

Chapter 6

Model Solution –Numerical Methods

The analytical solutions discussed in the previous chapter have the advantage of simplicity, as they can be solved using a hand calculator or in a spreadsheet. However, for many real-world problems, the differential equations are too difficult to solve analytically, or it may become impractical to do so. This often arises for instance because non-linear terms are included in the equations, or because the boundary conditions or forcing functions are a variable function of time. When this is the case, a numerical method must be used.

Numerical solutions are approximations where the continuous model equations are approached in discrete steps, both in time, and for spatial models, also in space (Fig. 6.1). The advantage of numerical methods is that there is virtually no limit to the complexity of problems that can be solved, but the price to be paid for this generality is the introduction of a new kind of errors, so-called numerical errors which must be controlled.

Generally, numerical solutions are obtained by writing a computer programme. As numerical models generally proceed by stepping through time, a model calculation is often called a simulation, or we talk about running a model.

6.1 Taylor Expansion

One important mathematical equation, the Taylor expansion, forms the backbone of many numerical methods, so we will discuss this equation first.

Taylor expansion allows evaluating, *for small h* , the unknown function value at some point $x + h$, when all the derivatives of the function at x are known. The Taylor

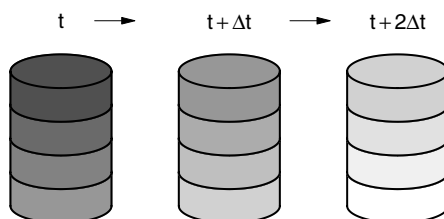


Fig. 6.1 Numerical solutions proceed by subdividing time and space in discrete steps

expansion function is a polynomial, which contains the function value at x (0th order derivative), the slope of the function at x (1st order derivative), the curvature (2nd order derivative), and higher-order derivatives of $f(x)$.

Intuitively, one may compare the derivation of Taylor expansion with forecasting the weather. Assume somebody asks you what the weather will be like tomorrow. The first best guess is to assume that the weather will not change during the day so that tomorrow will have the same weather as today.

Taylor-expansion-wise we assume that the function value did not change in the interval h such that the new value $f(x + h)$ is approximated as (Fig. 6.2A):

$$f(x + h) \approx f(x) \tag{6.1}$$

Of course this is only exact *if* the function $f(x)$ remains constant over the interval from x to $x + h$. If function $f(x)$ is constant, its first-order derivative $f' = df/dx$ is 0.

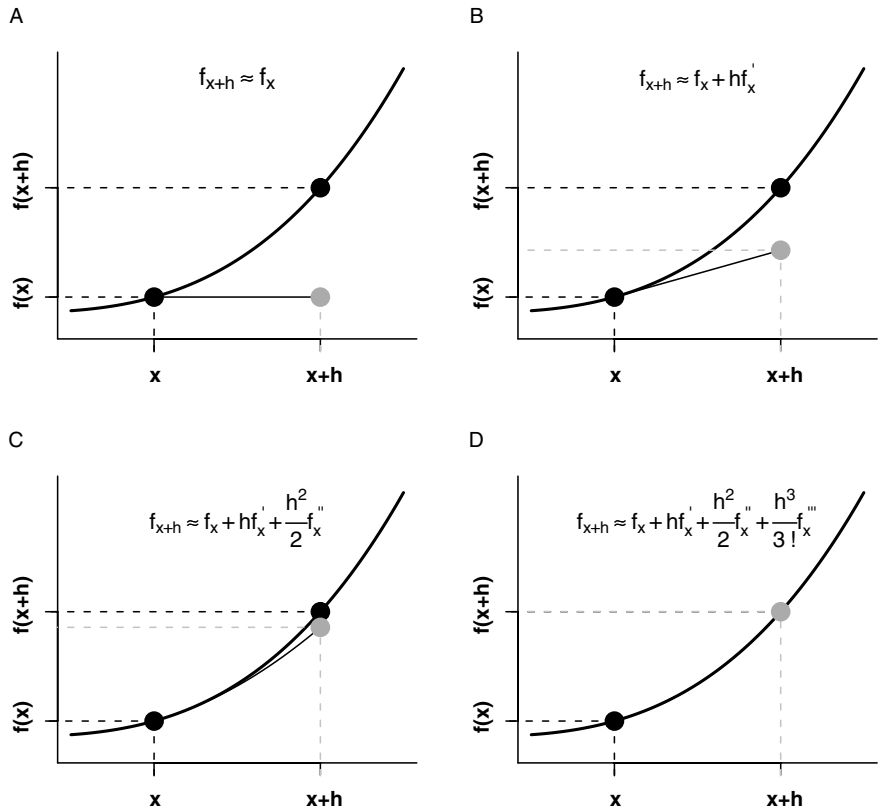


Fig. 6.2 The Taylor expansion of different order to predict a function value at $x+h$. **A.** uses only the function value at x (0th order derivative). **B.** also uses the rate of change (1st order derivative). **C.** includes the 2nd order derivative. **D.** up to 3rd order derivative. The grey dot denotes the estimated value at $x + h$; the black dot is the true function value at x and $x + h$

If the weather *does* change during the day, then our statement that tomorrow will be the same as today does not hold. We might argue that, as today it is getting warmer already (the rate of change of the weather), the weather will be similar as today, but somewhat warmer.

Similarly, in Taylor expansion, a better estimate is given by taking the (known) derivative into account (Fig. 6.2B):

$$f(x + h) \approx f(x) + h \cdot f'(x) \quad (6.2)$$

This formula is exact only if the derivative, $f'(x)$ remains constant in the interval h , i.e. if the second-order derivative, $f''(x)$ is zero.

Even better is to adjust for the change of this derivative (second order derivatives) (Fig. 6.2C).

$$f(x + h) \approx f(x) + h \cdot f'(x) + \frac{h^2}{2} f''(x) \quad (6.3)$$

Repeating this argument, taking ever-higher derivatives into account, leads to the Taylor series:

$$f(x + h) = f(x) + h \cdot f'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(x) + \dots + \frac{h^n}{n!} f^n(x) + \dots \quad (6.4)$$

and where $n!$ is the factorial $(n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1)$ and $f^n(x)$ denotes the n^{th} derivative of f at point x .

Similarly we may write (replacing h with $-h$) :

$$f(x - h) = f(x) - h \cdot f'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(x) + \dots \quad (6.5)$$

In the next sections we will see how Taylor expansion can be used to numerically integrate differential equations in time, or to approximate spatial derivatives.

6.2 Numerical Approximation and Numerical Errors

How is this Taylor expansion used?

Consider a very small h (< 1), say $h = 0.02$. The powers of h are: $h^2 = 0.0004$, $h^3 = 0.000008$, $h^4 = 0.00000016$,... $h^8 = 2.5610^{-14}$. The terms $n! = (1 \cdot 2 \cdot \dots \cdot n)$ by which the powers of h are divided, increase rapidly with n (1, 2, 6, 24, 120, 720, ...)

As the terms of the Taylor series become smaller and smaller for higher powers of h , at some point we may consider their value to be negligible.

Numerical methods typically proceed by truncating the Taylor series at some point, arguing that the error that is introduced by doing so is small enough to be unimportant. It is this truncation that is the reason for so-called numerical errors.

We write that a method is n -th order accurate, if the largest power of h included is h^n . Thus, the largest truncation error made at each step is proportional to h^{n+1} . This also means that halving the step h will divide the error by 2^{n+1} , i.e. by 8 for a second order method.

Why should we bother about the (small) errors in numerical solutions? First, of course, it is never pleasing to make errors. Scientists want to be accurate, and this alone is a compelling reason to try and reduce numerical errors to an acceptable level. There is a second reason, however, which relates to the (numerical) *stability* of the solution. When a small error is made in one time step, this error is sometimes compensated by another error in the following time step. For instance, if the first time step leads to a small underestimation of biomass at the end of the step, this may be compensated if this too-small biomass slightly overestimates the rate of increase, such that after two time steps the approximation is still reasonably close to the true solution of the model. If this is the case, we are lucky, and will have no problems of numerical stability. However, the reverse can also happen: small errors in one time step lead to magnification of errors in the next step, and so on... very soon the model solution will be far off-track (see Fig. 6.3A) or will start oscillating wildly (Fig. 6.3B). Particular problems of numerical stability may also arise if small errors lead to slightly negative values for state variables such as concentrations or biomass that, by definition, must be positive (Fig. 6.3B). You would be surprised how fast model algae can grow on negative model nitrate! Such negative values for a state variable may easily lead to growing oscillations in the numerical model solution, which have nothing to do with the real model solution.

One notorious example of truncation error, which is linked to the *spatial* approximation, is called 'numerical dispersion' and will be discussed at the end of this chapter.

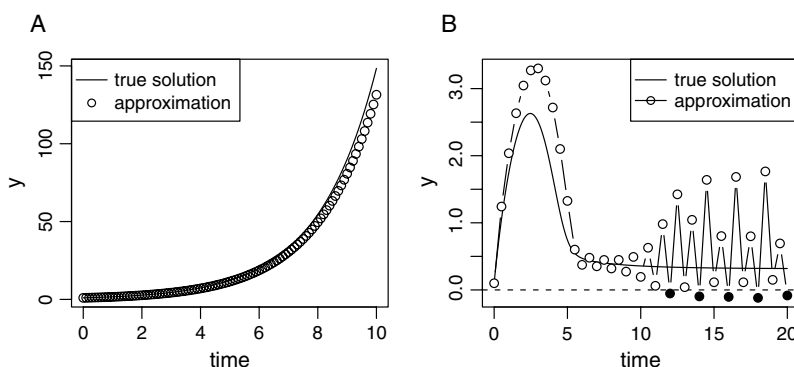


Fig. 6.3 Two examples of numerical errors. **A.** In this (exponential growth) model the error is small at first, but it is successively magnified; after 10 days, the approximated solution clearly deviates from the true solution. **B.** In this model, after 5 days of simulation the numerical solution starts to oscillate with ever increasing magnitude. These oscillations lead to negative values (*black dots*) after 10 days

6.3 Numerical Integration in Time – Basics

When a dynamic differential equation is solved numerically, the method jumps from one point in time to the next:

$$t_0 \Rightarrow t_1 = t_0 + \Delta t \Rightarrow t_2 = t_1 + \Delta t \Rightarrow t_3 \Rightarrow t_4 \Rightarrow \dots \Rightarrow t_n.$$

and where Δt is the time step. Note that this time step is not necessarily constant; some methods allow it to vary through time. For simplicity of notation, we will however explain the basic methods assuming a constant time step.

The model starts off with the specification of the initial conditions, i.e. the start values of the state variables C , at time t_0 (Fig. 6.4). Then, at each point in time (t), the rates of changes of the state variables, dC/dt are estimated. After this, the numerical integration method uses the values of the state variables plus their rates of changes to estimate the values of the state variables at the next point in time ($t+\Delta t$). This procedure is repeated until the total time has been simulated.

To estimate the values at the next time step, an update formula is needed.

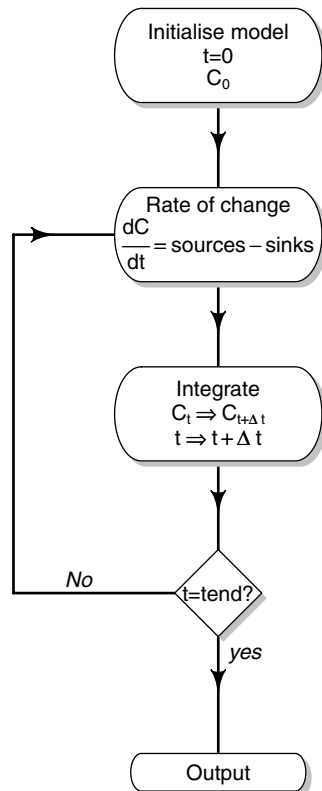


Fig. 6.4 Schematic representation of the numerical integration method

$$C^{t+\Delta t} = f\left(C^t, \frac{\partial C^t}{\partial t}, \Delta t\right) \quad (6.6)$$

6.3.1 Euler Integration

The simplest update formula is to use Taylor equation (Eq. 6.4), with $h = \Delta t$ and $f(x) = C^t$, and to ignore all terms of order higher than one:

$$\begin{aligned} C^{t+\Delta t} &= C^t + \Delta t \cdot \frac{\partial C^t}{\partial t} + O(\Delta t) \\ C^{t+\Delta t} &\approx C^t + \Delta t \cdot \frac{\partial C^t}{\partial t} \end{aligned} \quad (6.7)$$

where $O(\Delta t)$ stands for the error we make by truncating the Taylor series.

This is a reasonable approximation only for small enough Δt , as then the terms beyond linear are unimportant (i.e. for $\Delta t = 0.01$, $\Delta t^2 = 1e^{-4}$, $\Delta t^3 = 1e^{-6}$; whereas for $\Delta t = 1$, $\Delta t^2 = \Delta t^3 = 1$).

This method is called *forward differencing* or Euler integration and said to be first-order accurate, as we have ignored all terms of order higher than 1.

