Scientific Computing with MATLAB in Chemical Engineering and Biotechnology

Classroom Notes for KETA01 and KKKA05 at LTH

Carmen Arévalo

Revised 2010

Contents

List of Figures

List of Tables

Preface

This is a study aid to be used only in conjunction with attendance to the lectures of the courses KETA01 and KKKA05. The material here is based on lecture slides, classroom MATLAB examples, and blackboard annotations. The courses start with three computer sessions of introduction to MATLAB and are followed by 11 weekly classroom lectures complemented with a corresponding exercise in a computer lab. The eleven lectures develop the major themes of scientific computing especially directed to chemical and bio- engineering.

The computer exercises are compiled separately.

Chapter 1

Preliminaries

We assume you have a basic experience with MATLAB. There are many good MATLAB tutorials and primers on the web. Make sure you have gone through one or several of them. Check the bibliography for a short list available at present.

1.1 The colon operator

A colon (:) between two numbers tells MATLAB to generate the numbers between them.

```
>> x = 2:7
x =
     2     3     4     5     6     7
```

By using the colon twice one can prescribe a particular increment between the numbers.

```
>> y = 1:3:17
y =
     1     4     7    10    13    16
```

In conjunction with matrices, a colon can stand for a whole row or column. For matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

we have

```
>> A(:,1)
ans =
     1
     4
>> A(1,:)
ans =
     1     2     3
```

1.2 Element-by-element operations

In general, MATLAB will perform operations in a matrix-vector fashion. In order to carry out calculations in an element-by element fashion, you need to add a period (.) in front of the operator sign.

For example, to square each element of matrix A above, you need to write

```
>> A.^2
ans =
     1     4     9
    16    25    36
```

1.3 MATLAB Anonymous Functions

In mathematics you define a function by writing something like this

$$g(x) = \left(\frac{x}{2.4}\right)^3 - 2x + \cos\left(\frac{x\,\pi}{12}\right)$$

while in MATLAB the same function might be written as

```
g = @(x) (x/2.4)^3 - 2*x+\cos(x*pi/12)
```

Notice that the (x) in mathematical notation moves to the right hand side in MATLAB, and is always preceded by `@`. The structure of a MATLAB *anonymous function* is

```
"name_of_function" = @("name_of_variable")  "formula"
```

To evaluate this function we use the conventional mathematical notation. Thus, to find the value of the function at 3.1 and call it `y` we write the MATLAB command

```
y=g(3.1)
```

1.4 M-functions

Another way of defining a function in MATLAB is to construct an *M-function* with the following structure:

```
function  ["output_param"] = "function_name"("input_param")
"body_of_the_function" (can consist of several statements)
```

If there is only one output parameter, brackets may be omitted. If there are several input or output parameters, they should be separated by commas. M-functions must be saved in an M-file named `function_name.m` For example, the previous function `g` can be defined as an M-function by writing the following code and saving it to a file called `g.m`.

```
function  p = g(x)
% Evaluates a function g at x and calles the result p
% Input: x, any real number
% Output: p
p = (x/2.4)^3 - 2*x + cos(pi*x/12);
```

To evaluate the function at 3.1 and name the result `y` we write the MATLAB command

```
y = g(3.1)
```

The following is an example of an M-function with several input and output parameters. It must be saved in a file called `quadroot.m`

```
function  [x1,x2] = quadroot(a,b,c)
% Solves equation ax^2+bx+c=0
% Input: a,b,c, coefficients of the quadratic polynomial
% Output: x1,x2, the complex roots of the polynomial
t = -b/(2*a);
d = sqrt(t^2-c/a);
x1 = t + d;
x2 = t - d;
```

To find the roots of $3x^2-5x+7$ and call them `root1` and `root2`, write the MATLAB command

```
[root1, root2] = quadroot(3,-5,7)
```

1.5 The for-loop

The for-loop is used to execute some commands repeatedly. It always starts with the word for and ends with the word end. For example, if you wish to construct the row vector x with 50 elements

$$x(j) = \frac{1}{2^j + 3^j}, \quad j = 1, \ldots, 50$$

you could use the following for-loop

```
for j=1:50
    v(1,j) = 1/(2^j+3^j);
end
```

To construct the column vector $a = \begin{pmatrix} 1^2 & 2^2 & 3^2 \end{pmatrix}^T$, write

```
for j = 1:3
    a(j,1) = j^2;
end
```

To produce $w = \begin{pmatrix} 0 & 2 & 0 & 4 & 0 & 6 & 0 & 8 & 0 & 10 \end{pmatrix}^T$, you could start by constructing the vector $w = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix}^T$ and then zeroing the odd-indexed components with a for-loop:

```
w = 1:10;
for k = 1:2:10
    w(k) = 0;
end
```

To add -2 times the first row to the second and third rows of

$$A = \begin{pmatrix} 1 & 3 & 2 \\ 2 & 2 & 1 \\ 2 & 1 & 3 \end{pmatrix}$$

we write

```
for j = 2:3
    A(j,:) = A(j,:) - 2*A(1,:);
end
```

The result is

$$A = \begin{pmatrix} 1 & 3 & 2 \\ 0 & -4 & -3 \\ 0 & -5 & -1 \end{pmatrix}$$

Chapter 2

Nonlinear Equations

2.1 Motivation

A *zero* of a function $y = f(x)$ is a value of x that makes $f(x) = 0$. The zeros of a function are also called the *roots* of the equation $f(x) = 0$. There are many functions for which a zero cannot be computed easily. Such is the case of most nonlinear functions. For instance, a simple function such as

$$f(x) = \cos(x) - x$$

cannot be solved analytically. Common engineering principles that are modeled by nonlinear equations are heat balance, mass balance and energy balance.

2.1.1 The gas law

The *ideal gas law* is given by

$$pV = nRT$$

where p is the absolute pressure, V is the volume of the gas, n is the number of moles, R is the universal gas constant and T is the absolute temperature. This equation is only valid for some gases and for a small range of pressure and temperature [3]. A more general equation of state for gases is the van der Waals equation

$$\left(p + \frac{a}{v^2}\right)(v - b) = RT$$

where $v = V/n$ is the molal volume, and a and b are constants that depend on the particular gas.

Suppose you need to estimate the volume v for carbon dioxide and oxygen for all combinations of $T = 300, 500, 700$ K, and $p = 1, 10, 100$ atm, in order to choose appropriate containment vessels for these gases. You know the value $R = 0.082054$ L atm/(mol K), and the constants for carbon dioxide ($a = 3.592, b = 0.04267$) and oxygen ($a = 1.360, b = 0.03183$).

