a Function

$$\int_0^\pi \sin(x)dx = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x)dx$$

$$\approx \sum_{k=0}^{n-1} \frac{h}{2}(s_{k+1} + s_k),$$

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2} \sum_{k=0}^{n-1} (s_{k+1} + s_k).$$

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2} \left[s_0 + 2 \sum_{k=1}^{n-1} s_k + s_n \right]$$

Integrating a Function

```
def trapezoidal(f, a, b, n):
    h = (b-a)/float(n)
    I = f(a) + f(b)
    for k in xrange(1, n, 1):
        x = a + k*h
        I += 2*f(x)
    I *= h/2
    return I
```

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right]$$

```
from math import *
from scitools.StringFunction import StringFunction
import sys

def test(argv=sys.argv):
    f_formula = argv[1]
    a = eval(argv[2])
    b = eval(argv[3])
    n = int(argv[4])

    f = StringFunction(f_formula)
    I = trapezoidal(f, a, b, n)
    print 'Approximation of the integral: ', I

if __name__ == '__main__':
    test()
```

Integrating a Function

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

def g(t):
    return -a*exp(-a*t)*sin(pi*w*t) + pi*w*exp(-a*t)*cos(pi*w*t)

def G(t):  # integral of g(t)
    return exp(-a*t)*sin(pi*w*t)

a = 0.5
w = 1.0
t1 = 0
t2 = 4
exact = G(t2) - G(t1)
for n in 2, 4, 8, 16, 32, 64, 128, 256, 512:
    approx = trapezoidal(g, t1, t2, n)
    print 'n=%3d approximation=%12.5e  error=%12.5e' % \
          (n, approx, exact-approx)
```

Taylor Series

$$f(x_0 + h) \approx \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0)$$

$$f(x_0 + h) = \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0) + O(h^{n+1})$$

Programmeervaardigheden

Appendix C: Differential Equations

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

problem domain (e.g., chemistry, physics, biology, ...)
-> precisely describe domain problem
-> precisely describe domain solution
-> translate to precisely described mathematical problem

mathematical domain (e.g., linear algebra, differential equations, ...)
-> precisely describe mathematical solution
this may involve mathematical approximations
(e.g. only first 2 terms of a Taylor expansion)
-> translate to precisely described programming problem

programming domain
-> precisely describe programming solution
this may involve programming approximations
choose optimal data structures and algorithms
(e.g., floating point numbers are approximations of Real numbers;
lists or arrays to store sequences of numbers)
-> design and implement program
-> test program

Differential Equations

- Ordinary Differential Equations (ODEs)

$$u'(t) = u(t)$$

- Partial Differential Equations (PDEs)

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

$$u'(t) = t^3$$

analytical solution (family of solutions):

$$u(t) = \frac{1}{4}t^4 + C$$

Initial Condition (IC)
required to determine single solution

$$u(0) = 1$$

$$u'(t) = t^3$$

solution through explicit integration

$$\begin{aligned} u(t) &= u(0) + \int_0^t u'(\tau) d\tau \\ &= 1 + \int_0^t \tau^3 d\tau \\ &= 1 + \frac{1}{4} t^4. \end{aligned}$$

$$u'(t) = f(t) \qquad u(0) = u_0$$

$$u(t) = u_0 + \int_0^T f(\tau)d\tau$$

$$\int_0^T f(\tau)d\tau \qquad \tau_i = ih, \text{ and } y_i = f(\tau_i)$$

$$\int_0^T f(\tau)d\tau \approx \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right]$$

$$u(t) \approx u_0 + \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right]$$

Differential Equations

```
def integrate(T, n, u0):
    h = T/float(n)
    t = linspace(0, T, n+1)
    I = f(t[0])
    for k in iseq(1, n-1, 1):
        I += 2*f(t[k])
    I += f(t[-1])
    I *= (h/2)
    I += u0
    return I

from scitools.std import *
f_formula = sys.argv[1]
T  = eval(sys.argv[2])
u0 = eval(sys.argv[3])
n  = int(sys.argv[4])

f = StringFunction(f_formula, independent_variables='t')
print "Numerical solution of u'(t)=t**3: ", integrate(T, n, u0)
```

$$u(t) \approx u_0 + \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right]$$

Differential Equations

Terminal

```
integrate_ode.py't*exp(t**2)' 2 0 10
Numerical solution of u'(t)=t**3:  28.4066160877
```

Terminal