As an example of how such procedure might work in practice, let's consider a very simple algal growth model:

$$\begin{aligned} \frac{dALGAE}{dt} &= \text{Photosynthesis-Respiration-Grazing} \\ ALGAE^{t0} &= A_0 \end{aligned} \quad (6.8)$$

The numerical solution of this model starts at t_0 by setting the initial condition: $ALGAE = A_0$.

Using this value, and the parameters, the rate of change $dALGAE/dt$ at t_0 is calculated. Both the value of $ALGAE^{t0}$ and its rate of change are then used to calculate the value at $t_0 + \Delta t$.

$$\begin{aligned} ALGAE^{t0} &= A_0 \\ \left. \frac{dALGAE}{dt} \right|^{t0} &= \text{Photosynthesis}^{t0} - \text{Respiration}^{t0} - \text{Grazing}^{t0} \\ ALGAE^{t1} &= ALGAE^{t0} + \Delta t \cdot \left. \frac{dALGAE}{dt} \right|^{t0} \\ t_1 &= t_0 + \Delta t \end{aligned} \quad (6.9)$$

This procedure is then repeated for successive time steps.

$$\begin{aligned}
 \dots \\
 \left. \frac{d\text{ALGAE}}{dt} \right|^{tI} &= \text{Photosynthesis}^{tI} - \text{Respiration}^{tI} - \text{Grazing}^{tI} \\
 \text{ALGAE}^{tI+1} &= \text{ALGAE}^{tI} + \Delta t \cdot \left. \frac{d\text{ALGAE}}{dt} \right|^{tI}
 \end{aligned} \tag{6.10}$$

Unfortunately the first-order accuracy of the Euler method is not good enough for all applications.

By truncating the Taylor series at order 2, we effectively assume that the rate of change remains constant in the interval Δt . If this is not the case, then the method will not be very accurate (see Fig. 6.5A).

One remedy to obtain better accuracy is reducing the time step Δt . However, when the model's properties are really bad, it is possible that we will need to make the time step so small that the model never gets anywhere!

There are several other solutions to this problem, but before we (briefly) discuss them, we have a look at the criteria to which integration routines should obey.

6.3.2 Criteria for Numerical Integration

There are several criteria that determine the quality of the integrator. Here we discuss accuracy, stability, speed and memory requirements.

Accuracy is a measure for the correctness of the solution. As explained above, the discretization error which is due to the numerical approximation and made at each time step can be quantified by comparing the solution with the Taylor expansion.

$$C(t + \Delta t) = C(t) + \Delta t \times \frac{\partial C}{\partial t} + \frac{1}{2} \Delta t^2 \times \frac{\partial^2 C}{\partial t^2} + \dots \tag{6.11}$$

and we write that a method is of n th order, if the truncation error made at each step is proportional to Δt^{n+1} . Generally, higher order methods attain higher accuracy at larger time steps.

The *stability* of a method refers to its potential to lead to increasing oscillations between consecutive solution points (see Fig. 6.3B). Some models are especially prone to this type of problems. Clearly, we want an integration routine to be as accurate and stable as possible!

However, we also want to see the results within a reasonable time frame. This makes *speed* a third, albeit highly subjective, criterion. Speed depends on (1) the size of the time step, Δt , taken and (2) the number of function evaluations needed to advance from time t to time $t+\Delta t$. Euler, the computationally most simple of all algorithms requires only one function evaluation, whereas higher-order integration routines (see below) may require more evaluations. However, these methods can often run with a larger time step.

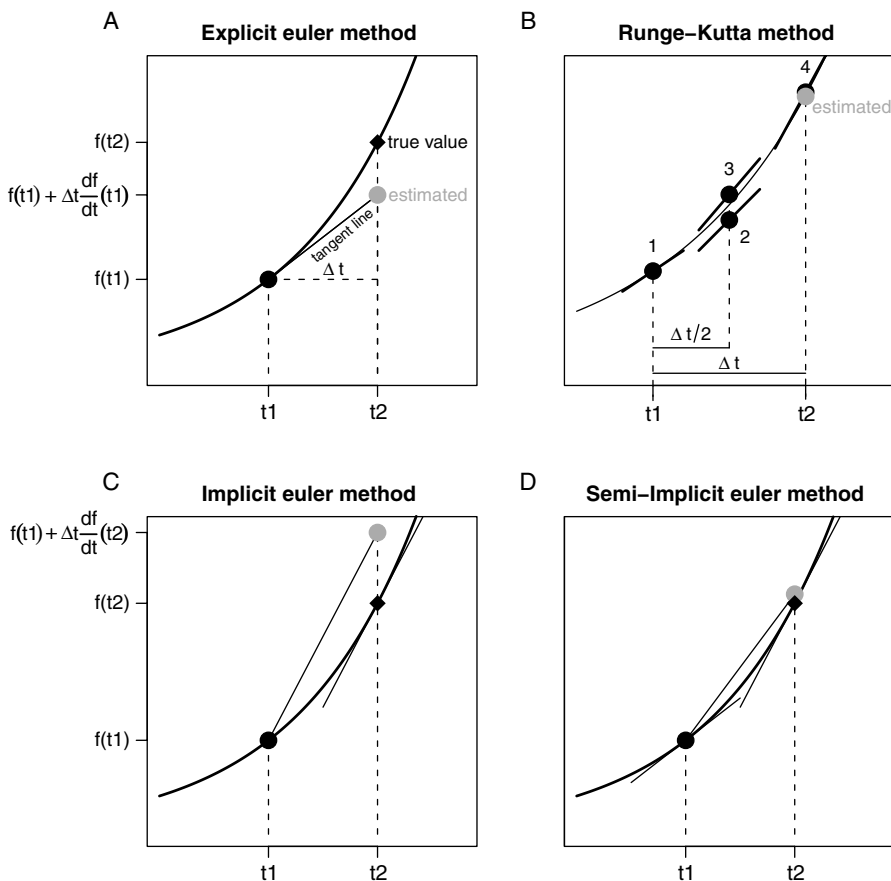


Fig. 6.5 **A.** The explicit Euler integration method. The function value at time t_2 is estimated based on the value at time t_1 and the rate of change (as denoted by the tangent line). It is assumed that the rate of change remains constant in the time step between t_1 and t_2 . As this is not the case, the estimated value considerably underestimates the true value. **B.** The 4-th order Runge-Kutta method. One function evaluation at the start point (1), two evaluations at the mid-point (2,3), and one evaluation at the end point (4) are combined to provide the final projected value (grey dot). **C.** The implicit Euler integration method uses the estimate of the rate of change at the next time step, t_2 . **D.** The semi-implicit method combines both the explicit and implicit approach

Finally, the more complex the integration routine, the more *memory* it requires. The reason for this is that for every time step more information (basically: higher and higher order derivatives in the Taylor expansion) must be remembered. For some large-scale applications with constituents described in many spatial cells, available memory may become a limiting factor.

A particularly hard-to-integrate type of problems are so-called ‘stiff’ sets of equations. The problem typically occurs when processes are modelled that have different inherent time scales. When one tries to model dynamics of bacterial population dynamics (typical time scales hours to days), together with a forcing (e.g. organic matter input) with a seasonal or year-to-year variation, then it can

be expected that the long-term result will mostly be influenced by the long-term varying factor. Bacterial populations, at any particular moment, will be more or less in equilibrium with the organic matter forcing that varies only slowly. Yet almost all computer resources needed to solve the model will go to the fast process. When using simple Euler integration, e.g., one will probably need a time step of minutes in order to simulate the bacterial dynamics correctly, while the process of real interest varies over months!

To comply with these –sometimes conflicting– requirements, more complex integration methods have been devised. These are briefly introduced in the next chapters.

Basically the improvements include:

- Performing extra function evaluations and interpolating between them. Some Runge-Kutta methods (Section 6.3.3) are based on this principle.
- Performing the simulation with a flexible time step. These integration routines use a finer subdivision of time only when it is actually needed, i.e. when environmental changes are most rapid. The principle is explained using 5th order Runge-Kutta (Section 6.3.4)
- Implicit methods of solution. This is a frequently used remedy against instability (but it comes at the cost of reduced accuracy).

Most modern integration routines somehow combine these features.

6.3.3 Interpolation Methods – 4th Order Runge-Kutta (**)

Runge-Kutta methods use extra evaluations of the differential equations at various positions in time, and interpolate between these values.

Most commonly used is the 4th order Runge-Kutta method, which requires four evaluations per time step Δt (Fig. 6.5 B). The methods' equations are the following:

$$\begin{aligned}
 k_1 &= \Delta t \cdot f(t_i, \mathbf{C}_i) \\
 k_2 &= \Delta t \cdot f(t_i + \Delta t/2, \mathbf{C}_i + k_1/2) \\
 k_3 &= \Delta t \cdot f(t_i + \Delta t/2, \mathbf{C}_i + k_2/2) \\
 k_4 &= \Delta t \cdot f(t_i + \Delta t, \mathbf{C}_i + k_3) \\
 \mathbf{C}_{i+1} &= \mathbf{C}_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4,
 \end{aligned} \tag{6.12}$$

where the k_i 's are the intermediate evaluations of the differential equations, and the last equation estimates the state variable's value at the next time step by interpolation. This method has a truncation error in the order of Δt^5 , i.e. it is a 4th order method.

Similarly as the Euler method, 4th order Runge-Kutta integration uses a fixed time step. For certain applications, using a fixed time step is not a good idea. For instance, a model that is applied over certain periods of time where changes occur very rapidly, and other periods where there is hardly any change at all, the size of (the constant) Δt must be chosen such as to perform well in the period where

changes are rapid. This may lead to unnecessary computation during periods where the model is relatively smooth.

6.3.4 Flexible Time Step Methods –5th Order Runge-Kutta (**)

If the solution varies rapidly in one region of the integration interval whilst staying nearly constant in another region, then a method that automatically adapts the time step is much better suited than the constant step size methods (as in previous section). Integration routines that use an adaptive step size, consist of two parts:

- the stepper routine which actually performs the integration, and
- a quality control routine that checks if the step taken is acceptable or not.

The time step is taken as large as possible, but such that the precision of the integration remains within predefined bounds.

The 5th order Runge-Kutta method for instance, combines the robustness of 4th order Runge-Kutta with the elegance of a flexible time step. The idea is to adapt Δt to some desired accuracy Δ_0 .

First, a ‘reference’ error Δ_1 is estimated from the difference between two Runge-Kutta applications, one with a large step Δt_1 , the other with two successive steps, with time step half the large stepsize. As in 4th order methods, the error Δ scales as Δt^5 and this is used to estimate the step size step Δt_0 producing the desired accuracy Δ_0 as:

$$\Delta t_0 = \Delta t_1 \cdot \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \quad (6.13)$$

6.3.5 Implicit and Semi-Implicit Integration Routines (**)

Implicit methods are less accurate than explicit methods (all the methods discussed before), but are much more stable.

Whereas in explicit methods, the new value of the state variable at time $t+\Delta t$ is calculated entirely in terms of the old value at time t , in *implicit* methods, the derivative is estimated *at the next time step* ($t+\Delta t$) (Fig. 6.5C) as follows:

$$\mathbf{C}^{t+\Delta t} = \mathbf{C}^t + \Delta t \cdot \left. \frac{\partial \mathbf{C}}{\partial t} \right|_{t+\Delta t} \quad (6.14)$$

As this derivative is a function of the unknown concentration at the next time step, it is itself unknown, and thus integration boils down to finding a solution of the (often nonlinear) set of equations. It is not necessary to know the fine details of how this is done; the important part is that, solving these equations requires the creation and inversion of an ($n*n$) matrix involving the Jacobian matrix (see

Appendix B.3.4), where n equals the number of state variables. For models with many state variables, this can be quite time-consuming.

In *Semi-implicit methods*, the derivative is approached partly at the current, partly at the next time step ($t+\Delta t$) (Fig. 6.5.D) as follows:

$$\frac{\mathbf{C}^{t+\Delta t} - \mathbf{C}^t}{\Delta t} = \gamma \cdot \left. \frac{\partial \mathbf{C}}{\partial t} \right|_t + (1 - \gamma) \cdot \left. \frac{\partial \mathbf{C}}{\partial t} \right|_{t+\Delta t} \quad (6.15)$$

and where γ is usually 0.5. This method is also known as the Crank-Nicholson scheme.

In explicit Euler and implicit Euler the truncation error is similar (both are 1st order accurate); however, implicit Euler is stable, such that larger time steps can be used. Note however, that, due to the inversion of the ($n \times n$) matrix, performing one time step is much more costly in the implicit method. The semi-implicit method is 2nd order accurate and also stable; its computational demand is comparable to the implicit method.

6.3.6 Which Integrator to Choose?

The good news is that, for those that use R as a modelling platform, a robust integration routine (`ode`) is available (package `deSolve`) that automatically selects the step size such as to achieve both reliability and efficiency. This is *the* method of choice and you can use it without worrying about the accuracy of the model; this is taken care of by the integration routine.

