

Chapter 2

Fundamentals of Modeling, Principles and MATLAB®

2.1 Model Types

As the term of ‘model’ has a wide variety of meanings, clearly a very narrow type of models is addressed in this book. Even within the special field of ‘environmental models’ only the sub-class of *deterministic models* is treated; statistical models do not appear, although statistics plays a significant role in environmental sciences and technology.

In deterministic models all variables and parameters are functions of independent space and time variables. The *independent variables* are referred to by the usual notation x , y , z and t . In most models, especially in the relatively simple examples presented here, it is sufficient to formulate the problem considering only a subset of these four variables.

Depending on the number of space dimensions, one speaks of 0D, 1D, 2D or 3D models. 0D models have no space dependency, only a time dependency. Ecological models concerning populations of biological species in an environmental compartment are in their majority of that type. As t is the only independent variable, the analytical formulation leads to *ordinary differential equations*. These are differential equations, which depend on one variable only; in contrast to *partial differential equations*, where there are at least two independent variables.

Models with no time dependency are denoted as steady, *steady state* or stationary. The corresponding terms for time dependent simulations are: unsteady or *transient*. A steady state is approached in real systems, if the internal processes have time enough to adjust to constant outer conditions. It is a necessary condition for steady state that exterior processes or parameters do not change in time. Otherwise steady conditions cannot be reached.

1D models include one space dimension only. Models for the soil compartment are mostly 1D, as the changes in vertical direction are of concern: seepage to the groundwater table or evaporation to the ground surface. Processes in rivers (image a water level peak or a pollutant plume moving downstream) can be regarded in 1D under certain conditions. Water from surface water bodies infiltrating into aquifers

may be described by a 1D approach, if relevant conditions do not change substantially in the vertical direction and along the shoreline.

1D steady state models lead to ordinary differential equations. Transient models, including at least one space direction, lead to partial differential equations. It is important to know about these differences, as mathematical solution techniques for both types of equations are different and different MATLAB® commands need to be used. Here we use MATLAB® for steady and unsteady modeling in 1D.

2D models include two space variables. One may distinguish between 2D horizontal and 2D vertical models. Terrestrial ecology is a typical field, where this type of model is suitable, describing the distribution or population of species on the land surface. In streams or estuaries or in shallow water models are often set up for vertically averaged variables, for which a 2D horizontal description results. Models for 2D vertical cross-sections are obtained,

- In groundwater flow, where several geological formations are to be included, but no variations of hydraulic conditions in one horizontal direction
- In cross-sections of streams
- In air pollution modeling, if no space direction is preferential around a source; in that case the single radial coordinate r replaces two horizontal space variables x and y

3D models are quite complex in most cases and will marginally appear in this book, as the focus is on simple explanatory examples. Numerical algorithms using the methods of Finite Differences, Finite Volumes or Finite Elements are the methods of choice for modeling in higher space dimensions, steady and unsteady. The MATLAB® ‘Partial Differential Toolbox’ can be recommended for the application of these algorithms.¹ As the focus here is on core MATLAB®, we leave numerical methods out. Instead it is outlined, how core MATLAB® can be applied for steady state modeling in higher dimensions, based on computing of analytical solutions.

2.2 Modeling Steps

The task of modeling can be sub-divided in several steps. The way from a real system to the working model contains different tasks, where every step depends on good performance of the previous step. The major steps are to build a conceptual model, to describe it by mathematical analysis, to solve the differential equations by computational methods and finally to perform post-processing tasks. A schematic overview of the procedure is given in Fig. 2.1 (see also: Holzbecher 1998).

¹ In the partial differential equations toolbox, the modeling of advection processes (see below) is difficult and requires numerical skills. All other processes can be simulated using the appropriate commands.

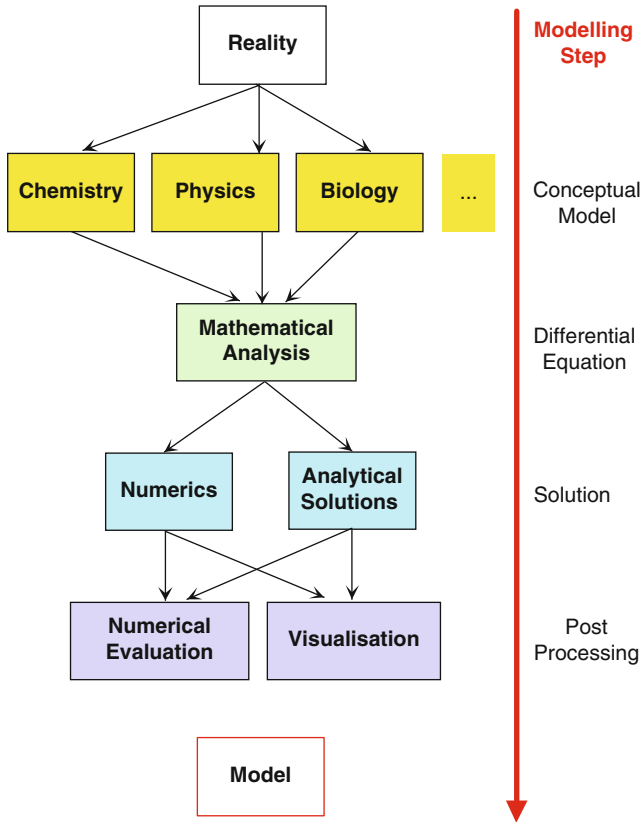


Fig. 2.1 Modeling steps

At first a conceptual model has to be formulated. From the available scientific and technical expertise and knowledge, as well as the experience with observations of the system in question, a concept has to be set up. The concept involves all processes, which could be relevant for the studied system. This first step is a qualitative one, i.e. there are no numbers involved yet. Scientific or technical expertise from the involved branches has to be included. In case of environmental problems advice has to be obtained from several disciplines mostly: chemistry, physics, biology, biochemistry, geology, biogeochemistry, ecology, hydrology, hydraulics, or hydrogeology.

