Generalizations of exponential decay models

Hans Petter Langtangen[1,2]

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

Feb 12, 2016

Contents

This chapter presents many mathematical models that all end up with ODEs of the type $u' = -au + b$. The applications are taken from biology, finance, and physics, and cover population growth or decay, interacting predator-prey populations, compound interest and inflation, radioactive decay, chemical and biochemical reaction, spreading of diseases, cooling of objects, compaction of geological media, pressure variations in the atmosphere, viscoelastic response in materials, and air resistance on falling or rising bodies.

Before we turn to the applications, however, we take a brief look at the technique of scaling, which is so useful in many applications.

1 Scaling

Real applications of a model $u' = -au + b$ will often involve a lot of parameters in the expressions for a and b. It can be quite a challenge to find relevant values of all parameters. In simple problems, however, it turns out that it is not always necessary to estimate all parameters because we can lump them into one or a few *dimensionless* numbers by using a very attractive technique called scaling. It

simply means to stretch the u and t axis in the present problem - and suddenly all parameters in the problem are lumped into one parameter if $b \neq 0$ and no parameter when $b = 0$!

1.1 Dimensionless variables

Scaling means that we introduce a new function $\bar{u}(\bar{t})$, with

$$\bar{u} = \frac{u - u_m}{u_c}, \quad \bar{t} = \frac{t}{t_c},$$

where u_m is a characteristic value of u, u_c is a characteristic size of the range of u values, and t_c is a characteristic size of the range of t where u shows significant variation. Choosing u_m, u_c, and t_c is not always easy and is often an art in complicated problems. We just state one choice first:

$$u_c = I, \quad u_m = b/a, \quad t_c = 1/a\,.$$

Inserting $u = u_m + u_c\bar{u}$ and $t = t_c\bar{t}$ in the problem $u' = -au + b$, assuming a and b are constants, results (after some algebra) in the *scaled problem*

$$\frac{d\bar{u}}{d\bar{t}} = -\bar{u}, \quad \bar{u}(0) = 1 - \beta,$$

where

$$\beta = \frac{b}{Ia}\,.$$

1.2 Dimensionless numbers

The parameter β is a dimensionless number. From the equation we see that b must have the same unit as the term au. The initial condition I must have the same unit as u, so Ia has the same unit as b, making the fraction $b/(Ia)$ dimensionless.

An important observation is that $\bar{u}$ depends on $\bar{t}$ and β. That is, only the special combination of $b/(Ia)$ matters, not what the individual values of b, a, and I are. The original unscaled function u depends on t, b, a, and I.

A second observation is striking: if $b = 0$, the scaled problem is independent of a and I! In practice this means that we can perform a single numerical simulation of the scaled problem and recover the solution of any problem for a given a and I by stretching the axis in the plot: $u = I\bar{u}$ and $t = \bar{t}/a$. For $b \neq 0$, we simulate the scaled problem for a few β values and recover the physical solution u by translating and stretching the u axis and stretching the t axis.

In general, scaling combines the parameters in a problem to a set of dimensionless parameters. The number of dimensionless parameters is usually much smaller than the number of original parameters. Section 11 presents an example where 11 parameters are reduced to one!

1.3 A scaling for vanishing initial condition

The scaling breaks down if $I = 0$. In that case we may choose $u_m = 0$, $u_c = b/a$, and $t_c = 1/b$, resulting in a slightly different scaled problem:

$$\frac{d\bar{u}}{d\bar{t}} = 1 - \bar{u}, \quad \bar{u}(0) = 0\,.$$

As with $b = 0$, the case $I = 0$ has a scaled problem with no physical parameters!

It is common to drop the bars after scaling and write the scaled problem as $u' = -u$, $u(0) = 1 - \beta$, or $u' = 1 - u$, $u(0) = 0$. Any implementation of the problem $u' = -au + b$, $u(0) = I$, can be reused for the scaled problem by setting $a = 1$, $b = 0$, and $I = 1 - \beta$ in the code, if $I \neq 0$, or one sets $a = 1$, $b = 1$, and $I = 0$ when the physical I is zero. Falling bodies in fluids, as described in Section 11, involves $u' = -au + b$ with seven physical parameters. All these vanish in the scaled version of the problem if we start the motion from rest!

Many more details about scaling are presented in the author's book *Scaling of Differential Equations* [1].

2 Evolution of a population

2.1 Exponential growth

Let N be the number of individuals in a population occupying some spatial domain. Despite N being an integer in this problem, we shall compute with N as a real number and view $N(t)$ as a continuous function of time. The basic model assumption is that in a time interval Δt the number of newcomers to the populations (newborns) is proportional to N, with proportionality constant $\bar{b}$. The amount of newcomers will increase the population and result in

$$N(t + \Delta t) = N(t) + \bar{b}N(t)\,.$$

It is obvious that a long time interval Δt will result in more newcomers and hence a larger $\bar{b}$. Therefore, we introduce $b = \bar{b}/\Delta t$: the number of newcomers per unit time and per individual. We must then multiply b by the length of the time interval considered and by the population size to get the total number of new individuals, $b\Delta t N$.

If the number of removals from the population (deaths) is also proportional to N, with proportionality constant $d\Delta t$, the population evolves according to

$$N(t + \Delta t) = N(t) + b\Delta t N(t) - d\Delta t N(t)\,.$$

Dividing by Δt and letting $\Delta t \to 0$, we get the ODE

$$N' = (b - d)N, \quad N(0) = N_0\,. \tag{1}$$

In a population where the death rate (d) is larger than then newborn rate (b), $b - d < 0$, and the population experiences exponential decay rather than exponential growth.

In some populations there is an immigration of individuals into the spatial domain. With I individuals coming in per time unit, the equation for the population change becomes

$$N(t+\Delta t) = N(t) + b\Delta t N(t) - d\Delta t N(t) + \Delta t I\,.$$

The corresponding ODE reads

$$N' = (b-d)N + I, \quad N(0) = N_0\,. \tag{2}$$

Emigration is also modeled by this I term if we just change its sign: $I < 0$. So, the I term models migration in and out of the domain in general.

Some simplification arises if we introduce a fractional measure of the population: $u = N/N_0$ and set $r = b - d$. The ODE problem now becomes

$$u' = ru + f, \quad u(0) = 1, \tag{3}$$

where $f = I/N_0$ measures the net immigration per time unit as the fraction of the initial population. Very often, r is approximately constant, but f is usually a function of time.

2.2 Logistic growth

The growth rate r of a population decreases if the environment has limited resources. Suppose the environment can sustain at most $N_{\max}$ individuals. We may then assume that the growth rate approaches zero as N approaches $N_{\max}$, i.e., as u approaches $M = N_{\max}/N_0$. The simplest possible evolution of r is then a linear function: $r(t) = \varrho(1 - u(t)/M)$, where ϱ is the initial growth rate when the population is small relative to the maximum size and there is enough resources. Using this $r(t)$ in (3) results in the *logistic model* for the evolution of a population (assuming for the moment that $f = 0$):

$$u' = \varrho(1 - u/M)u, \quad u(0) = 1\,. \tag{4}$$

Initially, u will grow at rate ϱ, but the growth will decay as u approaches M, and then there is no more change in u, causing $u \to M$ as $t \to \infty$. Note that the logistic equation $u' = \varrho(1 - u/M)u$ is *nonlinear* because of the quadratic term $-u^2\varrho/M$.

3 Compound interest and inflation

Say the annual interest rate is r percent and that the bank adds the interest once a year to your investment. If u^n is the investment in year n, the investment in year u^{n+1} grows to

$$u^{n+1} = u^n + \frac{r}{100}u^n\,.$$

In reality, the interest rate is added every day. We therefore introduce a parameter m for the number of periods per year when the interest is added. If n counts the periods, we have the fundamental model for compound interest:

$$u^{n+1} = u^n + \frac{r}{100m}u^n\,. \tag{5}$$

This model is a *difference equation*, but it can be transformed to a continuous differential equation through a limit process. The first step is to derive a formula for the growth of the investment over a time t. Starting with an investment u^0, and assuming that r is constant in time, we get

$$\begin{aligned} u^{n+1} &= \left(1 + \frac{r}{100m}\right) u^n \\ &= \left(1 + \frac{r}{100m}\right)^2 u^{n-1} \\ &\vdots \\ &= \left(1 + \frac{r}{100m}\right)^{n+1} u^0 \end{aligned}$$

Introducing time t, which here is a real-numbered counter for years, we have that $n = mt$, so we can write

$$u^{mt} = \left(1 + \frac{r}{100m}\right)^{mt} u^0\,.$$

The second step is to assume *continuous compounding*, meaning that the interest is added continuously. This implies $m \to \infty$, and in the limit one gets the formula

$$u(t) = u_0 e^{rt/100}, \tag{6}$$

which is nothing but the solution of the ODE problem

$$u' = \frac{r}{100}u, \quad u(0) = u_0\,. \tag{7}$$

This is then taken as the ODE model for compound interest if $r > 0$. However, the reasoning applies equally well to inflation, which is just the case $r < 0$. One may also take the r in (7) as the net growth of an investment, where r takes both compound interest and inflation into account. Note that for real applications we must use a time-dependent r in (7).

Introducing $a = \frac{r}{100}$, continuous inflation of an initial fortune I is then a process exhibiting exponential decay according to

$$u' = -au, \quad u(0) = I\,.$$

4 Newton's law of cooling

When a body at some temperature is placed in a cooling environment, experience shows that the temperature falls rapidly in the beginning, and then the

change in temperature levels off until the body's temperature equals that of the surroundings. Newton carried out some experiments on cooling hot iron and found that the temperature evolved as a "geometric progression at times in arithmetic progression", meaning that the temperature decayed exponentially. Later, this result was formulated as a differential equation: the rate of change of the temperature in a body is proportional to the temperature difference between the body and its surroundings. This statement is known as *Newton's law of cooling*, which mathematically can be expressed as

$$\frac{dT}{dt} = -k(T - T_s), \tag{8}$$

where T is the temperature of the body, T_s is the temperature of the surroundings (which may be time-dependent), t is time, and k is a positive constant. Equation (8) is primarily viewed as an empirical law, valid when heat is efficiently convected away from the surface of the body by a flowing fluid such as air at constant temperature T_s. The *heat transfer coefficient* k reflects the transfer of heat from the body to the surroundings and must be determined from physical experiments.

The cooling law (8) needs an initial condition $T(0) = T_0$.

5 Radioactive decay

An atomic nucleus of an unstable atom may lose energy by emitting ionizing particles and thereby be transformed to a nucleus with a different number of protons and neutrons. This process is known as radioactive decay[1]. Actually, the process is stochastic when viewed for a single atom, because it is impossible to predict exactly when a particular atom emits a particle. Nevertheless, with a large number of atoms, N, one may view the process as deterministic and compute the mean behavior of the decay. Below we reason intuitively about an ODE for the mean behavior. Thereafter, we show mathematically that a detailed stochastic model for single atoms leads to the same mean behavior.

5.1 Deterministic model

Suppose at time t, the number of the original atom type is $N(t)$. A basic model assumption is that the transformation of the atoms of the original type in a small time interval Δt is proportional to N, so that

$$N(t + \Delta t) = N(t) - a\Delta t N(t),$$

where $a > 0$ is a constant. The proportionality factor is $a\Delta t$, i.e., proportional to Δt since a longer time interval will produce more transformations. We can introduce $u = N(t)/N(0)$, divide by Δt, and let $\Delta t \to 0$:

$$\lim_{r \to 0} N_0 \frac{u(t + \Delta t) - u(t)}{\Delta t} = -aN_0 u(t)\,.$$

[1] `http://en.wikipedia.org/wiki/Radioactive_decay`

The left-hand side is the derivative of u. Dividing by the N_0 gives the following ODE for u:

$$u' = -au, \quad u(0) = 1\,. \tag{9}$$

The parameter a can for a given nucleus be expressed through the *half-life* $t_{1/2}$, which is the time taken for the decay to reduce the initial amount by one half, i.e., $u(t_{1/2}) = 0.5$. With $u(t) = e^{-at}$, we get $t_{1/2} = a^{-1}\ln 2$ or $a = \ln 2/t_{1/2}$.

5.2 Stochastic model

Originally, we have N_0 atoms. Up to some particular time t, each atom may either have decayed or not. If not, they have "survived". We want to count how many original atoms that have survived. The survival of a single atom at time t is a random event. Since there are only two outcomes, survival or decay, we have a Bernoulli trial[2]. Let p be the probability of survival (implying that the probability of decay is $1-p$). If each atom survives independently of the others, and the probability of survival is the same for every atom, we have N_0 Bernoulli trials, known as a *binomial experiment* from probability theory. The probability $P(N)$ that N out of the N_0 atoms have survived at time t is then given by the famous *binomial distribution*

$$P(N) = \frac{N_0!}{N!(N_0-N)!}p^N(1-p)^{N_0-N}\,.$$

The mean (or expected) value $\mathrm{E}[P]$ of $P(N)$ is known to be N_0p.

It remains to estimate p. Let the interval $[0,t]$ be divided into m small subintervals of length Δt. We make the assumption that the probability of decay of a single atom in an interval of length Δt is $\tilde{p}$, and that this probability is proportional to Δt: $\tilde{p} = \lambda\Delta t$ (it sounds natural that the probability of decay increases with Δt). The corresponding probability of survival is $1-\lambda\Delta t$. Believing that λ is independent of time, we have, for each interval of length Δt, a Bernoulli trial: the atom either survives or decays in that interval. Now, p should be the probability that the atom survives in all the intervals, i.e., that we have m successful Bernoulli trials in a row and therefore

$$p = (1-\lambda\Delta t)^m\,.$$

The expected number of atoms of the original type at time t is

$$\mathrm{E}[P] = N_0p = N_0(1-\lambda\Delta t)^m, \quad m = t/\Delta t\,. \tag{10}$$

To see the relation between the two types of Bernoulli trials and the ODE above, we go to the limit $\Delta t \to 0$, $m \to \infty$. It is possible to show that

$$p = \lim_{m\to\infty}(1-\lambda\Delta t)^m = \lim_{m\to\infty}\left(1-\lambda\frac{t}{m}\right)^m = e^{-\lambda t}$$

[2] `http://en.wikipedia.org/wiki/Bernoulli_trial`

This is the famous exponential waiting time (or arrival time) distribution for a Poisson process in probability theory (obtained here, as often done, as the limit of a binomial experiment). The probability of decay, or more precisely that at least one atom has undergone a transition, is $1 - p = 1 - e^{-\lambda t}$. This is the exponential distribution[3]. The limit means that m is very large, hence Δt is very small, and $\tilde{p} = \lambda \Delta t$ is very small since the intensity of the events, λ, is assumed finite. This situation corresponds to a very small probability that an atom will decay in a very short time interval, which is a reasonable model. The same model occurs in lots of different applications, e.g., when waiting for a taxi, or when finding defects along a rope.

5.3 Relation between stochastic and deterministic models

With $p = e^{-\lambda t}$ we get the expected number of original atoms at t as $N_0 p = N_0 e^{-\lambda t}$, which is exactly the solution of the ODE model $N' = -\lambda N$. This also gives an interpretation of a via λ or vice versa. Our important finding here is that the ODE model captures the mean behavior of the underlying stochastic model. This is, however, not always the common relation between microscopic stochastic models and macroscopic "averaged" models.

Also of interest, is that a Forward Euler discretization of $N' = -\lambda N$, $N(0) = N_0$, gives $N^m = N_0(1 - \lambda \Delta t)^m$ at time $t_m = m\Delta t$, which is exactly the expected value of the stochastic experiment with N_0 atoms and m small intervals of length Δt, where each atom can decay with probability $\lambda \Delta t$ in an interval.