However, there are some model problems that are faster solved with the more simple integration routines. We run a model because we want output at some (regular) time interval. If the time step Δt at which the Euler or 4th order Runge-Kutta method is accurate and stable, is similar to or larger than this output interval, then these simpler integration methods are probably the best choice, as they are much faster. However, before we can be sure that the model output produced by fixed time step methods is sufficiently accurate, we must check whether the constant time step is small enough. Here is a rule of thumb to guide you:

- First run the model with a certain timestep,
- double the timestep and inspect the difference between the two runs.
- If the difference is significant, halve the time step until the results stop changing significantly.
- If insignificant, double the time step until the results start to change significantly.
- To be on the safe side, take a timestep half the one that was just accurate enough.

For other problems, there exist especially-designed integration routines that perform the job optimally. One of these special cases are 1-dimensional reaction-transport equations. We give an example at the end of this chapter, when we model transport and death of marine zooplankton in an estuary.

6.4 Approximating Spatial Derivatives (*)

Numerical integration in time proceeds by jumping from one point in time to the next. Similarly, numerical approximation of spatial derivatives in one dimension calculate the value of the function in a finite number of nodes or layers only: x_1, x_2, x_3, \dots (Fig. 6.6A).

To show how this is done, we derive the numerical approximation for the general reaction-transport equation from Section 3.4.3. We start by approximating the equation written in flux-divergence form, after which we find suitable expressions for the advective and dispersive fluxes.

These derivations require that we keep track of positions in space very carefully. In particular, we must always be aware whether a concentration, surface, flux or distance refers to an interface between boxes, or to the centre of the boxes. The symbolism is detailed in Fig. 6.6.

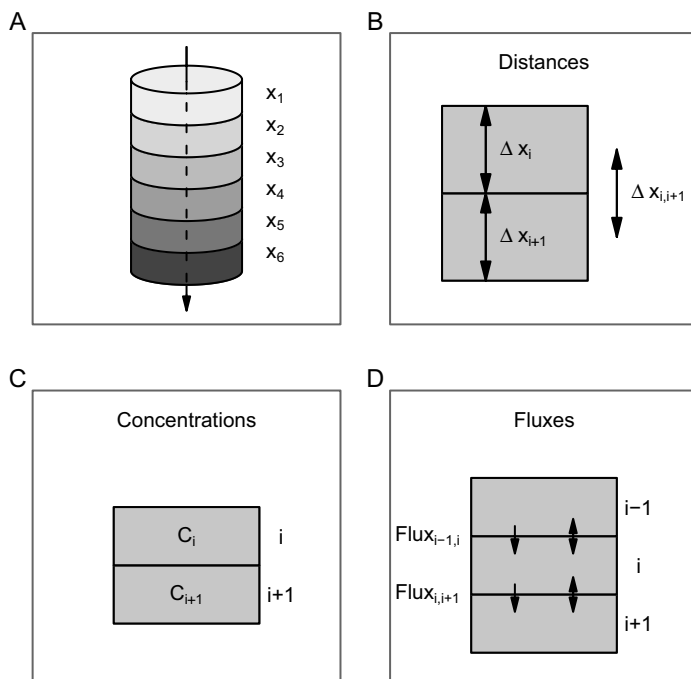


Fig. 6.6 **A.** Spatial derivatives are approximated by subdividing the spatial domain in a number of grid cells or boxes. Positions (x_i) are defined at the centre of the boxes. **B.** Relevant distances are either between centres of the boxes (e.g. $\Delta x_{i,i+1}$ between centres of boxes i and $i+1$) or lengths of boxes (Δx_i for length of box i). **C.** Concentrations, densities, etc.. are defined in the centre of the grid cells. **D.** Fluxes are defined on the box interfaces

6.4.1 Approximating the Flux Divergence Equation

As a first step, we numerically approximate the general reaction-transport equation written in the flux-gradient form, and where g is a first-order growth rate:

$$\frac{\partial C}{\partial t} = -\frac{1}{A} \cdot \frac{\partial A \cdot Flux}{\partial x} + g \cdot C \quad (6.16)$$

This formula is now discretized to represent the rate of change of the concentrations in each box i , C_i , as follows:

$$\frac{dC_i}{dt} = -\frac{1}{A_i} \frac{\Delta_i(A \cdot Flux)}{\Delta x_i} + g \cdot C_i \quad (6.17)$$

Where C_i is concentration in box i , Δx_i is the thickness of box i , A_i is the surface area in the middle of box i , and Δ_i denotes that the flux gradient is to be taken around box i .

At this point, it is useful to pinpoint that, when the model domain is divided in a number of discrete cells, we generally specify the concentrations or densities in the *middle* of boxes (Fig. 6.6B), while the fluxes are defined on the box *interfaces* (Fig. 6.6C). Fluxes as much leave a box as enter the next box, so it makes sense to prescribe them in between both boxes.

Keeping this in mind, the flux gradient for box i is then defined as the difference of the fluxes at the interface with the next box ($i,i+1$) and with the previous box ($i-1, i$). Thus we write:

$$\frac{dC_i}{dt} = -\frac{A_{i,i+1} \cdot Flux_{i,i+1} - A_{i-1,i} \cdot Flux_{i-1,i}}{A_i \cdot \Delta x_i} + g \cdot C_i \quad (6.18)$$

Here $Flux_{i-1,i}$ is the flux on the interface between cell $i-1$ and i , and $A_{i-1,i}$ is the surface area on this interface.

Depending on the problem, the surfaces at the interfaces and at the middle of the boxes may all be known (for instance, they may be defined as a continuous function of x). Alternatively, the surfaces at the interfaces may be derived by linear interpolation of the known surfaces at the middle of the boxes.

To make the numerical approximation complete, we need to find a suitable description for the advective and dispersive fluxes. It is simplest to start with the latter.

6.4.2 Approximating Dispersion

The continuous representation of the dispersive flux is:

$$Flux|_{dispersion} = -D \frac{\partial C}{\partial x} \quad (6.19)$$

which states that the flux is directed against the concentration gradient (hence the minus sign), i.e. there will be transport from high to low concentrations. As the flux is defined on the interface, the concentration gradient should also be defined on the interface. Thus it makes sense to approximate this gradient by simply taking the differences of the concentrations in the adjacent boxes, i.e. from box i to box $i-1$, and divide this by the distance between the centres of both boxes:

$$Flux_{i-1,i}|_{dispersion} = -D_{i-1,i} \cdot \frac{C_i - C_{i-1}}{\Delta x_{i-1,i}} \quad (6.20)$$

where $\Delta x_{i-1,i}$ is the ‘dispersion distance’, i.e. the distance from the centre of box $i-1$ to the centre of box i where the concentrations C_{i-1} and C_i are prescribed. $D_{i-1,i}$ is the value of the dispersion coefficient at the interface (see Fig. 6.6).

This is indeed how dispersive fluxes are approximated.

6.4.3 Approximating Advection

The advective fluxes are simply the product of a concentration times a velocity.

$$Flux|_{advection} = u \cdot C \quad (6.21)$$

With the flux defined on the interface, it is most logical to use the concentration on this interface to estimate the flux. However, as we model concentrations in the center of boxes the concentration on the interface is not known, but it can simply be estimated as an average of the concentrations of the two adjacent boxes.

Thus, the logical choice to approximate the advective flux on the interface between box $i-1$ and i is to interpolate spatially:

$$Flux_{i-1,i}|_{advection} = u \cdot \frac{C_i + C_{i-1}}{2} \quad (6.22)$$

This is called the ‘centred difference’ approximation, as it uses the central value of two adjacent concentrations. With equal box sizes, this is the (interpolated) concentration at the boundary between adjacent boxes. With unequal box sizes, this value is situated somewhere else, although the formula would be easily adapted to represent the interpolated concentration at the interface exactly.

Unfortunately, this is often not a good idea! When used to approximate the advective equation, the centred differencing has the undesired effect of producing negative concentrations. In numerical jargon it is said that the scheme is ‘non-monotone’.

Why is this so? The flux on the interface of box $i-1$ and i not only enters box i but it also leaves box $i-1$. In the centred difference approximation, the advective flux leaving box $i-1$ is a function of the concentration in box $i-1$, but also of the concentration in the next box, i . Now, assume the case where the concentration in this box ($i-1$) is zero, while it is positive in box i . Then the centred difference approximation will calculate a net efflux from box $i-1$, although there is nothing present to leave the box! Clearly, at the next time step, concentrations C_{i-1} , will be negative.

Such negative concentrations are biologically unrealistic and should be avoided. Moreover, these negative concentrations in turn often create oscillations, such that the scheme is also ‘unstable’.

It is much safer to make the flux leaving box $i-1$ a function of the concentration in the box $i-1$ only. Thus:

$$Flux_{i-1,i} \Big|_{advection} = u \cdot C_{i-1} \quad (6.23)$$

This is called a ‘backward’ approximation. If concentration is 0 in box $i-1$, then this formula predicts that there will be no efflux, hence negative concentration will not occur.

However, the world of numerics is never perfect. Whereas the backward differencing scheme does never produce negative concentrations, it is notorious for its ability to create so-called ‘numerical dispersion’. We will demonstrate this phenomenon later.

6.4.4 The Boundaries with the External World

We end this section with some consideration of what to do near the boundaries with the external world. In general, the boundaries will receive particular attention in the numerical solution of a model.

1. In case the boundary condition is prescribed as a *flux*, then life is simple: rather than calculating the flux as a function of concentrations (using Eq. (6.20) or Eq. (6.23)), we just plug in the imposed value in eq. 6.18. For instance, if at the upper boundary, i.e. at the interface with cell 1, a flux is prescribed with a value F , then the numerical approximation for the rate of change of concentration in box 1 is:

$$\frac{dC_1}{dt} = -\frac{A_{1,2} \cdot Flux_{1,2} - A_0 \cdot F}{A_1 \cdot \Delta x_1} + g \cdot C_1 \quad (6.24)$$

Where A_0 is the surface at the interface, and $Flux_{1,2}$ is estimated based on eq. 6.20 and eq. 6.23.

2. In contrast, in case the boundary condition is prescribed as a *concentration* defined on the edge, then it follows that the dispersion distance is only half the thickness of the box, i.e. we obtain:

$$\begin{aligned} Flux_{0,1} \Big|_{advection} &= u \cdot C_0^* \\ Flux_{0,1} \Big|_{dispersion} &= -D \cdot \frac{C_1 - C_0^*}{\Delta x_1/2} \end{aligned} \quad (6.25)$$

Using these formulas, we have transformed the boundary concentration prescription into a flux prescription, which can then be incorporated into the flux divergence equation:

$$\frac{dC_1}{dt} = - \frac{A_{1,2} \cdot Flux_{1,2} - A_0 \cdot (Flux_{0,1}|_{advection} + Flux_{0,1}|_{dispersion})}{A_1 \cdot \Delta x_1} + g \cdot C_1$$

3. Boundary conditions may also be prescribed as fluxes dependent on the concentration difference between an external concentration, and the concentration at one of the boundaries of the model (so-called Robin or evaporation boundary conditions):

$$Flux_{0,1} = \alpha(C_e - C_{0,1})$$

In principle, $C_{0,1}$ can be approximated from the known concentration at the centre of the first box C_1 , and the concentration gradient in the top part of the model, as was done in the previous case. In practice however, one usually takes the concentration in the upper box (C_1) as a sufficient approximation of the concentration at the interface.

Once the flux at the interface is fully defined it can, again, be plugged into the flux divergence equation.

6.5 Numerical Dispersion (***)

When approximating advective terms (Section 6.4.3), the backward differencing scheme (eq. 6.26) did not produce negative concentrations and was stable, but it is only first-order accurate.

$$Flux_{i-1,i}|_{advection} = u \cdot C_{i-1} \quad (6.26)$$

The truncation error, which is due to ignoring the 2nd and higher order terms from the Taylor expansion, manifests itself as a process that is called ‘*numerical dispersion*’ or numerical dissipation. This is one of the most famous types of numerical errors. It means that, even in the absence of true, physical or biological dispersion, existing sharp gradients will be smoothed because of a numerical artefact.

We illustrate the point using an example similar (but not equal) to the one from the book of Gurney and Nisbet (1998). Individuals of a population grow at a constant rate and without mortality. At the start of the simulation ten individuals are born, and we describe their size as a function of time.

This is a purely ‘advective’ model, where the advection rate is given by the growth rate, the increase in size of the individuals. If we denote the size of the individuals by x , the specification for this model and the backward differencing approximation is:

$$\begin{aligned}\frac{\partial N}{\partial t} &= -growth \cdot \frac{\partial N}{\partial x} \\ \frac{dN_i}{dt} &= \frac{growth \cdot N_{i-1} - growth \cdot N_i}{\Delta x} \\ N_0 &= 10\end{aligned}\tag{6.27}$$

where N_i is the number of individuals in size class i . If we select the size classes (Δx) such that the organisms move through one size class per day, then, in the real world, after the first 10 days all organisms are of size 10, after 20 days, all organisms are of size 20, etc. . . In Fig. 6.7 the dashed line represents the true solution after ten days of simulation.