Setting $n = 1$ in the ideal gas law, you can calculate the molal volumes for both gases, with :

$$v = \frac{V}{n} = \frac{RT}{p}.$$

If using the van der Waals equation you must solve the nonlinear equation

$$\left(P + \frac{a}{v^2}\right)(v - b) - RT = 0$$

where a, b, R are known and several values of P and T are given. To accomplish this task you need a numerical method that finds the roots of nonlinear equations, $f(x) = 0$.

2.1.2 Errors and residuals

To know how well you have accomplished the task of estimating the solution to $f(x) = 0$, you can take a look at the errors and residuals of your solution. Suppose you have found an approximate solution $\tilde{x}$.

- The *absolute error* of an approximate solution $\tilde{x}$ is $x - \tilde{x}$, where x is the exact solution. It can also be defined as $\tilde{x} - x$ or even as its absolute value, $|\tilde{x} - x|$.
- The *relative error* is the absolute error divided by the magnitude of the exact value, $|x - \tilde{x}|/|x|$. The percent relative error is the relative error expressed in terms of percentage, relative error $\times$ 100%.
- The *residual* for $f(x) = 0$ is $f(\tilde{x})$.

2.2 The Graphical Method

To obtain a rough estimate of the zero of a function, you can plot the function and observe where it crosses the x axis. Nevertheless, these estimates can be used as starting values for more precise methods that will be discussed later. With the MATLAB command `[x,y] = ginput(n)` you can obtain the x- and y-coordinates of n points from the current axes by clicking on the mouse after positioning the cursor on each desired point. Figure 2.1 illustrates this method.

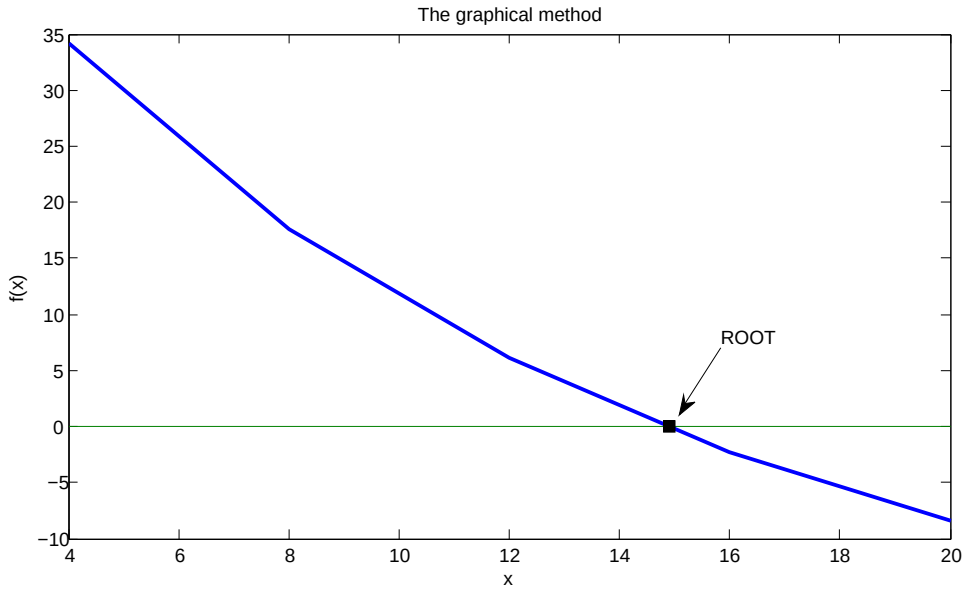

Figure 2.1: *The zero of the function is located at its intersection with the x axis.*

2.3 The Newton-Raphson Method

The most commonly used root-finding method is perhaps the Newton-Raphson method, commonly called the Newton method. This method is based on *linearizing* the given nonlinear function. This is based on the idea that if we zoom in a very small interval, the function appears to be a straight line. For instance, if you graph the function $f(x) = x^2 \sin(x^2/10)$ and zoom in twice at the point corresponding to $x = 4$, you get the plots in Figure 2.2.

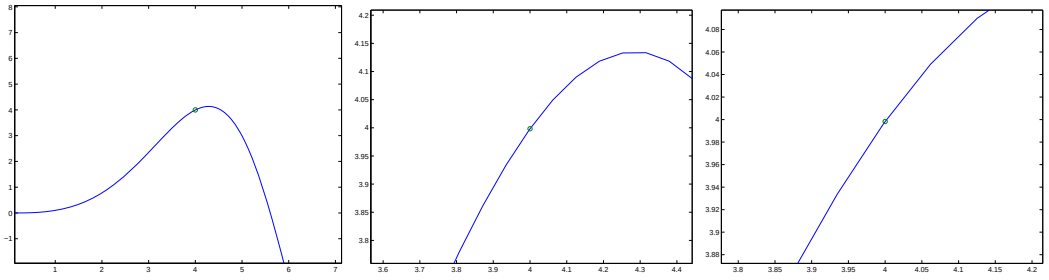

Figure 2.2: *The function $f(x) = x^2 \sin(x^2/10)$ can be locally seen as a linear function. Here we have zoomed-in around the value of the function at $x = 4$.*

2.3.1 A graphical description of the method

A tangent to the function at x_0 is extended to the x axis. The point x_1 where the tangent crosses the x-axis is usually an improved estimate of the point where the curve intersects the x axis. The tangent of the function at x_1 is then extended to the x axis, to obtain x_2. The process is repeated several times to obtain closer approximations to the zero of the function. See Figure 2.3.

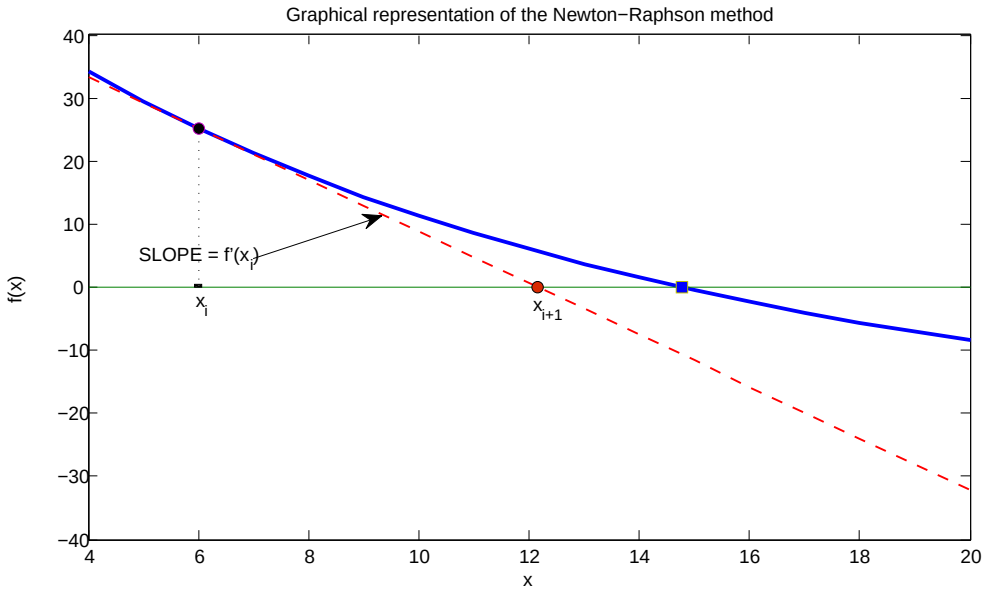

Figure 2.3: *The tangent to the function at x_i is extended to the x axis to estimate x_{i+1}.*

2.3.2 The Newton-Raphson formula

Since the slope of the tangent line is

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

you can rewrite this expression as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

This iterative formula is called the *Newton-Raphson formula* or just the *Newton formula*.