```
integrate_ode.py't*exp(t**2)' 2 0 20
Numerical solution of u'(t)=t**3:  27.2059977451
```

Terminal

```
integrate_ode.py't*exp(t**2)' 2 0 50
Numerical solution of u'(t)=t**3:  26.86441489
```

Terminal

```
integrate_ode.py't*exp(t**2)' 2 0 100
Numerical solution of u'(t)=t**3:  26.8154183399
```

$$\frac{1}{2}e^{2^2} - \frac{1}{2} \approx 26.799$$

Differential Equations

$$u'(t) = u(t) \quad u(0) = 1$$

$$\Delta t = 1/n \quad t_k = k\Delta t \qquad u(t_{k+1}) = u(t_k) + \Delta t u'(t_k) + O(\Delta t^2)$$

$$u'(t) = u(t) \qquad u'(t_k) \approx \frac{u(t_{k+1}) - u(t_k)}{\Delta t}$$

$$\frac{u(t_{k+1}) - u(t_k)}{\Delta t} \approx u(t_k)$$

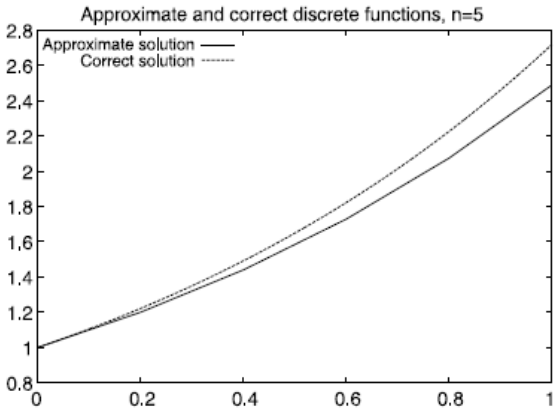
$$\frac{u_{k+1} - u_k}{\Delta t} = u_k$$

$$u_{k+1} = (1 + \Delta t)u_k$$

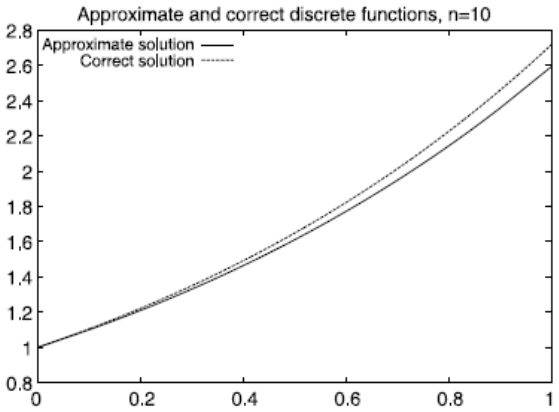
Differential Equations

```
def compute_u(u0, T, n):
    """Solve u'(t)=u(t), u(0)=u0 for t in [0,T] with n steps."""
    t = linspace(0, T, n+1)
    t[0] = 0
    u = zeros(n+1)
    u[0] = u0
    dt = T/float(n)
    for k in range(0, n, 1):
        u[k+1] = (1+dt)*u[k]
        t[k+1] = t[k] + dt
    return u, t

from scitools.std import *
n  = int(sys.argv[1])