Using the backward differencing scheme, the results are quite different: after 10 steps through time, the square pulse has been smoothened, by the ‘numerical dispersion’ error, into a bell-shaped curve, with a range more than 10 times the original range! (Fig. 6.7).

Faced with this problem, we can take several directions:

First of all, we can be very pragmatic about it. In the example, the square pulse of individuals moving through the size classes may be the true mathematical solution, but it is not very realistic. In real life, variation in growth rates of individuals will cause some individuals to attain larger size earlier; whilst other organisms will grow slower. Thus, biological variation will also cause a spread of the pulse as time proceeds. One might therefore argue that the smoothened pulse is more realistic than the square one. Moreover, in most applications, there will be a mixture of advective and diffusive terms, and in many cases, the ‘numerical’ dispersion will be but a fraction of the other dispersion terms. Under these circumstances, we may consider the backward numerical scheme to be sufficiently accurate.

Secondly, we can try to fix the problem by using more complex numerical schemes. These mathematical solutions reduce the effects of numerical dispersion

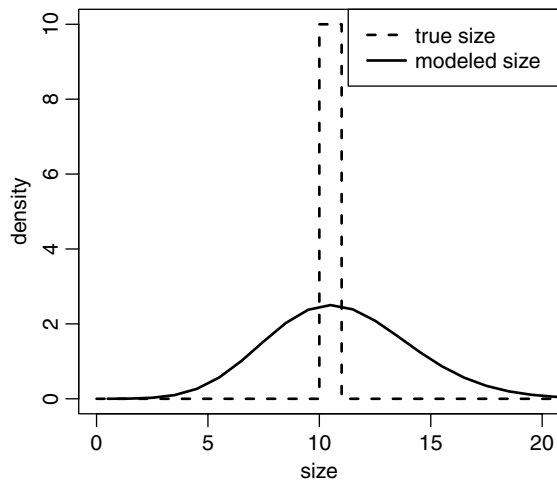


Fig. 6.7 Numerical dispersion. Although this growth model only includes advection, after 10 steps, the true square pulse has been smoothened spectacularly by numerical diffusion

by taking into account the higher order terms of the Taylor series, but most of these schemes are quite complex and still suffer some kind of artificial dispersion. One solution which may be appealing to biologists is provided in the book by Gurney and Nisbet (1998).

Thirdly, as the problem is mainly acute for large boxes, a solution is to increase the number of boxes. However, as this occurs at the expense of increasing computing time, there is a limit to the number of boxes that can be represented in a model.

Finally, we may decide to use centred differences after all, ignoring all warnings about the dangers of negativity. Under certain circumstances, there are perfectly good reasons to do so: as long as concentrations do not increase along the spatial axis, centred differences will never generate negative concentrations, yet they cause very limited numerical dispersion. For instance, sediment organic matter concentrations (nearly) always decrease with depth in the sediment. Thus, models that describe organic matter in a spatially explicit way often approach the advection term by means of centred differences.

6.5.1 Example: Sediment Model

We demonstrate the effect of numerical dispersion in different schemes using a simple model for inert tracers in a sediment, subject to a small dispersion term due to bioturbation ($D_b=0.1 \text{ cm}^2\text{y}^{-1}$) and an advective term due to sediment accretion ($u=1 \text{ cm y}^{-1}$). Three numerical schemes are compared: backward differences, centred differences, and a flexible scheme (called the Fiadeiro scheme, Fiadeiro and Veronis (1977)) that tries to optimise in between the other two methods.

For equal box sizes, the three schemes can all conveniently be modelled using a single formula, where the advective flux is expressed as a function of a composite concentration, using a parameter σ :

$$\begin{aligned} Flux_{i-1,i} \Big|_{advection} &= u \cdot C^* \\ C^* &= \sigma \cdot C_{i-1} + (1 - \sigma) \cdot C_i \end{aligned} \quad (6.28)$$

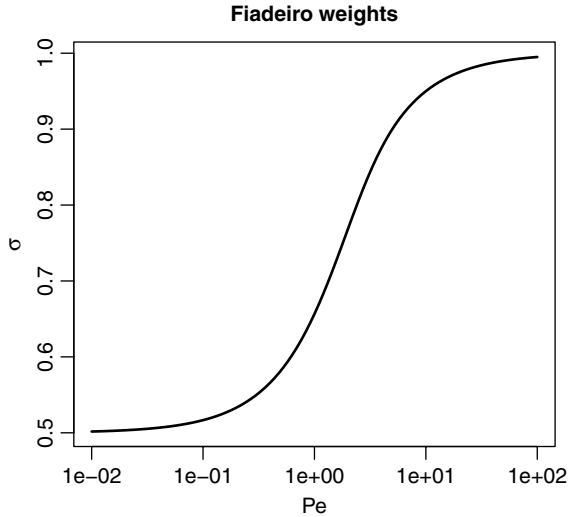
Clearly, in backward differences, σ equals 1. For centred differences, σ equals 0.5. The Fiadeiro scheme adapts the value of σ to the ‘Peclet number’ of the problem. The Peclet number is a dimensionless ratio expressing the relative importance of advective over dispersive processes:

$$Pe = \frac{u \cdot \Delta x}{D} \quad (6.29)$$

In the Fiadeiro scheme, σ is given by the impressive formula:

$$\sigma = \frac{1}{2} \left[1 + \frac{1}{\tanh(Pe)} - \frac{1}{Pe} \right] \quad (6.30)$$

Fig. 6.8 Dependence of the parameter σ on the Peclet number in the Fiadeiro scheme



(see Fig. 6.8). Clearly, the scheme switches from centred differences at low Peclet numbers (dispersion – dominated conditions) to backward differences in advection-dominated cases.

Comparison of the results for the passive tracer model (Fig. 6.9) shows that numerical dispersion dominates the results for backward differences, unless the number of boxes is very high (and computation time is accordingly very long). Centred difference results for a very low number of boxes are set somewhat apart, but this may partly be due to the rough specification of the initial conditions in this case. With 30 boxes, the result is almost indistinguishable from the result with 300 boxes. The Fiadeiro scheme in this case is closer to backward than to centred differences (D is rather low, so Pe is high), but still performs slightly better than the latter. When the dispersion coefficient is raised or the advection is decreased, the Fiadeiro scheme will approach the centred differences solution.

Finally note that in this case, the centred differences did not produce any instability.

6.6 Case Studies in R

In previous chapters we developed a number of models, but did not yet solve them. R-package `deSolve` contains several numerical methods to solve (systems of) differential equations. Function `ode` is the most general numerical integration routine from this package and is the method of choice for models whose dynamics are continuous, i.e. there are no sharp jumps in the forcing functions. In case the dynamics is discontinuous, we may run the model in several parts (e.g. Section 6.6.2).

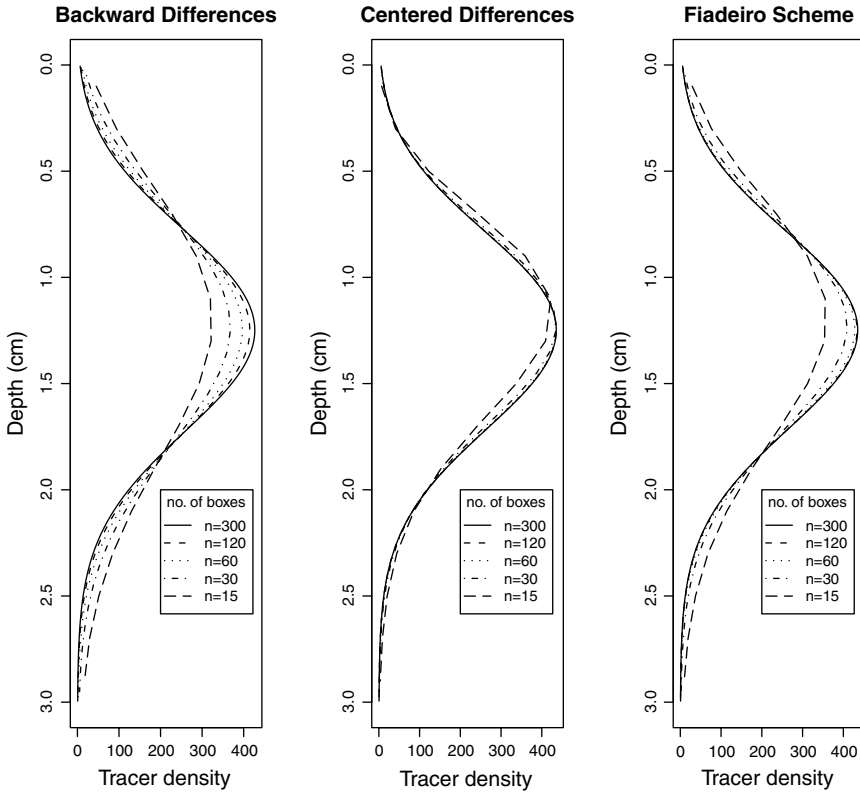
Luminophore distribution at $t = 1$ year

Fig. 6.9 Spatial distribution of inert tracers (e.g. luminophores, fluorescent tracer particles) added as a layer with concentration of $1000 \text{ particles cm}^{-3}$ in the top 0.5 cm at $t=0$. In the model, the particles are subject to dispersion by bioturbation ($Db = 0.1 \text{ cm}^2 \text{ y}^{-1}$) and advective movement due to sediment accretion ($u = 1 \text{ cm y}^{-1}$). After one year, the peak is situated between 1 and 1.5 cm , but depending on the numerical scheme, considerable numerical dispersion may have taken place. See text for details

6.6.1 Implementing the Enzymatic Reaction Model

We start by implementing the enzymatic reaction model of Section 2.4 in R. The reactions were



and the model equations:

$$\begin{aligned}
 \frac{d[D]}{dt} &= -k_1 \cdot [E] \cdot [D] + k_2 \cdot [I] \\
 \frac{d[I]}{dt} &= k_1 \cdot [E] \cdot [D] - k_2 \cdot [I] - k_3 \cdot [I] \cdot [F] \\
 \frac{d[E]}{dt} &= -k_1 \cdot [E] \cdot [D] + k_2 \cdot [I] + k_3 \cdot [I] \cdot [F] \\
 \frac{d[F]}{dt} &= -k_3 \cdot [I] \cdot [F] \\
 \frac{d[G]}{dt} &= k_3 \cdot [I] \cdot [F]
 \end{aligned} \tag{6.32}$$

There are three model parameters: the rate coefficients k_1 , k_2 and k_3 that are defined first. They are stored as a vector and assigned names and values.

```

params<-c(k1=0.01/24,
          k2=0.1/24,
          k3=0.1/24)

```

Similarly the state variable vector that contains the 5 chemical species is created and initial concentrations given.

```

state    <-c(D=100,
             I=10,
             E=1,
             F=1,
             G=0)

```

Next a function ('enzyme') is defined that calculates the rate of change of the state variables. Input to the function is the model time (not used here, but required by the calling routine), and the values of the state variables and the parameters. This function will be called by R's routine that solves the differential equations (ode).

The parameters and state variables are vectors, and their elements can be assessed by indexing (e.g. `state[1]`, `state[2]`, etc...). However, the code is more readable if we use their names instead. To do so, the vectors are treated as a list.

The statement `with(as.list(c(state,parameters)), {... })` does this.

The main part of the enzyme model calculates the rate of change of the state variables. At the end of the function, these rates of change are returned, packed as a list.

```
enzyme<-function(t,state,parameters)
{
  with(as.list(c(state,parameters)),{

    dD <- -k1*E*D + k2*I
    dI <-  k1*E*D - k2*I - k3*I*F
    dE <- -k1*E*D + k2*I + k3*I*F
    dF <-                               - k3*I*F
    dG <-                               k3*I*F

    list(c(dD,dI,dE,dF,dG))
  })
}
```

Now the model can be run. Below we choose to run the model for 300 h, and give output at 0.5 hourly intervals. R's function `seq()` creates the time sequence.

```
times      <-seq(0,300,0.5)
```

The model is solved using the function `ode`, which is included in R's package 'deSolve'. This package has to be loaded first, and if necessary installed (see Appendix A.2.6 for how to do this). The function `ode` takes as input the state variable vector, the times at which output is required, the model function that returns the rate of change and the parameter vector, in that order. It returns a matrix that contains the values of the state variables (columns) at the requested output times.

The output is converted to a data frame and stored in 'out'. Data frames have the advantage, that their columns can be accessed by name, rather than by number. For instance, 'out\$D' will take the outputted concentrations of substance D, and so on.

```
require(deSolve)
out <- as.data.frame(ode(state,times,enzyme,params))
```

Finally, the model output is plotted. The figures are arranged in two rows and two columns (`mflow`), and the size of the outer upper margin (the third margin) is increased (`oma`), such as to write a figure heading (`mtext`) (see Fig. 6.10).

