# ıtific software engineering for a simple C model

**Hans Petter Langtangen**[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Jul 14, 2014

## nts



**Exercises**

## Exercises and Problems





**Goal.**

This document illustrates *best practice* for developing scientific software in ar reliable way. Not only will the outlined techniques save a lost of human time, also help assure reproducible science and higher quality of computational i Key questions to be answered are

- How should I organize a program?
- How can I efficiently and safely provide input data and run my code?
- How can I verify that the implementation is correct?
- How should I reliably work with files and documents?
- How should I conduct large numerical experiments?

**hpl 1**: *Need to cover functions, classes, modules, cml, GUI, hand calc, vact num sol, MMS, qualitative results, Git, bitbucket/github, scripting, report*

# Sample problem and code

his first introduction to good programming habits in scientific computing will m mple mathematical problem to keep the mathematical details at the lowest pos troducing a series of computer science concepts. The simplicity of the mathe bviously prevents us from treating several techniques that are only meaning cientific software.

## .1 Mathematical problem

Ve consider a linear ordinary differential equation with variable coefficients:

$$u'(t) = -a(t)u(t) + b(t), \quad u(0) = U_0, \quad t \in (0, T]\,.$$

This problem is numerically solved by the so-called $\theta$-rule, which is a co nerge different formulas for the well-known Forward Euler, Backward Euler, and midpoint/central) schemes. We introduce a uniform time mesh $t_n = n\Delta t$, $n =$ eek $u(t)$ at the mesh points. The numerical approximation to $u(t_n)$ is denoted $u$ se the symbol $u$ both for the exact analytical solution of (1) and for the numerica e sometimes introduce $u_{\mathrm{e}}(t)$ to help distinguish the two types of solutions (i.e. exact")[1].

The $\theta$-rule leads to an explicit updating formula for $u^{n+1}$, given $u^n$:

**hpl 2**: *not ready. Need more here!!!!!!!*

[1] In the literature, it is more common to put a subscript (like $u_\Delta$ or $u_h$) on the numerical sol from the exact solution. However, we will use the variable u in the code for the numerical a omputed, and therefore adjust the mathematical notation to convenient conventions in the coc ave as close correspondence as possible between the implementation and the mathematics.

### at program implementation

### nction implementation

```
ule import *
```

### refixing functions by module names

```
odule as m
```

## er interfaces

programming practice to let programs read input from the user rather than req
it the source code when trying out new values of input parameters. One reaso
f the code has a danger of introducing bugs. Another reason is that it is easier
ork to supply data to a program instead of editing the program code. A third r
ogram that reads input can easily be run by another program, and in this way
a large number of runs in scientific investigations.

---

all make it a habit to equip any implementation of a numerical solver wit
riate user interface before testing out the code.

---

ıg input data can be done in many ways. We have to decide on desired *user i*
ve want to operate the program when providing input, and then use appropria
ıent the user interface. There are four basic types of user interface of relevanc
, listed here with increasing complexity of the implementation:

stions and answers in the terminal window

ımand-line arguments

ding data from file

phical user interfaces

conceptually simple, alternative 1 involves more typing than the other alternat
e abandoned. Below, we shall address alternative 2 and 4, which are most app
esent problem.

### reating command-line interfaces

nput from the command line is a simple and flexible way of interacting with t
ores all the command-line arguments in the list `sys.argv`, and there are, in p
of programming with command-line arguments in Python:

ide upon a sequence of parameters on the command line and read their values
ı the `sys.argv[1:]` list (`sys.argv[0]` is the just program name).



- Use option-value pairs (`-option value`) on the command line to overrid
  of input parameters, and utilize the `argparse.ArgumentParser` tool to i
  command line.

oth strategies will be illustrated next.

**eading a sequence of command-line arguments.** The `decay_plot_mp`
eeds the following input data: $I$, $a$, $T$, an option to turn the plot on or off (`m`
st of $\Delta t$ values.

The simplest way of reading this input from the command line is to say th
ommand-line arguments correspond to the first four points in the list above, in
ıat the rest of the command-line arguments are the $\Delta t$ values. The input giv
ın be a string among `'on'`, `'off'`, `'True'`, and `'False'`. The code for read
ıost conveniently put in a function:

```
import sys

def read_command_line():
    if len(sys.argv) < 6:
        print 'Usage: %s I a T on/off dt1 dt2 dt3 ...' % \
              sys.argv[0]; sys.exit(1)  # abort

    I = float(sys.argv[1])
    a = float(sys.argv[2])
    T = float(sys.argv[3])
    makeplot = sys.argv[4] in ('on', 'True')
    dt_values = [float(arg) for arg in sys.argv[5:]]

    return I, a, T, makeplot, dt_values
```

One should note the following about the constructions in the program above

- Everything on the command line ends up in a *string* in the list `sys.argv`. Ex
  to, e.g., a `float` object is required if the string as a number we want to c
- The value of `makeplot` is determined from a boolean expression, which
  the command-line argument is either `'on'` or `'True'`, and `False` otherwis
- It is easy to build the list of $\Delta t$ values: we simply run through the
  `sys.argv[5:]`, convert each command-line argument to `float`, and coll
  objects in a list, using the compact and convenient *list comprehension* syn

he loops over $\theta$ and $\Delta t$ values can be coded in a `main` function:

```
def main():
    I, a, T, makeplot, dt_values = read_command_line()
    for theta in 0, 0.5, 1:
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot)
            print '%3.1f %6.2f: %12.3E' % (theta, dt, E)
```

he complete program can be found in `decay_cml.py`[3].

[2] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`
[3] `http://tinyurl.com/jvzzcfn/decay/decay_cml.py`

**with an argument parser.** Python's `ArgumentParser` tool in the `argparse`
asy to create a professional command-line interface to any program. The docum
`ntParser`[4] demonstrates its versatile applications, so we shall here just list an
g basic features. On the command line we want to specify option-value pairs
., `-a 3.5 -I 2 -T 2`. Including `-makeplot` turns the plot on and excluding thi
plot off. The $\Delta t$ values can be given as `-dt 1 0.5 0.25 0.1 0.01`. Each pa
a sensible default value so that we specify the option on the command line on
lt value is not suitable.
roduce a function for defining the mentioned command-line options:

```
ne_command_line_options():
rt argparse
er = argparse.ArgumentParser()
er.add_argument('--I', '--initial_condition', type=float,
                default=1.0, help='initial condition, u(0)',
                metavar='I')
er.add_argument('--a', type=float,
                default=1.0, help='coefficient in ODE',
                metavar='a')
er.add_argument('--T', '--stop_time', type=float,
                default=1.0, help='end time of simulation',
                metavar='T')
er.add_argument('--makeplot', action='store_true',
                help='display plot or not')
er.add_argument('--dt', '--time_step_values', type=float,
                default=[1.0], help='time step values',
                metavar='dt', nargs='+', dest='dt_values')
rn parser
```

command-line option is defined through the `parser.add_argument` method. Alt
ke the short `-I` and the more explaining version `--initial_condition` can be
uments are `type` for the Python object type, a default value, and a help string
ed if the command-line argument `-h` or `-help` is included. The `metavar` a
he value associated with the option when the help string is printed. For exam
$I$ has this help output:

```
 python decay_argparse.py -h

--initial_condition I
                initial condition, u(0)
```

ture of this output is

```
 metavar, --initial_condition metavar
                    help-string
```

`makeplot` option is a pure flag without any value, implying a true value if th
d otherwise a false value. The `action='store_true'` makes an option for suc
y, the `-dt` option demonstrates how to allow for more than one value (separ
rough the `nargs='+'` keyword argument. After the command line is parsed, w
ere the values of the options are stored as attributes. The attribute name is spe

/docs.python.org/library/argparse.html



e `dist` keyword argument, which for the `-dt` option is `dt_values`. Without the
e value of an option `-opt` is stored as the attribute `opt`.

The code below demonstrates how to read the command line and extract the
ption:

```
ef read_command_line():
    parser = define_command_line_options()
    args = parser.parse_args()
    print 'I={}, a={}, T={}, makeplot={}, dt_values={}'.format(
        args.I, args.a, args.T, args.makeplot, args.dt_values)
    return args.I, args.a, args.T, args.makeplot, args.dt_values
```

The `main` function remains the same as in the `decay_cml.py` code based
`ys.argv` directly. A complete program featuring the demo above of `Argument`
the file `decay_argparse.py`[5].

## .2 Creating a graphical web user interface

he Python package Parampool[6] can be used to automatically generate a web-bas
*terface* (GUI) for our simulation program. Although the programming technic
mplifies the efforts to create a GUI, the forthcoming material on equipping
odule with a GUI is quite technical and of significantly less importance than
ake a command-line interface (Section 2.1). There is no danger in jumping rig

**Making a compute function.** The first step is to identify a function tha
omputations and that takes the necessary input variables as arguments. Th
*ompute function* in Parampool terminology. We may start with a copy of the
`lot_mpl.py`[7], which has a `main` function displayed in Section **??** for carrying
nd plotting for a series of $\Delta t$ values. Now we want to control and view the sa
om a web GUI.

To tell Parampool what type of input data we have, we assign default values
all arguments in the main function and call it `main_GUI`:

```
ef main_GUI(I=1.0, a=.2, T=4.0,
            dt_values=[1.25, 0.75, 0.5, 0.1],
            theta_values=[0, 0.5, 1]):
```

The compute function must return the HTML code we want for displaying th
age. Here we want to show plots of the numerical and exact solution for differe
$t$ values. The plots can be organized in a table with $\theta$ (methods) varying thro
nd $\Delta t$ varying through the rows. Assume now that a new version of the e
ot only returns the error `E` but also HTML code containing the plot. Then
`ain_GUI` function as

```
ef main_GUI(I=1.0, a=.2, T=4.0,
            dt_values=[1.25, 0.75, 0.5, 0.1],
            theta_values=[0, 0.5, 1]):
    # Build HTML code for web page. Arrange plots in columns
    # corresponding to the theta values, with dt down the rows
```

[5] http://tinyurl.com/jvzzcfn/decay/decay_argparse.py
[6] https://github.com/hplgit/parampool
[7] http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py

```
a2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
_text = '<table>\n'
dt in dt_values:
html_text += '<tr>\n'
for theta in theta_values:
    E, html = explore(I, a, T, dt, theta, makeplot=True)
    html_text += """

<b>%s, dt=%g, error: %s</b></center><br>


heta2name[theta], dt, E, html)
html_text += '</tr>\n'
_text += '</table>\n'
rn html_text
```

r than creating plot files and showing the plot on the screen, the new versio
unction makes a string with the PNG code of the plot and embeds that string i
s action is conveniently performed by Parampool's `save_png_to_str` function

```
atplotlib.pyplot as plt


(t, u, r-')
el('t')
el('u')

ampool.utils import save_png_to_str
t = save_png_to_str(plt, plotwidth=400)
```

we now write `plt.plot`, `plt.xlabel`, etc. The `html_text` string is long and
aracters that build up the PNG file of the current plot. The new `explore` func
of the above code snippet and return `html_text` along with `E`.