The next step is the formulation of the model in mathematical terms. Variables and parameters as functions of time and space are related to each other by mathematical expressions. Rearrangements and transformations of the expressions usually lead to the formulation of differential equations. Fundamental theoretical or empirical laws and principles are combined to finally yield differential equations. In the simplest case there is a single equation only, in general a system of equations emerges. It can be *ordinary differential equations*, which have a single independent

variably. More general *partial differential equations* depend on more than one independent variables. As will be shown in numerous examples the differential equations need to be accompanied by boundary and initial conditions, in order to complete the mathematical formulation.

With the next main step, according to the list in Fig. 2.1, we come to the computer. The solutions of the differential equations, under consideration of initial and boundary conditions, have to be calculated, which is always done on a computer. There is not a single strategy, which delivers these solutions and thus different paths have to be followed. Sometimes the solutions can be expressed by explicit formulae, which are quite easy to implement using any mathematical software. An example for such a situation with an *analytical solution* was presented in Chap. 1.3. With the exponential function the example formula is much simpler than in more general cases. However, in most cases even sophisticated analytical formulae do not suffice.

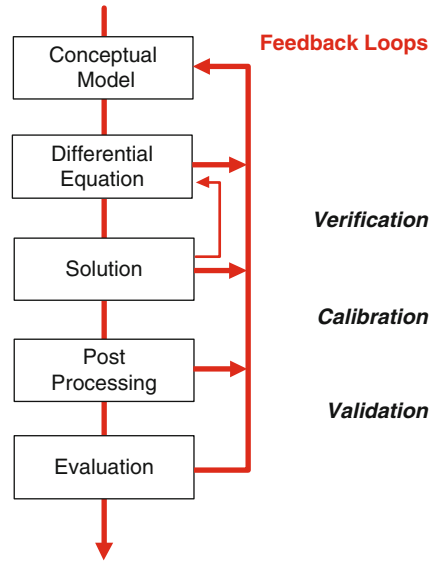
For most problem formulations in terms of differential equations numerical methods are required. As there is no explicit formula available, the solution has to be found approximately by so called *numerical methods*. It is sufficient to find such an approximate solution, as there are tolerance parameters, which mostly ensure to reach a good accuracy. Fortunately these methods need not to be implemented by the modeler: there are outworked strategies available in software packages. MATLAB® solvers for ordinary and partial differential equations will be presented.

In order to solve a problem usually some data need to be made available to the software program. Thus the computer very often comes into play even before the calculation of the solution is at stake. These tasks are referred to as *pre-processing*. A simple example is to transfer a parameter value from one physical unit into another. The computation of one parameter from another or from several others may be more complex. A more challenging task is the determination of parameter distributions within a model region based on some measured values. It will be shown how such tasks can be performed easily using MATLAB®.

After the approximate solution is computed (the computer always delivers approximate solutions, also when analytical solutions are evaluated!), there are usually several *post-processing* steps. Almost always the modeler and her/his client appreciate to have a graphical representation. Another task is compute fluxes for some key variables, may be in order to set up an entire balance for a model entity. For such a purpose numerical integration is a useful tool, which is also possible using MATLAB®. As another example the user may like to compare calculated and measure values. These few examples demonstrate that post-processing is a problem-specific task.

Finally the steps lead from a real system to a computer model. The step concept, sketched in Fig. 2.1, needs not to be followed strictly. In practise work will be on different steps at the same time. *Feedback loops* within the task list are necessary to improve earlier approaches, to correct errors and to adjust the model in view of measured data. Often new data come in after the start of modeling, which make adjustments necessary.

Fig. 2.2 Verification, calibration and validation as feedback loops between different model levels



The feedback loops within the system of tasks can be related to terms as verification, calibration and validation, which is visualized in Fig. 2.2. All these terms do not have a uniquely definition and thus may be found in slightly different contexts in the scientific literature. The term *verification* is mostly used in connection with software testing, which is a crucial step of software development. When a software code is entered and runs regularly without error messages from the computer, it is not sure that there are no ‘bugs’ in the code. In order to test the correct performance of the computer program, test cases are set up, to check, if the program delivers the correct answer. Test cases can be based on simple post-processing, on analytical solutions, on theoretical derivations and on inter-comparison with results from other codes.

Comparison with test cases, which are generally accepted in the concerned scientific community (so called benchmarks), is called *benchmarking*. Within the outlined step concept verification is thus a feedback loop in which it is checked, whether the computed solution delivers the solution of the differential equation. Unless it is a simple straight forward computation, the MATLAB® modeler also has to verify her/his implemented m-code. In case of errors the code *debugging* becomes necessary. How to ‘debug’ in MATLAB® is explained in Sect. 2.7.

The term *calibration* is used for the procedure of adjusting the model parameters for a specific application of the code. The term is almost identical to *parameter estimation* and strongly related to the term *inverse modeling*. Within the step concept the term means a feedback loop, in which the solution or some entity that is determined by post-processing is compared with values, obtained from the real site. In case the check is negative, usually some parameter values have to be adjusted in order to obtain a good fit. If that does not help, it may become necessary to make adjustments in the mathematical formulation or even in the conceptual model.

The work of *validation* is the most challenging. As the result of such work it is proven that a model is valid, i.e. behaves like the real system. The term usually is restricted to a field site application, but it sometimes also refers to the fact that a code can be applied for a certain type of applications. In the later case for each application site-specific parameters have to be included. Connected to software tools the term ‘valid’ remains slightly obscure as its concrete meaning is to be specified for each application field. It has to be clarified, which aspects of the real model should be represented by the model. As a model is not identical to the represented real system, there are always real world aspects for which the model is not sufficient.