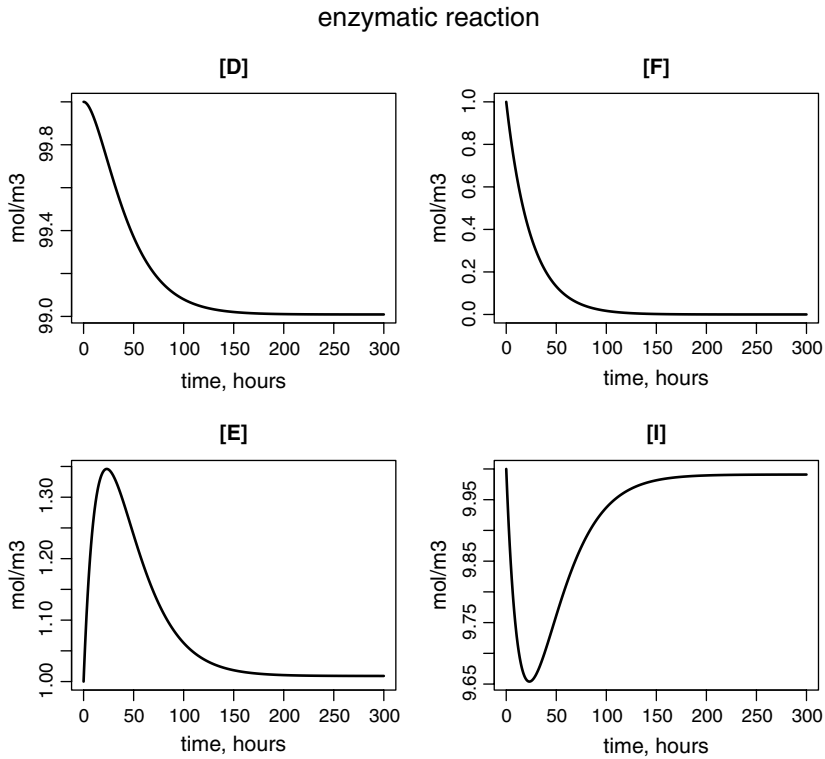


Fig. 6.10 Output of the enzymatic reaction model – for R-code see text

```
par(mfrow=c(2,2),      oma=c(0,0,3,0))

plot (times,out$D,type="l",main="[D] ",
      xlab="time,      hours", ylab="mol/m3",lwd=2)
plot (times,out$F,type="l",main="[F] ",
      xlab="time,      hours",ylab="mol/m3",lwd=2)
plot (times,out$E,type="l",main="[E] ",
      xlab="time,      hours",ylab="mol/m3",lwd=2)
plot (times,out$I,type="l",main="[I] ",
      xlab="time,      hours",ylab="mol/m3",lwd=2)
mtext(outer=TRUE,side=3,"enzymatic reaction ",cex=1.5)
```

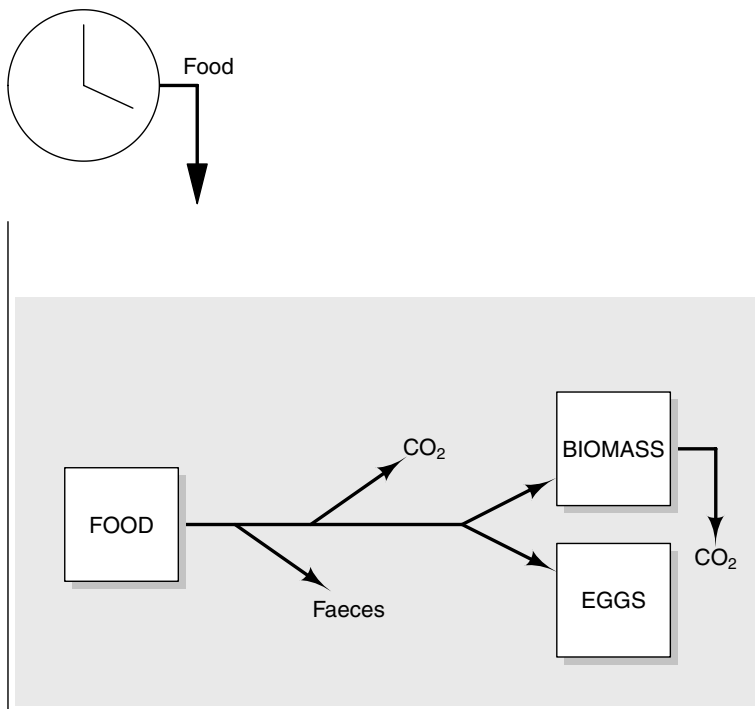


Fig. 6.11 Conceptual model of the DAPHNIA individual model

6.6.2 Growth of a *Daphnia* Individual

We now discuss a (more complex) model that describes the growth and reproduction of one individual of a zooplankton organism, belonging to the genus *Daphnia* (Cladocera). The model (Andersen, 1997), is based on the energy budget of the organism.

The conceptual model is depicted in Fig. 6.11. The weight of the individual (excluding the eggs), the accumulated egg mass and the food are the state variables. The model is embedded in a transfer culture experiment. This means they are reared in a bottle, and a fixed number of individuals are transferred to new medium at regular intervals.

The implementation of this model is interesting because the biomass of these organisms does not increase continuously. Rather, the individuals grow in-between molts, and the time span between molts depends on temperature. The transfer of organisms to new medium introduces a second temporal discontinuity in their growth.

We use the implementation in R to describe the model in detail.

The model *parameters* fall into three groups.

1. The weight of newborns (`neonateWeight`), weight at first reproduction, and maximal weight an organism can attain and the duration between moults (`instarDuration`):

```
neonateWeight <- 1.1 #µgC
reproductiveWeight <- 7.5 #µgC
maximumWeight <- 60.0 #µgC
instarDuration <- 3.0 #days
```

2. parameters that are used to describe the food ingestion, assimilation, and reproduction and respiration rate

```
ksFood <- 85.0 #µgC/l
IngestWeight <- 132.0 #µgC
maxIngest <- 1.05 #/day
assimileEff <- 0.8 #-

maxReproduction <- 0.8 #-
respirationRate <- 0.25 #/day
```

and

3. parameters that describe the experimental setup (time between transfers, the food concentration in the new medium, the number of individuals reared).

```
transferTime <- 2 # Days
foodInMedium <- 509 # µgC/l
numberIndividuals <- 32 # -
```

The three state variables are the individual weight of the organism, the total egg weight and the food in the medium. They are initialized whilst defining them:

```
state <-c(
  INDWEIGHT = neonateWeight ,
  EGGWEIGHT = 0 ,
  FOOD = foodInMedium
)
```

In between moults, the organisms ingest, respire, grow and reproduce. The ingestion rate per unit body weight decreases linearly with body size (variable `WeightFactor`), and is limited by food availability (half-saturation constant `ksFood`). Only part of ingested food is assimilated (parameter `assimileEff`). Assimilated food serves 3 purposes: somatic growth (increase of biomass), reproduction and maintenance (`Respiration`). The latter is the basal metabolism to support body functions and regeneration of lost material; it is first-order to individual weight.

The net balance between assimilation and respiration, if positive, is used for somatic growth and reproduction.

Reproduction only starts when the individual weight has exceeded the reproductive weight. The fraction allocated to reproduction increases hyperbolically with increasing individual weight. The maximal fraction is less than 1, such that body growth continues after reaching maturity.

Finally the rate of change of individual weight, egg weight and food availability is calculated. For the latter, we take into account the number of individuals (a parameter) that are in the medium. The function returns both the rates of changes and three ordinary variables.

```
model<-function(t,state,parameters)
{
  with(as.list(c(state)),{ # unpack the state variables

    # ingestion, size-dependent and food limited

    WeightFactor <- (IngestWeight-INDWEIGHT)/(IngestWeight-neonateWeight)
    MaxIngestion <- maxIngest*WeightFactor
    Ingestion     <- MaxIngestion*INDWEIGHT*FOOD / (FOOD + ksFood)

    Respiration   <- respirationRate * INDWEIGHT

    Growth        <- Ingestion*assimilEff - Respiration

    if (Growth <= 0. | INDWEIGHT <reproductiveWeight)
      Reproduction <- 0. else
    {
      WeightRatio   <- reproductiveWeight/INDWEIGHT
      Reproduction   <- maxReproduction * (1. - WeightRatio^2)
    }

    # rate of change
    dINDWEIGHT <- (1. -Reproduction) * Growth
    dEGGWEIGHT <-      Reproduction  * Growth
    dFOOD       <- -Ingestion * numberIndividuals

    list(c(dINDWEIGHT, dEGGWEIGHT, dFOOD),
         c(Ingestion   = Ingestion,
           Respiration  = Respiration,
           Reproduction = Reproduction))

  }) # end of with(as.list(...

} # end of model
```

During moults, increased energetic costs cause the organisms to loose weight. The weight loss is an allometric function of body length; length is an allometric function of individual weight.

```
Moulting  <- function ()
{
  with(as.list(c(state)),{

    refLoss  <-  0.24 #μgC
    cLoss    <-  3.1  #-

    INDLength  <-  (INDWEIGHT /3.0)^(1/2.6)

    WeightLoss <- refLoss * INDLength^cLoss
    return(INDWEIGHT - WeightLoss)
  })
}
```

Because of the setback of individual weight during moulting and due to the regular transfer of individuals to new medium, running the model proceeds in discrete steps. This is necessary because the integrator `ode` cannot cope with changes occurring in the values of the state variables: it just requires the values of the rates of changes.

We keep track of the next time at which moulting will occur and the next time at which individuals are transferred to new medium (`TimeMoult`, `TimeTransfer`). The integrator proceeds till either of these events occurs (`TimeOut`), and the state variables are each time reinitialised with the final condition of the integration procedure; this is on the last element of output.

If the organism moults, the individual weight is reset (using the function `Moulting`, defined above), the egg weight of the instar is reinitialised to 0, and the next time at which moulting (`TimeMoult`) will occur is calculated.

Transfer of the organisms to new medium, only affects the food concentration (`foodInMedium`); similar as for moulting, the next time at which transfer will occur (`transferTime`) is estimated.

```

TimeFrom      <- 0
TimeEnd       <- 40

TimeMoult     <- TimeFrom + instarDuration
TimeTransfer  <- TimeFrom + transferTime

Time          <- TimeFrom
Outdt         <- 0.1
out           <- NULL

while (Time < TimeEnd)
{
  TimeOut <- min(TimeMoult, TimeTransfer, TimeEnd)
  times   <- seq(Time, TimeOut, by=Outdt)
  out1    <- as.data.frame(ode(state, times, model, parms=0))
  out     <- rbind(out, out1)

  lout    <- nrow(out1)      # last element of output

  state   <-c(
    INDWEIGHT = out1[lout, "INDWEIGHT"],
    EGGWEIGHT = out1[lout, "EGGWEIGHT"],
    FOOD      = out1[lout, "FOOD"])

  if (Time >= TimeMoult)      # Moulting...
  {
    state[1] <- Moulting() # New weight individuals
    state[2] <- 0.         # Eggs = 0
    TimeMoult <- Time + instarDuration # next moult time
  }

  if (Time >= TimeTransfer) # New medium...
  {
    state[3] <- foodInMedium
    TimeTransfer <- Time + transferTime # next transfer time
  }

  Time <- TimeOut
}

```

Finally we plot the model results (Fig. 6.12), in two rows, two columns (mfrow). We increase the size of the 3rd outer margin (oma), as we write the main title of the model in this margin (mtext), enlarged with 50% (cex=1.5).

DAPHNIA model

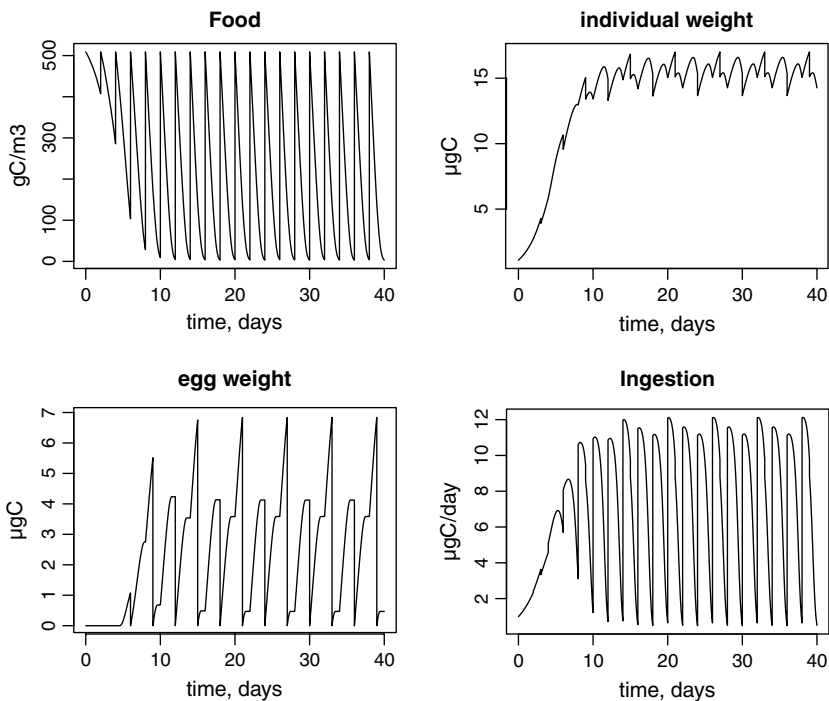


Fig. 6.12 Output generated by the DAPHNIA individual model, for a single individual per culture vessel – see text for R-code

```
par(mfrow=c(2,2),      oma=c(0,0,3,0))

plot (out$time,out$FOOD      ,type="l",main="Food",
      xlab="time, days",ylab="µgC/l")
plot (out$time,out$INDWEIGHT,type="l",main="individual weight",
      xlab="time, days",ylab="µgC")
plot (out$time,out$EGGWEIGHT ,type="l",main="egg weight",
      xlab="time, days",ylab="µgC")

plot (out$time,out$Ingestion ,type="l",main="Ingestion",
      xlab="time, days",ylab="µgC/ind/day")

mtext(outer=TRUE,side=3,"DAPHNIA model",cex=1.5)
```

It is illustrative to run the Daphnia model for two extreme cases:

1. population density of one organism per litre
2. population density of 50 individuals per litre

and to compare the trajectories of individual weight as a function of time for 1 and 50 organisms. We only show the results for a single individual per litre here. Try the model for the other scenario! The qualitative difference between both curves is due to the fact that the organisms become food limited when cultured at a density of 50 individuals, whereas the transfer regime is sufficient to prevent food limitation when only one organism grows in the culture vessel.