# Special test case: u'(t)=u, u(0)=1, t in [0,1]
T = 1; u0 = 1
u, t = compute_u(u0, T, n)
plot(t, u)
tfine = linspace(0, T, 1001) # for accurate plot
v = exp(tfine)               # correct solution
hold('on')
plot(tfine, v)
legend(['Approximate solution', 'Correct function'])
title('Approximate and correct discrete functions, n=%d' % n)
savefig('tmp.eps')
```

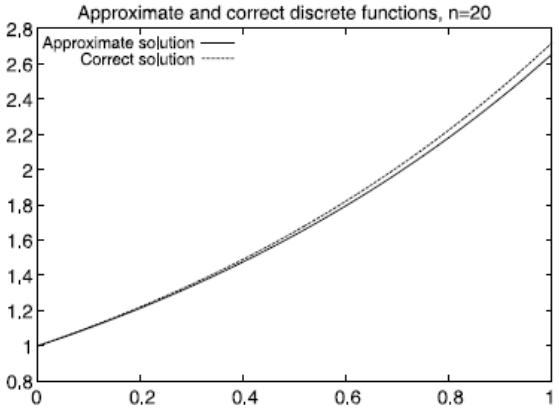
Differential Equations



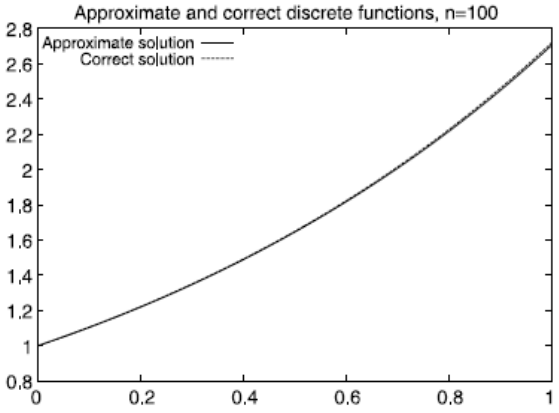
(a) $n = 5$



(b) $n = 10$



(c) $n = 20$

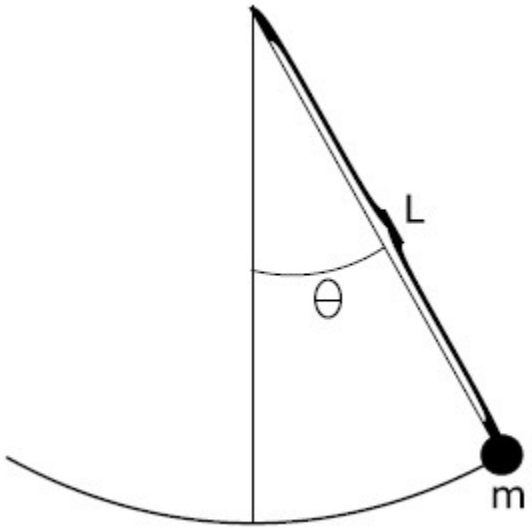


(d) $n = 100$

Differential Equations

$$\theta''(t) + \alpha \sin(\theta) = 0$$

$$\alpha = g/L$$



$$\theta'(t) = v(t),$$
$$v'(t) = -\alpha \sin(\theta)$$

$$\theta(0) = \theta_0$$
$$v(0) = v_0$$

$$\frac{\theta_{k+1} - \theta_k}{\Delta t} = v_k,$$
$$\frac{v_{k+1} - v_k}{\Delta t} = -\alpha \sin(\theta_k)$$

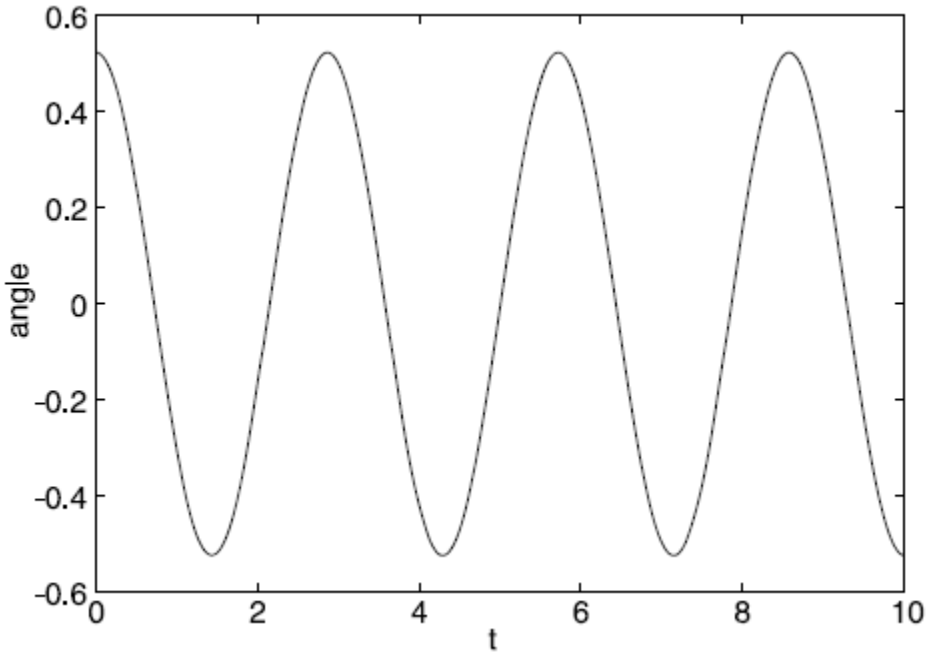
Differential Equations

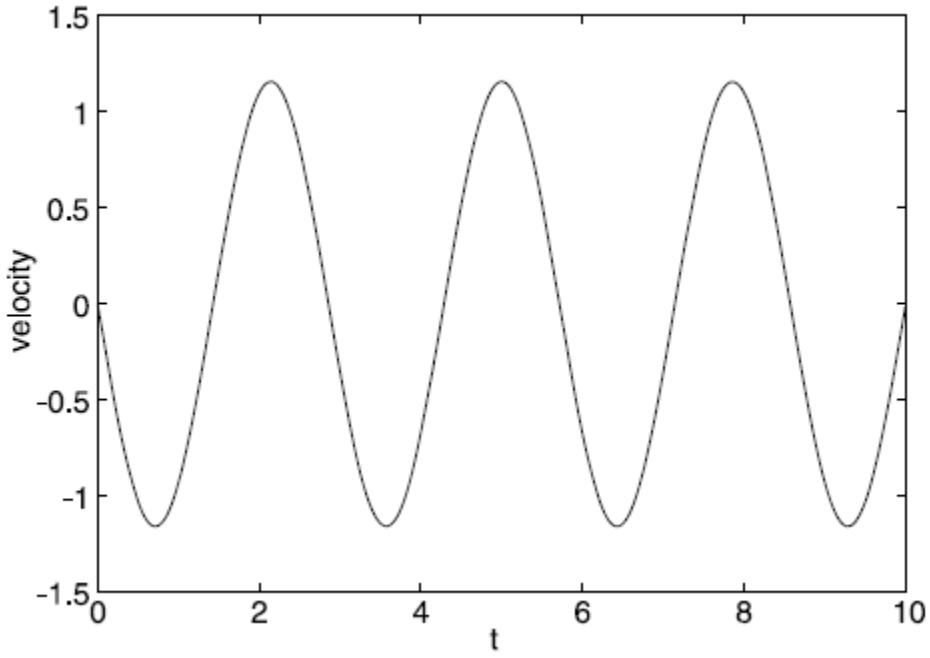
```
def pendulum(T, n, theta0, v0, alpha):
    """Return the motion (theta, v, t) of a pendulum."""
    dt = T/float(n)
    t = linspace(0, T, n+1)
    v = zeros(n+1)
    theta = zeros(n+1)
    v[0] = v0
    theta[0] = theta0
    for k in range(n):
        theta[k+1] = theta[k] + dt*v[k]
        v[k+1] = v[k] - alpha*dt*sin(theta[k+1])
    return theta, v, t