**ing the user interface.** The web GUI is automatically generated by the fo
ed in a file `decay_GUI_generate.py`[8]

```
rampool.generator.flask import generate
cay_GUI import main
e(main,
  output_controller='decay_GUI_controller.py',
  output_template='decay_GUI_view.py',
  output_model='decay_GUI_model.py')
```

he `decay_GUI_generate.py` program results in three new files whose names are s
l to `generate`:

`ay_GUI_model.py` defines HTML widgets to be used to set input data in t
rface,

`plates/decay_GUI_views.py` defines the layout of the web page,

`ay_GUI_controller.py` runs the web application.

need to run the last program, and there is no need to look into these files.

[8] `/tinyurl.com/jvzzcfn/decay/decay_GUI_generate.py`



**unning the web application.** The web GUI is started by

```
erminal> python decay_GUI_controller.py
```

pen a web browser at the location `127.0.0.1:5000`. Input fields for `I`, `a`, `T`, `heta_values` are presented. Setting the latter two to `[1.25, 0.5]` and `[1, 0` nd pressing *Compute* results in four plots, see Figure 1. With the technique ere, one can easily create a tailored web GUI for a particular type of applicati nteractively explore physical and numerical effects.

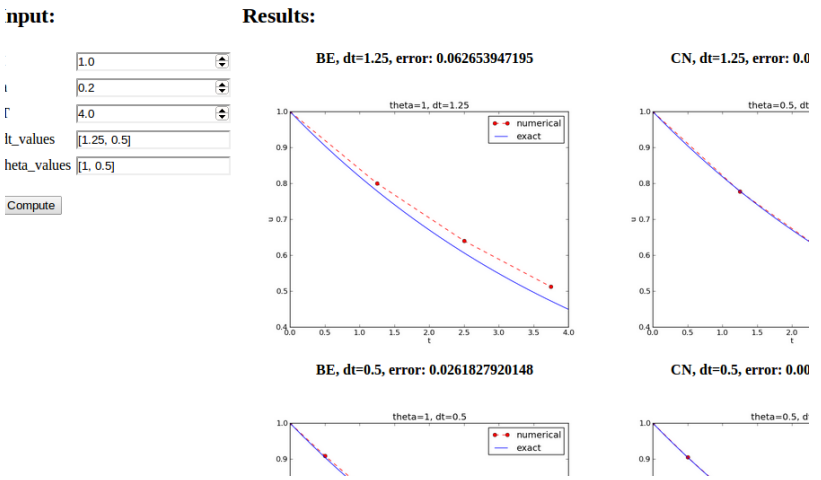



Figure 1: Automatically generated graphical web interface.

# Verification

## .1 Comparison with hand calculations

ne of the simplest and most powerful methods for verifying numerical codes is eps of the algorithm by hand and compare the results with those produced by resent case, we may choose some test problem and run three steps by hand. Pi **pl 3**: *Not ready. Time-dep a?*

## .2 Test function

**Caution: choice of parameter values.**

For the choice of values of parameters in verification tests one should stay away especially 0 and 1, as these can simplify formulas too much for test purposes.

$= 1$ the nominator in the formula for $u^n$ will be the same for all $a$ and $\Delta t$ v
ould therefore choose more "arbitrary" values, say $\theta = 0.8$ and $I = 0.1$.

### omparison with an exact discrete solution

s it is possible to find a closed-form *exact discrete solution* that fulfills the discre
equations. The implementation can then be verified against the exact discrete
ually the best technique for verification.

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,.$$

omputations with the $\theta$-rule results in

$$\begin{aligned}
u^0 &= I,\\
u^1 &= Au^0 = AI,\\
u^2 &= Au^1 = A^2 I,\\
&\vdots\\
u^n &= A^n u^{n-1} = A^n I\,.
\end{aligned}$$

hen established the exact discrete solution as

$$u^n = IA^n\,.$$

**on.**

ould be conscious about the different meanings of the notation on the left-
and side of (2): on the left, $n$ in $u^n$ is a superscript reflecting a counter of
$(t_n)$, while on the right, $n$ is the power in the exponentiation $A^n$.

arison of the exact discrete solution and the computed solution is done in the fo

```
fy_exact_discrete_solution():

exact_discrete_solution(n, I, a, theta, dt):
A = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
return I*A**n

a = 0.8; a = 2; I = 0.1; dt = 0.8
 int(8/dt)  # no of steps
 = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
 = array([exact_discrete_solution(n, I, a, theta, dt)
          for n in range(Nt+1)])
erence = abs(u_de - u).max()  # max deviation
= 1E-15  # tolerance for comparing floats
ess = difference <= tol
rn success
```



he complete program is found in the file `decay_verf2.py`[9] (`verf2` is a short name
ersion 2").

**Local functions.**

One can define a function inside another function, here called a *local function*
as *closure*) inside a *parent function*. A local function is invisible outside the pa
A convenient property is that any local function has access to all variables
parent function, also if we send the local function to some other function
(!). In the present example, it means that the local function `exact_discre`
does not need its five arguments as the values can alternatively be accessed
local variables defined in the parent function `verify_exact_discrete_solu`
send such an `exact_discrete_solution` without arguments to any other
`exact_discrete_solution` will still have access to `n`, `I`, `a`, and so forth define
function.

## .4 Computing convergence rates

/e expect that the error $E$ in the numerical solution is reduced if the mesh size
lore specifically, many numerical methods obey a power-law relation between

$$E = C\Delta t^r,$$

here $C$ and $r$ are (usually unknown) constants independent of $\Delta t$. The formu
s an asymptotic model valid for sufficiently small $\Delta t$. How small is normally l
ithout doing numerical estimations of $r$.

The parameter $r$ is known as the *convergence rate*. For example, if the conve
alving $\Delta t$ reduces the error by a factor of 4. Diminishing $\Delta t$ then has a greate
rror compared with methods that have $r = 1$. For a given value of $r$, we refer t
f $r$-th order. First- and second-order methods are most common in scientific c

**stimating $r$.** There are two alternative ways of estimating $C$ and $r$ base
mulations with corresponding pairs $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$, and $\Delta t_i < \Delta t_{i-1}$
ell size).

1. Take the logarithm of (3), $\ln E = r \ln \Delta t + \ln C$, and fit a straight line to
   $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$.
2. Consider two consecutive experiments, $(\Delta t_i, E_i)$ and $(\Delta t_{i-1}, E_{i-1})$. Divid
   $E_{i-1} = C\Delta t_{i-1}^r$ by $E_i = C\Delta t_i^r$ and solving for $r$ yields

$$r_{i-1} = \frac{\ln(E_{i-1}/E_i)}{\ln(\Delta t_{i-1}/\Delta t_i)}$$

or $i = 1, \ldots, m-1$.

The disadvantage of method 1 is that (3) might not be valid for the coarsest m
alues). Fitting a line to all the data points is then misleading. Method 2 compu
ates for pairs of experiments and allows us to see if the sequence $r_i$ converges

[9] `http://tinyurl.com/jvzzcfn/decay/decay_verf2.py`

2. The final $r_{m-2}$ can then be taken as the convergence rate. If the coarsest
fering rate, the corresponding time steps are probably too large for (3) to be vali
ime steps lie outside the asymptotic range of $\Delta t$ values where the error behaves

**ntation.** It is straightforward to extend the `main` function in the program `decay`
ments for computing $r_0, r_1, \ldots, r_{m-2}$ from (3):

```
h import log

():
, T, makeplot, dt_values = read_command_line()
{}  # estimated convergence rates
theta in 0, 0.5, 1:
E_values = []
for dt in dt_values:
    E = explore(I, a, T, dt, theta, makeplot=False)
    E_values.append(E)

# Compute convergence rates
m = len(dt_values)
r[theta] = [log(E_values[i-1]/E_values[i])/
            log(dt_values[i-1]/dt_values[i])
            for i in range(1, m, 1)]

theta in r:
print '\nPairwise convergence rates for theta=%g:' % theta
print ' '.join(['%.2f' % r_ for r_ in r[theta]])
rn r
```

am containing this `main` function is called `decay_convrate.py`[10].
object is a *dictionary of lists*. The keys in this dictionary are the $\theta$ values. For e
s the list of the $r_i$ values corresponding to $\theta = 1$. In the loop `for theta in r`,
`heta` takes on the values of the keys in the dictionary `r` (in an undetermined o
simply do a `print r[theta]` inside the loop, but this would typically yield ou
rgence rates with 16 decimals:

```
1919482274763, 1.1488178494691532, ...]
```

d, we format each number with 2 decimals, using a list comprehension to t
mbers, `r[theta]`, into a list of formatted strings. Then we join these strings
between to get a sequence of rates on one line in the terminal window. More g
`ist)` joins the strings in the list `list` to one string, with `d` as delimiter between `l`
etc.
s an example on the outcome of the convergence rate computations:

```
 python decay_convrate.py --dt 0.5 0.25 0.1 0.05 0.025 0.01

convergence rates for theta=0:
 1.07 1.03 1.02

convergence rates for theta=0.5:
 2.03 2.01 2.01

convergence rates for theta=1:
 0.99 1.00 1.00
```

/tinyurl.com/jvzzcfn/decay/decay_convrate.py



he Forward and Backward Euler methods seem to have an $r$ value which stab
ne Crank-Nicolson seems to be a second-order method with $r = 2$.

Very often, we have some theory that predicts what $r$ is for a numerical n
heoretical error measures for the $\theta$-rule point to $r = 2$ for $\theta = 0.5$ and $r = 1$
omputed estimates of $r$ are in very good agreement with these theoretical valu

**Why convergence rates are important.**

The strong practical application of computing convergence rates is for verific
convergence rates point to errors in the code, and correct convergence rates bu
that the implementation is correct. Experience shows that bugs in the code
the expected convergence rate.