A fundamental question is how accurate the ODE model is. The underlying stochastic model fluctuates around its expected value. A measure of the fluctuations is the standard deviation of the binomial experiment with N_0 atoms, which can be shown to be $\mathrm{Std}[P] = \sqrt{N_0 p(1-p)}$. Compared to the size of the expectation, we get the normalized standard deviation

$$\frac{\sqrt{\mathrm{Var}[P]}}{\mathrm{E}[P]} = N_0^{-1/2}\sqrt{p^{-1} - 1} = N_0^{-1/2}\sqrt{(1 - e^{-\lambda t})^{-1} - 1} \approx (N_0 \lambda t)^{-1/2},$$

showing that the normalized fluctuations are very small if N_0 is very large, which is usually the case.

5.4 Generalization of the radioactive decay modeling

The modeling in Section 5 is in fact very general, despite a focus on a particular physical process. We may instead of atoms and decay speak about a set of *items*, where each item can undergo a stochastic *transition* from one state to another. In Section 6 the item is a molecule and the transition is a chemical reaction, while in Section 7 the item is an ill person and the transition is recovering from the illness (or an immune person who loses her immunity).

[3] `http://en.wikipedia.org/wiki/Exponential_distribution`

From the modeling in Section 5 we can establish a deterministic model for a large number of items and a stochastic model for an arbitrary number of items, even a single one. The stochastic model has a parameter λ reflecting the probability that a transition takes place in a time interval of unit length (or equivalently, that the probability is $\lambda\Delta t$ for a transition during a time interval of length Δt). The probability of making a transition before time t is given by

$$F(t) = 1 - e^{-\lambda t}\,.$$

The corresponding probability density is $f(t) = F'(t) = e^{-\lambda t}$. The expected value of $F(t)$, i.e., the expected time to transition, is λ^{-1}. This interpretation of λ makes it easy to measure its value: just carry out a large number of experiments, measure the time to transition, and take one over the average of these times as an estimate of λ. The variance is λ^{-2}.

The deterministic model counts how many items, $N(t)$, that have undergone the transition (on average), and $N(t)$ is governed by the ODE

$$N' = -\lambda N(t), \quad N(0) = N_0\,.$$

6 Chemical kinetics

6.1 Irreversible reaction of two substances

Consider two chemical substances, A and B, and a chemical reaction that turns A into B. In a small time interval, some of the molecules of type A are transformed into molecules of B. This process is, from a mathematical modeling point of view, equivalent to the radioactive decay process described in the previous section. We can therefore apply the same modeling approach. If N_A is the number of molecules of substance A, we have that N_A is governed by the differential equation

$$\frac{dN_A}{dt} = -kN_A,$$

where (the constant) k is called the *rate constant* of the reaction. Rather than using the number of molecules, we use the *concentration* of molecules: $[A](t) = N_A(t)/N_A(0)$. We see that $d[A]/dt = N_A(0)^{-1}dN_A/dt$. Replacing N_A by $[A]$ in the equation for N_A leads to the equation for the concentration $[A]$:

$$\frac{d[A]}{dt} = -k[A], \quad t \in (0,T],\ [A](0) = 1,\ . \tag{11}$$

Since substance A is transformed to substance B, we have that the concentration of $[B]$ grows by the loss of $[A]$:

$$\frac{d[B]}{dt} = k[A], \quad [B](0) = 0\,.$$

The mathematical model can either be (11) or the system

$$\frac{d[A]}{dt} = -k[A], \qquad t \in (0,T] \tag{12}$$
$$\frac{d[B]}{dt} = k[A], \qquad t \in (0,T] \tag{13}$$
$$[A](0) = 1, \tag{14}$$
$$[B](0) = 0\,. \tag{15}$$

This reaction is known as a *first-order reaction*, where each molecule of A makes an independent decision about whether to complete the reaction, i.e., independent of what happens to any other molecule.

An n-th order reaction is modeled by

$$\frac{d[A]}{dt} = -k[A]^n, \tag{16}$$
$$\frac{d[B]}{dt} = k[A]^n, \tag{17}$$

for $t \in (0,T]$ with initial conditions $[A](0) = 1$ and $[B](0) = 0$. Here, n can be a real number, but is most often an integer. Note that the sum of the concentrations is constant since

$$\frac{d[A]}{dt} + \frac{d[B]}{dt} = 0 \quad \Rightarrow \quad [A](t) + [B](t) = \text{const} = [A](0) + [B](0) = 1 + 0\,.$$

6.2 Reversible reaction of two substances

Let the chemical reaction turn substance A into B and substance B into A. The rate of change of $[A]$ has then two contributions: a loss $k_A[A]$ and a gain $k_B[B]$:

$$\frac{d[A]}{dt} = -k_A[A] + k_B[B], \quad t \in (0,T],\ [A](0) = A_0\,. \tag{18}$$

Similarly for substance B,

$$\frac{d[B]}{dt} = k_A[A] - k_B[B], \quad t \in (0,T],\ [B](0) = B_0\,. \tag{19}$$

This time we have allowed for arbitrary initial concentrations. Again,

$$\frac{d[A]}{dt} + \frac{d[B]}{dt} = 0 \quad \Rightarrow \quad [A](t) + [B](t) = A_0 + B_0\,.$$

6.3 Irreversible reaction of two substances into a third

Now we consider two chemical substances, A and B, reacting with each other and producing a substance C. In a small time interval Δt, molecules of type A and B are occasionally colliding, and in some of the collisions, a chemical reaction occurs, which turns A and B into a molecule of type C. (More generally, M_A molecules of A and M_B molecules of B react to form M_C molecules of C.) The number of possible pairings, and thereby collisions, of A and B is $N_A N_B$, where N_A is the number of molecules of A, and N_B is the number of molecules of B. A fraction k of these collisions, $\hat{k}\Delta t N_A N_B$, features a chemical reaction and produce N_C molecules of C. The fraction is thought to be proportional to Δt: considering a twice as long time interval, twice as many molecules collide, and twice as many reactions occur. The increase in molecules of substance C is now found from the reasoning

$$N_C(t + \Delta t) = N_C(t) + \hat{k}\Delta t N_A N_B \,.$$

Dividing by Δt,

$$\frac{N_C(t + \Delta t) - N_C(t)}{\Delta t} = \hat{k} N_A N_B,$$

and letting $\Delta t \to 0$, gives the differential equation

$$\frac{dN_C}{dt} = \hat{k} N_A N_B \,.$$

(This equation is known as the important law of mass action[4] discovered by the Norwegian scientists Cato M. Guldberg and Peter Waage. A more general form of the right-hand side is $\hat{k} N_A^\alpha N_B^\beta$. All the constants $\hat{k}$, α, and β must be determined from experiments.)

Working instead with concentrations, we introduce $[C](t) = N_C(t)/N_C(0)$, with similar definitions for $[A]$ and $[B]$ we get

$$\frac{d[C]}{dt} = k[A][B] \,. \tag{20}$$

The constant k is related to $\hat{k}$ by $k = \hat{k} N_A(0) N_B(0)/N_C(0)$. The gain in C is a loss of A and B:

$$\frac{d[A]}{dt} = -k[A][B], \tag{21}$$

$$\frac{d[B]}{dt} = -k[A][B] \,. \tag{22}$$

[4] `https://en.wikipedia.org/wiki/Law_of_mass_action`

6.4 A biochemical reaction

A common reaction (known as Michaelis-Menton kinetics[5]) turns a substrate S into a product P with aid of an enzyme E. The reaction is a two-stage process: first S and E reacts to form a complex ES, where the enzyme and substrate are bound to each other, and then ES is turned into E and P. In the first stage, S and E react to produce a growth of ES according to the law of mass action:

$$\frac{d[S]}{dt} = -k_+[E][S],$$
$$\frac{d[ES]}{dt} = k_+[E][S]\,.$$

The complex ES reacts and produces the product P at rate $-k_v[ES]$ and E at rate $-k_-[ES]$. The total set of reactions can then be expressed by

$$\frac{d[ES]}{dt} = k_+[E][S] - k_v[ES] - k_-[ES], \tag{23}$$
$$\frac{d[P]}{dt} = k_v[ES], \tag{24}$$
$$\frac{d[S]}{dt} = -k_+[E][S] + k_-[ES], \tag{25}$$
$$\frac{d[E]}{dt} = -k_+[E][S] + k_-[ES] + k_v[ES]\,. \tag{26}$$

The initial conditions are $[ES](0) = [P](0) = 0$, and $[S] = S_0$, $[E] = E_0$. The constants k_+, k_-, and k_v must be determined from experiments.

7 Spreading of diseases

The modeling of spreading of diseases is very similar to the modeling of chemical reactions in Section 6. The field of epidemiology speaks about susceptibles: people who can get a disease; infectives: people who are infected and can infect susceptibles; and recovered: people who have recovered from the disease and become immune. Three categories are accordingly defined: S for susceptibles, I for infectives, and R for recovered. The number in each category is tracked by the functions $S(t)$, $I(t)$, and $R(t)$.

To model how many people that get infected in a small time interval Δt, we reason as with reactions in Section 6. The possible number of pairings ("collisions") between susceptibles and infected is SI. A fraction of these, $\beta\Delta tSI$, will actually meet and the infected succeed in infecting the susceptible, where β is a parameter to be empirically estimated. This leads to a loss of susceptibles and a gain of infected:

[5] `https://en.wikipedia.org/wiki/Michaelis-Menten_kinetics`

$$S(t+\Delta t) = S(t) - \beta \Delta t S I,$$
$$I(t+\Delta t) = I(t) + \beta \Delta t S I\,.$$

In the same time interval, a fraction $\nu \Delta t I$ of the infected is recovered. It follows from Section 5.4 that the parameter ν^{-1} is interpreted as the average waiting time to leave the I category, i.e., the average length of the disease. The $\nu \Delta t I$ term is a loss for the I category, but a gain for the R category:

$$I(t+\Delta t) = I(t) + \beta \Delta t S I - \nu \Delta t I, R(t+\Delta t) \qquad = R(t) + \nu \Delta t I\,.$$

Dividing these equations by Δt and going to the limit $\Delta t \to 0$, gives the ODE system

$$\frac{dS}{dt} = -\beta S I, \tag{27}$$
$$\frac{dI}{dt} = \beta S I - \nu I, \tag{28}$$
$$\frac{dR}{dt} = \nu I, \tag{29}$$

with initial values $S(0) = S_0$, $I(0) = I_0$, and $R(0) = 0$. By adding the equations, we realize that

$$\frac{dS}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0 \quad \Rightarrow \quad S + I + R = \text{const} = N,$$

where N is the total number in the population under consideration. This property can be used as a partial verification during simulations.

Equations (27)-(29) are known as the SIR model in epidemiology. The model can easily be extended to incorporate vaccination programs, immunity loss after some time, etc. Typical diseases that can be simulated by the SIR model and its variants are measles, smallpox, flu, plague, and HIV.

8 Predator-prey models in ecology

A model for the interaction of predator and prey species can be based on reasoning from population dynamics and the SIR model. Let $H(t)$ be the number of preys in a region, and let $L(t)$ be the number of predators. For example, H may be hares and L lynx, or rabbits and foxes.

The population of the prey evolves due to births and deaths, exactly as in a population dynamics model from Section 2.1. During a time interval Δt the increase in the population is therefore

$$H(t+\Delta t) - H(t) = a \Delta t H(t),$$

where a is a parameter to be measured from data. The increase is proportional to H, and the proportionality constant $a\Delta t$ is proportional to Δt, because doubling the interval will double the increase.

However, the prey population has an additional loss because they are eaten by predators. All the prey and predator animals can form LH pairs in total (assuming all individuals meet randomly). A small fraction $b\Delta t$ of such meetings, during a time interval Δt, ends up with the predator eating the prey. The increase in the prey population is therefore adjusted to

$$H(t+\Delta t) - H(t) = a\Delta t H(t) - b\Delta t H(t)L(t)\,.$$

The predator population increases as a result of eating preys. The amount of eaten preys is $b\Delta t LH$, but only a fraction $d\Delta t LH$ of this amount contributes to increasing the predator population. In addition, predators die and this loss is set to $c\Delta t L$. To summarize, the increase in the predator population is given by

$$L(t+\Delta t) - L(t) = d\Delta t L(t)H(t) - c\Delta t L(t)\,.$$

Dividing by Δt in the equations for H and L and letting $t \to 0$ results in

$$\lim_{\Delta t\to 0} \frac{H(t+\Delta t) - H(t)}{\Delta t} = H'(t) = aH(t) - bL(t)H(t),$$
$$\lim_{\Delta t\to 0} \frac{L(t+\Delta t) - L(t)}{\Delta t} = L'(t) = dL(t)H(t) - cL(t)\,.$$

We can simplify the notation to the following two ODEs:

$$H' = H(a - bL), \tag{30}$$
$$L' = L(dH - c)\,. \tag{31}$$

This is a so-called Lokta-Volterra model. It contains four parameters that must be estimated from data: a, b, c, and d. In addition, two initial conditions are needed for $H(0)$ and $L(0)$.

9 Decay of atmospheric pressure with altitude

9.1 The general model

Vertical equilibrium of air in the atmosphere is governed by the equation

$$\frac{dp}{dz} = -\varrho g\,. \tag{32}$$

Here, $p(z)$ is the air pressure, ϱ is the density of air, and $g = 9.807$ m/s^2 is a standard value of the acceleration of gravity. (Equation (32) follows directly from the general Navier-Stokes equations for fluid motion, with the assumption that the air does not move.)

The pressure is related to density and temperature through the ideal gas law

$$\varrho = \frac{Mp}{R^*T}, \tag{33}$$

where M is the molar mass of the Earth's air (0.029 kg/mol), R^* is the universal gas constant (8.314 Nm/(mol K)), and T is the temperature in Kelvin. All variables p, ϱ, and T vary with the height z. Inserting (33) in (32) results in an ODE with a variable coefficient:

$$\frac{dp}{dz} = -\frac{Mg}{R^*T(z)}p\,. \tag{34}$$

9.2 Multiple atmospheric layers

The atmosphere can be approximately modeled by seven layers. In each layer, (34) is applied with a linear temperature of the form

$$T(z) = \bar{T}_i + L_i(z - h_i),$$

where $z = h_i$ denotes the bottom of layer number i, having temperature $\bar{T}_i$, and L_i is a constant in layer number i. The table below lists h_i (m), $\bar{T}_i$ (K), and L_i (K/m) for the layers $i = 0, \ldots, 6$.

i	h_i	$\bar{T}_i$	L_i
0	0	288	-0.0065
1	11,000	216	0.0
2	20,000	216	0.001
3	32,000	228	0.0028
4	47,000	270	0.0
5	51,000	270	-0.0028
6	71,000	214	-0.002

For implementation it might be convenient to write (34) on the form

$$\frac{dp}{dz} = -\frac{Mg}{R^*(\bar{T}(z) + L(z)(z - h(z)))}p, \tag{35}$$

where $\bar{T}(z)$, $L(z)$, and $h(z)$ are piecewise constant functions with values given in the table. The value of the pressure at the sea level $z = 0$, $p_0 = p(0)$, is 101325 Pa.

9.3 Simplifications

Constant layer temperature. One common simplification is to assume that the temperature is constant within each layer. This means that $L = 0$.

One-layer model. Another commonly used approximation is to work with one layer instead of seven. This one-layer model[6] is based on $T(z) = T_0 - Lz$, with sea level standard temperature $T_0 = 288$ K and temperature lapse rate $L = 0.0065$ K/m.

10 Compaction of sediments

Sediments, originally made from materials like sand and mud, get compacted through geological time by the weight of new material that is deposited on the sea bottom. The porosity ϕ of the sediments tells how much void (fluid) space there is between the sand and mud grains. The porosity drops with depth, due to the weight of the sediments above. This makes the void space shrink, and thereby compaction increases.

A typical assumption is that the change in ϕ at some depth z is negatively proportional to ϕ. This assumption leads to the differential equation problem

$$\frac{d\phi}{dz} = -c\phi, \quad \phi(0) = \phi_0, \tag{36}$$

where the z axis points downwards, $z = 0$ is the surface with known porosity, and $c > 0$ is a constant.