To initiate the iterative process you need a *starting guess*. The Newton-Raphson method is a *local* method, which means that the starting point must be close enough to the true solution. If the starting point is too far away from the solution, the Newton-Raphson iteration might not converge. A good guess is provided by the graphical method.

To stop the process you need a *stopping criterion*. You can repeatedly apply the formula until one (or more) of these events occur

- $f(x_{i+1}) <$ tol, where `tol` is a small number
- absolute error $\approx |x_i - x_{i+1}| <$ tol
- relative error $\approx |x_i - x_{i+1}|/|x_{i+1}| <$ tol
- number of iterations $>$ N, where N is a given integer

After each Newton iteration, the number of correct figures approximately doubles. Table 2.1 shows the result of a Newton-Raphson iteration. Notice that the iterates have 0, 1, 2, 6 and 14 correct figures respectively.

i	x_i
0	0
1	0.500000000000000
2	0.566311003197218
3	0.567143165034862
4	0.567143290409781
5	0.567143290409784

Table 2.1: *The Newton-Raphson iterates rapidly converge to the true root.*

2.3.3 MATLAB nonlinear solvers

- To find the roots of polynomials, you can use the command `r = roots(p)` where `p` is the vector of polynomial coefficients. For instance, to find the zeros of $p(x) = x^4 - 0.1x^3 - 0.04x^2 + 3.5x - 2.5$ you can write the following commands

```
p=[1 -0.1 -0.04 3.5 -2.5];
r=roots(p)
```

- For other types of nonlinear functions, you can use the MATLAB function `fzero` by writing `x=fzero(fun, x0)` or `[x,residual]=fzero(fun, x0)`.
  - if `f` is defined as an anonymous function, write

```
f=@(t)exp(t/5)-0.3;
x = fzero(f, 2)
```

    or

```
[x, residual] = fzero(f, x0)
```

  - `f` can also be defined inside the call to `fzero`:

```
x = fzero(@(t)exp(t/5)-0.3, 2)
```

    or

```
[x, residual] = fzero(@(t)exp(t/5)-0.3, 2)
```

  - If `fun` is an M-function, you can write

```
x = fzero(@fun, x0)
```

    or

```
[x, residual] = fzero(@fun, x0)
```

2.4 Example

Let us calculate the roots of the van der Waals equation in 2.1.1 for carbon dioxide at a pressure of 1 atm. and a temperature of 300 K. We first define the function as an M-function by editing the file myfunction.m containing the following code:

```
function f=myfunction(v)
R=0.082054;
a=3.592;  b=0.04267;       %carbon dioxide
P=1;
T=300;
f=(P+a./v.^2).*(v-b)-R*T;
```

We now plot the function in Figure 2.4 to estimate the x intercept. We generate 100 points between 0 and 50 with the command `linspace`.

```
>> v=linspace(0,50);
>> y=myfunction(v);
>> plot(v,y,v,zeros(size(v)))
>> grid
>> v0=ginput(1)
v0 =
    2.447076612903226e+001                    0
```

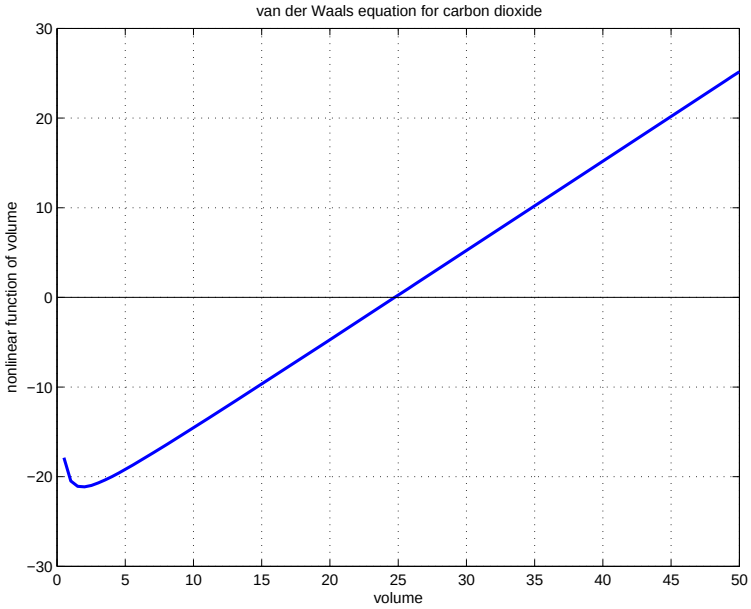

Figure 2.4: *Graphical method for the van der Waals equation.*

One way of solving the equation is to rewrite it as a cubic polynomial,

$$PV^3 - (RT + bP)V^2 + aV - ab = 0$$

and use the MATLAB command `roots`

```
>> p=[1 -0.082054*300-.04267 3.592 -3.592*.04267];
>> V=roots(p)
V =
  24.5126
   0.0731 + 0.0301i
   0.0731 - 0.0301i
```

As we do not know the exact solution, we cannot calculate the absolute or relative errors, but we can compute the residual by evaluating the polynomial at V = 24.5126.

```
>> r=p*[V(1)^3 V(1)^2 V(1) 1]'
r =
    3.637978807091713e-012
```

A small residual is an indicator of a close estimate.

Another way to solve the problem is to use `fzero` with an initial guess of 24.5, the estimate from the graphical method rounded to one decimal figure.

```
    >> [x, residual] = fzero(@myfunction, 24.5)
x =
  24.512588128441497
residual =
   -3.552713678800501e-015
```

2.5 Nonlinear systems of equations

You can use `fzero` to solve a single nonlinear equation, but for a system of nonlinear equations you need to use `fsolve`. This command is used in the same manner as `fzero`, but the system must be defined using an M-function.