**Debugging via convergence rates.** Let us experiment with bugs and see th
ne convergence rate. We may, for instance, forget to multiply by `a` in the den
pdating formula for `u[n+1]`:

```
u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt)*u[n]
```

unning the same `decay_convrate.py` command as above gives the expected c
). Why? The reason is that we just specified the $\Delta t$ values are relied on default
arameters. The default value of $a$ is 1. Forgetting the factor `a` has then no effec
nows how important it is to avoid parameters that are 1 or 0 when verifying i
unning the code `decay_v0.py` with $a = 2.1$ and $I = 0.1$ yields

```
erminal> python decay_convrate.py --a 2.1 --I 0.1  \
         --dt 0.5 0.25 0.1 0.05 0.025 0.01
..
airwise convergence rates for theta=0:
.49 1.18 1.07 1.04 1.02

airwise convergence rates for theta=0.5:
1.42 -0.22 -0.07 -0.03 -0.01

airwise convergence rates for theta=1:
.21 0.12 0.06 0.03 0.01
```

his time we see that the expected convergence rates for the Crank-Nicolson and
ethods are not obtained, while $r = 1$ for the Forward Euler method. The re
ate in the latter case is that $\theta = 0$ and the wrong `theta*dt` term in the denon
nyway.

The error

```
u[n+1] = ((1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
```

anifests itself through wrong rates $r \approx 0$ for all three methods. About the sa
om an erroneous initial condition, `u[0] = 1`, or wrong loop limits, `range(1,Nt`
n this simple problem, most bugs we can think of are detected by the conve
rovided the values of the input data do not hide the bug.

`ify_convergence_rate` function could compute the dictionary of list via `ma`
ıe final rate estimates ($r_{m-2}$) are sufficiently close to the expected ones. A tole
appropriate, given the uncertainty in estimating $r$:

```
fy_convergence_rate():
main()
= 0.1
cted_rates = {0: 1, 1: 1, 0.5: 2}
theta in r:
r_final = r[theta][-1]
diff = abs(expected_rates[theta] - r_final)
if diff > tol:
    return False
rn True  # all tests passed
```

k that `r[theta]` is a list and the last element in any list can be extracted by tl

## ftware engineering

ıt use of differential equation models requires software that is easy to test and fle
:ing up extensive numerical experiments. This section introduces three impo
ts:

Modules

Testing frameworks

mplementation with classes

ncepts are introduced using the differential equation problem $u' = -au$, $u(0) =$
le.

### aking a module

**)RY principle.**

evious sections have outlined numerous different programs, all of them having
)py of the `solver` function. Such copies of the same piece of code is agains
ant *Don't Repeat Yourself* (DRY) principle in programming. If we want to ch
lver function there should be *one and only one* place where the change needs
ned.

an up the repetitive code snippets scattered among the `decay_*.py` files, we
the various functions we want to keep for the future in one file, now called `dec`
stands for "module"). The following functions are copied to this file:

/tinyurl.com/jvzzcfn/decay/decay_mod.py



- `solver` for computing the numerical solution
- `verify_three_steps` for verifying the first three solution points against h
- `verify_discrete_solution` for verifying the entire computed solution a formula for the numerical solution
- `explore` for computing and plotting the solution
- `define_command_line_options` for defining option-value pairs on the co
- `read_command_line` for reading input from the command line, now extend with `sys.argv` directly and with an `ArgumentParser` object
- `main` for running experiments with $\theta = 0, 0.5, 1$ and a series of $\Delta t$ values, convergence rates
- `main_GUI` for doing the same as the `main` function, but modified for generation
- `verify_convergence_rate` for verifying the computed convergence rates a retically expected values

Ve use Matplotlib for plotting. A sketch of the `decay_mod.py` file, with complet
ıodified functions, looks as follows:

```
from numpy import *
from matplotlib.pyplot import *
import sys

def solver(I, a, T, dt, theta):
    ...

def verify_three_steps():
    ...

def verify_exact_discrete_solution():
    ...

def u_exact(t, I, a):
    ...

def explore(I, a, T, dt, theta=0.5, makeplot=True):
    ...

def define_command_line_options():
    ...

def read_command_line(use_argparse=True):
    if use_argparse:
        parser = define_command_line_options()
        args = parser.parse_args()
        print 'I={}, a={}, makeplot={}, dt_values={}'.format(
            args.I, args.a, args.makeplot, args.dt_values)
        return args.I, args.a, args.makeplot, args.dt_values
    else:
        if len(sys.argv) < 6:
            print 'Usage: %s I a on/off dt1 dt2 dt3 ...' % \
                  sys.argv[0]; sys.exit(1)

        I = float(sys.argv[1])
```

```
a = float(sys.argv[2])
T = float(sys.argv[3])
makeplot = sys.argv[4] in ('on', 'True')
dt_values = [float(arg) for arg in sys.argv[5:]]

return I, a, makeplot, dt_values

():
```

decay_mod.py file is already a module such that we can import desired func
grams. For example, we can in a file do

```
ay_mod import solver
olver(I=1.0, a=3.0, T=3, dt=0.01, theta=0.5)
```

er, it should also be possible to both use decay_mod.py as a module *and* exe
rogram that runs main(). This is accomplished by ending the file with a *test*

```
e__ == '__main__':
()
```

ay_mod.py is used as a module, __name__ equals the module name decay_mo
equals '__main__' when the file is run as a program. Optionally, we could run
sts if the word verify is present on the command line and verify_convergen
tested if verify_rates is found on the command line. The verify_rate
t be removed before we read parameter values from the command line, other
mand_line function (called by main) will not work properly.

```
e__ == '__main__':
verify' in sys.argv:
if verify_three_steps() and verify_discrete_solution():
    pass # ok
else:
    print 'Bug in the implementation!'
 'verify_rates' in sys.argv:
sys.argv.remove('verify_rates')
if not '--dt' in sys.argv:
    print 'Must assign several dt values'
    sys.exit(1)  # abort
if verify_convergence_rate():
    pass
else:
    print 'Bug in the implementation!'
:
# Perform simulations
main()
```

### refixing imported functions by the module name

atements of the form from module import * import functions and variables in m
urrent file. For example, when doing

```
py import *
plotlib.pyplot import *
```



e get mathematical functions like sin and exp as well as MATLAB-style functio
nd plot, which can be called by these well-known names. Unfortunately, i
omes confusing to know where a particular function comes from. Is it fr
atplotlib.pyplot? Or is it our own function?

An alternative import is

```
import numpy
import matplotlib.pyplot
```

nd such imports require functions to be prefixed by the module name, e.g.,

```
 = numpy.linspace(0, T, Nt+1)
u_e = I*numpy.exp(-a*t)
matplotlib.pyplot.plot(t, u_e)
```

his is normally regarded as a better habit because it is explicitly stated from
unction comes from.

The modules numpy and matplotlib.pyplot are so frequently used, and their
edious to write, so two standard abbreviations have evolved in the Python scie
ommunity:

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

 = np.linspace(0, T, Nt+1)
u_e = I*np.exp(-a*t)
lt.plot(t, u_e)
```

version of the decay_mod module where we use the np and plt prefixes is
ecay_mod_prefix.py[12].

The downside of prefixing functions by the module name is that mathemat
ke $e^{-at}\sin(2\pi t)$ get cluttered with module names,

```
umpy.exp(-a*t)*numpy.sin(2(numpy.pi*t)
# or
np.exp(-a*t)*np.sin(2*np.pi*t)
```

uch an expression looks like exp(-a*t)*sin(2*pi*t) in most other program
imilarly, np.linspace and plt.plot look less familiar to people who are used t
ho have not adopted Python's prefix style. Whether to do from module impo
odule depends on personal taste and the problem at hand. In these writin
odule import in shorter programs where similarity with MATLAB could be ar
here a one-to-one correspondence between mathematical formulas and Pytho
nportant. The style import module is preferred inside Python modules (see
emonstration).

## .3 Doctests

Ve have emphasized how important it is to be able to run tests in the program a
as solved by calling various verify* functions in the previous examples. Howe
ell-established procedures and corresponding tools for automating the execut

[12] http://tinyurl.com/jvzzcfn/decay/decay_mod_prefix.py

ly demonstrate two important techniques: *doctest* and *unit testing*. The corres
he modules `decay_mod_doctest.py`[13] and `decay_mod_nosetest.py`[14].
string (the first string after the function header) is used to document the pu
and their arguments. Very often it is instructive to include an example on hov
ion. Interactive examples in the Python shell are most illustrative as we can
sulting from function calls. For example, we can in the `solver` function inc
n calling this function and printing the computed `u` and `t` arrays:

```
er(I, a, T, dt, theta):

e u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.


u, t = solver(I=0.8, a=1.2, T=4, dt=0.5, theta=0.5)
for t_n, u_n in zip(t, u):
    print 't=%.1f, u=%.14f' % (t_n, u_n)
0, u=0.80000000000000
5, u=0.43076923076923
0, u=0.23195266272189
5, u=0.12489758761948
0, u=0.06725254717972
5, u=0.03621291001985
0, u=0.01949925924146
5, u=0.01049960113002
0, u=0.00565363137770
```

such interactive demonstrations are inserted in doc strings, Python's `doctest`[15]
ed to automate running all commands in interactive sessions and compare new
output appearing in the doc string. All we have to do in the current example is

```
> python -m doctest decay_mod_doctest.py
```

mand imports the `doctest` module, which runs all tests. No additional comma
is allowed when running doctests. If any test fails, the problem is reported, e.

```
 python -m doctest decay_mod_doctest.py
**********************************************
ay_mod_doctest.py", line 12, in decay_mod_doctest....
ample:
_n, u_n in zip(t, u):
rint 't=%.1f, u=%.14f' % (t_n, u_n)

, u=0.80000000000000
, u=0.43076923076923
, u=0.23195266272189
, u=0.12489758761948
, u=0.06725254717972

, u=0.80000000000000
, u=0.43076923076923
, u=0.23195266272189
, u=0.12489758761948
, u=0.06725254718756
```

/tinyurl.com/jvzzcfn/decay/decay_mod_doctest.py
/tinyurl.com/jvzzcfn/decay/decay_mod_nosetest.py
/docs.python.org/library/doctest.html



```
**********************************************************
 items had failures:
  1 of   2 in decay_mod_doctest.solver
**Test Failed*** 1 failures.
```

Note that in the output of `t` and `u` we write `u` with 14 digits. Writing all 1
ood idea: if the tests are run on different hardware, round-off errors might be d
`octest` module detects that the numbers are not precisely the same and rep
ne present application, where $0 < u(t) \leq 0.8$, we expect round-off errors to be
omparing 15 digits would probably be reliable, but we compare 14 to be on th
Doctests are highly encouraged as they do two things: 1) demonstrate how a
nd 2) test that the function works.
Here is an example on a doctest in the `explore` function:

```
ef explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).