The step concept, visualized in Fig. 2.1 and Fig. 2.2, is less a work schedule than a priority list. The mathematical formulation has to be based on a good conceptual model. If the mathematical formulation is insufficient, good solution techniques will not improve the model. One has to be sure that the solvers deliver accurate results, before putting extensive efforts into post-processing.

2.3 Fundamental Laws

The mathematical analytical formulation is based on fundamental principles and on empirical laws. From the former most important are the principles of conservation:

- Mass conservation
- Momentum conservation
- Energy conservation

Total mass, momentum and energy are preserved. If there are losses or gains, these are introduced in the conservation formulation as sources or sinks.

2.3.1 Conservation of Mass

The most nearby and most common application of the continuity equation is that for mass. There are two types of mass conservation. One type is the mass conservation of the medium, which can be solid, aqueous or gaseous. The mass is expressed in terms of a density ρ with a physical unit $[M/L^3]$. ‘M’ represents a mass unit and ‘L’ a length unit. For example the density of fresh water at a temperature of 4°C and the pressure of 101,325 Pa (1 atm) is 1,000 kg/m³.

The second formulation of mass conservation in a fluid concerns biogeochemical species within a fluid. In that case the mass is expressed in terms of the concentration c . The continuity equation then is formulated in terms of the concentration c of the species. The concentration also has the unit $[M/L^3]$ and measures the mass within a volume of fluid. The content of chloride Cl⁻ in seawater amounts to 19 g/l.

One has to extend the mass definition in situations, where the fluid does not fill the entire volume. Then the total mass per space volume is expressed by θc with porosity θ as additional factor. Porosity is dimensionless and measures the volumetric share of the fluid phase on the total volume. The physical dimension of the product is thus further $[M/L^3]$. In aquifers groundwater usually fills approximately 25% of the volume; thus holds: $\theta=0.25$.

The described concept can be extended to situations with several phases, where each phase has its own θ -value. In the unsaturated soil zone below the ground surface, above the groundwater table there are three phases present: the soil as solid phase, seepage water as liquid phase and soil air as gaseous phase.

The mass of a gas component is expressed in terms of partial pressure. According to the ideal gas law the product of pressure and volume is a constant, which changes only with temperature. Thus mass conservation can be formulated in terms of pressures instead of volumes. According to Dalton's Law² the pressures of a gas mixture have to be summed up to yield the total pressure.

2.3.2 Conservation of Momentum

The momentum of a fluid is expressed as the product $\rho \mathbf{v}$, where $\mathbf{v}$ denotes the velocity. The physical unit of momentum is $[M/(L^2T)]$, where the letter 'T' represents a time unit. As velocity is a vector, the momentum also is a vector, with one vector component for each space dimension of the model.

2.3.3 Conservation of Energy

The kinetic energy of a fluid is expressed as $\frac{1}{2}\rho v^2$ with the physical unit $[M/(LT^2)]$. If energy is measured in Joule, the given expression measures Joule per volume.

In problems, which include heat transfer, thermal energy is expressed in terms of temperature T . The energy content per volume is given by the product ρCT , where the new factor C is the specific heat capacity. The physical unit of C is $[L^2/(T^2K)]$, where 'K' represents the temperature measure unit, mostly °Celsius or °Kelvin. In many tables values for the product ρC can be found, which is addressed simply as heat capacity. Heat capacity has the unit $[M/(LT^2K)]$. If energy is measured in Joule ρC has the physical unit of J per volume and °Kelvin, while C is measured in the unit Joule per mass and °Kelvin.

² John Dalton (1766–1844), English chemist and physicist.

2.4 Continuity Equation for Mass

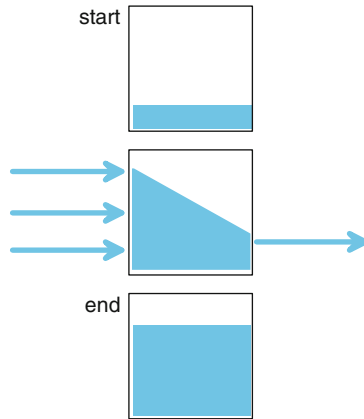
For the mathematical formulation of mass conservation consider the change of mass during the small time Δt within a control volume with spacing Δx , Δy and Δz , each for one direction in the three-dimensional space. There are two ways to calculate changes of mass. One method is to consider the mass within the control volume at the beginning and at the end of the time period and calculate the difference. The other method is to balance all fluxes across the boundaries of the volume. Balancing means that fluxes into the volume have to be taken as positive, while those leaving the volume are negative. In three-dimensional space six faces of the control volume have to be taken into account.

A simpler set-up for the one-dimensional space is depicted in Fig. 2.3. A box contains a certain amount of mass at the start of the time period, and a different amount at the end. During the time period there was influx on one face and outflux at the other. The graphical symbols in the equation at the bottom of the figure will be replaced by mathematical formulae in the following derivation.

The mass at the beginning and the end of the period t and $t+\Delta t$ is given by:

$$\theta \cdot c(x, t) \cdot \Delta x \Delta y \Delta z \quad \text{and} \quad \theta \cdot c(x, t + \Delta t) \cdot \Delta x \Delta y \Delta z$$

where θ denotes the share on the total volume. In case of a saturated porous medium θ denotes porosity. In the unsaturated zone, within soil for example, θ is the volumetric water saturation, when the aqueous phase is concerned. In the situation in which two fluids occupy the space (for example water and oil) the share of each