For example, to solve the system

$$\begin{aligned} x^2 - 3y + 2 &= 0 \\ x^3 - 4x^2 - xy + 1 &= 0 \end{aligned}$$

you can define the M-function `newf2`:

```
function f = newf2(x)
% Input: vector with 2 components
% Output: vector with 2 components
f1 = x(1).^2 - 3*x(2) + 2;
f2 = x(1).^3 - 4*x(1).^2 - x(1).*x(2) + 1;
f=[f1; f2];
```

and then write

```
[x, residual] = fsolve(@newf2, [0,0])
```

Here we chose the initial guess x = 0, y = 0.

2.6 Exercise

The following equation pertains to the concentration of a chemical in a completely mixed reactor:

$$c = c_{\text{in}}(1 - e^{-0.04t}) + c_0 e^{-0.04t}$$

If the initial concentration $c_0 = 4$ and the inflow concentration $c_{\text{in}} = 10$, compute the time required for c to be 93 percent of c_{in}.

Correct result: $t = 53.71$

Chapter 3

Systems of Linear Equations

3.1 Motivation

Systems of linear equations occur naturally in almost any field of science, but often they are the result of simplifying more complex or harder to solve equations, such as nonlinear or differential equations. It is easy and simple to solve a system of a small number of linear equations by hand, but for larger systems computers must be used.

A system of m linear equations with n unknowns has the form

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &=& b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &=& b_2 \\ & \vdots & \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &=& b_m \end{array} \tag{3.1}$$

In matrix-vector form it can be written as

$$\mathbf{A}\ \mathbf{x} = \mathbf{b} \tag{3.2}$$

where

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & & & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{pmatrix}$$

The system is said to be *square* if $m = n$, *overdetermined* if $m > n$ and *underdetermined* if $m < n$.

The square case: A is an $n \times n$ matrix

The diagonal $\{a_{11}, a_{22}, a_{23}, \ldots, a_{nn}\}$ is called the *main diagonal* of A. The *rank* of A is the number of independent rows (or columns) of A. A well-know theorem tells you that

$$\text{rank}(A) = n \Leftrightarrow A \text{ has an inverse} \tag{3.3}$$

When $\text{rank}(A) = n$ you also say A is *invertible*. If this is the case, you know there is one (unique) solution which can be expressed as $x = A^{-1}b$.

If A is not invertible, you say it is *singular*. Then $\text{rank}(A) < n$ and the system has infinitely many solutions or no solution, *depending on the right-hand side vector* b.

3.2 Gauss elimination

The basic principle of Gauss elimination is to transform your system to an upper triangular system by elementary row transformations. See Fig. 3.1.

3.2.1 Elementary row operations

Gaussian elimination uses *elementary row operations* to convert a square system into a triangular one. These three operations can be applied to individual equations without altering the solution:

- interchange the positions of any two rows,
- multiply a row by a non-zero scalar,
- replace a row by the sum of the row and a scalar multiple of another row.

3.2.2 Systems with triangular matrices

Once the original system has been converted to a triangular system by means of elementary row operations, you can solve the system row by row, starting with the matrix row that contains only one element.

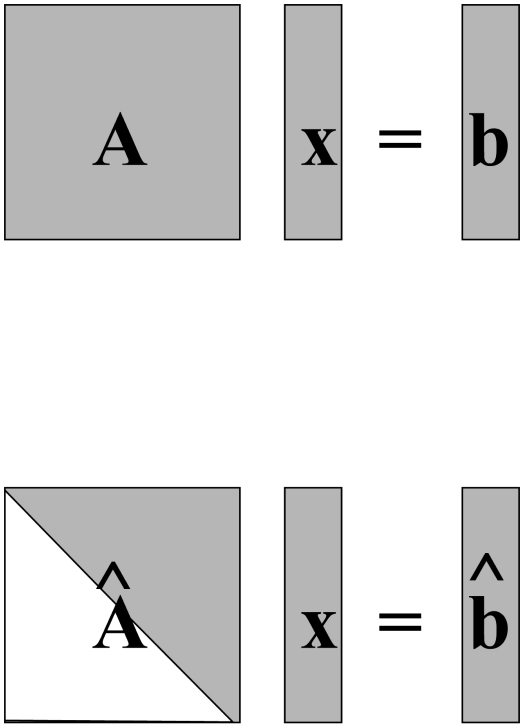

Figure 3.1: *After Gauss elimination the matrix is upper triangular.*

Back Substitution

If the matrix is upper triangular, as shown in Fig. 3.1, the original system is equivalent to

$$\begin{aligned}
\hat{a}_{11}x_1 + \hat{a}_{12}x_2 + \hat{a}_{13}x_3 + \cdots\cdots + \hat{a}_{1n}x_n &= \hat{b}_1 \\
\hat{a}_{22}x_2 + \hat{a}_{23}x_3 + \cdots\cdots + \hat{a}_{2n}x_n &= \hat{b}_2 \\
+ \hat{a}_{33}x_3 + \cdots\cdots + \hat{a}_{3n}x_n &= \hat{b}_3 \\
&\vdots \\
\hat{a}_{n-1n-1}x_{n-1} + \hat{a}_{n-1n}x_n &= \hat{b}_{n-1} \\
\hat{a}_{nn}x_n &= \hat{b}_n
\end{aligned}$$

and the last row can be solved for x_n:

$$x_n = \frac{\hat{b}_n}{\hat{a}_{nn}}.$$

Substituting this result in the (n-1)th equation, you can obtain x_{n-1},

$$x_{n-1} = \frac{\hat{b}_{n-1} - \hat{a}_{n-1n}x_n}{a_{n-1n-1}}.$$

Repeating the procedure to evaluate the remaining unknowns, you get the following formula:

$$x_i = \frac{\hat{b}_i - \sum_{j=i+1}^{n} \hat{a}_{ij}x_j}{\hat{a}_{ii}} \qquad \text{for } i = n, \ldots 1.^1$$

The elements that go in the divisor, $\hat{a}_{ii}$, are called *pivots*. Note that they cannot be zero.

Forward substitution

If you have a lower triangular system, the system is

$$\begin{aligned} l_{11}x_1 &= y_1 \\ l_{21}x_1 + l_{22}x_2 &= y_2 \\ l_{31}x_1 + l_{32}x_2 + l_{33}x_3 &= y_3 \\ &\vdots \\ l_{n1}x_1 + l_{n2}x_2 + \cdots\cdots + l_{nn}x_n &= y_n. \end{aligned}$$

You can start by solving the first equation to get x_1,

$$x_1 = \frac{y_1}{l_{11}}$$

and then substituting in the second equation to get x_2. Repeating the process yields the following formula:

$$x_i = \frac{y_i - \sum_{j=1}^{i-1} l_{ij}x_j}{l_{ii}} \qquad \text{for } i = 1, \ldots n.$$

3.2.3 Solving a square linear system with MATLAB

MATLAB M-function for back substitution

One way of constructing a back substitution algorithm in MATLAB is to use a `for`-loop. The MATLAB command `length()` gives the dimension of a vector, `size()` gives the dimension of a matrix and `zeros(n,m)` constructs a matrix with n rows and m columns with all elements equal to zero.