The upper part of the Earth's crust consists of many geological layers stacked on top of each other, as indicated in Figure 1. The model (36) can be applied for each layer. In layer number i, we have the unknown porosity function $\phi_i(z)$ fulfilling $\phi_i'(z) = -c_i z$, since the constant c in the model (36) depends on the type of sediment in the layer. Alternatively, we can use (36) to describe the porosity through all layers if c is taken as a piecewise constant function of z, equal to c_i in layer i. From the figure we see that new layers of sediments are deposited on top of older ones as time progresses. The compaction, as measured by ϕ, is rapid in the beginning and then decreases (exponentially) with depth in each layer.

When we drill a well at present time through the right-most column of sediments in Figure 1, we can measure the thickness of the sediment in (say) the bottom layer. Let L_1 be this thickness. Assuming that the volume of sediment remains constant through time, we have that the initial volume, $\int_0^{L_{1,0}} \phi_1 dz$, must equal the volume seen today, $\int_{\ell-L_1}^{\ell} \phi_1 dz$, where ℓ is the depth of the bottom of the sediment in the present day configuration. After having solved for ϕ_1 as a function of z, we can then find the original thickness $L_{1,0}$ of the sediment from the equation

$$\int_0^{L_{1,0}} \phi_1 dz = \int_{\ell-L_1}^{\ell} \phi_1 dz\,.$$

In hydrocarbon exploration it is important to know $L_{1,0}$ and the compaction history of the various layers of sediments.

[6] `http://en.wikipedia.org/wiki/Density_of_air`

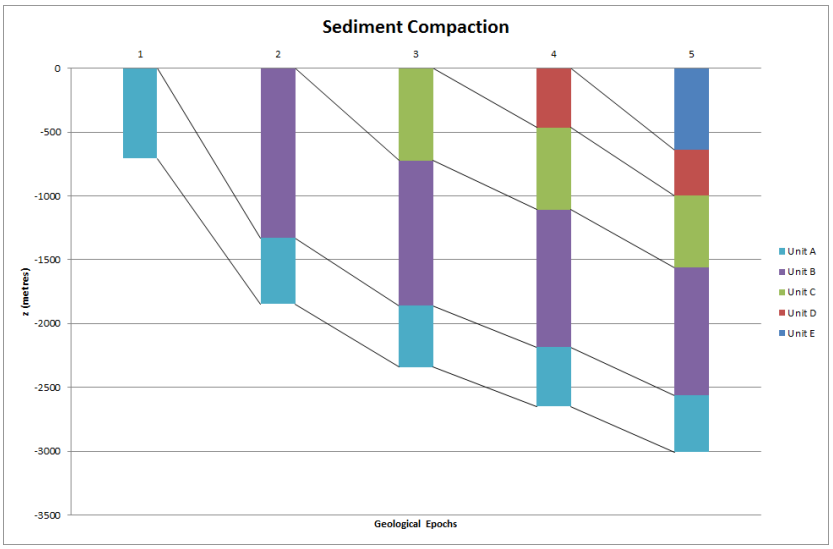


Figure 1: Illustration of the compaction of geological layers (with different colors) through time.

11 Vertical motion of a body in a viscous fluid

A body moving vertically through a fluid (liquid or gas) is subject to three different types of forces: the gravity force, the drag force[7], and the buoyancy force.

11.1 Overview of forces

Taking the upward direction as positive, the gravity force is $F_g = -mg$, where m is the mass of the body and g is the acceleration of gravity. The uplift or buoyancy force ("Archimedes force") is $F_b = \varrho g V$, where ϱ is the density of the fluid and V is the volume of the body.

The drag force is of two types, depending on the Reynolds number

$$\mathrm{Re} = \frac{\varrho d|v|}{\mu}, \tag{37}$$

where d is the diameter of the body in the direction perpendicular to the flow, v is the velocity of the body, and μ is the dynamic viscosity of the fluid. When $\mathrm{Re} < 1$, the drag force is fairly well modeled by the so-called Stokes' drag, which for a spherical body of diameter d reads

$$F_d^{(S)} = -3\pi d\mu v\,. \tag{38}$$

[7] `http://en.wikipedia.org/wiki/Drag_(physics)`

Quantities are taken as positive in the upwards vertical direction, so if $v > 0$ and the body moves upwards, the drag force acts downwards and become negative, in accordance with the minus sign in expression for $F_d^{(S)}$.

For large Re, typically $\mathrm{Re} > 10^3$, the drag force is quadratic in the velocity:

$$F_d^{(q)} = -\frac{1}{2} C_D \varrho A |v| v, \tag{39}$$

where C_D is a dimensionless drag coefficient depending on the body's shape, and A is the cross-sectional area as produced by a cut plane, perpendicular to the motion, through the thickest part of the body. The superscripts ^{q} and ^{S} in $F_d^{(S)}$ and $F_d^{(q)}$ indicate Stokes drag and quadratic drag, respectively.

11.2 Equation of motion

All the mentioned forces act in the vertical direction. Newton's second law of motion applied to the body says that the sum of these forces must equal the mass of the body times its acceleration a in the vertical direction.

$$ma = F_g + F_d^{(S)} + F_b\,.$$

Here we have chosen to model the fluid resistance by the Stokes drag. Inserting the expressions for the forces yields

$$ma = -mg - 3\pi d\mu v + \varrho g V\,.$$

The unknowns here are v and a, i.e., we have two unknowns but only one equation. From kinematics in physics we know that the acceleration is the time derivative of the velocity: $a = dv/dt$. This is our second equation. We can easily eliminate a and get a single differential equation for v:

$$m\frac{dv}{dt} = -mg - 3\pi d\mu v + \varrho g V\,.$$

A small rewrite of this equation is handy: We express m as $\varrho_b V$, where ϱ_b is the density of the body, and we divide by the mass to get

$$v'(t) = -\frac{3\pi d\mu}{\varrho_b V} v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,. \tag{40}$$

We may introduce the constants

$$a = \frac{3\pi d\mu}{\varrho_b V}, \quad b = g\left(\frac{\varrho}{\varrho_b} - 1\right), \tag{41}$$

so that the structure of the differential equation becomes obvious:

$$v'(t) = -av(t) + b\,. \tag{42}$$

The corresponding initial condition is $v(0) = v_0$ for some prescribed starting velocity v_0.

This derivation can be repeated with the quadratic drag force $F_d^{(q)}$, leading to the result

$$v'(t) = -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v + g\left(\frac{\varrho}{\varrho_b} - 1\right). \tag{43}$$

Defining

$$a = \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}, \tag{44}$$

and b as above, we can write (43) as

$$v'(t) = -a|v|v + b\,. \tag{45}$$

11.3 Terminal velocity

An interesting aspect of (42) and (45) is whether v will approach a final constant value, the so-called *terminal velocity* v_T, as $t \to \infty$. A constant v means that $v'(t) \to 0$ as $t \to \infty$ and therefore the terminal velocity v_T solves

$$0 = -av_T + b$$

and

$$0 = -a|v_T|v_T + b\,.$$

The former equation implies $v_T = b/a$, while the latter has solutions $v_T = -\sqrt{|b|/a}$ for a falling body ($v_T < 0$) and $v_T = \sqrt{b/a}$ for a rising body ($v_T > 0$).

11.4 A Crank-Nicolson scheme

Both governing equations, the Stokes' drag model (42) and the quadratic drag model (45), can be readily solved by the Forward Euler scheme. For higher accuracy one can use the Crank-Nicolson method, but a straightforward application of this method gives a nonlinear equation in the new unknown value v^{n+1} when applied to (45):

$$\frac{v^{n+1} - v^n}{\Delta t} = -a\frac{1}{2}(|v^{n+1}|v^{n+1} + |v^n|v^n) + b\,. \tag{46}$$

The first term on the right-hand side of (46) is the arithmetic average of $-|v|v$ evaluated at time levels n and $n+1$.

Instead of approximating the term $-|v|v$ by an arithmetic average, we can use a *geometric mean*:

$$(|v|v)^{n+\frac{1}{2}} \approx |v^n|v^{n+1}\,. \tag{47}$$

The error is of second order in Δt, just as for the arithmetic average and the centered finite difference approximation in (46). With the geometric mean, the resulting discrete equation

$$\frac{v^{n+1}-v^n}{\Delta t} = -a|v^n|v^{n+1} + b$$

becomes a *linear* equation in v^{n+1}, and we can therefore easily solve for v^{n+1}:

$$v^{n+1} = \frac{v_n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}}|v^n|}\,. \tag{48}$$

Using a geometric mean instead of an arithmetic mean in the Crank-Nicolson scheme is an attractive method for avoiding a nonlinear algebraic equation when discretizing a nonlinear ODE.

11.5 Physical data

Suitable values of μ are $1.8 \cdot 10^{-5}$ Pa s for air and $8.9 \cdot 10^{-4}$ Pa s for water. Densities can be taken as 1.2 kg/m^3 for air and as $1.0 \cdot 10^3$ kg/m^3 for water. For considerable vertical displacement in the atmosphere one should take into account that the density of air varies with the altitude, see Section 9. One possible density variation arises from the one-layer model in the mentioned section.

Any density variation makes b time dependent and we need $b^{n+\frac{1}{2}}$ in (48). To compute the density that enters $b^{n+\frac{1}{2}}$ we must also compute the vertical position $z(t)$ of the body. Since $v = dz/dt$, we can use a centered difference approximation:

$$\frac{z^{n+\frac{1}{2}} - z^{n-\frac{1}{2}}}{\Delta t} = v^n \quad \Rightarrow \quad z^{n+\frac{1}{2}} = z^{n-\frac{1}{2}} + \Delta t\, v^n\,.$$

This $z^{n+\frac{1}{2}}$ is used in the expression for b to compute $\varrho(z^{n+\frac{1}{2}})$ and then $b^{n+\frac{1}{2}}$.

The drag coefficient[8] C_D depends heavily on the shape of the body. Some values are: 0.45 for a sphere, 0.42 for a semi-sphere, 1.05 for a cube, 0.82 for a long cylinder (when the center axis is in the vertical direction), 0.75 for a rocket, 1.0-1.3 for a man in upright position, 1.3 for a flat plate perpendicular to the flow, and 0.04 for a streamlined, droplet-like body.

11.6 Verification

To verify the program, one may assume a heavy body in air such that the F_b force can be neglected, and further assume a small velocity such that the air resistance F_d can also be neglected. This can be obtained by setting μ and ϱ to zero. The motion then leads to the velocity $v(t) = v_0 - gt$, which is linear in t and therefore should be reproduced to machine precision (say tolerance 10^{-15}) by any implementation based on the Crank-Nicolson or Forward Euler schemes.

Another verification, but not as powerful as the one above, can be based on computing the terminal velocity and comparing with the exact expressions. The advantage of this verification is that we can also test the situation $\varrho \neq 0$.

[8] `http://en.wikipedia.org/wiki/Drag_coefficient`

As always, the method of manufactured solutions can be applied to test the implementation of all terms in the governing equation, but then the solution has no physical relevance in general.

11.7 Scaling

Applying scaling, as described in Section 1, will for the linear case reduce the need to estimate values for seven parameters down to choosing one value of a single dimensionless parameter

$$\beta = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d \mu I},$$

provided $I \neq 0$. If the motion starts from rest, $I = 0$, the scaled problem reads

$$\bar{u}' = 1 - \bar{u}, \quad \bar{u}(0) = 0,$$

and there is no need for estimating physical parameters (!). This means that there is a single universal solution to the problem of a falling body starting from rest: $\bar{u}(t) = 1 - e^{-\bar{t}}$. All real physical cases correspond to stretching the $\bar{t}$ axis and the $\bar{u}$ axis in this dimensionless solution. More precisely, the physical velocity $u(t)$ is related to the dimensionless velocity $\bar{u}(\bar{t})$ through

$$u = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d \mu} \bar{u}(t/(g(\varrho/\varrho_b - 1))) = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d \mu} (1 - e^{t/(g(\varrho/\varrho_b - 1))}) .$$

12 Viscoelastic materials

When stretching a rod made of a perfectly elastic material, the elongation (change in the rod's length) is proportional to the magnitude of the applied force. Mathematical models for material behavior under application of external forces use *strain* ε and *stress* σ instead of elongation and forces. Strain is relative change in elongation and stress is force per unit area. An elastic material has a linear relation between stress and strain: $\sigma = E\varepsilon$. This is a good model for many materials, but frequently the velocity of the deformation (or more precisely, the strain rate ε') also influences the stress. This is particularly the case for materials like organic polymers, rubber, and wood. When the stress depends on both the strain and the strain rate, the material is said to be viscoelastic. A common model relating forces to deformation is the Kelvin-Voigt model[9]:

$$\sigma(t) = E\varepsilon(t) + \eta\varepsilon'(t) . \tag{49}$$

Compared to a perfectly elastic material, which deforms instantaneously when a force is acting, a Kelvin-Voigt material will spend some time to elongate. For

[9] `https://en.wikipedia.org/wiki/Kelvin-Voigt_material`

example, when an elastic rod is subject to a constant force $\boldsymbol{\sigma}$ at $t = 0$, the strain immediately adjusts to $\varepsilon = \sigma/E$. A Kelvin-Voigt material, however, has a response $\varepsilon(t) = \frac{\sigma}{E}(1 - e^{Et/\eta})$. Removing the force when the strain is $\varepsilon(t_1) = I$ will for an elastic material immediately bring the strain back to zero, while a Kelvin-Voigt material will decay: $\varepsilon = Ie^{-(t-t_1)E/\eta)}$.

Introducing u for ε and treating $\boldsymbol{\sigma}(t)$ as a given function, we can write the Kelvin-Voigt model in our standard form

$$u'(t) = -au(t) + b(t), \tag{50}$$

with $a = E/\eta$ and $b(t) = \boldsymbol{\sigma}(t)/\eta$. An initial condition, usually $u(0) = 0$, is needed.

13 Decay ODEs from solving a PDE by Fourier expansions

Suppose we have a partial differential equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x,t),$$

with boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = I(x)$. One may express the solution as

$$u(x,t) = \sum_{k=1}^{m} A_k(t) e^{ikx\pi/L},$$

for appropriate unknown functions A_k, $k = 1, \ldots, m$. We use the complex exponential $e^{ikx\pi/L}$ for easy algebra, but the physical u is taken as the real part of any complex expression. Note that the expansion in terms of $e^{ikx\pi/L}$ is compatible with the boundary conditions: all functions $e^{ikx\pi/L}$ vanish for $x = 0$ and $x = L$. Suppose we can express $I(x)$ as

$$I(x) = \sum_{k=1}^{m} I_k e^{ikx\pi/L}.$$

Such an expansion can be computed by well-known Fourier expansion techniques, but those details are not important here. Also, suppose we can express the given $f(x,t)$ as

$$f(x,t) = \sum_{k=1}^{m} b_k(t) e^{ikx\pi/L}.$$

Inserting the expansions for u and f in the differential equations demands that all terms corresponding to a given k must be equal. The calculations result in the follow system of ODEs:

$$A_k'(t) = -\alpha \frac{k^2\pi^2}{L^2} + b_k(t), \quad k = 1, \ldots, m\,.$$

From the initial condition

$$u(x,0) = \sum_k A_k(0) e^{ikx\pi/L} = I(x) = \sum_k I_k e^{(ikx\pi/L)},$$

so it follows that $A_k(0) = I_k$, $k = 1, \ldots, m$. We then have m equations of the form $A_k' = -aA_k + b$, $A_k(0) = I_k$, for appropriate definitions of a and b. These ODE problems are independent of each other such that we can solve one problem at a time. The outlined technique is a quite common solution approach to partial differential equations.

Remark. Since a_k depends on k and the stability of the Forward Euler scheme demands $a_k \Delta t \leq 1$, we get that $\Delta t \leq \alpha^{-1} L^2 \pi^{-2} k^{-2}$ for this scheme. Usually, quite large k values are needed to accurately represent the given functions I and f so that Δt in the Forward Euler scheme needs to be very small for these large values of k. Therefore, the Crank-Nicolson and Backward Euler schemes, which allow larger Δt without any growth in the solutions, are more popular choices when creating time-stepping algorithms for partial differential equations of the type considered in this example.