The functional dependency of maximal ingestion, reproduction fraction, individual length and the weight loss during moulting on individual length (Fig. 6.13) is generated with the following R-script:

```
windows()
par (mfrow=c(2,2))
curve(maxIngest*(IngestWeight-x)/(IngestWeight-neonateWeight), 0,60,
      main="Max. ingestion rate",ylab="/d",xlab="ind. weight, µC",lwd=2)
curve(pmax(0., maxReproduction * (1.-(reproductiveWeight/x)^2)),0,60,
      main="fraction assimilate to reproduction ",ylab="-",
      xlab="ind. weight, µC",lwd=2)
curve(((x /3.0))^(1/2.6),0,60,
      main="Individual length",ylab="µm",xlab="ind. weight, µC",lwd=2)
curve(0.24*((x /3.0)^(1/2.6))^3.1,0,60,
      main="Weight loss during moulting",
      ylab="µg",xlab="ind. weight, µC",lwd=2)
```

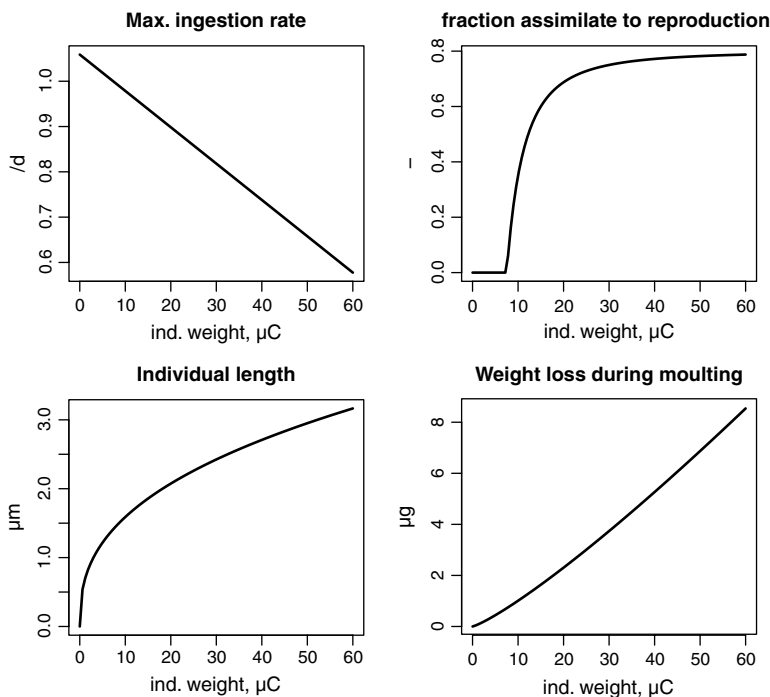


Fig. 6.13 Functional dependencies in the DAPHNIA individual model – see text for R-code

6.6.3 Zero-Dimensional Estuarine Zooplankton Model

Our third example is a very simple zero-dimensional (0-D) model, that describes marine zooplankton entering an estuary through the action of the tides. A significantly more complex and more realistic (1-D) model is implemented in Section 6.6.5. You may refer to this chapter to obtain more information about the rationale of the modelling. We use this simple example to demonstrate the use of the Euler integration method.

The estuarine zooplankton exchanges with the sea at a rate k (d^{-1}), and decays at a constant rate g (d^{-1}). Boundary concentrations (expressed in gram dry weight m^{-3}) in the sea are known and imposed as a forcing function (`ZOOsea`). The mass balance equation of zooplankton (`ZOO`) is:

$$\frac{d\text{ZOO}}{dt} = k \cdot (\text{ZOOsea} - \text{ZOO}) - g \cdot \text{ZOO} \quad (6.33)$$

Although it is perfectly possible to solve that model with R's function `ode`, the fact that we do not have control over the exact time at which the value of the forcing function needs to be estimated slows down the computation quite a bit. Here it is much more efficient to use Euler integration instead, as this gives total control over the time step used.

The implementation of the model starts by inputting the zooplankton concentration in the sea (a forcing function), as two vectors, one containing time values, the other the concentrations:

```
fZooTime = c(0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 340, 367)
fZooConc = c(20, 25, 30, 70, 150, 110, 30, 60, 50, 30, 10, 20, 20)
```

Then a function (`euler`) that runs the model from the initial time (`start`) to the end time (`end`) and with given time step (`delt`) and exchange rate and decay rate (`k`, `g`) is implemented.

We first form the sequence of time increments which the model will step through (`times`), and estimate the value of the forcing function at each of these time values (`ZOOsea`). R's function `approx` does that; it requires the `x` and `y` input data and the output `x` values (`xout`). In this case, the data `x` and `y`-values are in the `fZooTime` and `fZooConc` respectively, while mapping has to be performed on the output `times`.

We initialise the model with a reasonable guess of zooplankton concentration (5 gDWT/m³)

After declaring a matrix that is to contain the output (`out`), the model steps through the time sequence (`1 : (nt-1)`), each time calculating the input and decay rate, and the rate of change, `dZOO`. After that, the new value of the state variable is estimated by applying the Euler integration formula.

At each time the output is stored in the respective row of `out` (`out[i,]`). After finishing the time stepping, we assign column names to matrix `out`, and the output matrix is returned from the function.

```
euler <-function(start,end,delt,g,k)
{
  times      <- seq(start,end,delt)
  nt         <- length(times)

  ZOOsea     <- approx(fZooTime,fZooConc, xout=times)$y

  ZOO        <- 5
  out        <- matrix(ncol=3,nrow=nt)

  for (i in 1:(nt-1))
  {
    decay    <- g*ZOO
    input    <- k*(ZOOsea[i] - ZOO)

    out[i,] <- c(times[i],ZOO, input)

    dZOO     <- input - decay

    ZOO      <- ZOO + dZOO *delt
  }

  colnames(out) <- c("time","ZOO","input")
  return(as.data.frame(out))
}
```

Finally, the model is run for one year, the output plotted and a legend added (Fig. 6.14).

```
out <-euler(0,365,0.1,k=0.015,g=0.05)

par (oma=c(0,0,0,2))

plot(fZooTime,fZooConc,type="b",xlab="daynr",
     ylab="gC/m3",pch=15,lwd=2,
     main="Zooplankton model")
lines(out$time,out$ZOO,lwd=2,col="darkgrey")

legend("topright",c("Marine zooplankton","Estuarine zooplankton"),
     pch=c(15,NA),lwd=2,col=c("black","grey"))
```

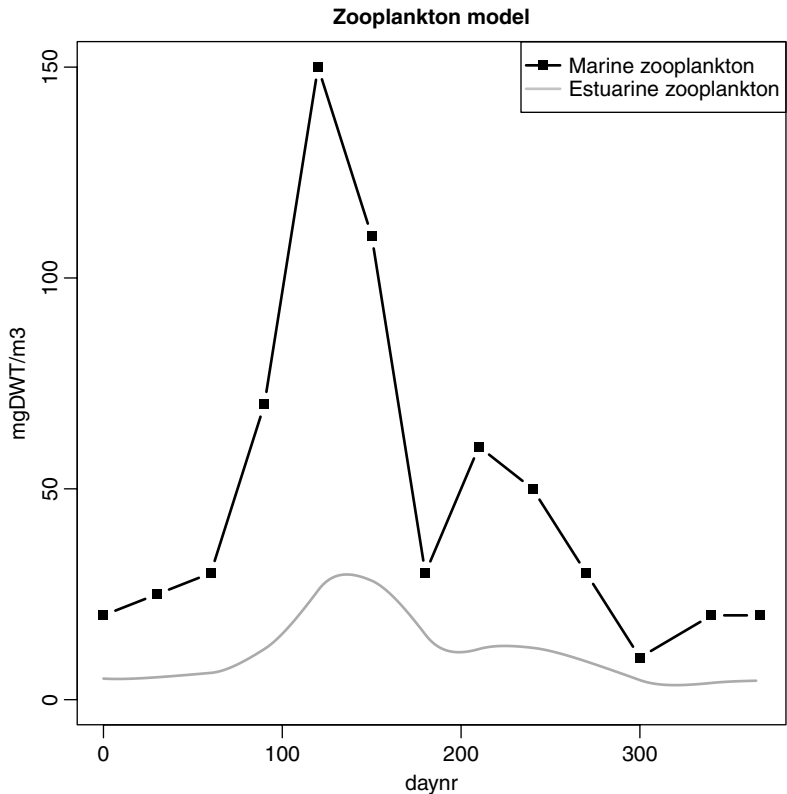



Fig. 6.14 Output generated with the 0-D estuarine zooplankton model – see text for R-code

6.6.4 Aphids on a Row of Plants: Numerical Solution of a Dispersion-Reaction Model

Aphids are serious pests for agriculture as their feeding on the crops reduces the vitality of the plants and makes them less attractive for human consumers. Once they have infected vegetables, they will slowly increase in numbers, spread and infest new plants. We implement and solve a simple model to mimic this behavior. A gardener has planted a row of lattuce, 60 m long. In the middle of the row, a small population of aphids invades the plants.

We assume that the aphids reproduce with a net rate of increase $g=0.01\text{ d}^{-1}$. This is a very low value, but it is a *net* rate of increase, i.e. difference between birth and mortality processes, and we assume that the plants are not very good food, so that the aphids just barely survive.

The aphids slowly disperse from their original location. This dispersal is by random movement, and can be modelled as a diffusion-type process with dispersion coefficient $K = 0.3\text{ m}^2\text{d}^{-1}$. At both ends of the row of plants, aphids that happen to

move outside will disappear forever (it is an absorbing boundary). Therefore we can set as spatial boundary conditions that densities are zero at the edge of the model domain, i.e. at $x=0\text{m}$ and $x=60\text{m}$.

We solve the model numerically, using a spatial step of 1 m. Thus, we divide the model domain into 60 boxes, each 1 m wide.

The implementation in R starts with the definition of the parameters and the calculation of distances of the boxes on the plant.

```
D      <- 0.3      # m2/day dispersion coefficient
g      <- 0.01     # /day net growth rate
delx   <- 1       # m thickness of boxes
numboxes <- 60

Distance <- seq(from=0.5, by=delx, length.out=numboxes)
```

It is simplest to solve the 1-D transport reaction model in the *flux-divergence* form (see Section 6.4.1).

Recapitulating, the flux-reaction model for the process is given by:

$$\frac{\partial N}{\partial t} = -\frac{1}{A} \cdot \frac{\partial A \cdot Flux}{\partial x} + g \cdot N \quad (6.34)$$

which, as the surface (A) is considered constant here, can be simplified:

$$\frac{\partial N}{\partial t} = -\frac{\partial Flux}{\partial x} + g \cdot N \quad (6.35)$$

And where the dispersive flux is given by:

$$Flux_{dispersion} = -D \frac{\partial N}{\partial x} \quad (6.36)$$

With D the dispersion coefficient.

We numerically approximate the rate of change of densities in one grid cell, i , as:

$$\frac{dN_i}{dt} = -\frac{\Delta_i Flux}{\Delta x_i} + g \cdot N_i = -\frac{Flux_{i,i+1} - Flux_{i-1,i}}{\Delta x_i} + g \cdot N_i \quad (6.37)$$

Where N_i is density in box i , Δx_i is the thickness of box i , and Δ_i denotes that the flux gradient is to be taken around box i , i.e. as the difference of the fluxes at the interface with the next box ($i,i+1$) and with the previous box ($i-1,i$). $Flux_{i-1,i}$ is the flux on the interface between cell $i-1$ and i .