from scitools.std import *
n = int(sys.argv[1])
T = eval(sys.argv[2])
v0 = eval(sys.argv[3])
theta0 = eval(sys.argv[4])
alpha = eval(sys.argv[5])

theta, v, t = pendulum(T, n, theta0, v0)
plot(t, v, xlabel='t', ylabel='velocity')
figure()
plot(t, theta, xlabel='t', ylabel='velocity')
```

$$\frac{\theta_{k+1} - \theta_k}{\Delta t} = v_k,$$

$$\frac{v_{k+1} - v_k}{\Delta t} = -\alpha \sin(\theta_k).$$

Differential Equations





Programmeervaardigheden

CH6: Files, Strings and Dictionaries

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

Reading data from a file

- A file is a *sequence of characters* (text)
- We can read ***text*** *in the file* into ***strings*** *in a program*
- Basic recipe:

```
infile = open('myfile.dat', 'r')
# read next line:
line = infile.readline()

# OR: read lines one by one:
for line in infile:
    <process line>

# OR: load all lines into a list of strings (lines):
lines = infile.readlines()
for line in lines:
    <process line>

infile.close()     # close the file for access by others
```

Example: Reading a file with numbers

Calculate the sum and average of all numbers in a file (each line contains a number, e.g., “234.56”):

```
infile = open('data1.txt', 'r')
lines = infile.readlines()
infile.close()
mean = 0
for line in lines:
    number = float(line)  # convert string to float!!!
    mean = mean + number
mean = mean/len(lines)    # number of lines read
print mean
```

Example: A mixture of text and numbers

- Calculate the sum and average of all numbers in a file (each line contains a word and a number, e.g. “Jan 81.2”):