    >>> for theta in 0, 0.5, 1:
    ...     E = explore(I=1.9, a=2.1, T=5, dt=0.1, theta=theta,
    ...                 makeplot=False)
    ...     print '%.10E' % E
    ...
    7.3565079236E-02
    2.4183893110E-03
    6.5013039886E-02
    """
    ...
```

his time we limit the output to 10 digits.

**Caution.**

Doctests requires careful coding if they use command-line input or print
terminal window. Command-line input must be simulated by filling `sys.argv`
`sys.argv = '-I 1.0 -a 5'.split`. The output lines of print statements m
exactly as they appear when running the statements in an interactive Pytho

## .4 Unit testing with nose

he unit testing technique consists of identifying small units of code, usually funct
nd write one or more tests for each unit. One test should, ideally, not depend o
ther tests. For example, the doctest in function `solver` is a unit test, and the do
`xplore` as well, but the latter depends on a working `solver`. Putting the error c
lotting in `explore` in two separate functions would allow independent unit te
ne design of unit tests impacts the design of functions. The recommended pract
esign and write the unit tests first and *then* implement the functions!
In scientific computing it is not always obvious how to best perform un
nits is naturally larger than in non-scientific software. Very often the solutio
athematical problem identifies a unit.

**e of nose.** The `nose` package is a versatile tool for implementing unit tests in
short explanation of the usage of nose:

lement tests in functions with names starting with `test_`. Such functions canr
arguments.

test functions perform assertions on computed results using `assert` functions f
`e.tools` module.

test functions can be in the source code files or be collected in separate files wit
`t*.py`.

es a very simple illustration of the three points. Assume that we have this funct
`ymod`:

```
le(n):
rn 2*n
```

in this file, or in a separate file `test_mymod.py`, we implement a test functio
s to test that the function `double` works as intended:

```
ose.tools as nt

_double():
lt = double(4)
ssert_equal(result, 8)
```

at `test_double` has no arguments. We need to do an `import mymod` or `fror`
`ouble` if this test resides in a separate file. Running

```
nosetests -s mymod
```

`nose` tool run all functions with names matching `test_*()` in `mymod.py`. Alter
functions are in some `test_mymod.py` file, we can just write `nosetests -s`. T
hen look for all files with names mathching `test*.py` and run all functions t
les.
you have nose tests in separate test files with names `test*.py` it is common t
in a subdirectory `tests`, or `*_tests` if you have several test subdirectories. I
`s -s` will then recursively look for all `tests` and `*_tests` subdirectories and
`test_*()` in all files `test_*.py` in these directories. Just one command can ther
tests in a directory tree!
ample of a `tests` directory with different types of `test*.py` files are found in
[6]. Note that these perform imports of modules in the parent directory. These
ll because the tests are supposed to be run by `nosetests -s` executed in the
(`decay`).

/tinyurl.com/jvzzcfn/decay/tests



**Tip.**

The `-s` option to `nosetests` assures that any print statement in the `test`
appears in the output. Without this option, `nosetests` suppressed whate
writes to the terminal window (standard output). Such behavior is annoyi
when developing and testing tests.

The number of failed tests and their details are reported, or an `OK` is printed i
The advantage with the `nose` package is two-fold:

1. tests are written and collected in a structured way, and
2. large collections of tests, scattered throughout a tree of directories, can b
   one command `nosetests -s`.

**lternative assert statements.** In case the `nt.assert_equal` function fin
rguments are equal, the test is a success, otherwise it is a failure and an ex
`ssertionError` is raised. The particular exception is the indicator that a test
Instead of calling the convenience function `nt.assert_equal`, we can use
`ssert` statement, which tests if a boolean expression is true and raises an *A*
therwise. Here, the statement is `assert result == 8`.
A completely manual alternative is to explicitly raise an `AssertionError`
`omputed result is wrong:

```
if result != 8:
    raise AssertionError()
```

**pplying nose.** Let us illustrate how to use the `nose` tool for testing key
`ecay_mod` module. Or more precisely, the module is called `decay_mod_unitt`
`erify*` functions removed as these now are outdated by the unit tests.
We design three unit tests:

1. A comparison between the computed $u^n$ values and the exact discrete sol
2. A comparison between the computed $u^n$ values and precomputed, verified
3. A comparison between observed and expected convergence rates.

hese tests follow very closely the code in the previously shown `verify*` func
ith comparing $u^n$, as computed by the function `solver`, to the formula for th
olution:

```
import nose.tools as nt
import decay_mod_unittest as decay_mod
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    """Return exact discrete solution of the theta scheme."""
    dt = float(dt)  # avoid integer division
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*factor**n
```

```
_exact_discrete_solution():

are result from solver against
ula for the discrete solution.

a = 0.8; a = 2; I = 0.1; dt = 0.8
int(8/dt)  # no of steps
 = decay_mod.solver(I=I, a=a, T=N*dt, dt=dt, theta=theta)
 = np.array([exact_discrete_solution(n, I, a, theta, dt)
             for n in range(N+1)])
 = np.abs(u_de - u).max()
ssert_almost_equal(diff, 0, delta=1E-14)
```

t.assert_almost_equal is the relevant function for comparing two real numbe
ument specifies a tolerance for the comparison. Alternatively, one can specify a
for the number of decimal places to be used in the comparison.
having carefully verified the implementation, we may store correctly computed r
t program or in files for use in future tests. Here is an example on how the c
solver function can be compared to what is considered to be correct results:

```
_solver():

are result from solver against
omputed arrays for theta=0, 0.5, 1.

8; a=1.2; T=4; dt=0.5  # fixed parameters
omputed = {
't': np.array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,
                3. ,  3.5,  4. ]),
0.5: np.array(
    [ 0.8       ,  0.43076923,  0.23195266, 0.12489759,
      0.06725255,  0.03621291,  0.01949926, 0.0104996 ,
      0.00565363]),
0: np.array(
    [  8.00000000e-01,   3.20000000e-01,
       1.28000000e-01,   5.12000000e-02,
       2.04800000e-02,   8.19200000e-03,
       3.27680000e-03,   1.31072000e-03,
       5.24288000e-04]),
1: np.array(
    [ 0.8       ,  0.5       ,  0.3125    ,  0.1953125 ,
      0.12207031,  0.07629395,  0.04768372,  0.02980232,
      0.01862645]),
}
theta in 0, 0.5, 1:
u, t = decay_mod.solver(I, a, T, dt, theta=theta)
diff = np.abs(u - precomputed[theta]).max()
# Precomputed numbers are known to 8 decimal places
nt.assert_almost_equal(diff, 0, places=8,
                       msg='theta=%s' % theta)
```

omputed object is a dictionary with four keys: 't' for the time mesh, and three
utions corresponding to $\theta = 0, 0.5, 1$.
g for special type of input data that may cause trouble constitutes a commor
ng unit tests. For example, the updating formula for $u^{n+1}$ may be incorrectly e
f unintended integer divisions. With

```
1; a = 1; I = 1; dt = 2
```

ıator and denominator in the updating expression,



```
(1 - (1-theta)*a*dt)
(1 + theta*dt*a)
```

valuate to 1 and 3, respectively, and the fraction 1/3 will call up integer division a
ad to u[n+1]=0. We construct a unit test to make sure solver is smart enou
roblem:

```
def test_potential_integer_division():
    """Choose variables that can trigger integer division."""
    theta = 1; a = 1; I = 1; dt = 2
    N = 4
    u, t = decay_mod.solver(I=I, a=a, T=N*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(N+1)])
    diff = np.abs(u_de - u).max()
    nt.assert_almost_equal(diff, 0, delta=1E-14)
```

The final test is to see that the convergence rates corresponding to $\theta = 0, 0.5$
, respectively:

```
def test_convergence_rates():
    """Compare empirical convergence rates to exact ones."""
    # Set command-line arguments directly in sys.argv
    import sys
    sys.argv[1:] = '--I 0.8 --a 2.1 --T 5 '\
                   '--dt 0.4 0.2 0.1 0.05 0.025'.split()
    r = decay_mod.main()
    for theta in r:
        nt.assert_true(r[theta])  # check for non-empty list

    expected_rates = {0: 1, 1: 1, 0.5: 2}
    for theta in r:
        r_final = r[theta][-1]
        # Compare to 1 decimal place
        nt.assert_almost_equal(expected_rates[theta], r_final,
                               places=1, msg='theta=%s' % theta)
```

Nothing more is needed in the test_decay_nose.py[17] file where the tests
osetests -s will report Ran 3 tests and an OK for success. Every time
ecay_mod_unittest module we can run nosetests to quickly see if the edits
n the verification tests.

**nstallation of nose.** The nose package does not come with a standard Pyt
nd must therefore be installed separately. The procedure is standard and describe
ages[18]. On Debian-based Linux systems the command is sudo apt-get instal
nd with MacPorts you run sudo port install py27-nose.

**Using nose to test modules with doctests.** Assume that mod is the name
ıat contains doctests. We may let nose run these doctests and report errors
ay using the code set-up

[17] http://tinyurl.com/jvzzcfn/decay/tests/test_decay_nose.py
[18] http://nose.readthedocs.org/en/latest/

```
octest
od

_mod():
ure_count, test_count = doctest.testmod(m=mod)
ssert_equal(failure_count, 0,
            msg='%d tests out of %d failed' %
            (failure_count, test_count))
```

o `doctest.testmod` runs all doctests in the module file `mod.py` and returns the
(`failure_count`) and the total number of tests (`test_count`). A real example
`test_decay_doctest.py`[19].

### lassical class-based unit testing

cal way of implementing unit tests derives from the JUnit tool in Java where
ds in a class for testing. Python comes with a module `unittest` for doing this
. While `nose` allows simple functions for unit tests, `unittest` requires deriving
m `unittest.TestCase` and implementing each test as methods with names `te`
. I strongly recommend to use `nose` over `unittest`, because it is much simpler a
t, but class-based unit testing is a very classical subject that computational s
ve some knowledge about. That is why a short introduction to `unittest` is i

**e of unittest.** We apply the `double` function in the `mymod` module introduce
ection as example. Unit testing with the aid of the `unittest` module consists of
`t_mymod.py` with the content