The flux on the interface, which is due to dispersion, is given by:

$$Flux_{i-1,i} = -D_{i-1,i} \cdot \frac{N_i - N_{i-1}}{\Delta x_{i-1,i}} \quad (6.38)$$

where $\Delta x_{i-1,i}$ is the dispersion distance, i.e. the distance from the centre of box $i-1$ to the centre of box i , where the densities N_i and N_{i-1} are prescribed. At the external boundaries of the first and last cell, the density is prescribed at the edge, thus it follows that the dispersion distances, $\Delta x_{0,1}$ and $\Delta x_{60,61}$ are only half the thickness of the boxes.

This is how we implement these equations in R: we start by defining the dispersion distances $\Delta x_{i-1,i}$ (`deltax`), taking into account the fact that they are only half the box thickness at the edges.

To estimate the fluxes (Eq. 6.38), we use R's function `diff`, which takes the gradient, (i.e. `diff(c(1,2,4))` would return a vector with elements $= 2 - 1$ and $4 - 2$ respectively). We take care to add the boundary densities ($0, \dots, 0$).

We then use Eq. (6.37) to calculate the rate of change of densities in each box. The rate of change is returned from the function, as a list.

```
deltax      <- c(0.5,rep(1,numboxes-1),0.5)

Aphid <-function(t,APHIDS,parameters)
{
  Flux      <- -D*diff(c(0,APHIDS,0))/deltax
  dAPHIDS   <- -diff(Flux)/delx + APHIDS*g

  list(dAPHIDS )
}
```

To run the model, we initialise the aphid densities. Density is set to 0 in all boxes, except for the two central ones (30,31), which are 1.

After specifying the times at which we want output, the integrator `ode` is called; output is stored in matrix `'out'`. The first column of this matrix contains time, the other columns the density values at each time. The last line of following code extracts all density values.

```
APHIDS      <- rep(0,times=numboxes)      # ind/m2
APHIDS[30:31] <- 1
state       <- c(APHIDS=APHIDS)

require(deSolve)
times       <- seq(0,200,by=1)
out         <- ode(state,times,Aphid,parms=0)
DENSITY     <- out[,2:(numboxes +1)]
```

Finally the output is plotted. First we create a temporal-spatial plot of the densities, using function `filled.contour`. The 3rd outer margin is increased (`oma`), as the main title will be written there (`mtext`) (Fig. 6.15).

Then a new window is opened and the initial, the intermediate and the final density versus distance and mean density versus time plotted (Fig. 6.16). For the latter, we take the means of the aphid densities, one for each row (`rowMeans`). The figures are outlined in two rows, two columns (`mfrow`).

```
par(oma=c(0,0,3,0))

filled.contour(x=times,y=Distance,z=DENSITY,color=topo.colors,
              xlab="days", ylab="Distance on plant", m,main="Density")

mtext(outer=TRUE,side=3,"Aphid model",cex=1.5)

windows()
par(mfrow=c(2,2),oma=c(0,0,3,0))

plot(Distance,DENSITY[1,],type="l",lwd=2,
     xlab="Distance, m",ylab="Density", main="initial condition")
plot(Distance,DENSITY[100,],type="l",lwd=2,
     xlab="Distance, m",ylab="Density", main="100 days")
plot(Distance,DENSITY[200,],type="l",lwd=2,
     xlab="Distance, m",ylab="Density", main="200 days")

meanAphid <- rowMeans(out[,2:ncol(out)])
plot(times,meanAphid ,type="l",xlab="time, days",ylab="/m2",
     main="Density versus time")

mtext(outer=TRUE,side=3,"Aphid model",cex=1.5)
```

6.6.5 Fate of Marine Zooplankton in an Estuary (***)

Estuaries are characterized by large salinity gradients, ranging from fresh to marine. The animals that live in these systems need to be adapted to these abiotic conditions. Typically, three different types of zooplankton are found in estuaries: freshwater species that are carried in the estuary through the river flow, endemic brackish-water species, and marine species that enter the estuary from the sea, through the action of the tides. One very well studied estuary is the Scheldt estuary, located in Belgium and the Netherlands. Each year, marine species progressively ‘invade’ this estuary, moving up the salinity gradient and reaching their maximal upstream position in summer, after which they start retreating back to the sea.

In order to investigate whether the occurrence of these animals is merely due to physical transport (mixing), or whether the marine species are able to increase in the estuary, a model was made that describes the dynamics of marine zooplankton in the Scheldt estuary (Soetaert and Herman, 1994).

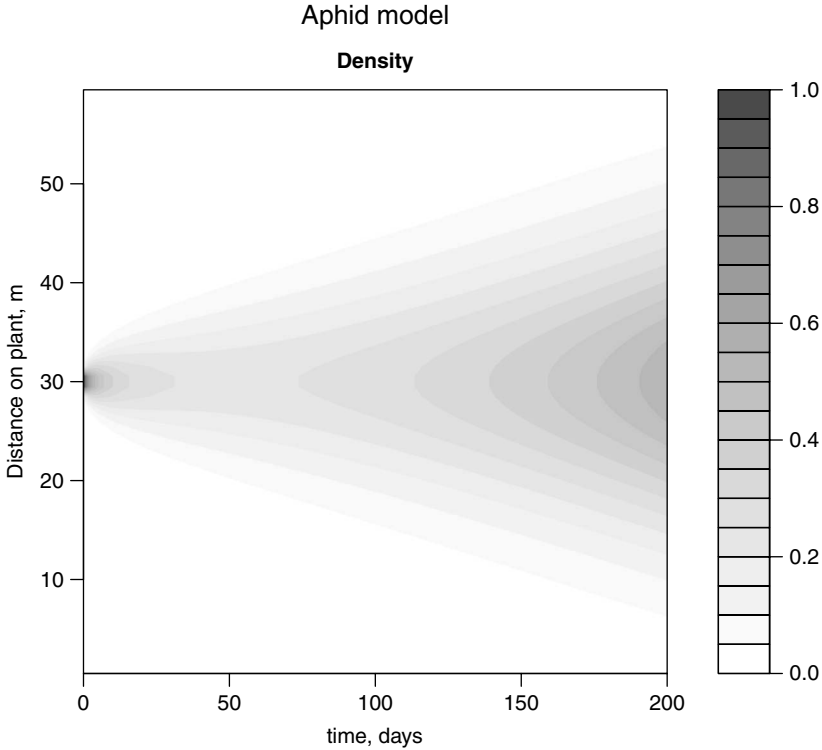


Fig. 6.15 Output generated with the Aphid dispersion-reaction model – see text for R-code

Marine zooplankton (C) is continuously driven to the sea by the freshwater flow (first term), and it is mixed into the estuary by the action of the tides (second term). To keep the model simple, the biological effects (third term) are represented by just one parameter, a net growth rate (g) of the zooplankton. If g is positive, the zooplankton grows in the estuary, if g is negative, there is only decay. The dynamic model reads (see Section 3.4.4):

$$\frac{\partial C}{\partial t} = -\frac{\partial}{\partial x}(Q \cdot C) + \frac{\partial}{\partial x}\left(A \cdot E \cdot \frac{\partial C}{\partial x}\right) + g \cdot C \quad (6.39)$$

Where C is zooplankton concentration ($\text{g dry weight m}^{-3}$), A is estuarine cross-sectional surface (m^2), Q is flow ($\text{m}^3 \text{d}^{-1}$), E is the tidal dispersion coefficient ($\text{m}^2 \text{d}^{-1}$) and x is the spatial position along the length axis (m). The spatial extent is subdivided into 100 boxes, extending from the river (box 1) to near the sea (box 100).

Aphid model

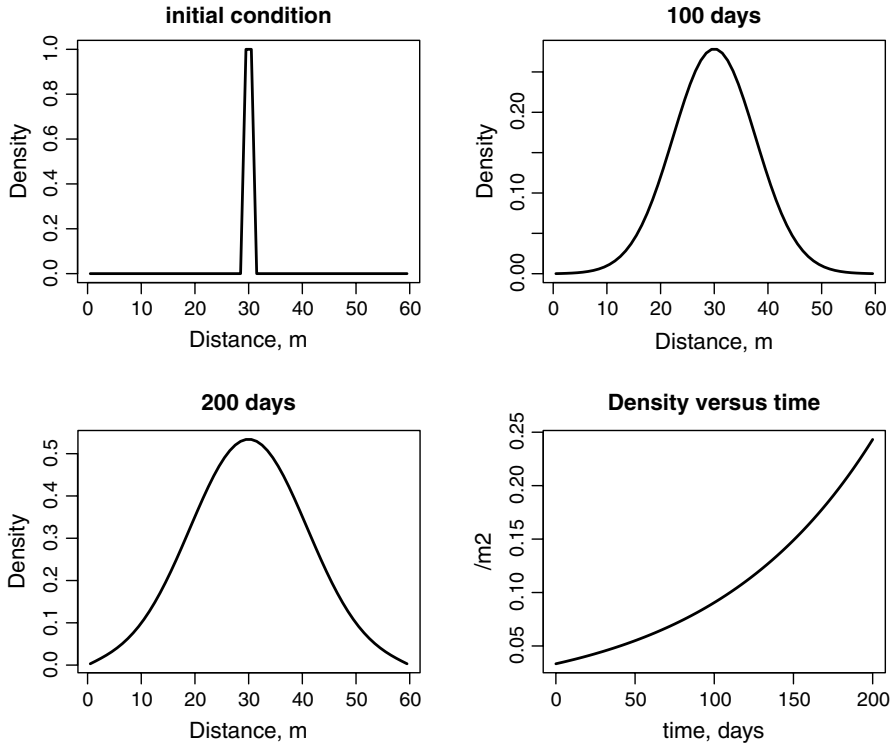


Fig. 6.16 Output generated with the Aphid dispersion-reaction model – see text for R-code

Numerical approximation

This partial differential equation is converted to an ordinary differential equation by approximating the spatial derivatives. It is simplest to start from the ‘flux divergence equation’:

$$\frac{\partial C}{\partial t} = -\frac{1}{A} \cdot \frac{\partial A \cdot J}{\partial x} + g \cdot C \quad (6.40)$$

And where the *total mass fluxes* $A \cdot J$ are defined as:

$$A \cdot J = Q \cdot C - A \cdot E_x \frac{\partial C}{\partial x} \quad (6.41)$$

To numerically approximate these equations, we keep in mind that fluxes are defined on the box interfaces, while concentrations are defined in the centre of the boxes. Thus, the rate of change of zooplankton concentration in box i can be approximated as:

$$\frac{dC_i}{dt} \approx -\frac{1}{A_i} \cdot \frac{\Delta_i(A \cdot J)}{\Delta x_i} + g \cdot C_i = -\frac{\Delta_i(A \cdot J)}{\Delta V_i} + g \cdot C_i \quad (6.42)$$

Where we have defined the volume of the box i as:

$$\Delta V_i = A_i \cdot \Delta x_i \quad (6.43)$$

and where Δ_i denotes that the mass flux gradient is to be taken around box i , i.e. as the difference of mass fluxes at the downstream ($i, i+1$) and upstream ($i-1, i$) interface:

$$\Delta_i(A \cdot J) = (A \cdot J)_{i, i+1} - (A \cdot J)_{i-1, i} \quad (6.44)$$

These mass fluxes are approximated as:

$$\begin{aligned} (A \cdot J)_{i-1, i} &\approx Q \cdot C_{i-1} - A_{i-1, i} \cdot E_{i-1, i} \cdot \left. \frac{\Delta C}{\Delta x} \right|_{i-1, i} \\ &= Q \cdot C_{i-1} - E_{i-1, i}^* \cdot \Delta C_{i-1, i} \end{aligned} \quad (6.45)$$

where we have defined the bulk dispersion coefficient E^* (units of $\text{m}^3 \text{d}^{-1}$) at the interface between box $i-1$ and i as:

$$E_{i-1, i}^* = \frac{A_{i-1, i} \cdot E_{i-1, i}}{\Delta x_{i-1, i}} \quad (6.46)$$

Note that we use $\Delta x_{i-1, i}$ here, i.e. the ‘dispersion distance’ from the centre of box $i-1$ to the centre of box i .

R- implementation

To make the model realistic, the estuarine morphology and physics is roughly patterned to the Scheldt estuary. This estuary is approximately 100 km long (`Length`). Combining the total length with the number of boxes (`nbox`), we calculate the distance, from the river (upstream boundary) to the upstream interfaces of each box (`IntDist`) and to the centre of the boxes (`Dist`). The cross-sectional area increases in a sigmoid fashion in this estuary, from around 4000 m^2 near the river to around 80000 m^2 near the estuarine mouth. We need to specify the cross-sectional surfaces both at the interfaces (`IntArea`) and in the centre of the boxes (`Area`). The latter is used to estimate the volumes of the boxes (`Volume`).