```
infile = open('data1.txt', 'r')
lines = infile.readlines()
infile.close()
mean = 0
for line in lines:
    words = line.split()      # list of words (delimiter ' ')
    number = float(words[1]) # second “word” is a number
    mean = mean + number
mean = mean/len(lines)
print mean
```

- line.split(delimiter) splits a string into the words separated by the delimiter (e.g. “,”, ”;”, “:”, ...)

Writing data to a file

- Collect what you want to write into strings and call the 'write' operation

```
outfile = open('myfile.txt', 'w') # new file (or overwrite)
outfile = open('myfile.txt', 'a') # append to existing file

outfile.write(string)

outfile.write(string + '\n')   # newlines are not automatic!

outfile.write('5')     # numbers as strings!

outfile.write('%14.8f' % myNumber)  # format

outfile.close()        # close the file to consolidate data!
```

- When working with files, do not forget to close them!

Dictionaries

- Lists and arrays are **collections** of objects, indexed by an integer index
- Dictionaries are lists which can use arbitrary (constant) objects, e.g. text, as index (*key*)
- Useful for associating data, e.g. names with numbers

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
# or
temps = dict(Oslo=13, London=15.4, Paris=17.5)

# application:
print 'The temperature in London is', temps['London']

temps['Madrid'] = 40.0   # add element
for city in temps:
    print (city, temps[city])

print temps.keys()      # list of cities
print temps.values()    # list of temperatures
```

Dictionary functionality

```
a = {}                                  # initialize an empty dictionary
a = {'point': [2,7], 'value':  3}       # initialize a dictionary
a = dict(point=[2,7], value=3)          # initialize a dictionary
a['hide'] = True                        # add new key-value pair to a dctnry
a['point']                              # get value corresponding to key point
'value' in a                            # True if value is a key in the dctnry
del a['point']                          # delete key-value pair from the dctnry
a.keys()                                # list of keys
a.values()                              # list of values
len(a)                                  # number of key-value pairs in dctnry
for key in a:                           # loop over keys in unknown order
for key in sorted(a.keys()):            # loop over keys in lexicographic order
isinstance(a, dict)                     # is True if a is a dictionary
```

Strings

- Strings can be considered to be tuples of characters
- Substrings correspond to slices, characters to elements
- Strings support some useful operations

```
s = 'Berlin: 18.4 C at 4 pm'
s[1]   # 'e'
s[8:]  # '18.4 C at 4 pm'

s.find('pm')   # 20   (start position of substring)
s.find('Oslo') # -1   (not found)

s.replace(' ', '—')   # 'Berlin:--18.4--C--at--4--pm'

s.split(':') # ['Berlin', ' 18.4 C at 4 pm']

q.splitlines() # use cross-platform newline to split ...
q.strip() # strip off whitespace on both sides
```

Some string operations

```
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]           # extract substring
s.find(':')       # index where first ':' is found
s.split(':')      # split into substrings
s.split()         # split wrt whitespace
'Berlin' in s     # test if substring is in s
s.replace('18.4', '20')
s.lower()         # lower case letters only
s.upper()         # upper case letters only
s.split()[4].isdigit() # true if fifth word is a number
s.strip()         # remove leading/trailing blanks
', '.join(list_of_words)  # concatenate using ', '
```

Example: Read pairs of numbers from a file

- Read pairs of numbers from a file organised like this:

```
(1.3,0)   (-1,2)     (3,-1.5)
(0,1)     (1,0)      (1,1)
(0,-0.01) (10.5,-1)  (2.5,-2.5)
```

- Python program:

```
lines = open('read_pairs.dat', 'r').readlines()

pairs = []   # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1] # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1); n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair) # add 2-tuple
```

Programmeervaardigheden

Differential Equations

Hans Vangheluwe

(based on slides by Hans Petter Langtangen, uni. Oslo and updates by Hans Schippers, UA)