```
nittest
ymod

stMyCode(unittest.TestCase):
test_double(self):
result = mymod.double(4)
self.assertEqual(result, 8)

e__ == '__main__':
test.main()
```

s run by executing the test file `test_mymod.py` as a standard Python program
port in `unittest` for automatically locating and running all tests in all test f
tree.
who have experience with object-oriented programming will see that the di
using `unittest` and `nose` is minor.

**tration of unittest.** The same tests as shown for the nose framework are
ith the `TestCase` classes in the file `test_decay_unittest.py`[20]. The tests are i
lifference being that with `unittest` we must write the tests as methods in a c
functions have slightly different names.

/tinyurl.com/jvzzcfn/decay/tests/test_decay_doctest.py
/tinyurl.com/jvzzcfn/decay/tests/test_decay_nose.py



```
import unittest
import decay_mod_unittest as decay
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*factor**n

class TestDecay(unittest.TestCase):

    def test_exact_discrete_solution(self):
        ...
        diff = np.abs(u_de - u).max()
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_solver(self):
        ...
        for theta in 0, 0.5, 1:
            ...
            self.assertAlmostEqual(diff, 0, places=8,
                                   msg='theta=%s' % theta)

    def test_potential_integer_division():
        ...
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_convergence_rates(self):
        ...
        for theta in r:
            ...
            self.assertAlmostEqual(...)

if __name__ == '__main__':
    unittest.main()
```

## .6 Implementing simple problem and solver classes

he $\theta$-rule was compactly and conveniently implemented in a function `solver` i
ore complicated problems it might be beneficial to use classes and introduce a
old the definition of the physical problem, a class `Solver` to hold the data and
o numerically solve the problem, and a class `Visualizer` to make plots. This
lustrated, resulting in code that represents an alternative to the `solver` and `ex`
ound in the `decay_mod` module.

Explaining the details of class programming in Python is considered beyond
ext. Readers who are unfamiliar with Python class programming should first c
any electronic Python tutorials or textbooks to come up to speed with concep
ython classes before reading on. The author has a gentle introduction to class
cientific applications in [1], see Chapter 7 and 9 and Appendix E. Other useful

- The Python Tutorial: `http://docs.python.org/2/tutorial/classes.h`
- Wiki book on Python Programming: `http://en.wikibooks.org/wiki/Pyt`
  `Classes`
- tutorialspoint.com: `http://www.tutorialspoint.com/python/python_c`
  `htm`

**blem class.** The purpose of the problem class is to store all information ab
tical model. This usually means all the physical parameters in the problem.
ample with exponential decay we may also add the exact solution of the ODI
lass. The simplest form of a problem class is therefore

```
py import exp

oblem:
__init__(self, I=1, a=1, T=10):
self.T, self.I, self.a = I, float(a), T

u_exact(self, t):
I, a = self.I, self.a
return I*exp(-a*t)
```

in the `u_exact` method have written `self.I*exp(-self.a*t)`, but using local v
allows the formula `I*exp(-a*t)` which looks closer to the mathematical exp
is is not an important issue with the current compact formula, but is beneficial
ed problems with longer formulas to obtain the closest possible relationship l
mathematics. My coding style is to strip off the `self` prefix when the code e
tical formulas.
ass data can be set either as arguments in the constructor or at any time late

```
= Problem(T=5)
T = 8
dt = 1.5
```

ogrammers prefer `set` and `get` functions for setting and getting data in classe
ted via *properties* in Python, but I consider that overkill when we just have a f
class.)
uld be convenient if class `Problem` could also initialize the data from the co
his end, we add a method for defining a set of command-line options and a
the local attributes equal to what was found on the command line. The defaul
l with the command-line options are taken as the values provided to the cons
`blem` now becomes

```
oblem:
__init__(self, I=1, a=1, T=10):
self.T, self.I, self.a = I, float(a), T

define_command_line_options(self, parser=None):
if parser is None:
    import argparse
    parser = argparse.ArgumentParser()

parser.add_argument(
    '--I', '--initial_condition', type=float,
    default=self.I, help='initial condition, u(0)',
    metavar='I')
parser.add_argument(
    '--a', type=float, default=self.a,
    help='coefficient in ODE', metavar='a')
parser.add_argument(
    '--T', '--stop_time', type=float, default=self.T,
    help='end time of simulation', metavar='T')
return parser
```



```
    def init_from_command_line(self, args):
        self.I, self.a, self.T = args.I, args.a, args.T

    def exact_solution(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

bserve that if the user already has an `ArgumentParser` object it can be sup
oes not have any, class `Problem` makes one. Python's `None` object is used to
ariable is not initialized with a proper value.

**he solver class.** The solver class stores data related to the numerical solut
rovides a function `solve` for solving the problem. A problem object must
onstructor so that the solver can easily look up physical data. In the present ex
elated to the numerical solution method consists of $\Delta t$ and $\theta$. We add, as in th
nctionality for reading $\Delta t$ and $\theta$ from the command line:

```
lass Solver:
    def __init__(self, problem, dt=0.1, theta=0.5):
        self.problem = problem
        self.dt, self.theta = float(dt), theta

    def define_command_line_options(self, parser):
        parser.add_argument(
            '--dt', '--time_step_value', type=float,
            default=0.5, help='time step value', metavar='dt')
        parser.add_argument(
            '--theta', type=float, default=0.5,
            help='time discretization parameter', metavar='dt')
        return parser

    def init_from_command_line(self, args):
        self.dt, self.theta = args.dt, args.theta

    def solve(self):
        from decay_mod import solver
        self.u, self.t = solver(
            self.problem.I, self.problem.a, self.problem.T,
            self.dt, self.theta)

    def error(self):
        u_e = self.problem.exact_solution(self.t)
        e = u_e - self.u
        E = sqrt(self.dt*sum(e**2))
        return E
```

ote that we here simply reuse the implementation of the numerical method from
odule. The `solve` function is just a *wrapper* of the previously developed sta
nction.

**he visualizer class.** The purpose of the visualizer class is to plot the nu
ored in class `Solver`. We also add the possibility to plot the exact solution
roblem and solver objects is required when making plots so the constructor mus
o these objects:

```
lass Visualizer:
    def __init__(self, problem, solver):
        self.problem, self.solver = problem, solver
```

```
plot(self, include_exact=True, plt=None):
"""
Add solver.u curve to the plotting object plt,
and include the exact solution if include_exact is True.
This plot function can be called several times (if
the solver object has computed new solutions).
"""
if plt is None:
    import scitools.std  as plt # can use matplotlib as well

plt.plot(self.solver.t, self.solver.u, '--o')
plt.hold('on')
theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
name = theta2name.get(self.solver.theta, '')
legends = ['numerical %s' % name]
if include_exact:
    t_e = linspace(0, self.problem.T, 1001)
    u_e = self.problem.exact_solution(t_e)
    plt.plot(t_e, u_e, 'b-')
    legends.append('exact')
plt.legend(legends)
plt.xlabel('t')
plt.ylabel('u')
plt.title('theta=%g, dt=%g' %
          (self.solver.theta, self.solver.dt))
plt.savefig('%s_%g.png' % (name, self.solver.dt))
return plt
```

lt object in the `plot` method is worth a comment. The idea is that `plot` ca l solution curve to an existing plot. Calling `plot` with a `plt` object (which h: clib.pyplot or `scitools.std` object in this implementation), will just add th ver.u as a dashed line with circles at the mesh points (leaving the color of th plotting tool). This functionality allows plots with several solutions: just mak / data is set in the problem and/or solver classes, the solver's `solve()` method i nost recent numerical solution is plotted by the `plot(plt)` method in the vi ercise 6 describes a problem setting where this functionality is explored.

**ng the objects.** Eventually we need to show how the classes `Problem`, `Solv` `er` play together:

```
():
lem = Problem()
er = Solver(problem)
= Visualizer(problem, solver)

ad input from the command line
er = problem.define_command_line_options()
er = solver. define_command_line_options(parser)
 = parser.parse_args()
lem.init_from_command_line(args)
er. init_from_command_line(args)

lve and plot
er.solve()
rt matplotlib.pyplot as plt
ort scitools.std as plt
= viz.plot(plt=plt)
solver.error()
 is not None:
print 'Error: %.4E' % E
show()
```



The file `decay_class.py`[21] constitutes a module with the three classes and th

**Test the understanding.**

Implement the problem in Exercise **??** in terms of problem, solver, and visu Equip the classes and their methods with doc strings with tests. Also includ

## .7 Improving the problem and solver classes

he previous `Problem` and `Solver` classes containing parameters soon get much hen the number of parameters increases. Much of this code can be parameteriz ore compact. For this purpose, we decide to collect all parameters in a diction ith two associated dictionaries `self.types` and `self.help` for holding associa nd help strings. Provided a problem, solver, or visualizer class defines these thre ne constructor, using default or user-supplied values of the parameters, we ca lass `Parameters` with general code for defining command-line options and read s methods for setting and getting a parameter. A `Problem` or `Solver` class ommand-line functionality and the set/get methods from the `Parameters` clas

**generic class for parameters.** A simplified version of the parameter class

```
class Parameters:
    def set(self, **parameters):
        for name in parameters:
            self.prms[name] = parameters[name]

    def get(self, name):
        return self.prms[name]

    def define_command_line_options(self, parser=None):
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        for name in self.prms:
            tp = self.types[name] if name in self.types else str
            help = self.help[name] if name in self.help else None
            parser.add_argument(
                '--' + name, default=self.get(name), metavar=name,
                type=tp, help=help)

        return parser

    def init_from_command_line(self, args):
        for name in self.prms:
            self.prms[name] = getattr(args, name)
```

he file `class_decay_oo.py`[22] contains a slightly more advanced version of cl here we in the `set` and `get` functions test for valid parameter names and raise nformative messages if any name is not registered.