The effect of the tides is present up till the upstream boundary, where a sluice separates the estuary from the river. Here the tidal dispersion coefficient is 0 (`Eriver`); we assume that the dispersion coefficient increases linearly towards the mouth (`Esea`), where it equals 350 $\text{m}^2 \text{s}^{-1}$ (this is only a rough approximation). The tidal dispersion coefficient E ($\text{m}^2 \text{d}^{-1}$), is defined at the upstream interface of each box,

hence the linear interpolation uses `IntDist`. It is used to estimate the bulk dispersion coefficient (`Estar`, units of $\text{m}^3 \text{d}^{-1}$).

```
nbox      <- 100
Length    <- 100000                                # m

dx         <- Length/nbox                           # m

IntDist    <- seq(0,by=dx,length.out=nbox+1)        # m
Dist       <- seq(dx/2,by=dx,length.out=nbox)        # m

IntArea    <- 4000 + 76000 * IntDist^5 / (IntDist^5+50000^5) # m2
Area       <- 4000 + 76000 * Dist^5 / (Dist^5+50000^5)      # m2

Volume     <- Area*dx                                # m3

Erivier    <- 0                                       # m2/d
Esea       <- 350*3600*24                             # m2/d
E          <- Erivier + IntDist/Length * Esea         # m2/d

Estar      <- E * IntArea/dx                          # m3/d
```

The freshwater flow Q in this estuary fluctuates relatively smoothly from around $150 \text{ m}^3 \text{ s}^{-1}$ in winter to $50 \text{ m}^3 \text{ s}^{-1}$ in summer; we mimic this as a simple sine wave (parameters `meanFlow`, `ampFlow` and `phaseFlow`).

As the model is one-dimensional, and includes dispersion, we need to specify two boundary conditions, near the river and sea boundaries. Concentration of marine zooplankton in the river is set to 0 (parameter `riverZoo`), whilst at the seaward boundary, true measurements, at monthly intervals are imposed. These measurements are inputted as two vectors: the sampling time (`fZooTime`) and the zooplankton concentration at the sea boundary (`fZooConc`).

```
fZooTime = c(0, 30,60,90,120,150,180,210,240,270,300,340,367)
fZooConc = c(20,25,30,70,150,110, 30, 60, 50, 30, 10, 20, 20)

# the model parameters:
pars      <- c(riverZoo = 0.0,                # river zooplankton conc
               g         =-0.05,              # /day growth rate
               meanFlow  = 100*3600*24,        # m3/d, mean river flow
               ampFlow   = 50*3600*24,        # m3/d, amplitude
               phaseFlow = 1.4)                # - phase of river flow
```

The model itself is in a function called `Zootran` that estimates the zooplankton rate of change. It starts with estimating the current flow values (`Flow`) and inter-

polating the time series (`fZooTime`, `fZooConc`) to the current simulation time (`t`); the interpolation uses R's function `approx`.

Then the mass input fluxes due to advection and dispersion are estimated (`Input`), and the rate of change calculated as the sum of transport (flux gradient) and growth. The rate of change is returned as a list.

```
require(deSolve)

Zootran <-function(t,Zoo,pars)
{
  with (as.list(pars),{

    Flow    <- meanFlow+ampFlow*sin(2*pi*t/365+phaseFlow)
    seaZoo  <- approx(fZooTime, fZooConc, xout=t)$y
    Input   <- +Flow * c(riverZoo, Zoo) +
              -Estar* diff(c(riverZoo, Zoo, seaZoo))
    dZoo    <- -diff(Input)/Volume + g*Zoo
    list(dZoo)

  })
}
```

To run the model, we provide initial conditions (`ZOOP`), one value for each box, and a times sequence (one year, daily output). The model is best integrated with one of `deSolve`'s integration routines which is especially designed for solving one-dimensional reaction-transport models (`ode.band`). We specify that the model includes only one species (`nspec`).

```
ZOOP  <- rep(0,times=nbox)
times <- 1:365
out   <- ode.band(times=times,y=ZOOP,func=Zootran,parms=pars,nspec=1)
```

Finally, the output is depicted as a filled contour (Fig. 6.17).

```
par(oma=c(0,0,3,0))      # set margin size

filled.contour(x=times,y=Dist/1000,z=out[,-1],
               color= terrain.colors,xlab="time, days",
               ylab= "Distance, km",main="Zooplankton, mg/m3")
mtext(outer=TRUE,side=3,"Marine Zooplankton in the Scheldt",cex=1.5)
```

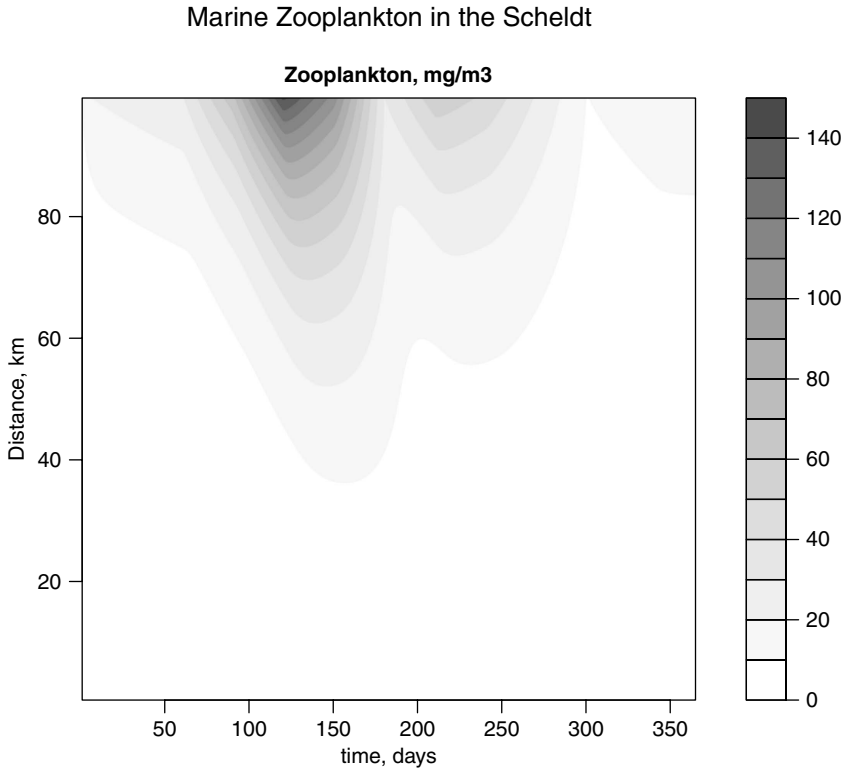


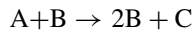
Fig. 6.17 Output generated with the Estuarine zooplankton transport-reaction model – see text for R-code

6.7 Projects

6.7.1 Numerical Solution of the Autocatalytic Reaction in a Flow-Through Stirred Tank

Implement and solve the model described in Section 3.6.1.

Two chemicals A and B are fed into a flow-through stirred tank where an auto-catalytic reaction occurs between A and B, and that produces substance C.



The rate of change of the concentrations [A], [B] and [C] is given by:

$$\frac{d[A]}{dt} = d_r \cdot (A_{in} - [A]) - k \cdot [A] \cdot [B]$$

$$\begin{aligned}\frac{d[B]}{dt} &= d_r \cdot (B_{in} - [B]) + k \cdot [A] \cdot [B] \\ \frac{d[C]}{dt} &= -d_r \cdot [C] + k \cdot [A] \cdot [B]\end{aligned}\quad (6.47)$$

Use following parameter settings:

$d_r = 0.05 \text{ s}^{-1}$, $k = 0.05 \text{ (mmol m}^{-3})^{-1} \text{ s}^{-1}$, $A_{in} = 1 \text{ mmol m}^{-3}$, $B_{in} = 0.1 \text{ mmol m}^{-3}$ and for initial conditions for $A(t=0) = 1 \text{ mmol m}^{-3}$, $B(t=0) = 0.1 \text{ mmol m}^{-3}$ and $C(t=0) = 0$.

Run the model for 300 seconds. Use `ode` to solve the model. Plot the concentration of substances A, B and C as a function of time (results: see Fig. 6.18).

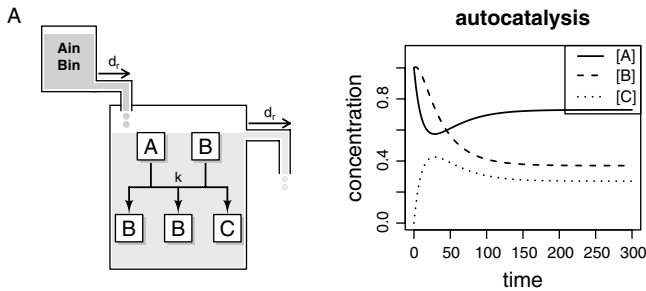


Fig. 6.18 Conceptual model (*left*) and output generated with the autocatalysis in a flow-through stirred tank model (*right*)

6.7.2 Numerical Solution of a Nutrient-Algae Chemostat Model – Euler Integration

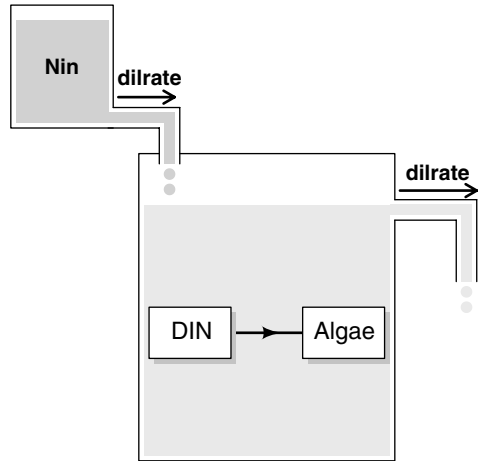
The next model is similar to the previous, except that algae are growing in a chemostat, that is poor in nutrients. Culture medium is pumped continuously into the vessel, where it is mixed homogeneously with the existing contents. An identical amount of the existing contents in the vessel is removed by this process. It is assumed that light is in surplus, so that the algal growth is limited only by nutrient availability (Fig. 6.19).

The model equations are:

$$\begin{aligned}\frac{d\text{PHYTO}}{dt} &= p_{max} \cdot \frac{\text{DIN}}{\text{DIN} + k_s} \cdot \text{PHYTO} - \text{dilrate} \cdot \text{PHYTO} \\ \frac{d\text{DIN}}{dt} &= -p_{max} \cdot \frac{\text{DIN}}{\text{DIN} + k_s} \cdot \text{PHYTO} - \text{dilrate} \cdot (\text{DIN} - \text{N}_{in})\end{aligned}\quad (6.48)$$

Use the following parameter values:

Fig. 6.19 Schematic representation of the chemostat model



$pMax$	1 d^{-1}
ks	1 mmol N m^{-3}
Nin	10 mmol N m^{-3}
$Dilrate$	0.24 d^{-1}

And initial conditions:

$DIN(t_0)$	$0.1 \text{ mmol N m}^{-3}$
$PHYTO(t_0)$	1 mmol N m^{-3}

Solve the model numerically using Euler integration:

$$C^{t+\Delta t} = C^t + \Delta t \cdot \frac{dC^t}{dt} \quad (6.49)$$

Tasks:

1. Simulate the dynamics of phytoplankton – DIN for 20 days.
Use 0.1 day as the time step.
Start by creating parameters and assigning initial values to the state variables. Then loop over time, starting with time = 0 and increasing time each iteration with the time step. At each iteration, first calculate the rate of change of DIN and the rate of change of algae, and then, update the concentrations using Euler integration.
Make a plot of nutrient and algal concentrations versus time.
Make a plot of algae versus nutrients.
2. Change the timestep of the model solution.
First decrease it, then increase it in steps of 0.1 d. always simulate a 20-day period.
What happens to your model solution when you decrease and increase the time step?

3. Set the timestep back to 0.1 day. Now increase p_{max} from its ‘standard’ value of 1 d^{-1} in steps of 0.5 d^{-1} .

Watch the stability of your solution. Conclude about the relation between time step length and dynamic properties of the model.

6.7.3 Rain of Organic Matter in the Ocean: Numerical Solution of the Advection-Reaction Model

Use the R-code of the aphid model (Section 6.6.4) as a template to implement a numerical solution of organic matter sinking through the aphotic part of an oceanic water column. The analytical solution was treated in Section 5.5.1.

The depth of this water column is 400 m. Organic matter is raining down from the productive euphotic zone. As it sinks through the water column, it is being degraded.

Assume that

- The organic matter has a constant sinking velocity u of 50 m.d^{-1} . The sinking of the organic matter is an advective process.
- Model the degradation as a first-order process, with a degradation rate k of 0.2 d^{-1} .
- The flux of organic matter from the euphotic zone, i.e. the upper boundary for our model, is prescribed as $Flux = 100 \text{ mmolC.m}^{-2}.\text{d}^{-1}$. At the lower end of the water column, material leaves the water column to settle on the bottom.
- There is no diffusive mixing.

Run the model for a sufficiently long time, such that the organic matter concentrations stop changing. Compare the final concentration gradient with the analytical solution.

6.7.4 AQUAPHY Model Implementation

Implement the AQUAPHY algal growth model (Section 2.9.2), under fluctuating light conditions.

A Practical Guide to Ecological Modelling
Using R as a Simulation Platform

Soetaert, K.; Herman, P.M.J.

2009, XV, 372 p., Hardcover

ISBN: 978-1-4020-8623-6