MSDL/AnSyMo, University of Antwerp

2nd Semester 2013-2014

Differential Equations

- A (ordinary) differential equation (ODE) has the form:

$$u'(t) = f(u(t), t)$$

  and the solution is a function u(t)
- An initial condition $u(0)=u_0$ ensures a unique solution
- Some examples of ODEs:

$$u' = \alpha u$$ exponential growth

$$u' = \alpha u\left(1-\frac{u}{R}\right)$$ logistic growth

$$u' = -b|u|u+g$$ velocity of body in fluid

Solving a general ODE numerically

- Given u'=f(u,t) and $u(0)=u_0$, the Forward Euler method generates a sequence $u_1,u_2,u_3,...$ values of u at times $t_1,t_2,t_3,...$:

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)$$

  where $t_k = k\Delta t$

- General idea of implementation:

```
u = [u0]; t = [0]
...
for k in range(n):
  unew = u[k] + dt * f(u[k],t[k])
  u.append(unew)
  t.append(t[k] + dt)
...
```

Forward Euler function

```
def ForwardEuler(f, dt, u0, T):
    """Solve u'=f(u,t), u(0)=u0, in steps of dt until t=T"""

    u = []; t = []   # u[k] is the solution at time t[k]
    u.append(u0)
    t.append(0)
    n = int(round(T/dt))
    for k in range(n):
        unew = u[k] + dt*f(u[k], t[k])
        u.append(unew)
        tnew = t[-1] + dt
        t.append(tnew)
    return numpy.array(u), numpy.array(t)
```