Principle of mass conservation

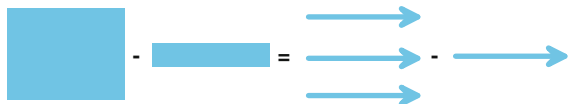
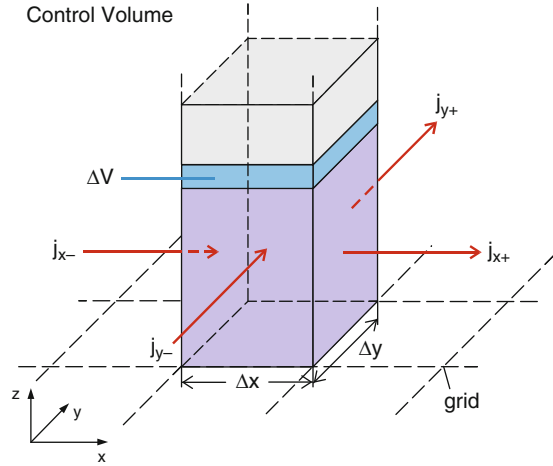


Fig. 2.3 Illustration for the derivation of the mass conservation equation

Fig. 2.4 Illustration of a control volume in two space dimensions x and y



phase has to be taken into account, too. $\Delta x \Delta y \Delta z$ is the volume. c denotes the concentration, measured as [mass/volume]. The change of mass per time is given by:

$$\theta \cdot \frac{c(x, t + \Delta t) - c(x, t)}{\Delta t} \cdot \Delta x \Delta y \Delta z$$

Fluxes in x -direction are given across faces of the control volume:

$$\theta j_{x-}(x, t) \Delta y \Delta z \quad \text{and} \quad \theta j_{x+}(x, t) \Delta y \Delta z$$

where j_{x-} denotes mass flux per area across the left face of the volume, in negative x -direction. Analogously j_{x+} denotes the mass flux in x -direction across the right face, in positive x -direction (see Fig. 2.4). The fluxes may change spatially and temporally which do the brackets indicate. Both fluxes are positive, if they add mass to the control volume, and negative otherwise. The physical unit of mass flux is $[M/(L^2 \cdot T)]$. The term $\theta \Delta y \Delta z$ denotes the area, through which flow takes place.³

The balance between both flux terms is thus given by:

$$\theta(j_{x-}(x, t) - j_{x+}(x, t)) \Delta y \Delta z$$

For simplicity the fluxes across the four other faces are neglected for the derivation at this point. One may assume here that the flux components in y - and

³ It is generally assumed that the volumetric share and the area share compared to the entire volume or area respectively are both quantified by the same number, here θ . This is not necessarily true. Especially in technical systems such as filters both ratios may vary significantly. The practitioner in the field is usually happy, if there is one value at all.

z-direction are zero. As stated above, both formulations measure the change of mass and thus need to be equal:

$$\theta \frac{c(x, t + \Delta t) - c(x, t)}{\Delta t} \cdot \Delta x \Delta y \Delta z = \theta (j_{x-}(x, t) - j_{x+}(x, t)) \Delta y \Delta z \quad (2.1)$$

Division through the volume $\Delta x \Delta y \Delta z$ and porosity θ yields:

$$\frac{c(x, t + \Delta t) - c(x, t)}{\Delta t} = - \frac{j_{x+}(x, t) - j_{x-}(x, t)}{\Delta x} \quad (2.2)$$

From this equation a differential equation can be derived by the transition of the finite grid spacing Δx and time step Δt to infinitesimal expressions, e.g. by the limits $\Delta x \rightarrow 0$ and $\Delta t \rightarrow 0$. It follows:

$$\frac{\partial c}{\partial t} = - \frac{\partial}{\partial x} j_x \quad (2.3)$$

which is a differential formulation for the principle of mass conservation. The presumption for the differentiation procedure is that the functions c and j_x , are sufficiently smooth, mathematically speaking differentiable, which is usually taken for granted. Equation 2.3 is valid for one-dimensional transport and is the basis for the mathematical analysis of transport processes. The unit of the equation is $[M/(L^3 \cdot T)]$.

Formulation (2.3) is valid if there are no internal sources or sinks for the concerned biogeochemical species. Sources and sinks are understood here in the most general sense: each process, which creates or destroys some species mass, can contribute to such a source or sink. In the remainder of this volume we will see examples, where chemical reactions and inter-phase exchange of species can be included in that way.

Easily the given mathematical formulation can be extended to consider sources and sinks additionally. If these are described by a source- or sinkrate $q(x, t)$ $[M/(L^3 \cdot T)]$, which may vary spatially and temporally, one simply has to add a corresponding integral term

$$\int_{\Delta x \Delta t} q(x, t) dt dx$$

on the right side of (2.1) and (2.2). The term is positive, if mass is added (source) and negative, if mass is removed (sink). In the derivation of (2.3) the integral term had to be differentiated, which leads to the general transport equation in one space dimension:

$$\theta \frac{\partial c}{\partial t} = - \frac{\partial}{\partial x} \theta j_x + q \quad (2.4)$$

Flux components in y - and z -direction can also be taken into account, based on formulae analogous to formula for the x -direction. The fluxes j_{y-}, j_{y+}, j_{z-} and j_{z+} have to be introduced, balanced and the balances added on the right side of (2.1) and (2.2). Taking the limits $\Delta y \rightarrow 0$ and $\Delta z \rightarrow 0$ one obtains:

$$\theta \frac{\partial c}{\partial t} = - \left(\frac{\partial}{\partial x} \theta j_x + \frac{\partial}{\partial y} \theta j_y + \frac{\partial}{\partial z} \theta j_z \right) + q \quad (2.5)$$

which is the generalized formulation of the mass conservation for three space dimensions. Using the formal ∇ -operator (speak: ‘nabla’),

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} \text{ in 3D, } = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \text{ in 2D, } = \frac{\partial}{\partial x} \text{ in 1D} \quad (2.6)$$

the equation can be written more compactly:

$$\theta \frac{\partial c}{\partial t} = -\nabla \bullet \theta \mathbf{j} + q \quad (2.7)$$