[1] If a summation has a initial index that is greater than the final index, as in $\sum_{j=1}^{0}$, its value is defined to be zero.

```
function x=backsub(U,y)
% back substitution for the upper triangular system Ux=y
% Input: U, upper triangular n x n matrix; y, n x 1 vector
% Output: x, n x 1 vector solution to Ux=y
n=length(y); % finds dimension of system
x=zeros(size(y)); % makes x a column vector
for i=n:-1:1
    rowsum=0; % initializes the sum to 0
    for j=i+1:n
        rowsum=rowsum+U(i,j)*x(j);
    end
    x(i)=(y(i)-rowsum)/U(i,i);
end;
```

MATLAB **M-function for Gauss Elimination**

```
function x=gauss(A,b)
% Gauss elimination for the system Ax=b
% first constructs an equivalent upper triangular system
% and then solves the system
n=length(b);   % we extract the size of the system U=[A b];
% matrix U is extended system, A + right hand side
% construct U
for j=1:n
    pivot=U(j,j); % at row j all elements before U(j,j) are 0
    for i=j+1:n
        m(i)=U(i,j)/pivot; % multipliers for rows below diagonal
        U(i,j)=0; % eliminate elements below the diagonal
        % row i - m * row j
        U(i,j+1:n+1)=U(i,j+1:n+1)-m(i)*U(j,j+1:n+1);
    end
end;
r=U(:,n+1); % extract new right hand side (last column)
U=U(:,1:n); % extract U (first n columns)
x=backsub(U,r); % solve Ux=r using function backsub
```

3.2.4 Solution of Ax = b in MATLAB

In MATLAB there are two ways of solving a square linear system.

- The solution can be computed by multiplying the right hand side by the inverse of the system matrix,
  `x = inv(A)*b;  % OK for small matrices.`

- The solution can be computed by Gauss elimination followed by back elimination, MATLAB does this with the *backslash* operator,
  `x = A\b;  % preferred method.`

3.2.5 Example

Conservation of Mass in a System of Reactors

The *conservation of mass* principle is one of the pillars of chemical engineering. Over a period of time, this principle applied to a material that passes in and out of a volume can be expressed as

Accumulation = inputs - outputs.

If the inputs are equal to the outputs, the accumulation is zero and the mass in the volume remains constant. This condition is called *steady-state*. The conservation of mass principle can be used to determine steady-state concentrations of a system of coupled reactors such as the one shown in Fig. 3.2. We now apply the conservation

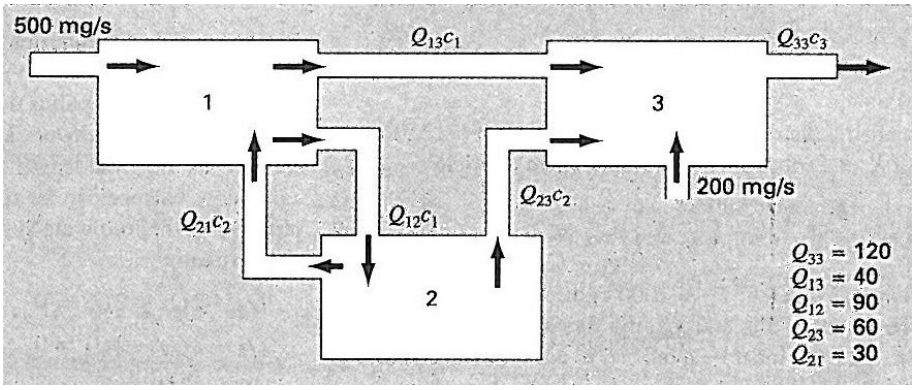

Figure 3.2: *Three reactors linked by pipes. The flow rates Q are in* $\mathrm{m}^3/\mathrm{min}$*, and the concentrations c are in* ml/m^3*.*

of mass to determine the steady-state concentrations of this system. We first develop the mass-balance equations for each reactor:

- Reactor 1: $500 + Q_{21}c_2 = Q_{12}c_1 + Q_{13}c_1$
- Reactor 2: $Q_{12}c_1 = Q_{21}c_2 + Q_{23}c_2$
- Reactor 3: $200 + Q_{13}c_1 + Q_{23}c_2 = Q_{33}c_3$

or

$$\begin{aligned}(Q_{12} + Q_{13})c_1 - Q_{21}c_2 &= 500\\ Q_{12}c_1 - (Q_{21} + Q_{23})c_2 &= 0\\ -Q_{13}c_1 - Q_{23}c_2 + Q_{33}c_3 &= 200\end{aligned}$$

These equations can be written in matrix-vector form as $Qc = b$,

$$\begin{pmatrix}40+90 & -30 & 0\\ 90 & -(30+60) & 0\\ -40 & -60 & 120\end{pmatrix}\begin{pmatrix}c_1\\ c_2\\ c_3\end{pmatrix} = \begin{pmatrix}500\\ 0\\ 200\end{pmatrix}$$

In MATLAB we can write

```
Q = [130 -30 0; 90 -90 0;-40 -60 120];
b = [500; 0; 200];
c = Q\b
```

and the result will be $c_1 = c_2 = 5, c_3 = 5.8333$.

The inverse matrix

The MATLAB command `inv(Q)` finds the inverse of matrix `Q`. For the problem above, you get

$$Q^{-1} = \begin{pmatrix}0.0100 & -0.0033 & 0\\ 0.0100 & -0.0144 & 0\\ 0.0083 & -0.0083 & 0.0083\end{pmatrix}$$

Each element $Q^{-1}(i, j)$ (recall that i is the row number and j is the column number) signifies the change in concentration of reactor i due to a unit change in loading to reactor j.

Looking at column 3 of A^{-1}, you see that the first two rows have zeros. This means that a loading to reactor 3 will have no impact on reactors 1 and 2. Loadings to reactors 1 or 2 will affect the entire system, as columns 1 and 2 have no zeros. Note the consistency of these observations with Fig. 3.2.

3.3 LU factorization

Suppose you want to convert the system $Ax = b$ into an upper triangular system using Gauss elimination. You initially set $U = A$ and $L = I$, where I is the *identity matrix*,

$$I = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

In MATLAB you can construct the $n \times n$ identity matrix with the command

```
I=eye(n);
```

You perform Gauss elimination as usual, so that the matrix U becomes an upper triangular matrix, but you also save in matrix L the *multipliers* used to reduce column entries to zeros. The multipliers are

$$l_{i,j} = \frac{u_{i,j}}{u_{j,j}} \quad \text{for} \quad i = j, \dots, n$$

where j is the number of the actual column – or the number of the step in the algorithm – and i is the number of the row we want to change. L will be a lower triangular matrix,

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots\cdots & 0 & 0 \\ l_{21} & 1 & 0 & \cdots\cdots & 0 & 0 \\ l_{31} & l_{32} & 1 & \cdots\cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \\ & & & \cdots\cdots & 1 & 0 \\ l_{n1} & l_{n2} & l_{n3} & \cdots\cdots & l_{nn-1} & 1 \end{pmatrix}.$$

If you now multiply matrices L and U, you obtain the system matrix A. The fact is you have factorized the system matrix A in the form

$$\mathbf{A = LU}$$

where L is a lower triangular matrix and U is the upper triangular matrix resulting from the Gauss elimination process.