[21] http://tinyurl.com/jvzzcfn/decay/decay_class.py
[22] http://tinyurl.com/jvzzcfn/decay/class_decay_oo.py

**blem class.** A class `Problem` for the problem $u' = -au$, $u(0) = I$, $t \in (0,$
rs input $a$, $I$, and $T$ can now be coded as

```
oblem(Parameters):

ical parameters for the problem u'=-a*u, u(0)=I,
 t in [0,T].

__init__(self):
self.prms = dict(I=1, a=1, T=10)
self.types = dict(I=float, a=float, T=float)
self.help = dict(I='initial condition, u(0)',
                 a='coefficient in ODE',
                 T='end time of simulation')

exact_solution(self, t):
I, a = self.get('I'), self.get('a')
return I*np.exp(-a*t)
```

**er class.** Also the solver class is derived from class `Parameters` and works v
es, and `help` dictionaries in the same way as class `Problem`. Otherwise, the
ar to class `Solver` in the `decay_class.py` file:

```
lver(Parameters):
__init__(self, problem):
self.problem = problem
self.prms = dict(dt=0.5, theta=0.5)
self.types = dict(dt=float, theta=float)
self.help = dict(dt='time step value',
                 theta='time discretization parameter')

solve(self):
from decay_mod import solver
self.u, self.t = solver(
    self.problem.get('I'),
    self.problem.get('a'),
    self.problem.get('T'),
    self.get('dt'),
    self.get('theta'))

error(self):
try:
    u_e = self.problem.exact_solution(self.t)
    e = u_e - self.u
    E = np.sqrt(self.get('dt')*np.sum(e**2))
except AttributeError:
    E = None
return E
```

**alizer class.** Class `Visualizer` can be identical to the one in the `decay_c`
he class does not need any parameters. However, a few adjustments in the `plot`
ry since parameters are accessed as, e.g., `problem.get('T')` rather than `pro`
ls are found in the file `class_decay_oo.py`.
, we need a function that solves a real problem using the classes `Problem`, `Solv`
`er`. This function can be just like `main` in the `decay_class.py` file.
dvantage with the `Parameters` class is that it scales to problems with a large nu
nd numerical parameters: as long as the parameters are defined once via a dic
act code in class `Parameters` can handle any collection of parameters of any si



# 5 Performing scientific experiments

**Goal.**

This section explores the behavior of a numerical method for a differential equ
computer experiments. In particular, it is shown how scientific experiments
and reported. We address the ODE problem

$$u'(t) = -au(t), \quad u(0) = I, \quad t \in (0, T],$$

numerically discretized by the $\theta$-rule:

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a \Delta t} u^n, \quad u^0 = I\,.$$

Our aim is to plot $u^0, u^1, \ldots, u^N$ together with the exact solution $u_{\mathrm{e}} = Ie^-$
choices of the parameters in this numerical problem: $I$, $a$, $\Delta t$, and $\theta$. We
interested in how the discrete solution compares with the exact solution
parameter is varied and $\theta$ takes on the three values corresponding to the F
Backward Euler, and Crank-Nicolson schemes ($\theta = 0, 1, 0.5$, respectively).

## .1 Software

verified implementation for computing the numerical solution $u^n$ and plotting
he exact solution $u_{\mathrm{e}}$ is found in the file `decay_mod.py`[23]. This program admit
rguments to specify a series of $\Delta t$ values and will run a loop over these values
We make a slight edit of how the plots are designed: the numerical solution i
ne type `'r-o'` (dashed red lines with dots at the mesh points), and the `show`
emoved to avoid a lot of plot windows popping up on the computer screen
f the plot are still stored in files via `savefig`). The slightly modified progra
`xperiments/decay_mod.py`[24]. All files associated with the scientific investigat
a subdirectory `experiments`.

Running the experiments is easy since the `decay_mod.py` program already ha
and $\Delta t$ implemented. An experiment with $I = 1$, $a = 2$, $T = 5$, and $dt = 0.5$,
un by

```
erminal> python decay_mod.py --I 1 --a 2 --makeplot \
         --T 5 --dt 0.5 0.25 0.1 0.05
```

## .2 Combining plot files

he `decay_mod.py` program generates a lot of image files, e.g., `FE_*.png`,
`N_*.png`. We want to combine all the `FE_*.png` files in a table fashion i
wo images in each row, starting with the largest $\Delta t$ in the upper left corner an
alue as we go to the right and down. This can be done using the `montage`[25] pro

[23] `http://tinyurl.com/jvzzcfn/decay/decay_mod.py`
[24] `http://tinyurl.com/jvzzcfn/decay/experiments/decay_mod.py`
[25] `http://www.imagemagick.org/script/montage.php`

white areas around the plots can be cropped away by the `convert -trim` co
ining white can be made transparent for HTML pages with a non-white backgr
and `convert -transparent white`.
lot files in the PDF format with names `FE_*.pdf`, `BE_*.pdf`, and `CN_*.pdf` are g
should be combined using other tools: `pdftk` to combine individual plots into
plot per page, and `pdfnup` to combine the pages into a table with multiple p
resulting image often has some extra surrounding white space that can be rem
`op` program. The code snippets below contain all details about the usage of
`pdftk`, `pdfnup`, and `pdfcrop`.
ng manual commands is boring, and errors may easily sneak in. Both for aut
ork and documenting the operating system commands we actually issued
t, we should write a *script* (little program). An alternative is to write the co
ython notebook and use the notebook as the script. A plain script as a standard
n a separate text file will be used here.

**ducible science.**

t that automates running our computer experiments will ensure that the experi
ily be rerun by ourselves or others in the future, either to check the results or
periments with other input data. Also, whatever we did to produce the resu
ented in every detail in the script. Automating scripts are therefore essenti
our research *reproducible*, which is a fundamental principle in science.

ript takes a list of $\Delta t$ values on the command line as input and makes three c
ne for each $\theta$ value, displaying the quality of the numerical solution as $\Delta t$ va

```
python decay_exper0.py 0.5 0.25 0.1 0.05
```

images `FE.png`, `CN.png`, `BE.png`, `FE.pdf`, `CN.pdf`, and `BE.pdf`, each with fo
ding to the four $\Delta t$ values. Each plot compares the numerical solution with t
latter image is shown in Figure 2.
, the script should be scalable in the sense that it works for any number of $\Delta$
he case for this particular implementation:

```
os, sys

_experiments(I=1, a=2, T=5):
he command line must contain dt values
len(sys.argv) > 1:
 dt_values = [float(arg) for arg in sys.argv[1:]]
e:
 print 'Usage: %s dt1 dt2 dt3 ...' %  sys.argv[0]
 sys.exit(1)  # abort

un module file as a stand-alone application
 = 'python decay_mod.py --I %g --a %g --makeplot --T %g' % \
   (I, a, T)
values_str = ' '.join([str(v) for v in dt_values])
 += ' --dt %s' % dt_values_str
nt cmd
```



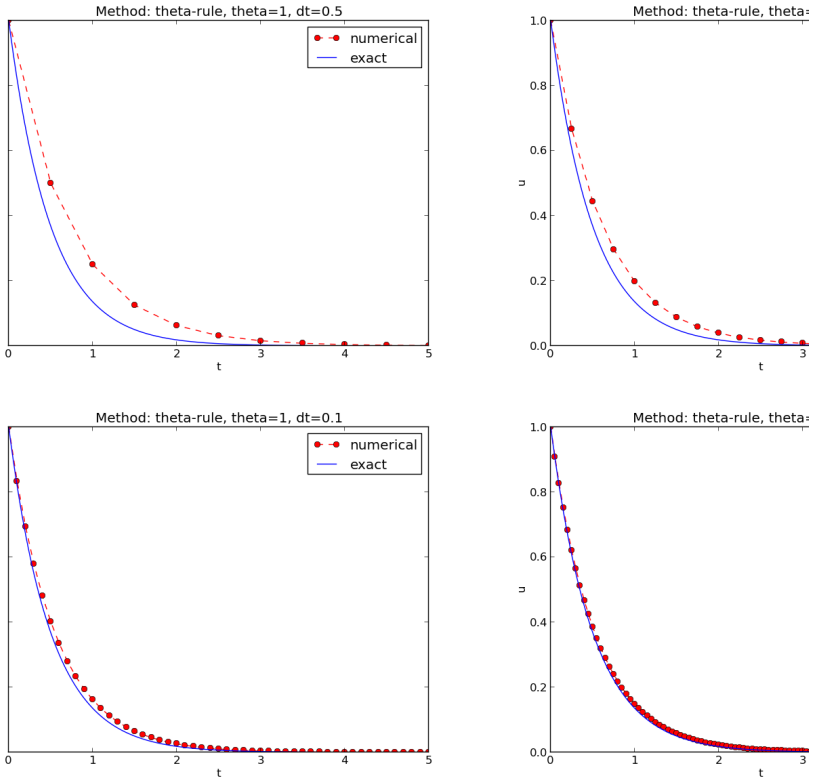



Figure 2: Illustration of the Backward Euler method for four time step

```
    failure = os.system(cmd)
    if failure:
        print 'Command failed:', cmd; sys.exit(1)

    # Combine images into rows with 2 plots in each row
    image_commands = []
    for method in 'BE', 'CN', 'FE':
        pdf_files = ' '.join(['%s_%g.pdf' % (method, dt)
                              for dt in dt_values])
        png_files = ' '.join(['%s_%g.png' % (method, dt)
                              for dt in dt_values])
        image_commands.append(
            'montage -background white -geometry 100%' +
            ' -tile 2x %s %s.png' % (png_files, method))
        image_commands.append(
            'convert -trim %s.png %s.png' % (method, method))
        image_commands.append(
            'convert %s.png -transparent white %s.png' %
            (method, method))
        image_commands.append(
            'pdftk %s output tmp.pdf' % pdf_files)
        num_rows = int(round(len(dt_values)/2.0))
```

```
image_commands.append(
    'pdfnup --nup 2x%d tmp.pdf' % num_rows)
image_commands.append(
    'pdfcrop tmp-nup.pdf %s.pdf' % method)

cmd in image_commands:
print cmd
failure = os.system(cmd)
if failure:
    print 'Command failed:', cmd; sys.exit(1)

emove the files generated above and by decay_mod.py
m glob import glob
enames = glob('*_*.png') + glob('*_*.pdf') + \
         glob('*_*.eps') + glob('tmp*.pdf')
filename in filenames:
os.remove(filename)

me__ == '__main__':
_experiments()
```

s available as `experiments/decay_exper0.py`[26].
ay comment upon many useful constructs in this script:

`oat(arg) for arg in sys.argv[1:]]` builds a list of real numbers from mand-line arguments.

`lure = os.system(cmd)` runs an operating system command, e.g., another p execution is successful only if `failure` is zero.

uccessful execution usually makes it meaningless to continue the program, and t abort the program with `sys.exit(1)`. Any argument different from 0 signifie puter's operating system that our program stopped with a failure.

`s_%s.png' % (method, dt) for dt in dt_values]` builds a list of filename of numbers (`dt_values`).