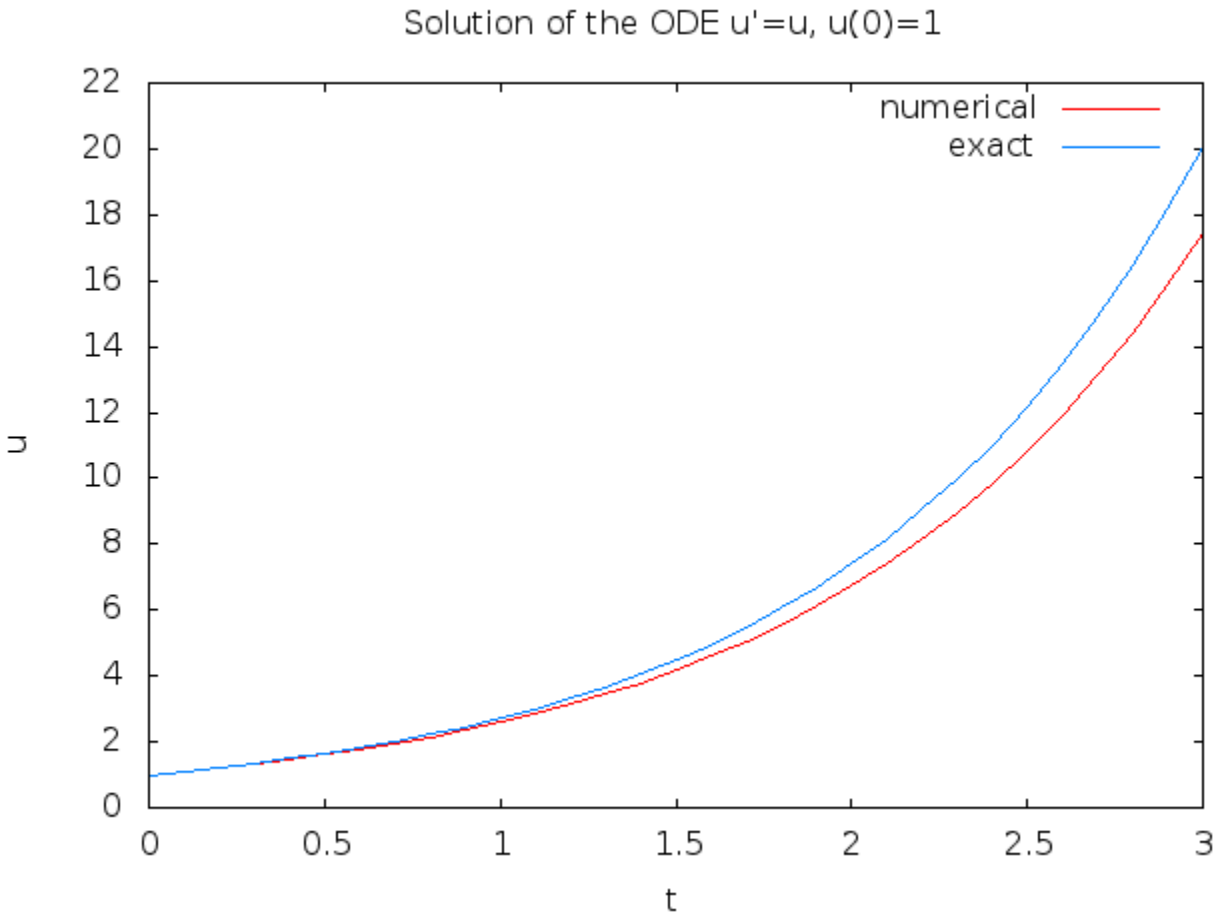
Example ODE

Solve u'=u, u(0)=1 for t ∈ [0,3] with Δt = 0.1

```
from scitools.std import plot, exp

def f(u, t):
    return u

u0 = 1
T = 3
dt = 0.1
u, t = ForwardEuler(f, dt, u0, T)

u_exact = exp(t)  # analytic (exact) solution

plot(t, u, 'r-', t, u_exact, 'b-',
     xlabel='t', ylabel='u', legend=('numerical', 'exact'),
     title="Solution of the ODE u'=u, u(0)=1")
```

Example ODE

Solve u'=u, u(0)=1 for t ∈ [0,3] with Δt = 0.1



Harmonic Oscillator

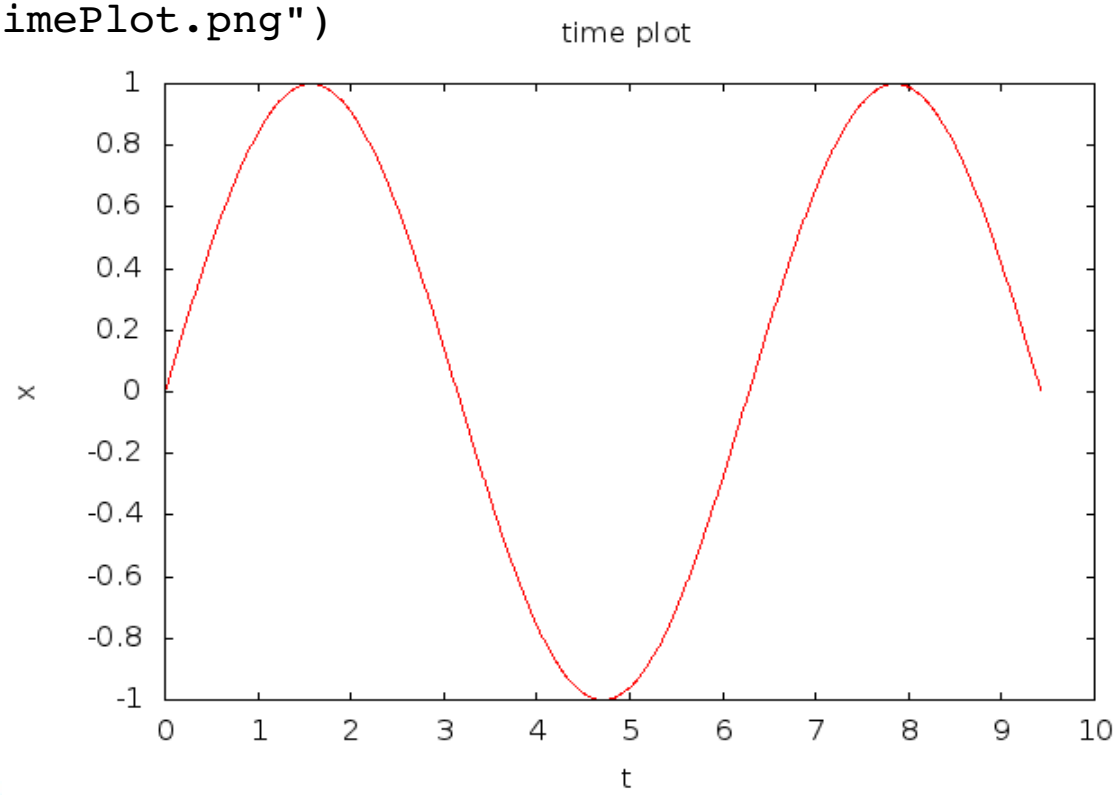
```
# d²x/dt² = -x; x(0) = 0, dx/dt(0) = 1

# dx/dt = y   ; x(0) = 0
# dy/dt = -x  ; y(t) = 1

# du/dt = rhs(u,t)
# u = [x,y]
# u0 = u(0) = [0, 1]
```