3.3.1 Solving after the LU factorization

The original system $Ax = b$ can be written as $LUx = b$. Defining $y = Ux$ leads to $Ly = b$. Once you have matrices L and U, the system can be solved in two steps:

1. solve the lower triangular system $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$,
2. solve the upper triangular system $\mathbf{Ux} = \mathbf{y}$ for the solution $\mathbf{x}$.

3.3.2 Gauss elimination with LU factorization in MATLAB

Gauss elimination can be programmed as an LU factorization followed by one forward substitution and one back substitution.

```
function [x,L,U]=gausslu(A,b)
% Gauss elimination for the system Ax=b using LU factorization
% Output: x solution , U upper triangular , L lower triangular
n=length(b);   % extract size of system
U=A ;          % initialize U
L=eye(n);      % initialize L
% construct U and L
for j=1:n
    pivot=U(j,j);
    for i=j+1:n
        L(i,j)=U(i,j)/pivot;
        U(i,j)=0;
        U(i,j+1:n)=U(i,j+1:n)-L(i,j)*U(j,j+1:n);
    end
end;
y=forsub(L,b);    % solve Ly=b
x=backsub(U,y);   % solve Ux=y
```

3.3.3 Gauss Elimination vs LU factorization

You have seen that Gauss elimination and LU factorization are the two sides of a coin. Suppose you want solve $Ax = b$. Gauss Elimination operates on both A and b

$$\begin{array}{rcl} [A \quad b] & \rightarrow & [U \quad \hat{b}] \\ Ax = b & \rightarrow & Ux = \hat{b} \end{array}$$

while LU factorization operates on A only and saves information in L for *later* operations on b.

$$\begin{array}{rcl} [A] & \rightarrow & [LU] \\ Ax = b & \rightarrow & Ly = b,\ Ux = y \end{array}$$

3.3.4 Example

Consider the small system

$$\begin{pmatrix} 2 & 0 & 1 \\ -3 & 4 & -2 \\ 1 & 7 & -5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ -3 \\ 6 \end{pmatrix}$$

and note the two main steps required by each of the two methods, as described in Table 3.3.4.

First Gauss Step	First LU step
	U $\qquad$ L
$\begin{pmatrix} 2 & 0 & 1 \\ 0 & 4 & -1/2 \\ 0 & 7 & -11/2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ 4 \end{pmatrix}$	$\begin{pmatrix} 2 & 0 & 1 \\ 0 & 4 & -1/2 \\ 0 & 7 & -11/2 \end{pmatrix}$ $\quad$ $\begin{pmatrix} 1 & 0 & 0 \\ -3/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix}$
Second Gauss Step	**Second LU step**
$\begin{pmatrix} 2 & 0 & 1 \\ 0 & 4 & -1/2 \\ 0 & 0 & -37/8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \\ -5/4 \end{pmatrix}$	$\begin{pmatrix} 2 & 0 & 1 \\ 0 & 4 & -1/2 \\ 0 & 0 & -37/8 \end{pmatrix}$ $\quad$ $\begin{pmatrix} 1 & 0 & 0 \\ -3/2 & 1 & 0 \\ 1/2 & 7/4 & 1 \end{pmatrix}$

Table 3.1: *Steps required in Gauss elimination and in LU factorization*

After these two steps, Gauss elimination needs one back substitution, while LU factorization needs one back and one forward substitution.

3.4 Row pivoting

The LU or Gauss algorithm breaks down when a pivot is equal or very close to zero, as the algorithm will attempt to divide by zero. For example, the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

has a zero pivot. Nevertheless, this does not necessarily imply that you cannot perform Gauss elimination. Row exchange is an elementary row operation and can be done without altering the solution to the system of equations. If you interchange rows in this example the pivot is 1.

Exchanging rows during during the process in order to have a nonzero pivot is called *row pivoting*. If Gauss elimination still fails, then it means A is singular and the system does not have a (unique) solution. In practice row interchange is also made to ensure that the largest row element becomes the pivot element. This minimizes the risk of large numerical errors that can occur when dividing by a very small number.

3.4.1 Pivoting and permutation matrices

Interchanging rows of a matrix is equivalent to multiplying by a permutation matrix. The following matrix

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

puts rows in the order (2, 3, 1):

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad = \quad \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{pmatrix}$$

A permutation matrix is an identity matrix with interchanged rows.

Not all matrices have an LU factorization, as they may have a zero pivot, but for every matrix there is a permutation matrix P such that

$$PA = LU.$$

If the matrix A is singular, then the matrix U will also be singular, and in fact it will have at least one diagonal element equal to zero.

3.4.2 LU factorization with pivoting

Suppose you have the (P)LU factorization of A,

$$PA = LU$$

The system $Ax = b$ is equivalent to $PAx = Pb$, so

$$LUx = Pb$$

To solve after factorizing,

1. solve $Ly = Pb$ for y,
2. solve $Ux = y$ for the solution x.

3.4.3 Example

How to find the (P)LU factorization of a matrix A using MATLAB. With the command `[L,U,P] = lu(A)` you get matrices L, U and P such that $LU = PA$ with L lower triangular, U upper triangular, and P a permutation matrix.

```
>>A = [0 4 6;2 4 3;1 5 9];
>>[L,U,P] = lu(A)
L =
    1.0000         0         0
         0    1.0000         0
    0.5000    0.7500    1.0000
U =
     2     4     3
     0     4     6
     0     0     3
P =
     0     1     0
     1     0     0
     0     0     1
```

3.5 Why use LU factorization at all?

- If you want to solve several systems with the same matrix but with different right-hand side vectors,

$$\begin{aligned} Ax_1 &= b_1, \\ Ax_2 &= b_2, \\ &\vdots \\ Ax_n &= b_n, \end{aligned}$$

  you can perform just one Gauss elimination step (LU factorization of A) and then solve by doing one forward and one back substitution for each right-hand side, instead of performing n Gauss eliminations on the same matrix.

- Another practical application of the LU factorization is the efficient calculation of matrix inverses. The inverse of matrix can be computed by generating solutions to n different systems with the same matrix. In particular, one needs to solve n systems of the form $Ax = e_k$, where e_k is the vector with all components equal to zero except the k-component, which is equal to one.