`montage`, `convert`, `pdftk`, `pdfnup`, and `pdfcrop` commands for creating co res are stored in a list and later executed in a loop.

`b('*_*.png')` returns a list of the names of all files in the current directory w name matches the Unix wildcard notation[27] `*_*.png` (meaning any text, unc text, and then `.png`).

`remove(filename)` removes the file with name `filename`.

## terpreting output from other programs

that run other programs, like `decay_exper0.py` does, will often need to i om those programs. Let us demonstrate how this is done in Python by extrac between $\theta$, $\Delta t$, and the error $E$ as written to the terminal window by the `decay_` when being executed by `decay_exper0.py`. We will

l the output from the `decay_mod.py` program

/tinyurl.com/jvzzcfn/decay/experiments/decay_exper0.py
/en.wikipedia.org/wiki/Glob_(programming)



- interpret this output and store the $E$ values in arrays for each $\theta$ value
- plot $E$ versus $\Delta t$, for each $\theta$, in a log-log plot

The simple `os.system(cmd)` call does not allow us to read the output fr nstead we need to invoke a bit more involved procedure:

```
from subprocess import Popen, PIPE, STDOUT
p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
output, dummy = p.communicate()
failure = p.returncode
if failure:
    print 'Command failed:', cmd; sys.exit(1)
```

The command stored in `cmd` is run and all text that is written to the standard standard error is available in the string `output`. Or in other words, the text in ppeared in the terminal window while running `cmd`.

Our next task is to run through the `output` string, line by line, and if the cu , $\Delta t$, and $E$, we split the line into these three pieces and store the data. The tructure is a dictionary `errors` with keys `dt` to hold the $\Delta t$ values in a list, and old the corresponding $E$ values in a list. The relevant code lines are

```
errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
for line in output.splitlines():
    words = line.split()
    if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
        # typical line: 0.0   1.25:    7.463E+00
        theta = float(words[0])
        E = float(words[2])
        errors[theta].append(E)
```

Note that we do not bother to store the $\Delta t$ values as we read them from `out` already have these values in the `dt_values` list.

We are now ready to plot $E$ versus $\Delta t$ for $\theta = 0, 0.5, 1$:

```
import matplotlib.pyplot as plt
plt.loglog(errors['dt'], errors[0], 'ro-')
plt.hold('on')
plt.loglog(errors['dt'], errors[0.5], 'b+-')
plt.loglog(errors['dt'], errors[1], 'gx-')
plt.legend(['FE', 'CN', 'BE'], loc='upper left')
plt.xlabel('log(time step)')
plt.ylabel('log(error)')
plt.title('Error vs time step')
plt.savefig('error.png')
plt.savefig('error.pdf')
```

lots occasionally need some manual adjustments. Here, the axis of the log-log p e adapt them strictly to the data, see Figure 3. To this end, we need to com ax $E$, and later specify the extent of the axes:

```
# Find min/max for the axis
E_min = 1E+20; E_max = -E_min
for theta in 0, 0.5, 1:
    E_min = min(E_min, min(errors[theta]))
    E_max = max(E_max, max(errors[theta]))
```

```
og(errors['dt'], errors[0], 'ro-')
([min(dt_values), max(dt_values), E_min, E_max])
```

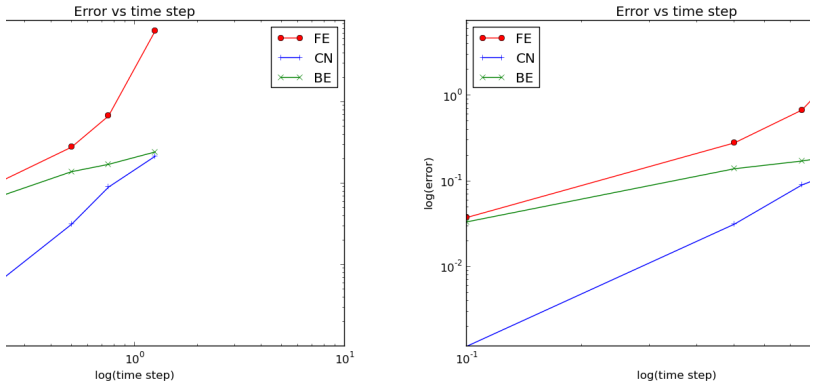



Figure 3: Default plot (left) and manually adjusted axes (right).

omplete program, incorporating the code snippets above, is found in `exper` `per1.py`[28]. This example can hopefully act as template for numerous other o needs to run experiments, extract data from the output of programs, make pl several plots in a figure file. The `decay_exper1.py` program is organized as a files can then easily extend the functionality, as illustrated in the next sectio

## aking a report

ts of running computer experiments are best documented in a little report co em to be solved, key code segments, and the plots from a series of experime part of the report containing the plots should be automatically generated by t orms the set of experiments, because in that script we know exactly which inp used to generate a specific plot, thereby ensuring that each figure is connecte a. Take a look at an example at `http://tinyurl.com/k3sdbuv/writing_r` `cloud/` to see what we have in mind.

**TML.** Scientific reports can be written in a variety of formats. Here we beg L[29] format which allows efficient viewing of all the experiments in any web am `decay_exper1_html.py`[30] calls `decay_exper1.py` to perform the experime statements for creating an HTML file with a summary, a section on the mathe a section on the numerical method, a section on the `solver` function implemen nd a section with subsections containing figures that show the results of expe is varied for $\theta = 0, 0.5, 1$. The mentioned Python file contains all the details for

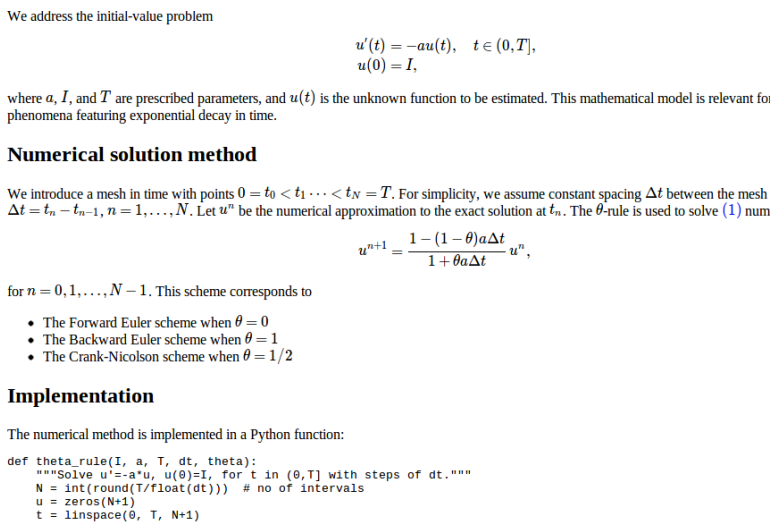
/tinyurl.com/jvzzcfn/decay/experiments/decay_exper1.py
/en.wikipedia.org/wiki/HTML
/tinyurl.com/jvzzcfn/decay/experiments/decay_exper1_html.py



is HTML report[31]. You can view the report on `http://tinyurl.com/k3s` `eports//_static/report_html.html`.

**TML with MathJax.** Scientific reports usually need mathematical form athematical typesetting. In plain HTML, as used in the `decay_exper1_html.p` se just the keyboard characters to write mathematics. However, there is an exte alled MathJax[32], which allows formulas and equations to be typeset with L icely rendered in web browsers, see Figure 4. A relatively small subset of LaTeX upported, but the syntax for formulas is quite rich. Inline formulas are look lik hile equations are surrounded by `$$` signs. Inside such signs, one can use `\` nnumbered equations, or `\begin{equation}` and `\end{equation}` surrounc umbered equations, or `\begin{align}` and `\end{align}` for multiple aligned eed to be familiar with mathematical typesetting in LaTeX[33].

The file `decay_exper1_mathjax.py`[34] contains all the details for turning th TML report into web pages with nicely typeset mathematics. The correspondir e studied to see all details of the mathematical typesetting.



We address the initial-value problem

$$u'(t) = -au(t), \quad t \in (0,T],$$
$$u(0) = I,$$

where $a$, $I$, and $T$ are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant fo phenomena featuring exponential decay in time.

**Numerical solution method**

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_N = T$. For simplicity, we assume constant spacing $\Delta t$ between the mesh $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N$. Let $u^n$ be the numerical approximation to the exact solution at $t_n$. The $\theta$-rule is used to solve (1) num

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\, u^n,$$

for $n = 0, 1, \ldots, N-1$. This scheme corresponds to

- The Forward Euler scheme when $\theta = 0$
- The Backward Euler scheme when $\theta = 1$
- The Crank-Nicolson scheme when $\theta = 1/2$