With the different forms of the ∇ -operator the short notation of the continuity (2.7) is valid in one-, two- or three-dimensional space. On the right side the ∇ -operator is multiplied by the flux vector $\theta \mathbf{j} = \theta \begin{pmatrix} j_x \\ j_y \\ j_z \end{pmatrix}$ as a vector product. In the formulae, here and in the following, the $\bullet$ denotes the standard *vector product*,⁴ which in MATLAB® is applied by using the $\star$ multiplication and the transpose of one column vector. Examine with the following command:

```
[1;2]’ * [1;2];
```

An advantage of the formulation (2.7) is that it is valid for one-, two- and three-dimensional situations. The number of components in the flux-vector and ∇ -operator is equal to the number of space dimensions. In two dimensions, as illustrated in Fig. 2.4, the flux vector has two components. The illustration is concerned with a fluid, for which the mass conservation principle can also be applied, as for any other chemical species. When the fluid density is not changing,

⁴ In three dimensions for vectors arbitrary vectors $\mathbf{u}$ and $\mathbf{v}$:

$$\mathbf{u} \bullet \mathbf{v} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} \bullet \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = u_x v_x + u_y v_y + u_z v_z, \text{ not to be confused with the cross-product } \mathbf{u} \times \mathbf{v};$$

another formulation, found in the literature is: $\frac{\partial c}{\partial t} = -\text{div} \mathbf{j}$; divergence ‘div’ is another expression for a vector product with the nabla-operator.

one may express mass change by volume change (ΔV), which itself is represented by a change of the ‘water’-table.

The derived equation for mass conservation alone is not yet sufficient for a complete mathematical formulation. There are too many unknown variables, namely concentration c , and the components of the flux vector $\mathbf{j}$. In order to reduce the number of unknowns, one has to utilize a formulation, in which the flux terms are connected with the concentrations. Finally an equation will result, in which c is the only unknown variable.

The advective flux is simply given by the product of the concentration and the velocity. In the three-dimensional case the three flux components are determined by the three velocity components:

$$j_x = v_x c \quad j_y = v_y c \quad j_z = v_z c \quad (2.8)$$

Independent of the dimension, using the scalar multiplication one may write in vector notation:

$$\mathbf{j} = \mathbf{v}c \quad \text{or} \quad \mathbf{j} = c\mathbf{v} \quad (2.9)$$

In formulae we mostly omit the multiplication sign. Note that on the right side of (2.9) there is scalar multiplication: the scalar variable c is multiplied with the vector variable $\mathbf{v}$, as already outlined in Chap. 1.

2.5 MATLAB® M-files

The command window is good for an introduction into MATLAB®. It demonstrates that operations, which someone used to compute on a scientific calculator, can be better performed using MATLAB®. However, MATLAB® can do much more.

In MATLAB® a high level programming language is included. It is often called ‘M’. Files, containing ‘M’ source code, are stored in files with extension ‘.M’. The programming work with m-files replaces extensive operating in the command window. Only for certain tasks, the command window will remain the most direct and simple way to compute with MATLAB®. Program writing and editing, not only with MATLAB®, is done by use of a text editor.

MATLAB has a built-in editor. The editor can be called in different ways. The easiest way is to use

```
edit
```

in the command window. Other convenient ways of calling the editor are given in the next section. Options for the editor are set in the ‘Preferences’ sub-menu of ‘File’ (see Fig. 2.5). It is also possible to use other than the built-in editors. With the

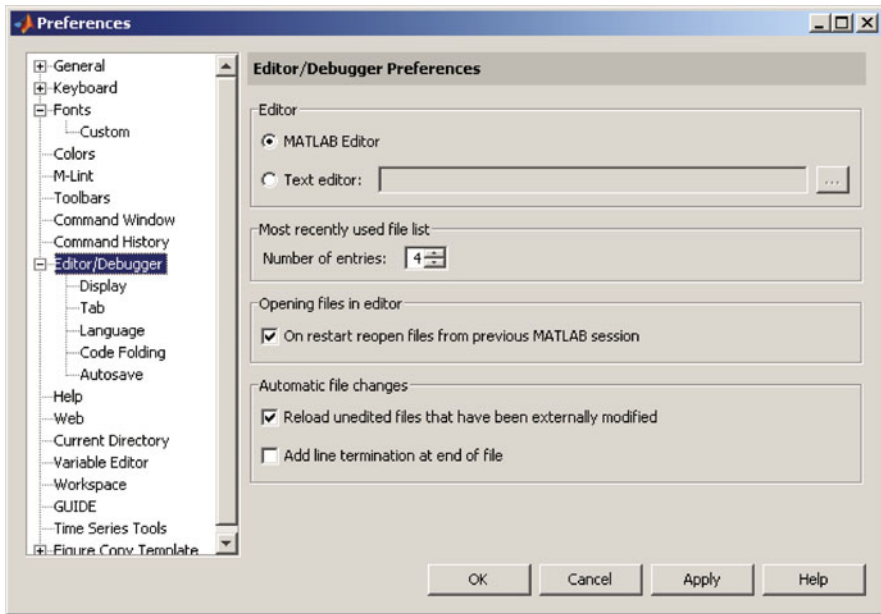


Fig. 2.5 Preferences, relevant for the editor

editor the programmer works on m-files. In the first chapter it was shown how MATLAB® operations are stated in the command window. Even after few exercises the user will recognize that it is often necessary to give a former command again or to give it in a slightly different form; maybe just with a parameter name altered. It was already shown that the command history view of the MATLAB® graphical user interface is an appropriate tool to go back to former commands. As already mentioned an alternative way is to use the up- and down-arrow buttons of the keyboard.

There is another alternative, which for most purposes turns out to be so powerful that most users prefer it for their normal work in comparison to the command window. MATLAB® command sequences can be gathered and stored as files. The extension of these files is simply *.m*; for that reason they are called *M-files*. What the MATLAB® user does with M-files, is what programmers do with other programming languages, as FORTRAN, JAVA or some C variant, just to mention some names.