- The LU factorization can also be used to evaluate how sensitive a system matrix is to small errors.

Example

Suppose we have the three reactors from Example 3.2.5, and we wish to establish the concentrations in each reactor at steady-state when the input to Reactor 1 is 400, 300, 200 and 100 mg/sec. For each value we have a system of equations with the same matrix as before but with right-hand sides equal to

$$\begin{pmatrix} 400 \\ 0 \\ 200 \end{pmatrix}, \begin{pmatrix} 300 \\ 0 \\ 200 \end{pmatrix}, \begin{pmatrix} 200 \\ 0 \\ 200 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 100 \\ 0 \\ 200 \end{pmatrix}$$

If we run the MATLAB script

```
Q = [130 -30 0; 90 -90 0; -40 -60 120];
b=[400 300 200 100; 0 0 0 0; 200 200 200 200];
[L,U,P]=lu(Q);
for k=1:4
    y(:,k)=forsub(L,P*b(:,k));
    x(:,k)=backsub(U,y(:,k));
end
```

we get the following results:

$$\begin{pmatrix} 4.0000 \\ 4.0000 \\ 5.0000 \end{pmatrix}, \begin{pmatrix} 3.0000 \\ 3.0000 \\ 4.1667 \end{pmatrix}, \begin{pmatrix} 2.0000 \\ 2.0000 \\ 3.3333 \end{pmatrix} \text{ and } \begin{pmatrix} 1.0000 \\ 1.0000 \\ 2.5000 \end{pmatrix}$$

3.6 Exercise

Determine the concentration of chloride in each of the Great Lakes using the information shown in Fig. 3.3.

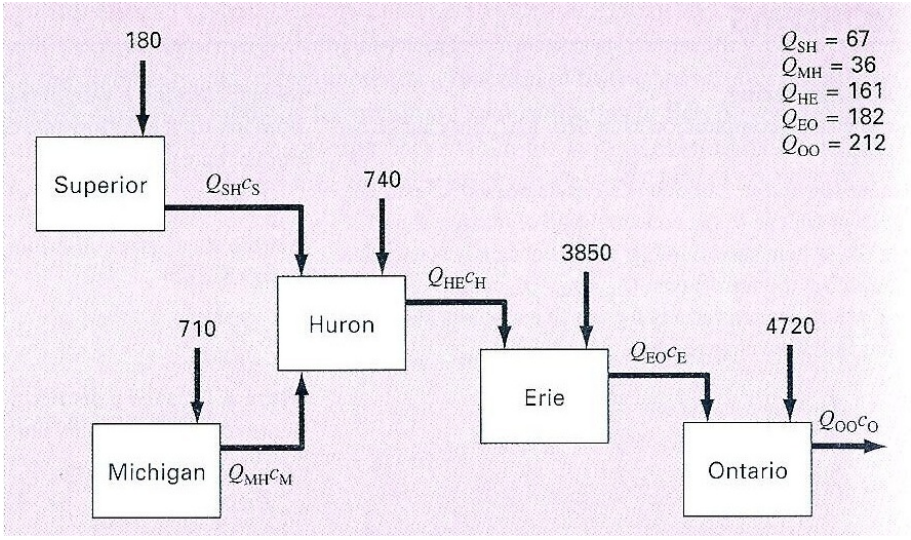

Figure 3.3: *A chloride balance for the Great Lakes.*

Correct result: $c_S = 2.6866, c_M = 19.7222, c_H = 10.1242, c_E = 30.1099, c_O = 48.1132$.

Chapter 4

Overdetermined Systems

4.1 Motivation

When you have a linear system of equations with more equations than unknowns (variables), you call it an *overdetermined* system. Each equation can be thought of as being a condition imposed on the variables. When the equations are inconsistent, that is, when all the equations cannot be satisfied at the same time, you might still want to find the best solution possible, or the set of values that comes the closest to being a solution to the overdetermined system.

In *hypothesis testing* mathematical models are compared with measured data. If the model parameters are not know, you might want to determine those that best fit the observed data; if you have alternative models with known parameters, you might want to compare the different models by comparing the observed data with the values predicted by each model.

The Oat Porridge Problem

Suppose you want to make oat porridge from a box that states that you need the following quantities:

Portion	Oats	Water	Salt
1	1 dl	2.5 dl	0.5 ssp
2	2 dl	4.5 dl	1 ssp
4	4 dl	9 dl	0.5 tsp

Table 4.1: *Data on the cereal box.*

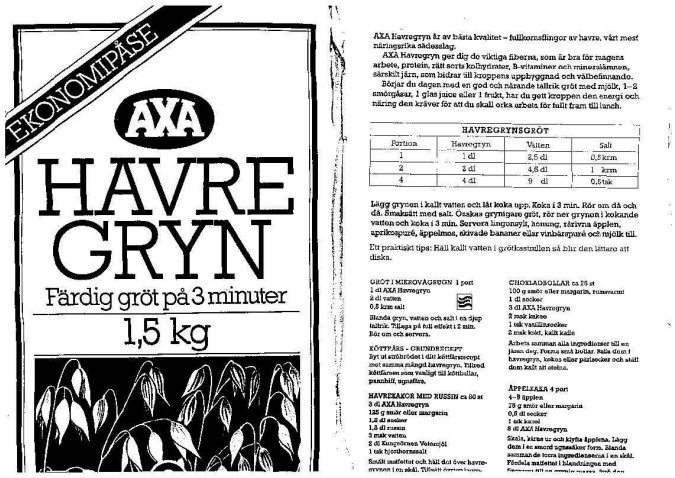

Figure 4.1: *Making oat porridge from a cereal box.*

but today you actually want to make porridge for 3 persons, and on Sunday you will need to cook porridge for 300 people. You ask yourself

- how much water is needed for 3 portions?
- how much water is needed for 300 portions?

To answer these questions properly, you first have to set up a correct mathematical model and then determine the unknown parameters in the model. You can start by using MATLAB to plot the amount of water vs. the number of portions as given on the box.

```
> portions=[1;2;4];
> water=[2.5;4.5;9];
>> plot(portions,water,'-o','MarkerEdgeColor','k',...
   'MarkerFaceColor','r','MarkerSize',12)
>> axis([0 5 0 10])
```

If you had hoped to use a linear model, you see in Fig. 4.2 that there is no straight line passing through these 3 points.

The proposed solution is to find an approximating function that fits the data without necessarily passing through the points. This process is called *curve fitting*.