Harmonic Oscillator

```
#
http://docs.scipy.org/doc/scipy/reference/generated/scipy.inte
grate.odeint.html#scipy.integrate.odeint

from scipy.integrate import odeint

from numpy import pi, arange
t = arange(0, 3*pi, 0.01)

# initial conditions
u0 =[0,1]

# right-hand-sides of equations
def rhs(vectarg, time):
   return [vectarg[1],-vectarg[0]]
```

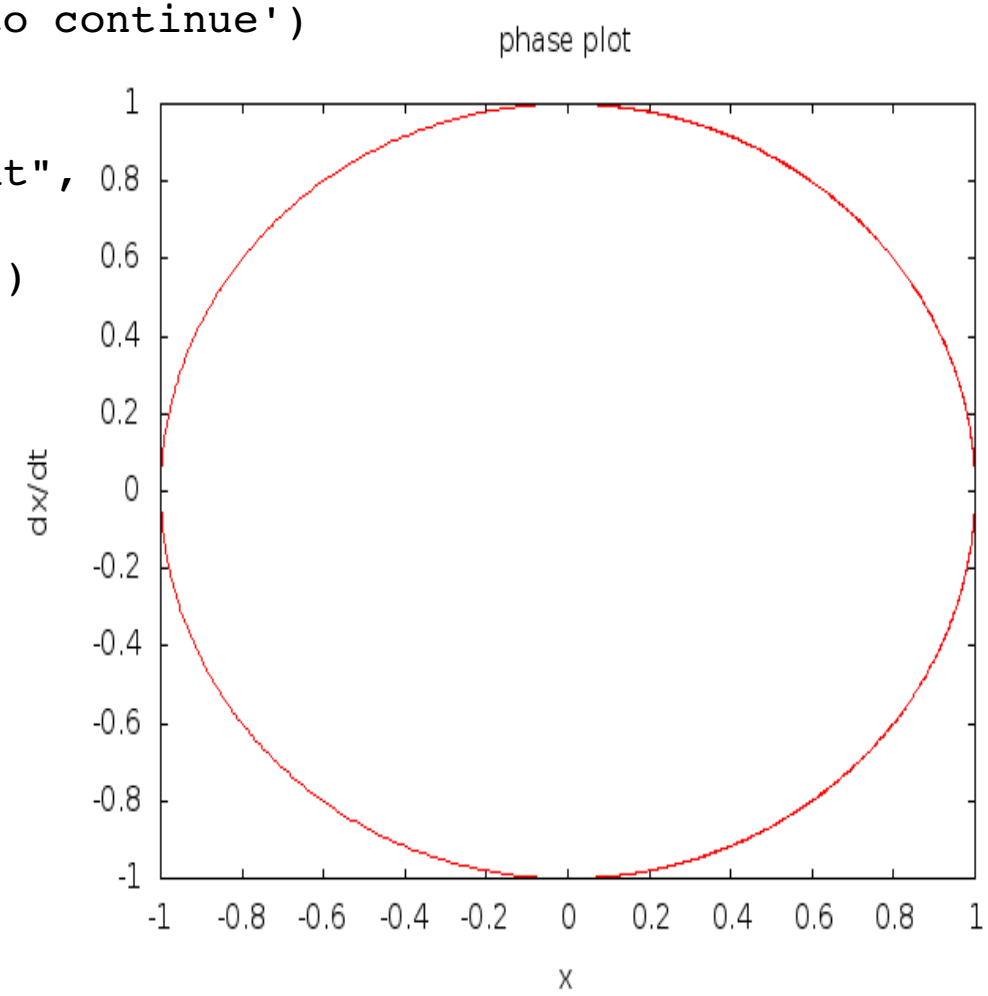
Harmonic Oscillator

```
# solve the equations
u = odeint(rhs, u0, t)

from scitools.std import plot
plot(t, u[:,0], xlabel="t", ylabel="x", title="time plot",
hardcopy="timePlot.png")
```



Harmonic Oscillator

```
raw_input('press ENTER to continue')

plot(u[:,0], u[:,1],
xlabel="x", ylabel="dx/dt",
title="phase plot",
hardcopy="phasePlot.png")
```



In Matlab ...

http://www.mathworks.com/help/matlab/ref/ode23.html

General-purpose solver: ode45