In order to create a new M-file we use the 'New→M-file' entries of the 'File' main menu, as shown in Fig. 2.6. In the same menu there is an entry for opening an already existing M-file.

A simple example demonstrates the procedure. As just described, create a new M-file. The MATLAB® editor appears. The main menu of the editor is depicted in Fig. 2.7. Type the following commands:

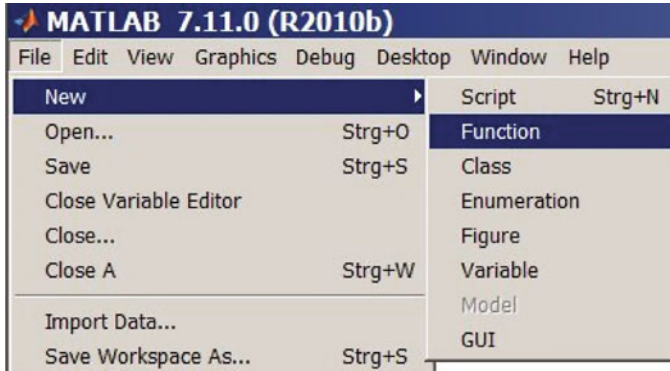


Fig. 2.6 File submenu entries for M-files and other files

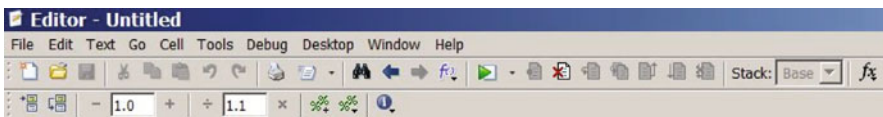


Fig. 2.7 MATLAB® editor; main menu entries of the graphical user interface

```
c0 = 1;
t = [0:0.1:1];
f = c0*exp(-lambda*t);
plot (t,f);
```

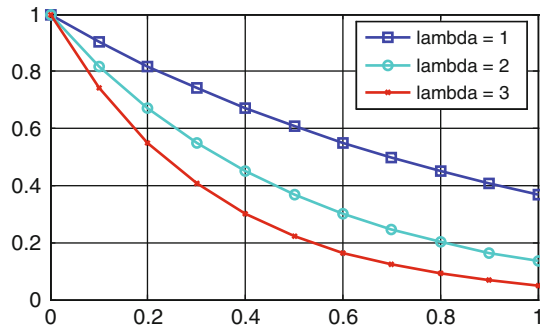
Use the ‘Save’ or ‘Save as...’ entries in the ‘File’ menu of the editor, or the corresponding button, to save the file under the name ‘*example.m*’. Return to the command window and type:

```
lambda = 1;
example;
```

A graph showing concentrations decreasing with time appears.⁵ All commands of the ‘*example.m*’ file are executed, when **example** is used as command. A bundle of graphs can now be plotted in the same figure without much typing work:

⁵ Depending on the specific installation and organization on the computer, the user could have a problem here, if MATLAB® does not find the ‘*example.m*’ file. The user should store the file in a directory, which is included in the MATLAB® path list. Use the command **path**, to see the current path list of the installation. In case of problems store the file in another directory or use the **addpath** command, to add another directory in the path list. There may also be a problem, if the user chooses an M-file name that already exists within the directories of the path list. Use the **which** command in order to check, if your stored version of the file is the most nearby version in the MATLAB® system on your machine!

Fig. 2.8 Graphical output of 'example.m' file



```
hold on;
lambda = 2;
example;
```

A second graph appears. Proceed further:

```
lambda = 3;
example;
legend ('lambda = 1', 'lambda = 2', 'lambda = 3');
```

A graph, showing stronger degradation with increasing **lambda** – value, appears. It should look similar to Fig. 2.8

The example shows that it is convenient to call repeatedly appearing command sequences in an M-file. The demonstration application is a mixture of commands on the command window, and of editing a file. Alternatively the entire command sequence can be started in an M-file. There is also an alternative way to start the current M-file: the button  in the main menu of the editor.⁶ The procedure is

exemplified in the next section. There is an alternative way to create an M-file directly from the command history. One or several command in the command history can be highlighted. By using the right mouse button one gets several options. One option is to create a M-file directly from the selected commands.

There are two types of M-files: *scripts* and *functions*. Scripts are simply files containing a sequence of commands. By typing the filename, the commands are executed subsequently. Variables, which are not specified or initialized in the script have to be available in the workspace, when the script is executed. Variables, specified in a script, are available in the workspace after execution, if they are not cleared explicitly.

Like scripts functions contain sequences of commands, and are stored with the extension '.m'. Formally they differ from scripts as the first no-comment line starts

⁶The button does not appear, when the editor is called outside of MATLAB®, for example by double-click on a M-file from the operating system.

with the **function** keyword. What follows is a list of output parameters, the equality sign, the function name and a list of input parameters:

```
function [out1, out2,...] = myfun (in1, in2, ...)
```

Also alike scripts functions are called by using the function name and *formal parameter* sets for input and output.

```
[out1, out2,...] = myfun (in1, in2, ...)
```

either from the command window, from a script or from another function. The filename must be identical with the function name, i.e. in the example: 'myfun.m'. Formal parameters in the calling command and the function are identified on basis of the parameter lists. They need **not** be identical! For example the call:

```
myfun (5, A)
```

results in a continuation of the execution with the function commands, where the parameter **in1** obtains the value 5, and **in2** gets the value of variable **A**. No output is taken from the function in this case. By using

```
[B, A] = myfun (5, A)
```

B and **A** obtain values calculated during execution of the function. Then the function commands are executed, the execution returns to the calling statement and proceeds with the following command.