14 Exercises

Problem 1: Radioactive decay of Carbon-14

The Carbon-14[10] isotope, whose radioactive decay is used extensively in dating organic material that is tens of thousands of years old, has a half-life of $5,730$ years. Determine the age of an organic material that contains 8.4 percent of its initial amount of Carbon-14. Use a time unit of 1 year in the computations. The uncertainty in the half time of Carbon-14 is ± 40 years. What is the corresponding uncertainty in the estimate of the age?

Hint 1. Let A be the amount of Carbon-14. The ODE problem is then $A'(t) = -aA(t)$, $A(0) = I$. Introduced the scaled amount $u = A/I$. The ODE problem for u is $u' = -au$, $u(0) = 1$. Measure time in years. Simulate until the first mesh point t_m such that $u(t_m) \leq 0.084$.

Hint 2. Use simulations with $5,730 \pm 40$ y as input and find the corresponding uncertainty interval for the result.

Solution. We need a tailored solver function for this exercise:

[10] `http://en.wikipedia.org/wiki/Carbon-14`

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

def solver(I, a, u_crit, dt, theta):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,t_m] until u <= u_crit
    with steps of dt. Return t_m.
    """
    # Use list for u and t since we do not know how many points
    # that are needed
    dt = float(dt)               # avoid integer division
    u = []
    t = []

    u.append(I)                  # assign initial condition
    t.append(0)
    while u[-1] > u_crit:
        u_new = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[-1]
        u.append(u_new)
        t.append(t[-1] + dt)
    return t[-1]


half_life = 5730
a  = np.log(2)/half_life
print 'Age:', solver(I=1, a=a, u_crit=0.084, dt=10, theta=0.5)
```

Running this code gives an age of 20,480 years.

The uncertainty can be estimated by the following code:

```
half_life_min = 5730 - 40
half_life_max = 5730 + 40
a_min = np.log(2)/half_life_min
a_max = np.log(2)/half_life_max
age_min = solver(I=1, a=a_max, u_crit=0.084, dt=10, theta=0.5)
age_max = solver(I=1, a=a_min, u_crit=0.084, dt=10, theta=0.5)
print 'Uncertainty: [%g, %g]' % (age_min, age_max)
```

Filename: `carbon14`.

Exercise 2: Derive schemes for Newton's law of cooling

Show in detail how we can apply the ideas of the Forward Euler, Backward Euler, and Crank-Nicolson discretizations to derive explicit computational formulas for new temperature values in Newton's law of cooling (see Section 4):

$$\frac{dT}{dt} = -k(T - T_s(t)), \quad T(0) = T_0\,.$$

Here, T is the temperature of the body, $T_s(t)$ is the temperature of the surroundings, t is time, k is the heat transfer coefficient, and T_0 is the initial temperature of the body. Summarize the discretizations in a θ-rule such that you can get the three schemes from a single formula by varying the θ parameter.

Solution. The idea of the Forward Euler scheme is to sample the ODE at $t = t_n$ and apply a forward difference approximation to the derivative:

$$\frac{T^{n+1}-T^n}{\Delta t} = -k(T^n - T_s(t_n))\,.$$

The Backward Euler applies a backward difference instead:

$$\frac{T^{n}-T^{n-1}}{\Delta t} = -k(T^n - T_s(t_n))\,.$$

The Crank-Nicolson scheme samples the ODE at $t_{n+\frac{1}{2}}$, applies a centered difference approximation, and an arithmetic mean approximation to $T^{n+\frac{1}{2}}$:

$$\frac{T^{n+1}-T^n}{\Delta t} = -k(T^{n+\frac{1}{2}} - T_s(t_{n+\frac{1}{2}})) \approx -k(\frac{1}{2}(T^n + T^{n+1}) - T_s(t_{n+\frac{1}{2}}))\,.$$

For each scheme we solve with respect to the unknown T^{n+1} (note that we switch index from n to $n+1$ in the Backward Euler scheme):

$$\begin{aligned}
T^{n+1} &= T^n - k\Delta t(T^n - T_s(t_n)),\\
T^{n+1} &= \frac{T^n + k\Delta t T_s(t_{n+1})}{1 + k\Delta t},\\
T^{n+1} &= \frac{T^n - \frac{1}{2}k\Delta t T^n + k\Delta t T_s(t_{n+\frac{1}{2}})}{1 + \frac{1}{2}k\Delta t}\,.
\end{aligned}$$

A θ scheme can be formulated as

$$T^{n+1} = \frac{T^n - (1-\theta)k\Delta t T^n + k\Delta t T_s((1-\theta)t_n + \theta t_{n+1})}{1 + \theta k\Delta t}$$

Filename: `schemes_cooling`.

Exercise 3: Implement schemes for Newton's law of cooling

The goal of this exercise is to implement the schemes from Exercise 2 and investigate several approaches for verifying the implementation.

a) Implement the θ-rule from Exercise 2 in a function

```
cooling(T0, k, T_s, t_end, dt, theta=0.5)
```

where `T0` is the initial temperature, `k` is the heat transfer coefficient, `T_s` is a function of `t` for the temperature of the surroundings, `t_end` is the end time of the simulation, `dt` is the time step, and `theta` corresponds to θ. The `cooling` function should return the temperature as an array `T` of values at the mesh points and the time mesh `t`.

Solution. Here is an appropriate function:

```
import numpy as np

def cooling(T0, k, T_s, t_end, dt, theta=0.5):
    """
    Solve T'=-k(T-T_s(t)), T(0)=T0,
    for t in (0,t_end] with steps of dt.
    T_s(t) is a Python function of t.
    theta=0.5 means Crank-Nicolson, 1 is Backward
    Euler, and 0 is Forward Euler scheme.
    """
    dt = float(dt)                  # avoid integer division
    Nt = int(round(t_end/dt))       # no of time intervals
    t_end = Nt*dt                   # adjust to fit time step dt
    T = np.zeros(Nt+1)              # array of T[n] values
    t = np.linspace(0, t_end, Nt+1) # time mesh
    T[0] = T0                       # set initial condition
    for n in range(0, Nt):          # n=0,1,...,Nt-1
        T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
        dt*k*(theta*T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
        (1 + dt*theta*k)
    return T, t
```

b) In the case $\lim_{t\to\infty} T_s(t) = C = \text{const}$, explain why $T(t) \to C$. Construct an example where you can illustrate this property in a plot. Implement a corresponding test function that checks the correctness of the asymptotic value of the solution.

Solution. Apply the limit to the ODE:

$$\lim_{t\to\infty} \frac{dT}{dt} = -k(\lim_{t\to\infty} T - \lim_{t\to\infty} T_s)\,.$$

Assuming steady state behavior, $dT/dt \to \infty$ as $t \to \infty$. Then we get

$$0 = -k(\lim_{t\to\infty} T - C),$$

which means

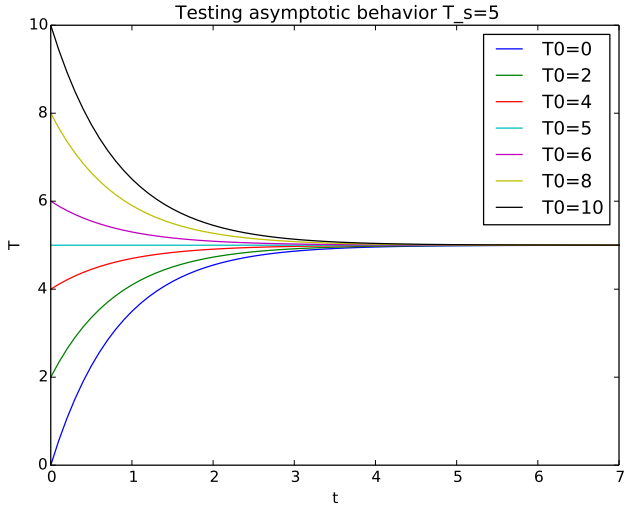
$$\lim_{t\to\infty} T = C\,.$$

A corresponding test function takes the form

```
def test_asymptotic():
    """
    Test that ``any'' initial condition leads to
    the same asymptotic behavior when T_s=constant.
    """
    import matplotlib.pyplot as plt
    plt.figure()
    T_s = 5.
    k = 1.2
    dt = 0.1
    tol = 0.01  # tolerance for testing asymptotic value
    t_end = 7    # make sure t_end is large enough for tol
    T0_values = [0, 2, 4, 5, 6, 8, 10] # test many cases
```

```
    for T0 in [0, 2, 4, 5, 6, 8, 10]:
        u, t = cooling(T0, k, lambda t: T_s, t_end, dt)
        plt.plot(t, u)

        assert abs(u[-1] - T_s) < tol, '%s != %s' % (u[-1], T_s)

    plt.legend(['T0=%g' % T0 for T0 in T0_values])
    plt.title('Testing asymptotic behavior T_s=%g' % T_s)
    plt.xlabel('t')
    plt.ylabel('T')
    plt.savefig('tmp1.png');  plt.savefig('tmp1.pdf')
    plt.show()
```

Note that we have added a plot in the test function for convenience. Letting test functions perform plotting is, however, not a good idea if you want to run a large set of tests.



c) A piecewise constant surrounding temperature,

$$T_s(t) = \begin{cases} C_0, & 0 \leq t \leq t^* \\ C_1, & t > t^*, \end{cases}$$

corresponds to a sudden change in the environment at $t = t^*$. Choose $C_0 = 2T_0$, $C_1 = \frac{1}{2}T_0$, and $t^* = 3/k$. Plot the solution $T(t)$ and explain why it seems physically reasonable.

Solution. First we implement a general tool for piecewise constant functions:

```
class Piecewise(object):
    """Class for holding a piecewise constant function."""
    def __init__(self, C0, C1, t_star):
        self.C0, self.C1 = C0, C1
        self.t_star = t_star
```

```
    def __call__(self, t):
        """
        Return value of piecewise constant function.
        t can be float or numpy array.
        """
        if isinstance(t, (float,int)):
            if t <= self.t_star:
                T_s = self.C0
            elif t > self.t_star:
                T_s = self.C1
        else:
            # assume numpy array
            T_s = np.piecewise(t,
                               [t <= self.t_star, t > self.t_star],
                               [self.C0, self.C1])
            # Alternative
            # T_s = np.where(t <= self.t_star, C0, C1)
        return T_s
```

It is convenient to scale the problem such that we do not need to find physically relevant values for k. A common scaling of T is

$$\bar{T} = \frac{T - T_0}{T_s - T_0},$$

when T_s is constant since then $\bar{T} \in [0, 1]$. Here, we may choose the long-term value of T_s in the denominator such that $\lim_{t\to\infty} \bar{T} = 1$, i.e.,

$$\bar{T} = \frac{T - T_0}{0.5T_0 - T_0} = -2\frac{T - T_0}{T_0},$$

but it leads to a shift in the sign of the temperature on the right-hand side of the ODE, and we cannot reuse the code for the original problem in the dimensionless case. We therefore avoid the negative sign and use a temperature scale $2T_0 - T_0$,

$$\bar{T} = \frac{T - T_0}{2T_0 - T_0} = \frac{T - T_0}{T_0},$$

which gives $\bar{T}$ varying from 0 initially to a maximum of 1 and finally to a minimum of $-\frac{1}{2}$. We scale T_s by its maximum value $2T_0$ so $\bar{T}_s \in [0, 1]$:

$$\bar{T}_s(\bar{t}) = \frac{T_s(t)}{\max_t T_s(t)} = \frac{T_s(t_c\bar{t})}{2T_0} = \begin{cases} 1, & \bar{t} < t^*/t_c, \\ \frac{1}{4}, & \bar{t} \geq t^*/t_c \end{cases}$$

where t_c is the time scale. Inserted in the ODE we get

$$\frac{T_0}{t_c}\frac{d\bar{T}}{d\bar{t}} = -k(T_0\bar{T} + T_0 - 2T_0\bar{T}_s,$$

leading to

$$\frac{d\bar{T}}{d\bar{t}} = -kt_c(\bar{T} + 1 - 2\bar{T}_s)\,.$$

A natural choice is $t_c = 1/k$ so we get the scaled problem

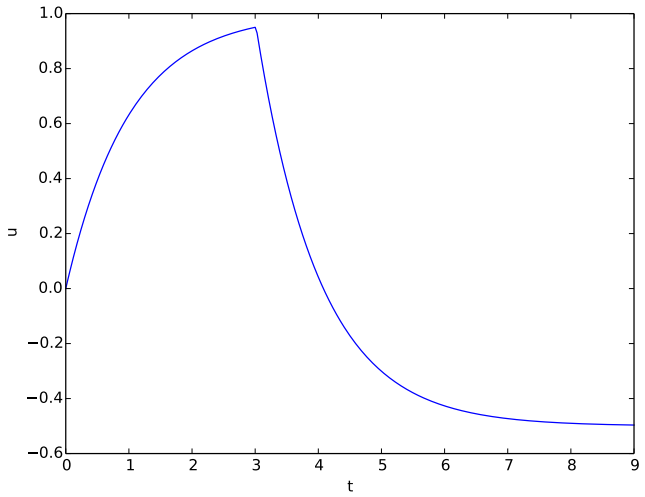
$$\frac{d\bar{T}}{d\bar{t}} = -(\bar{T} + 1 - 2\bar{T}_s) = -(\bar{T} - (2\bar{T}_s - 1)), \quad \bar{T}(0) = 0\,.$$

We can simulate this problem using the code for the original problem by choosing $k = 1$, $T_0 = 0$, and $T_s = (2-1) = 1$ for $t < 3$ and $T_s = (2\frac{1}{4} - 1) = -\frac{1}{2}$ for $t > 3$.

The appropriate code becomes

```
def simulate_piecewise_constant_Ts():
    """
    Simulate scaled problem: T' = -(T - (2T_s-1)), T(0)=0,
    where T_s=1 for t < 3 and -0.5 for t > 3.
    """
    k = 1
    T0 = 0
    t_star = 3.0
    C0 = 1
    C1 = -0.5
    T_s = Piecewise(C0, C1, t_star)
    dt = t_star/100.0
    T, t = cooling(T0, k, T_s, t_end=3*t_star, dt=dt, theta=0.5)
    import matplotlib.pyplot as plt
    plt.figure()
    plt.plot(t, T)
    plt.xlabel('t');  plt.ylabel('u')
    plt.savefig('tmp2.png');  plt.savefig('tmp2.pdf')
    plt.show()
```

The plot looks like this:



The result is reasonable because first $T_s = 1$ and the body's temperature will try to rise from 0 to 1, and it almost gets there in the time $[0, 3]$, before $T_s = -0.5$ and then the body is cooled down to -0.5 as t increases, and this is also the asymptotic value.

d) We know from the ODE $u' = -au$ that the Crank-Nicolson scheme can give non-physical oscillations for $\Delta t > 2/a$. In the present problem, this results indicates that the Crank-Nicolson scheme give undesired oscillations for $\Delta t > 2/k$. Discuss if this a potential problem in the physical case from c).

Solution. In the unscaled problem, the first stage of the simulation covers the time interval $[0, t^*] = [0, 3/k]$. It makes sense to choose Δt significantly smaller than $3/k$, and the stability limit $2/k$ is a too large step. The next time level is then $4/k$, and it sounds reasonable to include the point $t^* = 3/k$ as a mesh point. Oscillations would then occur if we choose $\Delta t = 3/k$, but this means only one step through the first interval $[0, t^*]$, which is a very coarse mesh. Halving Δt is still a coarse mesh, but then there cannot be oscillations.

e) Find an expression for the exact solution of $T' = -k(T - T_s(t))$, $T(0) = T_0$. Construct a test case and compare the numerical and exact solution in a plot.

Find a value of the time step Δt such that the two solution curves cannot (visually) be distinguished from each other. Many scientists will claim that such a plot provides evidence for a correct implementation, but point out why there still may be errors in the code. Can you introduce bugs in the `cooling` function and still achieve visually coinciding curves?

Hint. The exact solution can be derived by multiplying (8) by the integrating factor e^{kt}.

Solution. Multiplication of e^{kt}, using the product rule for differentiation "backwards", and integrating from 0 to t, results in

$$\int_0^t (e^{kt}T)'dt = k\int_0^t e^{kt}T_s dt\,.$$

The left-hand side becomes $e^{kt}T(t) - T_0$. Multiplying by e^{-kt} then gives

$$T(t) = T_0 e^{-kt} + ke^{-kt}\int_0^t e^{k\tau}T_s(\tau)d\tau,$$

which is the general expression for the exact solution.

As a check, we consider the case where T_s is constant. That problem can easily be solved by introducing $u = T - T_s$, resulting in $u' = -ku$, $u(0) = T_0 - T_s$, with solution $u(t) = (T_0 - T_s)e^{-kt}$, and consequently $T = T_s + (T_0 - T_s)e^{-kt}$. With a constant T_s in the general solution above, the solution becomes

$$\begin{aligned} T(t) &= T_0 e^{-kt} + ke^{-kt}\int_0^t e^{kt}T_s dt \\ &= T_0 e^{-kt} + ke^{-kt}T_s k^{-1}(e^{kt} - 1) \\ &= T_0 e^{-kt} + T_s - T_s e^{-kt} \\ &= T_s + (T_0 - T_s)e^{-kt}, \end{aligned}$$

as desired.

We choose the same test problem as in c) and use SymPy to do the integration. A function doing the integration and returning Python functions for the formulas for $t < t^*$ and $t \geq t^*$ is convenient:

```
def T_exact_symbolic(verbose=False):
    """Compute the exact solution formula via sympy."""
    # sol1: solution for t < t_star,
    # sol2: solution for t > t_star
    import sympy as sym
    T0 = sym.symbols('T0')
    k = sym.symbols('k', positive=True)
    # Piecewise linear T_sunction
    t, t_star, C0, C1 = sym.symbols('t t_star C0 C1')
    T_s = C0
    I = sym.integrate(sym.exp(k*t)*T_s, (t, 0, t))
    sol1 = T0*sym.exp(-k*t) + k*sym.exp(-k*t)*I
    sol1 = sym.simplify(sym.expand(sol1))
    if verbose:
        # Some debugging print
        print 'solution t < t_star:', sol1
        #print sym.latex(sol1)
    T_s = C1
    I = sym.integrate(sym.exp(k*t)*C0, (t, 0, t_star)) + \
        sym.integrate(sym.exp(k*t)*C1, (t, t_star, t))
    sol2 = T0*sym.exp(-k*t) + k*sym.exp(-k*t)*I
    sol2 = sym.simplify(sym.expand(sol2))
    if verbose:
        print 'solution t > t_star:', sol2
        #print sym.latex(sol2)

    # Convert to numerical functions
    exact0 = sym.lambdify([t, C0, k, T0],
                          sol1, modules='numpy')
    exact1 = sym.lambdify([t, C0, C1, t_star, k, T0],
                          sol2, modules='numpy')
    return exact0, exact1
```

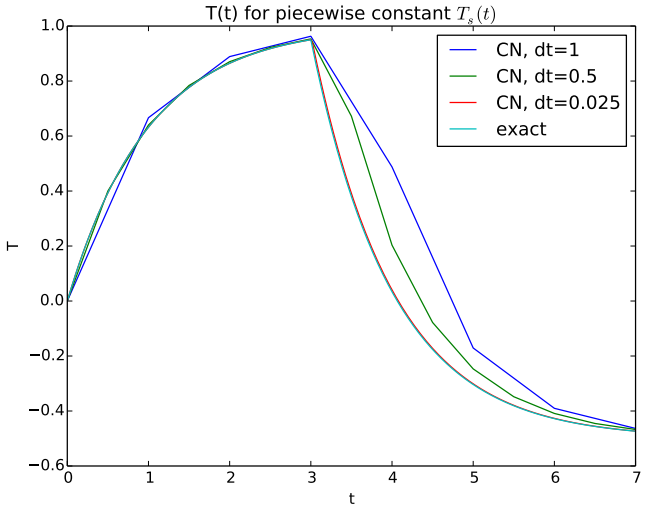
Then we need a function that can evaluate the exact solution as a mesh function:

```
def evaluate_T_exact(t, k, T0, C0, C1, t_star, verbose=False):
    """
    Return exact (analytical) solution of the problem.
    Exact solution is produced by sympy.
    """
    exact0, exact1 = T_exact_symbolic()
    # exact0/1 works with t as numpy array
    if isinstance(t, (float,int)):
        if t < t_star:
            return exact0(t, C0, k, T0)
        else:
            return exact1(t, C0, C1, t_star, k, T0)
    else:
        # assume numpy array
        return np.where(
            t < t_star,
            exact0(t, C0, k, T0),
            exact1(t, C0, C1, t_star, k, T0))
```

Finally we can run the comparison:

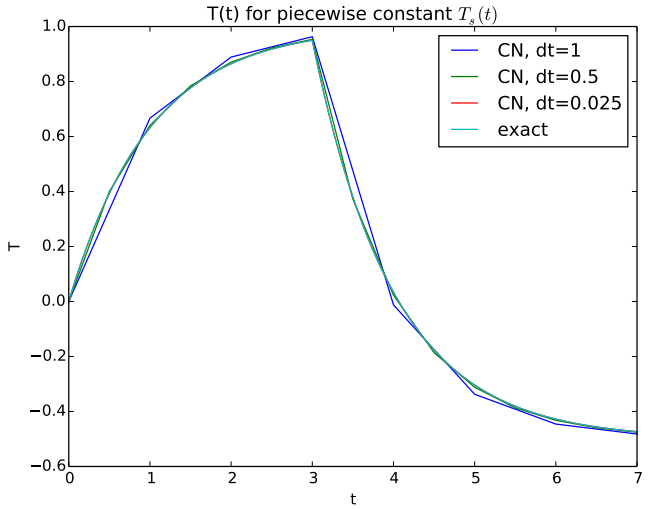
```
def compare_numerical_and_exact_solution():
    """
    Compare exact and numerical solution with piecewise
    constant surrounding temperature. Use scaled problem
    from function simulate_piecewise_constant_Ts.
    """
    T0 = 0
    k = 1
    C0 = 1
    C1 = -0.5
    t_star = 3
    t_end = 7

    T_s = Piecewise(C0, C1, t_star)

    import matplotlib.pyplot as plt
    plt.figure()
    dt_values = [1, 0.5, 0.025]
    #dt_values = [0.025]
    for dt in dt_values:
        T, t = cooling(T0, k, T_s, t_end, dt, theta=0.5)
        plt.plot(t, T)

    t_e = np.linspace(0, t_end, 1001)  # find mesh for T_e
    # Could use sym.Rational(1,2) instead of 0.5, but not necessary
    # when we are not interested in symbolic formulas
    T_e = evaluate_T_exact(t_e, k, T0, C0, C1, t_star)
    plt.plot(t_e, T_e)
    plt.legend(['CN, dt=%g' % dt for dt in dt_values] + ['exact'])
    plt.title('T(t) for piecewise constant $T_s(t)$')
    plt.xlabel('t')
    plt.ylabel('T')
    plt.savefig('tmp3.png');  plt.savefig('tmp3.pdf')
    plt.show()
```

The Δt values were found after some trial and error, but they illustrate crude approximations and one with the biggest possible Δt such that the exact solution and the numerical solution cannot be visually distinguished:

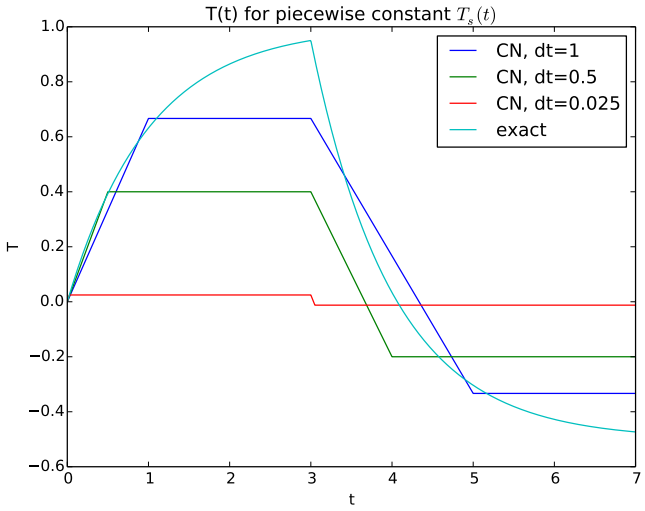


We can now start to introduce bugs in the `cooling` function to see if it is possible to have some Δt and still find coinciding curves.

Bug 1: Wrong time level in the T_s function. We replace `T_s[n]` by `T_s[n+1]` in the implementation of the scheme and rerun the case. Now the lowest Δt is still on top of the exact solution, but the numerical solution on the two coarser meshes are more accurate! This is because we lower the surrounding temperature somewhat earlier in the buggy scheme and this reduces the "overshoot" on the coarsest meshes in the figure above.



Bug 2: Wrong time level in the T function. We can replace `T[n]` by `T[n+1]` on the right-hand side of the scheme. This is a serious error since `T[n+1]` is not yet computed and therefore equal to zero when `T` was made by calling `np.zeros`. The results are also nonsense, and one would immediately look for a bug.



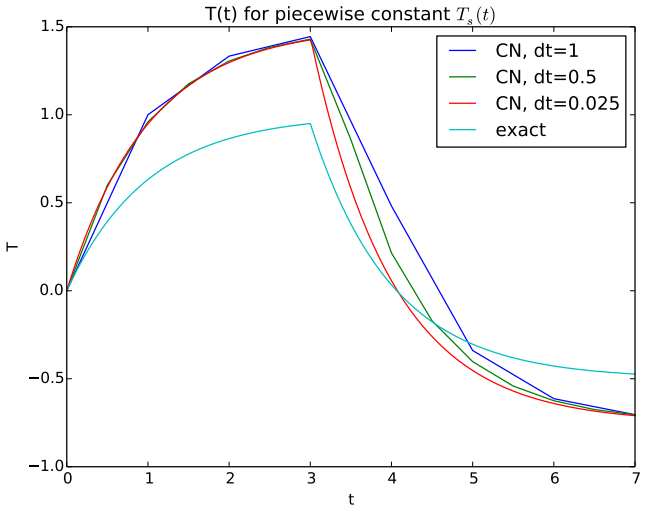
Bug 3: Missing θ in a term. Let us forget to multiply by `theta` in the nominator of the scheme, i.e., we replace

```
T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
         dt*k*(theta*T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
         (1 + dt*theta*k)
```

by

```
T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
         dt*k*(T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
         (1 + dt*theta*k)
```

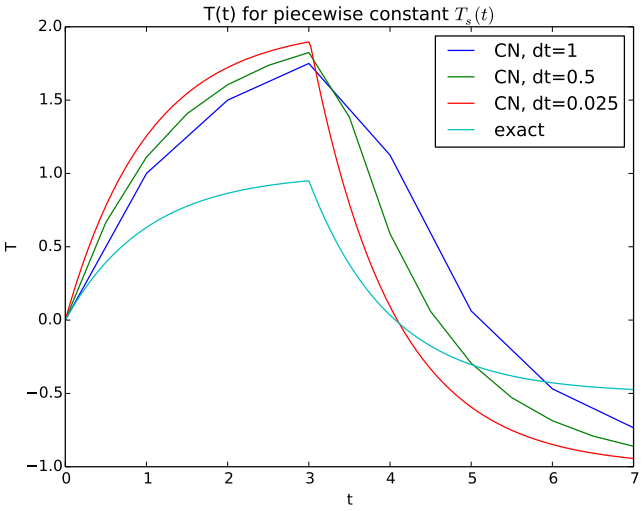
This error leads to convergence towards a wrong solution:



If we did not have the exact solution, one could be led to think that the solution was correct, but it is non-physical since we do not expect the temperature to rise from T_0 to a level *above* the surrounding temperature. The plot shows that $\bar{T} > 1$, the value of the (scaled) surrounding temperature. Note that if we used the Backward Euler scheme instead of the Crank-Nicolson scheme, this bug would have no effect!

Bug 4: Missing `k` in the updating formula. Obviously, when we solve the scaled problem where $k = 1$ by definition, such a programming mistake has no effect. Otherwise, k influences the time scale, so there will be a stretch of the time axis in the numerical solution and this should be easily detected in a plot.

Bug 5: Using `1-theta` instead of `theta`. Such an error is not detectable in the Crank-Nicolson scheme, but will have a significant effect in the other schemes. As a test, we replace `1-theta` in the nominator by `theta`. This leads to $T = 0$ in the Forward Euler scheme, but a reasonable shape in the Backward Euler scheme, although the solution becomes larger than the surrounding temperature (1 in the scaled problem).



f) Implement a test function for checking that the solution returned by the `cooling` function is identical to the exact numerical solution of the problem (to machine precision) when T_s is constant.

Hint. The exact solution of the discrete equations in the case T_s is a constant can be found by introducing $u = T - T_s$ to get a problem $u' = -ku$, $u(0) = T_0 - T_s$. The solution of the discrete equations is then of the form $u^n = (T_0 - T_s)A^n$ for some amplification factor A. The expression for T^n is then $T^n = T_s(t_n) + u^n = T_s + (T_0 - T_s)A^n$. We find that

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

The test function, testing several θ values for a quite coarse mesh, may take the form

```
def test_discrete_solution():
    """
    Compare the numerical solution with an exact solution of the scheme
    when the T_s is constant.
    """
    T_s = 10
    T0 = 2
    k = 1.2
    dt = 0.1   # can use any mesh
    N_t = 6    # any no of steps will do
    t_end = dt*N_t
    t = np.linspace(0, t_end, N_t+1)

    for theta in [0, 0.5, 1, 0.2]:
        u, t = cooling(T0, k, lambda t: T_s , t_end, dt, theta)
        A = (1 - (1-theta)*k*dt)/(1 + theta*k*dt)
        u_discrete_exact = T_s + (T0-T_s)*A**(np.arange(len(t)))
```

```
        diff = np.abs(u - u_discrete_exact).max()
        print 'diff computed and exact discrete solution:', diff
        tol = 1E-14
        success = diff < tol
        assert success, 'diff=%g' % diff
```

Running this function shows that the `diff` variable is `3.55E-15` as maximum so a tolerance of 10^{-14} is appropriate. This is a good test that the `cooling` function works!
Filename: `cooling`.

Exercise 4: Find time of murder from body temperature

A detective measures the temperature of a dead body to be 26.7 C at 2 pm. One hour later the temperature is 25.8 C. The question is when death occurred.

Assume that Newton's law of cooling (8) is an appropriate mathematical model for the evolution of the temperature in the body. First, determine k in (8) by formulating a Forward Euler approximation with one time steep from time 2 am to time 3 am, where knowing the two temperatures allows for finding k. Assume the temperature in the air to be 20 C. Thereafter, simulate the temperature evolution from the time of murder, taken as $t = 0$, when $T = 37$ C, until the temperature reaches 25.8 C. The corresponding time allows for answering when death occurred.

Solution. A Forward Euler step from T^0 to T^1 reads

$$T^1 = T_0 + -k\Delta t(T^0 - T_s),$$

and solving with respect to k results in

$$k = \frac{T^1 - T^0}{\Delta t(T_s - T_0)}\,.$$

We implement this formula in a function,

```
def estimate_k(T0, T1, Ts, dt):
    return float(T1 - T0)/(dt*(Ts - T0))
```

We have $T_0 = 26.7$ C, $T_1 = 25.8$ C, $T_s = 20$ C, and $\Delta t = 1$ h, i.e., $\Delta t = 3600$ s. The proper call is therefore

```
k = estimate_k(26.7, 25.8, 20, 3600)
```

For the simulation we use the Forward Euler method,

$$T^{n+1} = T^n - k\Delta t(T^n - T_s),$$

and simulate as long as $T > 25.8$ C:

```
T = 37
Ts = 20
from cooling import cooling
while T > 25.8:
    T = T - k*dt*(T - Ts)
    t+= dt

minutes, seconds = divmod(t, 60)
hours, minutes = divmod(minutes, 60)
print """
The death occurred %d hours, %d minutes,
and %g seconds before 3am.""" % (hours, minutes, seconds)
```

The result of running the code becomes

Terminal

```
Terminal> python detective.py
k=3.73134e-05

The death occurred 8 hours, 0 minutes,
and 19 seconds before 3am.
```

Filename: `detective`.

Exercise 5: Simulate an oscillating cooling process

The surrounding temperature T_s in Newton's law of cooling (8) may vary in time. Assume that the variations are periodic with period P and amplitude a around a constant mean temperature T_m:

$$T_s(t) = T_m + a\sin\left(\frac{2\pi}{P}t\right). \tag{51}$$

Simulate a process with the following data: $k = 0.05$ min^{-1}, $T(0) = 5$ C, $T_m = 25$ C, $a = 2.5$ C, and $P = 1$ h, $P = 10$ min, and $P = 6$ h. Plot the T solutions and T_s in the same plot.

Solution. We can reuse the `cooling` function from Exercise 3 to do the simulations:

```
import numpy as np

def cooling(T0, k, T_s, t_end, dt, theta=0.5):
    """
    Solve T'=-k(T-T_s(t)), T(0)=T0,
    for t in (0,t_end] with steps of dt.
    T_s(t) is a Python function of t.
    theta=0.5 means Crank-Nicolson, 1 is Backward
    Euler, and 0 is Forward Euler scheme.
    """
    dt = float(dt)                  # avoid integer division
    Nt = int(round(t_end/dt))       # no of time intervals
    t_end = Nt*dt                   # adjust to fit time step dt
    T = np.zeros(Nt+1)              # array of T[n] values
```

```
    t = np.linspace(0, t_end, Nt+1) # time mesh
    T[0] = T0                       # set initial condition
    for n in range(0, Nt):          # n=0,1,...,Nt-1
        T[n+1] = ((1 - dt*(1 - theta)*k)*T[n] + \
        dt*k*(theta*T_s(t[n+1]) + (1 - theta)*T_s(t[n])))/ \
        (1 + dt*theta*k)
    return T, t
```

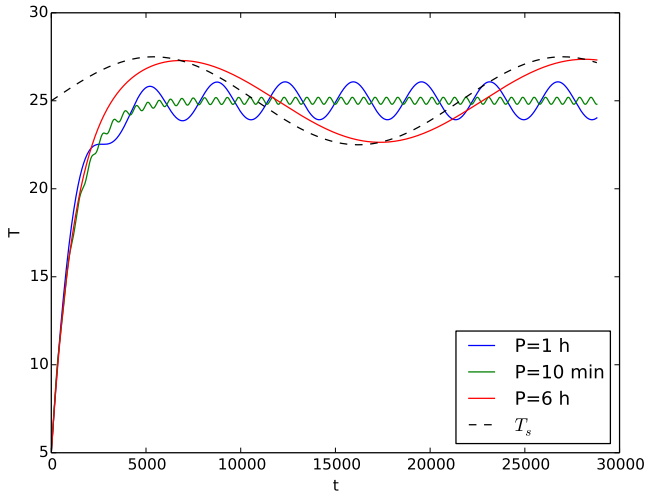
The challenge is to use the right units for the input data. We can use Celsius for temperature since it has the same increments as Kelvin. Time quantities should be measured in seconds:

$$k = 20\ \text{min}^{-1} = \frac{20}{60}\ \text{s}^{-1},$$
$$P = (1\ \text{h} = 3600\ \text{s}, 10\ \text{min} = 600\ \text{s}, 6\ \text{h} = 6 \cdot 3600\ \text{s})\,.$$

To achieve reasonable accuracy, we choose Δt as 40 steps per the shortest period of the T_s oscillations: $\Delta t = 600/40$. With some trials we find an appropriate simulation interval for all three cases to be $[0, 8]$ h.

The code becomes

```
from cooling import cooling
from numpy import pi, sin

def T_s(t):
    return Tm + a*sin((2*pi/P)*t)

Tm = 25
a = 2.5
P_values = [3600, 600, 3600*6]
k = 0.05/60
T0 = 5

import matplotlib.pyplot as plt
for P in P_values:
    T, t = cooling(T0, k, T_s, t_end=8*3600, dt=600/40)
    plt.plot(t, T)
plt.plot(t, T_s(t), 'k--')  # T_s for largest P to show amplitude
legends = ['P=1 h', 'P=10 min', 'P=6 h', '$T_s$']
plt.legend(legends, loc='lower right')
plt.xlabel('t'); plt.ylabel('T')
plt.savefig('tmp.png');  plt.savefig('tmp.pdf')
plt.show()
```



Discussion of the results. We see that it takes some time to increase the temperature from T_0 to oscillations around T_m. When T_s oscillates fast ($P = 10$ min), k is not large enough so that T can reach the surrounding temperature in the time available before the surrounding temperature decreases. However, for large P (6 h), there is almost enough time to heat and cool the object to reach the maximum and minimum temperatures of the surroundings.
Filename: `osc_cooling`.

Exercise 6: Simulate stochastic radioactive decay

The purpose of this exercise is to implement the stochastic model described in Section 5 and show that its mean behavior approximates the solution of the corresponding ODE model.

The simulation goes on for a time interval $[0, T]$ divided into N_t intervals of length Δt. We start with N_0 atoms. In some time interval, we have N atoms that have survived. Simulate N Bernoulli trials with probability $\lambda\Delta t$ in this interval by drawing N random numbers, each being 0 (survival) or 1 (decay), where the probability of getting 1 is $\lambda\Delta t$. We are interested in the number of decays, d, and the number of survived atoms in the next interval is then $N - d$. The Bernoulli trials are simulated by drawing N uniformly distributed real numbers on $[0, 1]$ and saying that 1 corresponds to a value less than $\lambda\Delta t$:

```
# Given lambda_, dt, N
import numpy as np
uniform = np.random.uniform(N)
Bernoulli_trials = np.asarray(uniform < lambda_*dt, dtype=np.int)
d = Bernoulli_trials.size
```

Observe that `uniform < lambda_*dt` is a boolean array whose true and false values become 1 and 0, respectively, when converted to an integer array.

Repeat the simulation over $[0, T]$ a large number of times, compute the average value of N in each interval, and compare with the solution of the corresponding ODE model. Filename: `stochastic_decay`.

Problem 7: Radioactive decay of two substances

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a half-life $A_{1/2}$, while substance B decay to form type A nuclei with a half-life $B_{1/2}$. Letting u_A and u_B be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$\frac{1}{\ln 2}u'_A = u_B/B_{1/2} - u_A/A_{1/2}, \tag{52}$$

$$\frac{1}{\ln 2}u'_B = u_A/A_{1/2} - u_B/B_{1/2}, \tag{53}$$

with $u_A(0) = u_B(0) = 1$.

a) Make a simulation program that solves for $u_A(t)$ and $u_B(t)$.

b) Verify the implementation by computing analytically the limiting values of u_A and u_B as $t \to \infty$ (assume $u'_A, u'_B \to 0$) and comparing these with those obtained numerically.

c) Run the program for the case of $A_{1/2} = 10$ minutes and $B_{1/2} = 50$ minutes. Use a time unit of 1 minute. Plot u_A and u_B versus time in the same plot. Filename: `radioactive_decay_2subst`.

Exercise 8: Simulate a simple chemical reaction

Consider the simple chemical reaction where a substance A is turned into a substance B according to

$$\frac{d[A]}{dt} = -k[A],$$
$$\frac{d[B]}{dt} = k[A],$$

where $[A]$ and $[B]$ are the concentrations of A and B, respectively. It may be a challenge to find appropriate values of k, but we can avoid this problem by working with a scaled model (as explained in Section 1). Scale the model above, using a time scale $1/k$, and use the initial concentration of $[A]$ as scale for $[A]$ and $[B]$. Show that the scaled system reads

$$\frac{du}{dt} = -u,$$
$$\frac{dv}{dt} = u,$$

with initial conditions $u(0) = 1$, and $v(0) = \alpha$, where $\alpha = [B](0)/[A](0)$ is a dimensionless number, and u and v are the scaled concentrations of $[A]$ and $[B]$, respectively. Implement a numerical scheme that can be used to find the solutions $u(t)$ and $v(t)$. Visualize u and v in the same plot. Filename: `chemcial_kinetics_AB`.

Exercise 9: Simulate an n-th order chemical reaction

An n-order chemical reaction, generalizing the model in Exercise 8, takes the form

$$\frac{d[A]}{dt} = -k[A]^n,$$
$$\frac{d[B]}{dt} = k[A]^n,$$

where symbols are as defined in Exercise 8. Bring this model on dimensionless form, using a time scale $[A](0)^{n-1}/k$, and show that the dimensionless model simplifies to

$$\frac{du}{dt} = -u^n,$$
$$\frac{dv}{dt} = u^n,$$

with $u(0) = 1$ and $v(0) = \alpha = [B](0)/[A](0)$. Solve numerically for $u(t)$ and show a plot with u for $n = 0.5, 1, 2, 4$. Filename: `chemcial_kinetics_ABn`.

Exercise 10: Simulate a biochemical process

The purpose of this exercise is to simulate the ODE system (23)-(26) modeling a simple biochemical process.

a) Scale (23)-(26) such that we can work with dimensionless parameters, which are easier to prescribe. Introduce

$$\bar{Q} = \frac{[ES]}{Q_c}, \quad \bar{P} = \frac{P}{P_c}, \quad \bar{S} = \frac{S}{S_0}, \quad \bar{E} = \frac{E}{E_0}, \quad \bar{t} = \frac{t}{t_c},$$

where appropriate scales are

$$Q_c = \frac{S_0 E_0}{K}, \quad P_c = Q_c, \quad t_c = \frac{1}{k_+ E_0},$$

with $K = (k_v + k_-)/k_+$ (Michaelis constant). Show that the scaled system becomes

$$\frac{d\bar{Q}}{d\bar{t}} = \alpha(\bar{E}\bar{S} - \bar{Q}), \tag{54}$$

$$\frac{d\bar{P}}{d\bar{t}} = \beta\bar{Q}, \tag{55}$$

$$\frac{d\bar{S}}{d\bar{t}} = -\bar{E}\bar{S} + (1 - \beta\alpha^{-1})\bar{Q}, \tag{56}$$

$$\epsilon\frac{d\bar{E}}{d\bar{t}} = -\bar{E}\bar{S} + \bar{Q}, \tag{57}$$

where we have three dimensionless parameters

$$\alpha = \frac{K}{E_0}, \quad \beta = \frac{k_v}{k_+ E_0}, \quad \epsilon = \frac{E_0}{S_0}\,.$$

The corresponding initial conditions are $\bar{Q} = \bar{P} = 0$ and $\bar{S} = \bar{E} = 1$.

Solution. Replacing the unknowns and t by their dimensionless equivalents leads to

$$\begin{aligned}
\frac{d\bar{Q}}{d\bar{t}} &= t_c k_+ \frac{E_0 S_0}{Q_c}\bar{E}\bar{S} - t_c(k_v + k_-)\bar{Q}, \\
\frac{d\bar{P}}{d\bar{t}} &= t_c k_v \frac{Q_c}{P_c}\bar{Q}, \\
\frac{d\bar{S}}{d\bar{t}} &= -t_c k_+ E_0 \bar{E}\bar{S} + t_c k_- \frac{Q_c}{S_0}\bar{Q}, \\
\frac{d\bar{E}}{d\bar{t}} &= -t_c k_+ S_0 \bar{E}\bar{S} + t_c(k_- + k_v)\frac{Q_c}{E_0}\bar{Q}\,.
\end{aligned}$$

Inserting the choice of scales brings us to the given equations, after quite some algebra and identifying coefficients in terms of the provided dimensionless numbers.

b) Implement a function for solving (54)-(57).

Solution. Let us use Odespy to solve the differential equations, although a plain Forward Euler scheme will be fine.

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

def solver(alpha, beta, epsilon, T, dt=0.1):
    def f(u, t):
        Q, P, S, E = u
        return [
            alpha*(E*S - Q),
            beta*Q,
            -E*S + (1-beta/alpha)*Q,
            (-E*S + Q)/epsilon,
            ]

    Nt = int(round(T/dt))
    t_mesh = np.linspace(0, Nt*dt, Nt+1)

    solver = odespy.RK4(f)
    solver.set_initial_condition([0, 0, 1, 1])
    u, t = solver.solve(t_mesh)
    Q = u[:,0]
    P = u[:,1]
    S = u[:,2]
    E = u[:,3]
    return Q, P, S, E
```

c) There are two conservation equations implied by (23)-(26):

$$[ES] + [E] = E_0, \tag{58}$$

$$[ES] + [S] + [P] = S_0\,. \tag{59}$$

Derive these two equations. Use these properties in the function in b) to do a partial verification of the solution at each time step.

Solution. Adding (23) and (26) shows that

$$\frac{d[ES]}{dt} + \frac{d[E]}{dt} = 0,$$

and therefore $[ES] + [E] = \text{const}$. Since $[ES](0) = 0$ and $[E](0) = E_0$, the constant is E_0 at $t = 0$ and will remain so. Similarly, adding (23), (25), and (24) shows that their time derivatives sum up to zero, and therefore $[ES]+[S]+[P] = \text{const}$. Since $[P](0) = 0$, the constant must be $0 + S_0 + 0 = S_0$.

To use the conservation as a consistency check in the software, we need to find the equivalent dimensionless versions:

$$[ES] + [E] = E_0 \quad \Rightarrow \quad Q_c\bar{Q} + E_0\bar{E} = E_0,$$

and from this we get, after a little algebra,

$$\alpha^{-1}\epsilon^{-1}\bar{Q} + \bar{E} = 1\,.$$

The other conservation equation becomes

$$\bar{Q} + \alpha\bar{S} + \bar{P} = \alpha\,.$$

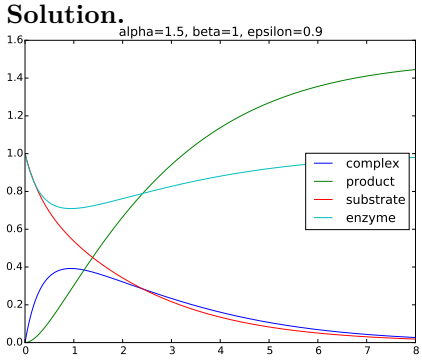
The implementation may go like

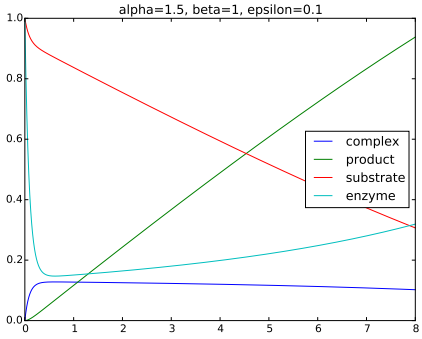
```
computed = Q[n+1]/(alpha*epsilon) + E[n+1]
expected = 1
diff1 = abs(computed - expected)

computed = Q[n+1] + alpha*S[n+1] + P[n+1]
expected = alpha
diff2 = abs(computed - expected)

tol = 1E-14
if diff1 < tol or diff2 < tol:
    print '*** Consistency check failed:', diff1, diff2
```