**Implementation**

The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)
```

Figure 4: Report in HTML format with MathJax.

**TEX.** The *de facto* language for mathematical typesetting and scientific r aTeX[36]. A number of very sophisticated packages have been added to the eriod of three decades, allowing very fine-tuned layout and typesetting. For ou rmat[37], see Figure 5 for an example, LaTeX is the definite choice when it comes

[31] `http://tinyurl.com/k3sdbuv/writing_reports//_static/report_html.html.html`
[32] `http://www.mathjax.org/`
[33] `http://en.wikibooks.org/wiki/LaTeX/Mathematics`
[34] `http://tinyurl.com/jvzzcfn/decay/experiments/decay_exper1_html.py`
[35] `http://tinyurl.com/k3sdbuv/writing_reports//_static/report_mathjax.html.html`
[36] `http://en.wikipedia.org/wiki/LaTeX`
[37] `http://tinyurl.com/k3sdbuv/writing_reports//_static/report.pdf`

guage used to write the reports has typically a lot of commands involving bac
ɜs[38]. For output on the web, using HTML (and not the PDF directly in the
LaTeX struggles with delivering high quality typesetting. Other tools, especially
ɛr results and can also produce nice-looking PDFs. The file `decay_exper1_1a`
v to generate the LaTeX source from a program.

## 3 Implementation

The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)

    u[0] = I
    for n in range(0, N):
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

## 4 Numerical experiments

We define a set of numerical experiments where $I$, $a$, and $T$ are fixed, while $\Delta t$ and $\theta$ are varied. In particular, $I = 1$, $a = 2$, $\Delta t = 1.25, 0.75, 0.5, 0.1$.

Figure 5: Report in PDF format generated from LaTeX source.

Sphinx[39] is a typesetting language with similarities to HTML and LaTeX, but wi
ıg. It has recently become very popular for software documentation and mathe
phinx can utilize LaTeX for mathematical formulas and equations (via MathJax
Unfortunately, the subset of LaTeX mathematics supported is less than in full M
ılar, numbering of multiple equations in an `align` type environment is not sup
ıx syntax[40] is an extension of the reStructuredText language. An attractive fe
its rich support for fancy layout of web pages[41]. In particular, Sphinx can e
with various layout *themes* that give a certain look and feel to the web site a
le of contents, navigation, and search facilities, see Figure 6.

**vn.** A recently popular format for easy writing of web pages is Markdown[42].
ɛry much like one would do in email, using spacing and special characters to n
ıe code instead of heavily tagging the text as in LaTeX and HTML. With
one can go from Markdown to a variety of formats. HTML is a common output
., epub, XML, OpenOffice, MediaWiki, and MS Word are some other possibili

/tinyurl.com/k3sdbuv/writing_reports//_static/report.tex.html
/sphinx.pocoo.org/
/tinyurl.com/k3sdbuv/writing_reports//_static/report_sphinx.rst.html
/tinyurl.com/k3sdbuv/writing_reports//_static/sphinx-cloud/index.html
/daringfireball.net/projects/markdown/
/johnmacfarlane.net/pandoc/



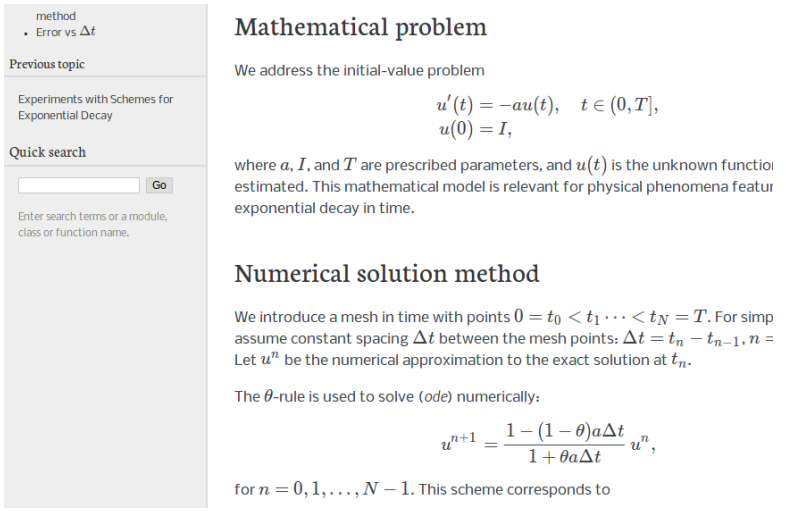



Figure 6: Report in HTML format generated from Sphinx source

**Viki formats.** A range of wiki formats are popular for creating notes on the
ocuments which allow groups of people to edit and add content. Apart from M
iki format used for Wikipedia), wiki formats have no support for mathematical
lso limited tools for displaying computer code in nice ways. Wiki formats a
uitable for scientific reports compared to the other formats mentioned here.

**DocOnce.** Since it is difficult to choose the right tool or format for writing a sc
advantageous to write the content in a format that easily translates to LaTeX,
Markdown, and various wikis. DocOnce[45] is such a tool. It is similar to Pa
ɔme special convenient features for writing about mathematics and programmi
modest[46], somewhere between LaTeX and Markdown. The program `dec`
emonstrates how to generate (and write) DocOnce code for a report.

**Worked example.** The HTML, LaTeX (PDF), Sphinx, and DocOnce formats
ɛport whose content is outlined above, are exemplified with source codes an
ɛb pages associated with this teaching material: `http://tinyurl.com/k3s`
`eports/`.

### .5 Publishing a complete project

report documenting scientific investigations should be accompanied by all t
ata used for the investigations so that others have a possibility to redo the wor
ualify of the results. This possibility is important for *reproducible research* anc
ɛliable scientific conclusions.

[44] http://www.mediawiki.org/wiki/MediaWiki
[45] https://github.com/hplgit/doconce
[46] http://tinyurl.com/k3sdbuv/writing_reports//_static/report.do.txt.html

ray of documenting a complete project is to make a directory tree with all
ferably, the tree is published at some project hosting site like Bitbucket, Git
le[47] so that others can download it as a tarfile, zipfile, or clone the files directly
ntrol system like Mercurial or Git. For the investigations outlined in Section
e a directory tree with files

```
.py
::
cay_mod.py
::
src:
  decay_exper1_mathjax.py
  make_report.sh
  run.sh
pub:
  report.html
```

directory holds source code (modules) to be reused in other projects, the se
l installs such software, the `doc` directory contains the documentation, with `src`
the documentation and `pub` for ready-made, published documentation. The `run`
e Bash script listing the `python` command we used to run `decay_exper1_mathj`
te the experiments and the `report.html` file.

## ercises

### e 1: Refactor a flat program in terms of a function

e ODEs of the form

$$u' = f(t), \quad u(0) = I, \ t \in (0, T]$$

ıd the solution by straightforward integration:

$$u(t) = \int_0^t f(\tau)d\tau\,.$$

ıte $u(t)$ for $t \in [0, T]$, we introduce a uniform time mesh with points $t_n = n$
Trapezoidal rule to approximate the integral. Suppose we have computed the nu
ation $u^n$ to $u(t_n)$. We have

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(\tau)d\tau\,.$$

Trapezoidal rule we get

$$u^{n+1} = u^n + \frac{1}{2}\Delta t(f(t_n) + f(t_{n+1}))\,.$$

ng value is $u^0 = I$. A corresponding implementation for the case $f(t) = 2t + 1$

```
s 2*t + 1

py import *
                # time step
```

/hplgit.github.com/teamods/bitgit/html/



```
Nt = int(round(T/dt))  # no of mesh points
u = zeros(Nt+1)
t = linspace(0, T, Nt+1)
for n in range(Nt-1):
    u[n+1] = u[n] + 0.5*dt*(2*t[n]+1 + 2*t[n+1]+1)
```

his is a flat program. Refactor the program as a function `solver(f, I, T, dt`
ython implementation of the mathematical function $f(t)$ that is to be integra
alue of `solver` is the pair (`u`, `t`) representing the solution values and the assoc
ilename: `integrate.py`.

**emarks.** Many prefer to do a first implementation of an algorithm as a fl
ardcode formulas, here the $f(t)$, into the algorithm. Unfortunately, this codin
ifficult to reuse a well-tested algorithm. When the flat program works, it is strong
o *refactor* the code (i.e., rearrange the statements) such that general algorithms
ı Python functions. The function arguments should be chosen such that the
pplied for a large class of problems, here all problems that can be expressed as

### xercise 2: Compare methods for a given time mesh

Iake a program that imports the `solver` function from the `decay_mod` mod
ınction `compare(dt, I, a)` for comparing, in a plot, the methods correspondir
nd the exact solution. This plot shows the accuracy of the methods for a given t
ıput data for the problem from the command line using appropriate functions ir
ıodule (the `-dt` option for giving several time step values can be reused: just u
ep value for the computations). Filename: `decay_compare_theta.py`.

### roblem 3: Write a doctest

ype in the following program and equip the `roots` function with a doctest:

```
import sys
# This sqrt(x) returns real if x>0 and complex if x<0
from numpy.lib.scimath import sqrt

def roots(a, b, c):
    """
    Return the roots of the quadratic polynomial
    p(x) = a*x**2 + b*x + c.

    The roots are real or complex objects.
    """
    q = b**2 - 4*a*c
    r1 = (-b + sqrt(q))/(2*a)
    r2 = (-b - sqrt(q))/(2*a)
    return r1, r2

a, b, c = [float(arg) for arg in sys.argv[1:]]
print roots(a, b, c)
```

Iake sure to test both real and complex roots. Write out numbers with 14 digits c
`octest_roots.py`.

### roblem 4: Write a nose test

Iake a nose test for the `roots` function in Problem 3. Filename: `test_roots.`

### n 5: Make a module

$$q(t) = \frac{RAe^{at}}{R + A(e^{at} - 1)}\,.$$

ython module `q_module` containing two functions `q(t)` and `dqdt(t)` for compu
respectively. Perform a `from numpy import *` in this module. Import `q` and
le using the "star import" construction `from q_module import *`. All objects a
e is given by `dir()`. Print `dir()` and `len(dir())`. Then change the import c
ule.py to `import numpy as np`. What is the effect of this import on the nu
`dir()` in a file that does `from q_module import *`? Filename: `q_module.py`

### e 6: Make use of a class implementation

to solve the exponential decay problem $u' = -au$, $u(0) = I$, for several $\Delta t$ val
, 1. For each $\Delta t$ value, we want to make a plot where the three solutions corres
).5, 1 appear along with the exact solution. Write a function `experiment` to acc
e function should import the classes `Problem`, `Solver`, and `Visualizer` fi
ass[48] module and make use of these. A new command-line option `--dt_valu`
to allow the user to specify the $\Delta t$ values on the command line (the opti
ta implemented by the `decay_class` module have then no effect when runn
nt function). Note that the classes in the `decay_class` module should *not* be r
`decay_class_exper.py`.

### e 7: Generalize a class implementation

the file `decay_class.py`[49] where the exponential decay problem $u' = -au$, $u(0$
ted via the classes `Problem`, `Solver`, and `Visualizer`. Extend the classes to
general problem

$$u'(t) = -a(t)u(t) + b(t), \quad u(0) = I,\ t \in (0, T],$$

$\theta$-rule for discretization.
case with arbitrary functions $a(t)$ and $b(t)$ the problem class is no longer guara
n exact solution. Let the `u_exact` in class `Problem` return `None` if the exact solu
ular problem is not available. Modify classes `Solver` and `Visualizer` accordi
st functions `test_*()` for the nose testing tool in the module. Also add a demo
environment suddenly changes (modeled as an abrupt change in the decay ra

$$a(t) = \begin{cases} 1, & 0 \leq t \leq t_p, \\ k, & t > t_p, \end{cases}$$

s the point of time the environment changes. Take $t_p = 1$ and make plots that il
of having $k \gg 1$ and $k \ll 1$. Filename: `decay_class2.py`.

### e 8: Generalize an advanced class implementation

ercise 7 by utilizing the class implementations in `decay_class_oo.py`[50]. F
ass3.py`.

/tinyurl.com/jvzzcfn/decay/decay_class.py
/tinyurl.com/jvzzcfn/decay/decay_class.py
/tinyurl.com/jvzzcfn/decay/decay_class_oo.py



## References

] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in
Science and Engineering. Springer, third edition, 2012.