Functions may contain subfunctions. Subfunctions are not visible outside the file where they are defined. Normally functions return when the end of the function is reached. The

```
return
```

statement can be used to force an early return. If there are functions with identical names, the ones defined as subfunctions have priority. In general the programmer may use the

```
which
```

command to find the location of function with highest priority. The

```
nargin
nargout
```

commands deliver the numbers for input resp. output parameters. This can be used to deal with varying formal parameters, for example to allow the user to call the file with a reduced number of parameters. This is feasible if for missing parameters default values are specified.

2.6 Ifs and Loops in MATLAB®

In M-files quite often commands or sequences of commands are to be executed under certain conditions only. This can be easily programmed using the `if`, a keyword that is available in all programming languages. The following command sequence in pseudo-programming language explains its application:

```
if condition
    commands1
else
    commands2
end
```

If the `condition` is fulfilled, `commands1` are executed. Otherwise `commands2` are executed. The `else` block can be omitted if there is nothing to be done, if the `condition` is not fulfilled. Conditions can be formulated differently. Instead of a rigorous definition here four mainly self-explaining examples:

```
a==b   a>1   a<b   a
```

In order to be valid conditions, `a` and `b` must be of same data type. There are two equality signs in the first example in order to distinguish a condition from an assignment. In the second example the condition is clear, if `a` is a number. But the condition is also valid if `a` is of different data type, which the reader may explore. The last of the four conditions is fulfilled if `a` is a positive non-zero number, otherwise not. The letter represents the value of a condition, and could also be defined like this in MATLAB®:

```
c = (a>=0);
if c a=1; end
```

By the example sequence the value of `a` is changed to 1, if the former value was positive or equal to zero. Variables as `c` of the example are so-called *logicals*. These are variables that can take only two values: ‘true’ or ‘false.’ As other programming languages MATLAB® treats logicals as binary, i.e. they have values 1 for ‘true’ and 0 for ‘false.’

```
2>1
ans =
    1
```

Logical variables can be combined by logical operators (and, or, not and exclusive or), as given in Table 2.1. Example:

```
if A>0 & F==1
```

Table 2.1 Defining table for logical operators

Input parameter		and	or ^a	not	xor
A	B	A & B	A B	~A	xor(A,B)
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

^a The | symbol is reached by Alt-Strg->

It is also possible to check the type or any other characteristic of a variable. For that purpose there are several `is...` keywords (which have at least one variable as formal parameter):

- `isfloat` checks if a variable is a float, i.e. a single or double
- `islogical` checks if a variable is a logical
- `isscalar` checks if variable is a scalar
- `isvector` checks if variable is a vector
- `isfinite` checks if variable is finite
- `isinf` checks if variable is infinite
- `isnan` checks if variable is not a number
- `isnan` checks if variable is not a number
- `isempty` checks whether an array is empty
- `isnan` checks if variable is not a number

Look for `is*` in the help system (see Chap. 1.5) to obtain a complete list of all such commands.

The `if`, `else` and `end` keywords are reserved and should not be used by the MATLAB® user as function or variable names. The `if`, `else` and `end` keywords are reserved and should not be used by the MATLAB® user as function or variable names. An `if` without `else` is just a condition. Using the `if-else` construction, one has the option to bifurcate into one of two branches. A bifurcation into several branches can be realized by using nested if-else commands:

```
if condition1
    if condition2
        commands1
    else
        commands2
    end
else
    if condition3
        commands3
    else
        commands4
    end
end
```

There is also the possibility for a direct multiple branching. For this purpose MATLAB® has the `switch-case` construction. It is exemplified in the following examples:

```
switch i
case 0
    commands1
case 1
    commands2
case 2
    commands3
end
```

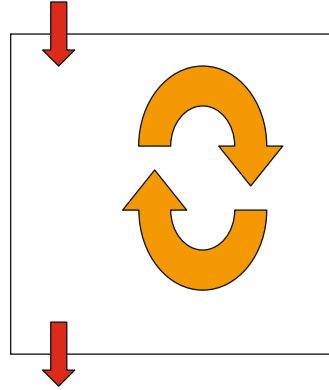
In order to use the functionality of a single M-file for the example from the preceding section, the programming technique of *loops* has to be applied. How loops work, is best explained by an example. The most common `for`-loop is used in the command sequence:

```
c0 = 1;
t = [0:0.1:1];
figure; hold on;
for lambda = 1:1:5
    f = c0*exp(-lambda*t);
    plot (t,f);
end
legend ('1','2','3','4','5');
```

`for` is a MATLAB® keyword, which is indicated by a different color. MATLAB® keywords may not be used as variable names. The statement, starting with the `for` keyword, starts a loop, which ends at the line with the `end` keyword. `lambda` is the loop variable in this example (the user is free to choose the name of the variable). In the first run through the loop `lambda` has the value 1. The commands within the loop are executed with that parameter value; here the function vector `f` is evaluated and a graph is plotted in the figure. Then the commands are executed again with the variable parameter taking the next value. In the example `lambda` gets the values 2, 3, 4 and finally 5. After the last run through the loop the execution continues with the next statement after `end`. In the example after the loop, the final command adds the legend in the figure. As described in Sect. 2.5 store the entire M-file under a new name and call that name in the

command window; or use the  button (old version:  (Fig. 2.9)).

Note that commands within loops are indented, as shown in the example above. This is not required by MATLAB®, but it is highly recommended, as it enhances the readability of the M-files significantly. Loops may be nested, i.e. inner loops may appear within outer loops. In that case it is important to recognize the corresponding `end` statements. It is good practise to indent commands with every inner loop even further. In that way the programmer visually obtains a connection between the start of a loop (with either a `for` or a `while` keyword) and the corresponding `end`.

Fig. 2.9 Illustration of a loop

Like in any other programming language, also in MATLAB®, there are several other types of loops. Another main type is the **while**-loop. Here a condition is checked at the start of the loop, and the commands of the loop (between **while** and **end** statements) are executed as long as the condition is fulfilled. Here the use has to take care that some of the variables appearing in the condition change during the loop; otherwise an infinite loop is produced.