d) Simulate a case with $T = 8$, $\alpha = 1.5$, $\beta = 1$, and two ϵ values: 0.9 and 0.1.

```
def demo():
    alpha = 1.5
    beta = 1
    epsilon = 0.1
    T = 8
    dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.01
    Q, P, S, E = solver(alpha, beta, epsilon, T, dt)
    plt.plot(t, Q, t, P, t, S, t, E)
    plt.legend(['complex', 'product', 'substrate', 'enzyme'],
               loc='upper right')
    plt.title('alpha=%g, beta=%g, epsilon=%g' %
              (alpha, beta, epsilon))
    plt.savefig('tmp.png');  plt.savefig('tmp.pdf')
    plt.show()

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

Solution.





Filename: `biochem`.

Exercise 11: Simulate spreading of a disease

The SIR model (27)-(29) can be used to simulate spreading of an epidemic disease.

a) Estimating the parameter β is difficult so it can be handy to scale the equations. Use $t_c = 1/\nu$ as time scale, and scale S, I, and R by the population size $N = S(0) + I(0) + R(0)$. Show that the resulting dimensionless model becomes

$$\frac{d\bar{S}}{d\bar{t}} = -R_0\bar{S}\bar{I}, \tag{60}$$

$$\frac{d\bar{I}}{d\bar{t}} = R_0\bar{S}\bar{I} - \bar{I}, \tag{61}$$

$$\frac{d\bar{R}}{d\bar{t}} = I, \tag{62}$$

$$\bar{S}(0) = 1 - \alpha, \tag{63}$$

$$\bar{I}(0) = \alpha, \tag{64}$$

$$\bar{R}(0) = 0, \tag{65}$$

where R_0 and α are the only parameters in the problem:

$$R_0 = \frac{N\beta}{\nu}, \quad \alpha = \frac{I(0)}{N}\,.$$

A quantity with a bar denotes a dimensionless version of that quantity, e.g, $\bar{t}$ is dimensionless time, and $\bar{t} = \nu t$.

Solution. We introduce

$$\bar{t} = \frac{t}{\nu^{-1}}, \quad \bar{S} = \frac{S}{N}, \quad \bar{I} = \frac{I}{N}, \quad \bar{R} = \frac{R}{N}\,.$$

Inserting these expressions in the governing equations and dividing by νN gives the listed dimensionless ODEs. The scaled initial condition for $\bar{S}(0)$ follows from $\bar{S}(0) = S(0)/N = (N - I(0))/N = 1 - \alpha$, since initially, $R(0) = 0$ and therefore $N = S(0) + I(0)$.

b) Show that the R_0 parameter governs whether the disease will spread or not at $t = 0$.

Hint. Spreading means $dI/dt > 0$.

Solution. For dI/dt to be positive, we must have $(R_0\bar{S}-1)\bar{I} > 0$, i.e., $R_0\bar{S}-1 > 0$ since $\bar{I} \geq 0$. At $t = 0$, we get $R_0\bar{S}(0) > 1$ as the criterion, or

$$\tilde{R}_0\bar{S}(0) = \frac{N\beta}{\nu}\frac{S(0)}{N} = \frac{S(0)\beta}{\nu} > 1\,.$$

The dimensionless parameter $S(0)\beta/\nu$ is denoted by R_0 in the epidemiology literature and known as the *basic reproductive number*.

c) Implement the scaled SIR model. Check at every time step, as a verification, that $\bar{S} + \bar{I} + \bar{R} = 1$.

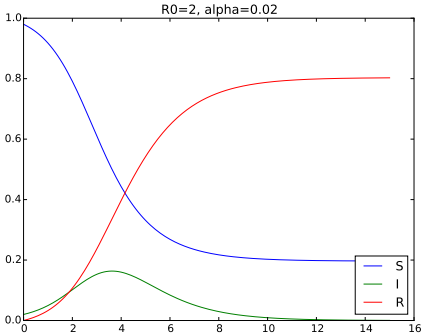
Solution. We may use Odespy and the RK4 method to solve the system:

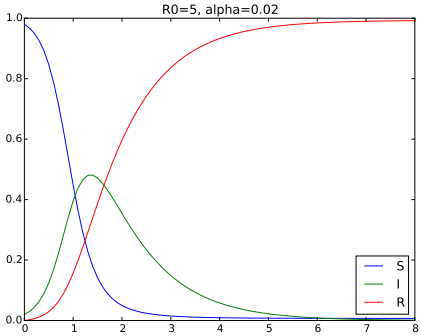
```
import odespy
import numpy as np
import matplotlib.pyplot as plt
import sys

def solver(R0, alpha, T, dt=0.1):
    def f(u, t):
        S, I, R = u
        return [
            -R0*S*I,
            R0*S*I - I,
            I]

    Nt = int(round(T/dt))
    t_mesh = np.linspace(0, Nt*dt, Nt+1)

    solver = odespy.RK4(f)
    solver.set_initial_condition([1-alpha, alpha, 0])
    u, t = solver.solve(t_mesh)
    S = u[:,0]
    I = u[:,1]
    R = u[:,2]
    # Consistency check
    N = 1
    tol = 1E-15
    for i in range(len(S)):
        if abs(S[i] + I[i] + R[i] - N) > tol:
            print 'Consistency error: S+I+R=%g != %g' % \
                  (S[i] + I[i] + R[i], N)
    return S, I, R, t
```

d) Simulate the spreading of a disease where $R_0 = 2,5$ and 2 percent of the population is infected at time $t = 0$.

Solution. The given data means that $\bar{I}(0) = \alpha = 0.02$ and $\bar{S}(0) = 0.98$.

```
def demo():
    alpha = 0.02
    R0 = 5
    T = 8
    dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.1
    S, I, R, t = solver(R0, alpha, T, dt)
    plt.plot(t, S, t, I, t, R)
    plt.legend(['S', 'I', 'R'], loc='lower right')
    plt.title('R0=%g, alpha=%g' % (R0, alpha))
    plt.savefig('tmp.png');  plt.savefig('tmp.pdf')
    plt.show()
```





Filename: `SIR`.

Exercise 12: Simulate predator-prey interaction

Section 8 describes a model for the interaction of predator and prey populations, such as lynx and hares.

a) Scale the equations (30)-(31). Use the initial population $H(0) = H_0$ of H has scale for H and L, and let the time scale be $1/(bH_0)$.

Solution. We introduce dimensionless time $\bar{t} = t/t_c$, where $t_c^{-1} = bH_0$, and scale H and L as $\bar{H} = H/H_0$ and $\bar{L} = L/H_0$. Inserted in the equations, we arrive at

$$\frac{H_0}{t_c}\frac{d\bar{H}}{d\bar{t}} = H_0\bar{H}(a - bH_0\bar{L}),$$
$$\frac{H_0}{t_c}\frac{d\bar{L}}{d\bar{t}} = H_0\bar{L}(dH_0\bar{H} - c)\,.$$

Inserting $t_c^{-1} = bH_0$ and rearranging simplify the equations to

$$\frac{d\bar{H}}{d\bar{t}} = \frac{a}{bH_0}\bar{H} - \bar{L}\bar{H},$$
$$\frac{d\bar{L}}{d\bar{t}} = \frac{d}{b}\bar{L}\bar{H} - \frac{c}{bH_0}\bar{L})\,.$$

The initial conditions become $\bar{H}(0) = 1$, $\bar{L}(0) = L_0/H_0$. With the dimensionless parameters

$$\alpha = \frac{a}{bH_0}, \quad \beta = \frac{d}{b}, \quad \gamma = \frac{c}{bH_0}, \quad \delta = \frac{H_0}{L_0},$$

we can write the dimensionless problem as

$$
\begin{aligned}
\frac{d\bar{H}}{d\bar{t}} &= \alpha\bar{H} - \bar{L}\bar{H}, \\
\frac{d\bar{L}}{d\bar{t}} &= \beta\bar{L}\bar{H} - \gamma\bar{L}), \\
\bar{H}(0) &= 1, \\
\bar{L}(0) &= \delta\,.
\end{aligned}
$$

The quantity bH_0 is the number of eaten preys per predator. Then α measures the ratio of natural population growth of the prey, due to nutrition, and the number of eaten preys per predator. The β parameter measures the fraction of the eaten preys and the amount of this that actually leads to population growth of the predator. The number γ reflects the ratio of predator deaths and the eaten preys per predator, and δ is the initial fraction of preys and predators.

b) Implement the scaled model from a). Run illustrating cases how the two populations develop.

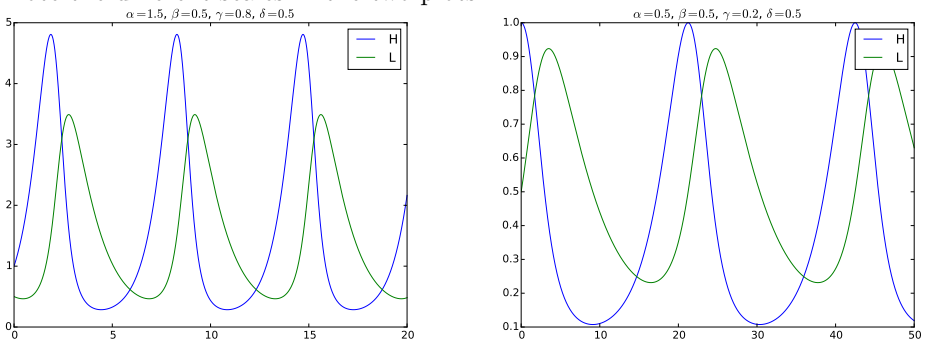
Solution. Here are two functions:

```
import odespy
import numpy as np
#import scitools.std as plt
import matplotlib.pyplot as plt
fig_counter = 0  # used in plot file names

def simulate(alpha, beta, gamma, delta, T=20):
    def f(u, t, alpha, beta, gamma):
        H, L = u
        return [alpha*H - L*H, beta*L*H - gamma*L]


    t = np.linspace(0, T, 1501)
    solver = odespy.RK4(f, f_args=(alpha, beta, gamma))
    solver.set_initial_condition([1, delta])
    u, t = solver.solve(t)
    H = u[:,0]
    L = u[:,1]

    plt.figure()
    global fig_counter
    fig_counter += 1
    plt.plot(t, H, t, L)
    plt.legend(['H', 'L'])
    plt.title(r'$\alpha=%g$, $\beta=%g$, $\gamma=%g$, $\delta=%g$'
              % (alpha, beta, gamma, delta))
    plt.savefig('tmp%d.png' % fig_counter)
    plt.savefig('tmp%d.pdf' % fig_counter)
    return H, L, t

def demo():
    simulate(alpha=1.5, beta=0.5, gamma=0.8, delta=0.5, T=20)
    simulate(alpha=0.5, beta=0.5, gamma=0.2, delta=0.5, T=50)
```

We must call `demo()` from the main program.

Here are two examples. The one to the left has a good birth-death rate compared to how many that are eaten by predators ($\alpha = 1.5$), half of the eaten preys contribute to growth of the predator population ($\beta = 0.5$), the death rate of predators is slightly smaller than how many preys they eat (per predator) ($\gamma = 0.8$), and we start out with twice as many preys as predators.

The figure to the right features less nutrition for the preys compared to being eaten by predators ($\alpha = 0.5$), and a lower death rate in the predator population. Note the different scales in the two plots.



c) The scaling in a) used a scale for H and L based on the initial condition $H(0) = H_0$. An alternative scaling is to make the ODEs as simple as possible by introducing separate scales H_c and L_c for H and L, respectively. Fit H_c, L_c, and the time scale t_c such that there are as few dimensionless parameters as possible in the ODEs. Scale the initial conditions. Compare the number and type of dimensionless parameters with a).

Solution. Inserting

$$\bar{t} = \frac{t}{c_c}, \quad \bar{H} = \frac{H}{H_c}, \quad \bar{L} = \frac{L}{L_c},$$

in the ODEs (30)-(31), we arrive at

$$\frac{H_c}{t_c}\frac{d\bar{H}}{d\bar{t}} = H_c\bar{H}(a - bL_c\bar{L}),$$
$$\frac{L_c}{t_c}\frac{d\bar{L}}{d\bar{t}} = L_c\bar{L}(dH_c\bar{H} - c)\,.$$

Now we divide by H_c and L_c in the H and L equations, respectively, and multiply by t_c:

$$\frac{d\bar{H}}{d\bar{t}} = t_c\bar{H}(a - bL_c\bar{L}),$$
$$\frac{d\bar{L}}{d\bar{t}} = t_c\bar{L}(dH_c\bar{H} - c)\,.$$

Choosing $t_c = 1/a$ and $t_c a L_c = 1$, i.e., $L_c = a/b$, makes the first equation free of parameters: $\bar{H}' = \bar{H}(1 - \bar{L})$. Factoring out c in the equation for L and choosing $H_c d/c = 1$, i.e., $H_c = c/d$, leaves us with the L equation as $\bar{L}' = (c/a)\bar{L}(\bar{H} - 1)$. The ratio c/a equals γ/α from a) and is here called μ.

We remark that $L_c = a/b$ and $H_c = c/d$ could also be found from the *stationary points* of the ODE system, where $H' = L' = 0$. The stationary points are in our case $H = L = 0$ and $H = c/d$ and $L = a/b$.

The initial conditions become $\bar{H}(0) = H_0/H_c = H_0 d/c = \beta/\gamma = \nu$, and $\bar{L}(0) = L_0/L_c = L_0 b/a = \delta/\alpha = \omega$.

The resulting dimensionless problem takes the form

$$\frac{d\bar{H}}{d\bar{t}} = \bar{H}(1 - \bar{L}), \tag{66}$$

$$\frac{d\bar{L}}{d\bar{t}} = \mu\bar{L}(\bar{H} - 1) = \gamma\alpha^{-1}\bar{L}(\bar{H} - 1), \tag{67}$$

$$\bar{H}(0) = \nu = \beta/\gamma, \tag{68}$$

$$\bar{L}(0) = \omega = \delta/\alpha, \tag{69}$$

with

$$\mu = \frac{c}{a}, \quad \nu = H_0\frac{d}{c}, \quad \omega = L_0\frac{b}{a}\,.$$

The unknowns $\bar{H}$ and $\bar{L}$ now have less intuitive scalings,

$$\bar{H} = \frac{Hd}{c}, \quad \bar{L} = \frac{Lb}{a},$$

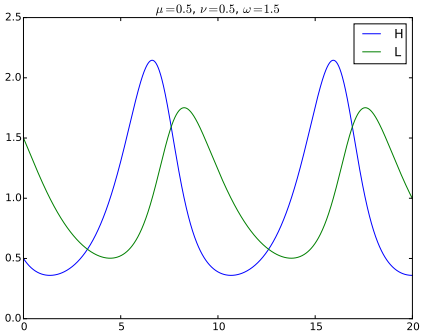
while time is measured in the units based on the exponential growth due to births and deaths of preys (a). The number of dimensionless parameters is one less since we have one more scale (for L_c) at our disposal. Simplicity in one initial conditions in a) is exchanged with more simplicity in the ODEs, which now have only one dimensionless parameter. Note that ν and ω must be different from unity to avoid $\bar{H} \neq 0$ and $\bar{L} \neq 0$ because of the factors $1 - L$ and $H - 1$ in the equations that can make $\bar{H}' = 0$ and $\bar{L}' = 0$.

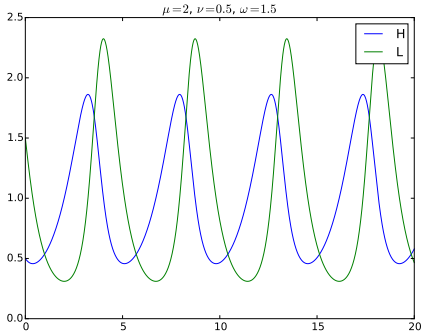
d) Compute with the scaled model from c) and create plots to illustrate the typical solutions.

Solution. Modified `simulate` and `demo` go as follows.