If in larger loops it is necessary to have another outlet from the loop aside from the default at the **end**, one may use the **break** command. In nested loops the command only concerns that one loop, in which the statement is placed, i.e. execution proceeds with the command after the next end of loop statement.

continue is a related command: the execution is skipped not for the entire concerned loop (as with **break**), but only for the remaining statements in the current run through the loop.

Loops can also be used in the MATLAB® command mode. A trivial example is:

```
for i = 1:4
```

Note that no prompt appears after pressing the return key. The command is not yet executed. Enter:

```
a(i) = i
end
```

When the **end** command is entered the entire loop is executed and the prompt appears in the command window again. Note that in the specification of the loop variable **i** only two values are given. The increment 1 is used as default in such a situation.

In MATLAB® the direct notation of loops can often be avoided by using vector- or vector-matrix notations. For an example examine the ‘elegant’ solution implementing the Horner scheme, as shown above. By matrix-vector multiplication loops are implicitly performed, as there is a sum over the products of matrix elements. Utilizing this type of looping not only is much shorter and more clearly

arranged, it also is usually much more efficient, as special vectorized implementations of the basic operations on a near-machine level are utilized.

2.7 Debugging of M-files

It was already said that the M-file, currently shown in the editor, is executed by pressing the  button in the editor menu list. Alternative ways to do the same

thing are (1) pressing the F5 button, or (2) select the sub-menu entry 'Run' from the 'Debug' main entry. For the exploration of M-files some other debugging commands are very convenient, which are explained here.

Debugging is a term, which came up with computer programming. Debugging is the task to find and correct errors, so called bugs. If a program or an M-file does not behave as intended, if there is still at least one erroneous statement, it is convenient to stop the program execution at a certain point and watch the execution of the following steps in detail. For such tasks several tools are available in the MATLAB® editor in the 'Debug' submenu.

A *breakpoint*, at which execution stops, can be set by the user easily by using the column of '-' signs left from the line number counter in the editor window. A mouse click initiates the bar to change into a red circle, indicating the location of a breakpoint, as shown in Fig. 2.10. When the program is run, execution stops at the first breakpoint encountered. A green arrow indicates the current position of the command execution (see Fig. 2.10).

Now the user may check the current values in variables. Moving the cursor through the editor window will make the contents of variables pop-up in small boxes. Figure 2.11 depicts an example: variable *T*, which was touched by the cursor, is a 1×1 double variable and currently contains the value 4. Another method for checking variables at a breakpoint is more convenient, if the variable is a huge array, and its contents can not be shown appropriately in a small box. The user can change into the MATLAB® command window and examine the workspace, as described in Chap. 1.

The  button in the editor's main menu initiates the execution of the next line only. Alternatives are the keyboard F10 button or the 'Step' entry in the 'Debug' sub-menu. Now each command can be checked step by step, examining the effect of the command on the variables involved. The green arrow on the left side of the text window moves further with every step, always indicating the next command, which is not yet executed. Although the 'Debug' commands and options were designed for the programmer to find bugs in programs, these are also convenient tools for novices to understand M-files, written by others. Novices are urged to try the debug functionality on the loop of the M-file of the previous section.

There are further debugging tools, for which the reader may view the MATLAB® help. It is important to know that the  button (or key F5) always

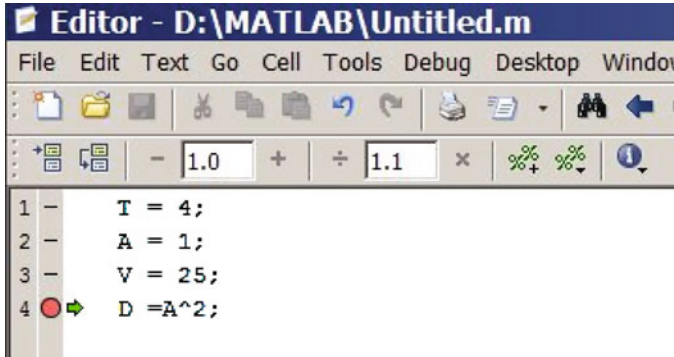


Fig. 2.10 Illustration of debugging; set of breakpoint and stop of execution

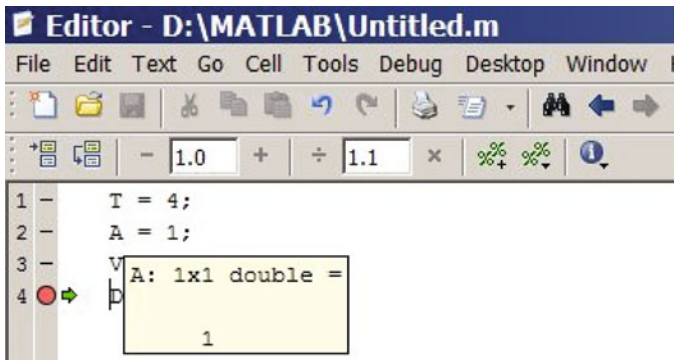


Fig. 2.11 Illustration of debugging; variable check

stands for continuation: starting from the current position the M-file will be executed until it reaches the next breakpoint (it is possible to set several breakpoints!) or the end of the M-file. The  button stops execution; alternatively select sub-menu entry 'Exit Debug Mode.' Breakpoints are deleted by a click, which changes the red circle back to a bar. All breakpoints are cleared by using the  button, or by selection of the corresponding submenu entry.

Reference

Holzbecher E (1998) Modeling density-driven flow in porous media. Springer, Berlin, p 286

Environmental Modeling

Using MATLAB

Holzbecher, E.

2012, XIX, 410 p. 273 illus. in color. With online
files/update., Hardcover

ISBN: 978-3-642-22041-8