```
def simulate2(mu, nu, omega, T=20):
    def f(u, t, mu):
        H, L = u
        return [H*(1-L), mu*L*(H-1)]


    t = np.linspace(0, T, 1501)
    solver = odespy.RK4(f, f_args=(mu,))
```

```
    solver.set_initial_condition([nu, omega])
    u, t = solver.solve(t)
    H = u[:,0]
    L = u[:,1]

    plt.figure()
    global fig_counter
    fig_counter += 1
    plt.plot(t, H, t, L)
    plt.legend(['H', 'L'])
    plt.title(r'$\mu=%g$, $\nu=%g$, $\omega=%g$' % (mu, nu, omega))
    plt.savefig('tmp%d.png' % fig_counter)
    plt.savefig('tmp%d.pdf' % fig_counter)
    return H, L, t

def demo2():
    simulate2(mu=0.5, nu=0.5, omega=1.5, T=20)
    simulate2(mu=2, nu=0.5, omega=1.5, T=20)
```





Filename: `predator_prey`.

Exercise 13: Simulate the pressure drop in the atmosphere

We consider the models for atmospheric pressure in Section 9. Make a program with three functions,

- one computing the pressure $p(z)$ using a seven-layer model and varying L,
- one computing $p(z)$ using a seven-layer model, but with constant temperature in each layer, and
- one computing $p(z)$ based on the one-layer model.

How can these implementations be verified? Should ease of verification impact how you code the functions? Compare the three models in a plot. Filename: `atmospheric_pressure`.

Exercise 14: Make a program for vertical motion in a fluid

Implement the Stokes' drag model (40) and the quadratic drag model (43) from Section 11, using the Crank-Nicolson scheme and a geometric mean for $|v|v$ as explained, and assume constant fluid density. At each time level, compute

the Reynolds number Re and choose the Stokes' drag model if $\mathrm{Re} < 1$ and the quadratic drag model otherwise.

The computation of the numerical solution should take place either in a stand-alone function or in a solver class that looks up a problem class for physical data. Create a module and equip it with pytest/nose compatible test functions for automatically verifying the code.

Verification tests can be based on

- the terminal velocity (see Section 11),
- the exact solution when the drag force is neglected (see Section 11),
- the method of manufactured solutions (see Section **??**) combined with computing convergence rates (see Section **??**).

Use, e.g., a quadratic polynomial for the velocity in the method of manufactured solutions. The expected error is $\mathcal{O}(\Delta t^2)$ from the centered finite difference approximation and the geometric mean approximation for $|v|v$.

A solution that is linear in t will also be an exact solution of the discrete equations in many problems. Show that this is true for linear drag (by adding a source term that depends on t), but not for quadratic drag because of the geometric mean approximation. Use the method of manufactured solutions to add a source term *in the discrete equations for quadratic drag* such that a linear function of t is a solution. Add a test function for checking that the linear function is reproduced to machine precision in the case of both linear and quadratic drag.

Apply the software to a case where a ball rises in water. The buoyancy force is here the driving force, but the drag will be significant and balance the other forces after a short time. A soccer ball has radius 11 cm and mass 0.43 kg. Start the motion from rest, set the density of water, ϱ, to 1000 kg/m^3, set the dynamic viscosity, μ, to 10^{-3} Pa s, and use a drag coefficient for a sphere: 0.45. Plot the velocity of the rising ball. Filename: `vertical_motion`.

Project 15: Simulate parachuting

The aim of this project is to develop a general solver for the vertical motion of a body with quadratic air drag, verify the solver, apply the solver to a skydiver in free fall, and finally apply the solver to a complete parachute jump.

All the pieces of software implemented in this project should be realized as Python functions and/or classes and collected in one module.

a) Set up the differential equation problem that governs the velocity of the motion. The parachute jumper is subject to the gravity force and a quadratic drag force. Assume constant density. Add an extra source term to be used for program verification. Identify the input data to the problem.

b) Make a Python module for computing the velocity of the motion. Also equip the module with functionality for plotting the velocity.

Hint 1. Use the Crank-Nicolson scheme with a geometric mean of $|v|v$ in time to linearize the equation of motion with quadratic drag.

Hint 2. You can either use functions or classes for implementation. If you choose functions, make a function `solver` that takes all the input data in the problem as arguments and that returns the velocity (as a mesh function) and the time mesh. In case of a class-based implementation, introduce a problem class with the physical data and a solver class with the numerical data and a `solve` method that stores the velocity and the mesh in the class.

Allow for a time-dependent area and drag coefficient in the formula for the drag force.

c) Show that a linear function of t does not fulfill the discrete equations because of the geometric mean approximation used for the quadratic drag term. Fit a source term, as in the method of manufactured solutions, such that a linear function of t is a solution of the discrete equations. Make a test function to check that this solution is reproduced to machine precision.

d) The expected error in this problem goes like Δt^2 because we use a centered finite difference approximation with error $\mathcal{O}(\Delta t^2)$ and a geometric mean approximation with error $\mathcal{O}(\Delta t^2)$. Use the method of manufactured solutions combined with computing convergence rate to verify the code. Make a test function for checking that the convergence rate is correct.

e) Compute the drag force, the gravity force, and the buoyancy force as a function of time. Create a plot with these three forces.

Hint. You can either make a function `forces(v, t, plot=None)` that returns the forces (as mesh functions) and `t`, and shows a plot on the screen and also saves the plot to a file with name stored in `plot` if `plot` is not `None`, or you can extend the solver class with computation of forces and include plotting of forces in the visualization class.

f) Compute the velocity of a skydiver in free fall before the parachute opens.

Hint. Meade and Struthers [2] provide some data relevant to skydiving[11]. The mass of the human body and equipment can be set to 100 kg. A skydiver in spread-eagle formation has a cross-section of 0.5 m^2 in the horizontal plane. The density of air decreases with altitude, but can be taken as constant, 1 kg/m^3, for altitudes relevant to skydiving (0-4000 m). The drag coefficient for a man in upright position can be set to 1.2. Start with a zero velocity. A free fall typically has a terminating velocity of 45 m/s. (This value can be used to tune other parameters.)

[11] `http://en.wikipedia.org/wiki/Parachuting`

g) The next task is to simulate a parachute jumper during free fall and after the parachute opens. At time t_p, the parachute opens and the drag coefficient and the cross-sectional area change dramatically. Use the program to simulate a jump from $z = 3000$ m to the ground $z = 0$. What is the maximum acceleration, measured in units of g, experienced by the jumper?

Hint. Following Meade and Struthers [2], one can set the cross-section area perpendicular to the motion to 44 m^2 when the parachute is open. Assume that it takes 8 s to increase the area linearly from the original to the final value. The drag coefficient for an open parachute can be taken as 1.8, but tuned using the known value of the typical terminating velocity reached before landing: 5.3 m/s. One can take the drag coefficient as a piecewise constant function with an abrupt change at t_p. The parachute is typically released after $t_p = 60$ s, but larger values of t_p can be used to make plots more illustrative.
Filename: `parachuting`.

Exercise 16: Formulate vertical motion in the atmosphere

Vertical motion of a body in the atmosphere needs to take into account a varying air density if the range of altitudes is many kilometers. In this case, ϱ varies with the altitude z. The equation of motion for the body is given in Section 11. Let us assume quadratic drag force (otherwise the body has to be very, very small). A differential equation problem for the air density, based on the information for the one-layer atmospheric model in Section 9, can be set up as

$$p'(z) = -\frac{Mg}{R^*(T_0 + Lz)}p, \tag{70}$$

$$\varrho = p\frac{M}{R^*T}\,. \tag{71}$$

To evaluate $p(z)$ we need the altitude z. From the principle that the velocity is the derivative of the position we have that

$$z'(t) = v(t), \tag{72}$$

where v is the velocity of the body.

Explain in detail how the governing equations can be discretized by the Forward Euler and the Crank-Nicolson methods. Discuss pros and cons of the two methods. Filename: `falling_in_variable_density`.

Exercise 17: Simulate vertical motion in the atmosphere

Implement the Forward Euler or the Crank-Nicolson scheme derived in Exercise 16. Demonstrate the effect of air density variation on a falling human, e.g., the famous fall of Felix Baumgartner[12]. The drag coefficient can be set to 1.2. Filename: `falling_in_variable_density`.

[12] `http://en.wikipedia.org/wiki/Felix_Baumgartner`

Problem 18: Compute $y = |x|$ by solving an ODE

Consider the ODE problem

$$y'(x) = \begin{cases} -1, & x < 0, \\ 1, & x \geq 0 \end{cases} \qquad x \in (-1, 1], \quad y(1-) = 1,$$

which has the solution $y(x) = |x|$. Using a mesh $x_0 = -1$, $x_1 = 0$, and $x_2 = 1$, calculate by hand y_1 and y_2 from the Forward Euler, Backward Euler, Crank-Nicolson, and Leapfrog methods. Use all of the former three methods for computing the y_1 value to be used in the Leapfrog calculation of y_2. Thereafter, visualize how these schemes perform for a uniformly partitioned mesh with $N = 10$ and $N = 11$ points. Filename: `signum`.

Problem 19: Simulate fortune growth with random interest rate

The goal of this exercise is to compute the value of a fortune subject to inflation and a random interest rate. Suppose that the inflation is constant at i percent per year and that the annual interest rate, p, changes randomly at each time step, starting at some value p_0 at $t = 0$. The random change is from a value p^n at $t = t_n$ to $p_n + \Delta p$ with probability 0.25 and $p_n - \Delta p$ with probability 0.25. No change occurs with probability 0.5. There is also no change if p^{n+1} exceeds 15 or becomes below 1. Use a time step of one month, $p_0 = i$, initial fortune scaled to 1, and simulate 1000 scenarios of length 20 years. Compute the mean evolution of one unit of money and the corresponding standard deviation. Plot the mean curve along with the mean plus one standard deviation and the mean minus one standard deviation. This will illustrate the uncertainty in the mean curve.

Hint 1. The following code snippet computes p^{n+1}:

```
import random

def new_interest_rate(p_n, dp=0.5):
    r = random.random()  # uniformly distr. random number in [0,1)
    if 0 <= r < 0.25:
        p_np1 = p_n + dp
    elif 0.25 <= r < 0.5:
        p_np1 = p_n - dp
    else:
        p_np1 = p_n
    return (p_np1 if 1 <= p_np1 <= 15 else p_n)
```

Hint 2. If $u_i(t)$ is the value of the fortune in experiment number i, $i = 0, \ldots, N-1$, the mean evolution of the fortune is

$$\bar{u}(t) = \frac{1}{N} \sum_{i=0}^{N-1} u_i(t),$$

and the standard deviation is

$$s(t) = \sqrt{\frac{1}{N-1}\left(-(\bar{u}(t))^2 + \sum_{i=0}^{N-1}(u_i(t))^2\right)}.$$

Suppose $u_i(t)$ is stored in an array `u`. The mean and the standard deviation of the fortune is most efficiently computed by using two accumulation arrays, `sum_u` and `sum_u2`, and performing `sum_u += u` and `sum_u2 += u**2` after every experiment. This technique avoids storing all the $u_i(t)$ time series for computing the statistics.
Filename: `random_interest`.

Exercise 20: Simulate a population in a changing environment

We shall study a population modeled by (3) where the environment, represented by r and f, undergoes changes with time.

a) Assume that there is a sudden drop (increase) in the birth (death) rate at time $t = t_r$, because of limited nutrition or food supply:

$$r(t) = \begin{cases} \varrho, & t < t_r, \\ \varrho - A, & t \geq t_r, \end{cases}$$

This drop in population growth is compensated by a sudden net immigration at time $t_f > t_r$:

$$f(t) = \begin{cases} 0, & t < t_f, \\ f_0, & t \geq t_a, \end{cases}$$

Start with ϱ and make $A > \varrho$. Experiment with these and other parameters to illustrate the interplay of growth and decay in such a problem.

b) Now we assume that the environmental conditions changes periodically with time so that we may take

$$r(t) = \varrho + A\sin\left(\frac{2\pi}{P}t\right).$$

That is, the combined birth and death rate oscillates around ϱ with a maximum change of $\pm A$ repeating over a period of length P in time. Set $f = 0$ and experiment with the other parameters to illustrate typical features of the solution.
Filename: `population.py`.

Exercise 21: Simulate logistic growth

Solve the logistic ODE (4) using a Crank-Nicolson scheme where $(u^{n+\frac{1}{2}})^2$ is approximated by a *geometric mean*:

$$(u^{n+\frac{1}{2}})^2 \approx u^{n+1}u^n.$$

This trick makes the discrete equation linear in u^{n+1}. Filename: `logistic_CN`.

Exercise 22: Rederive the equation for continuous compound interest

The ODE model (7) was derived under the assumption that r was constant. Perform an alternative derivation without this assumption: 1) start with (5); 2) introduce a time step Δt instead of m: $\Delta t = 1/m$ if t is measured in years; 3) divide by Δt and take the limit $\Delta t \to 0$. Simulate a case where the inflation is at a constant level I percent per year and the interest rate oscillates: $r = -I/2 + r_0 \sin(2\pi t)$. Compare solutions for $r_0 = I, 3I/2, 2I$. Filename: `interest_modeling`.

Exercise 23: Simulate the deformation of a viscoelastic material

Stretching a rod made of polymer will cause deformations that are well described with a Kelvin-Voigt material model (49). At $t = 0$ we apply a constant force $\sigma = \sigma_0$, but at $t = t_1$, we remove the force so $\sigma = 0$. Compute numerically the corresponding strain (elongation divided by the rod's length) and visualize how it responds in time.

Hint. To avoid finding proper values of the E and η parameters for a polymer, one can scale the problem. A common dimensionless time is $\bar{t} = tE/\eta$. Note that ε is already dimensionless by definition, but it takes on small values, say up to 0.1, so we introduce a scaling: $\bar{u} = 10\varepsilon$ such that $\bar{u}$ takes on values up to about unity.

Show that the material model then takes the form $\bar{u}' = -\bar{u} + 10\sigma(t)/E$. Work with the dimensionless force $F = 10\sigma(t)/E$, and let $F = 1$ for $\bar{t} \in (0, \bar{t}_1)$ and $F = 0$ for $\bar{t} \geq \bar{t}_1$. A possible choice of t_1 is the characteristic time η/E, which means that $\bar{t}_1 = 1$.

Solution. Inserting the scaling $\bar{u} = 10\varepsilon$ and $\bar{t} = tE/\eta$ in the ODE for ε results in

$$\frac{E}{10\eta}\frac{d\bar{u}}{d\bar{t}} = -\frac{E}{\eta 10}\bar{u} + \frac{\sigma(t)}{\eta}\,.$$

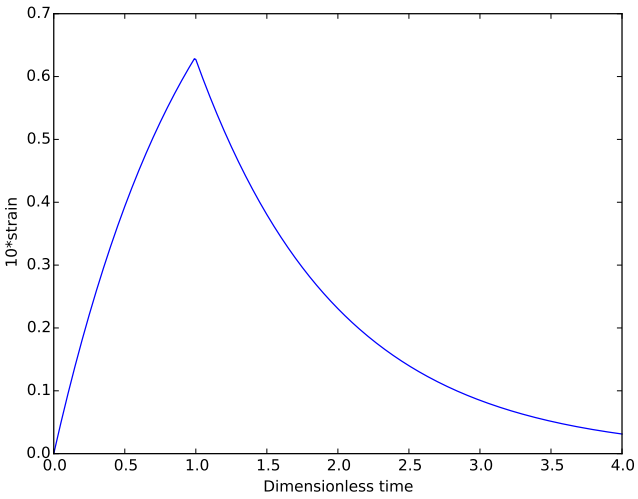
This reduces to

$$\frac{d\bar{u}}{d\bar{t}} = -\bar{u} + \frac{10\sigma}{E}\,.$$

We can use the module `decay_vc` for solving $u' = -a(t)u + b(t)$:

```
# Scaled model: u' = -u + b(t), b=1 for t < 1 else 0
eps, t = solver(I=0, a=lambda t: 1,
                b=lambda t: 1 if t < 1 else 0,
                T=4, dt=0.01, theta=0.5)

import matplotlib.pyplot as plt
```

```
plt.plot(t, eps)
plt.xlabel('Dimensionless time'); plt.ylabel('10*strain')
plt.savefig('tmp.png');  plt.savefig('tmp.pdf')
plt.show()
```



Filename: `KelvinVoigt`.

References

[1] H. P. Langtangen and G. K. Pedersen. *Scaling of Differential Equations*. SimulaSpringerBrief. Springer, 2015. `http://tinyurl.com/qfjgxmf/web`.

[2] D. B. Meade and A. A. Struthers. Differential equations in the new millenium: the parachute problem. *International Journal of Engineering Education*, 15(6):417–424, 1999.

Index