In this example, you could use a straight line, $y = kx + m$, as a model. Note that a straight line **cannot** pass through the 3 data points. You will learn how to

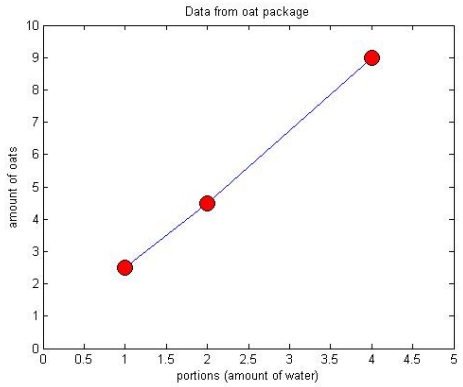

Figure 4.2: *Plot of data points for the oat porridge problem.*

choose the "best fit" for a given data set, that is, how to determine the parameters k and m so that the model $y = kx + m$ best describes the relation between the amount of water (y) and the number of portions (x).

4.1.1 A linear model

Let the model be $y = c_1x + c_0$. Ideally, you would like all points to be on the line, that is, you would like each and every set of points (x, y) to satisfy the equation:

$$\begin{aligned} 2.5 &= 1 \cdot c_1 + c_0, \\ 4.5 &= 2 \cdot c_1 + c_0, \\ 9 &= 4 \cdot c_1 + c_0. \end{aligned}$$

This is an overdetermined system with 3 equations and 2 unknowns that can be written in matrix-vector form as

$$\begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} 2.5 \\ 4.5 \\ 9 \end{pmatrix}.$$

4.1.2 An improved linear model

As for 0 portions you need 0 dl oats, you can improve the data by adding the point (0,0) to the given data.

$$\begin{aligned} 2.5 &= 1 \cdot c_1 + c_0, \\ 4.5 &= 2 \cdot c_1 + c_0, \\ 9 &= 4 \cdot c_1 + c_0, \\ 0 &= 0 \cdot c_1 + c_0. \end{aligned}$$

This is an overdetermined system with 4 equations and 2 unknowns,

$$\begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 4 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} 2.5 \\ 4.5 \\ 9 \\ 0 \end{pmatrix}.$$

4.1.3 Yet another linear model: the homogeneous equation

The solution to the previous model does not pass through (0,0), so instead of adding the point $(0, 0)$ to the data, you can take the model to be homogeneous (insisting that it passes through the point $(0, 0)$),

$$y = c_1 x.$$

The equations are

$$\begin{aligned} 2.5 &= 1c_1, \\ 4.5 &= 2c_1, \\ 9 &= 4c_1. \end{aligned}$$

This is an overdetermined system of 3 equations and 1 unknown,

$$\begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} c_1 = \begin{pmatrix} 2.5 \\ 4.5 \\ 9 \end{pmatrix}.$$

4.2 Solution of Overdetermined Systems — Curve Fitting

Consider linear systems of the form

$$Ax = y$$

where A is a matrix with m rows and n columns and $m > n$. Overdetermined linear systems rarely have a solution, but can a *solution to the system* be defined in some sense?

4.2.1 Least squares solution

If the system you are considering does not have a solution, you cannot find a vector v such that $Av = y$, but you may be able to find a vector $\hat{v}$ such that $A\hat{v}$ is as close as possible to y

$$|y - A\hat{v}| = \min_v |y - Av| = \min_c |r(v)|,$$

where $|w|$ is the (Euclidean) length of vector w:

$$|w| = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2} = \sqrt{w^T \cdot w}.$$

The residual vector $r(\hat{v}) := y - A\hat{v}$ is required to be as small as possible. As the vector $\hat{v}$ will minimize

$$|y - Av| = \sqrt{(y_1 - Av_1)^2 + (y_2 - Av_2)^2 + \cdots + (y_n - Av_n)^2},$$

$\hat{v}$ is called the **least squares** solution.

4.2.2 Is there a unique solution?

Finding a vector $\hat{v}$ with

$$|y - A\hat{v}| = \min_v |y - Av| = \min_v |r(v)|$$

is mathematically equivalent to solving the *normal equations*:

$$A^T A\hat{v} = A^T y.$$

Note that the number of rows of A is equal to the number of equations and the number of columns is equal to the number of parameters in the model. The matrix $A^T A$ is a square matrix of dimension equal to the number of parameters.

4.2.3 Solving the normal equations in MATLAB

In MATLAB you can either directly look for the solution of the normal equation

```
>> p = (A'*A)\(A'*y)
```

or you can just use the backslash operator with the original system matrix,

```
>> p = A\y
```

In this case, if the system is overdetermined, MATLAB will solve it as a least squares problem.

4.2.4 Quadratic Curve Fitting

Suppose you want to fit a quadratic polynomial

$$p(x) = c_2x^2 + c_1x + c_0$$

to the following data:

i	0	1	2	3	4
x_i	-1.0	-0.5	0.0	0.5	1.0
y_i	1.0	0.5	0.0	0.5	2.0

Table 4.2: *Data for quadratic fitting.*

Set up the desired conditions $p(x_i) = y_i$ and write the system in matrix-vector form:

$$\begin{pmatrix} x_0^2 & x_0 & 1 \\ x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \\ x_4^2 & x_4 & 1 \end{pmatrix} \begin{pmatrix} c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

This rectangular 5×3 linear system has a very special matrix. Each column is formed from right to left by the x-values raised to successive powers, 0,1,2, etc. Matrices with this structure are called Vandermonde matrices. You will encounter them again in the next chapter.

You can use MATLAB to solve the overdetermined system and plot the resulting parabola using 200 points in the interval $[-1, 1]$.

```
>> x=[-1.0;-0.5;0.0;0.5;1.0];
>> y=[1.0;0.5;0.0;0.5;2.0];
>> A=[x.^2 x ones(length(x),1)];
>> p=A\y;
>> xplot=linspace(-1,1,200);
>> yplot=p(1)*xplot.^2+p(2)*xplot+p(3);
>> plot(x,y,'o',xplot,yplot)
```

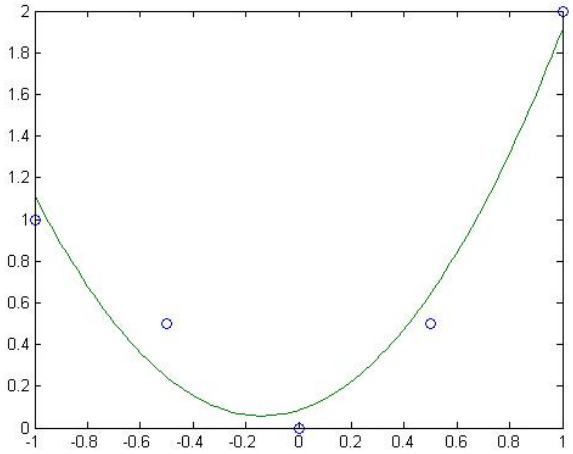

Figure 4.3: *Plot of the fitted quadratic polynomial.*

4.2.5 The Oat Porridge Regression Line

If you now retake the problem in Section 4.1 with the added data (0,0), you can solve the normal equations and get the straight line

$$y = 2.23\,x + 0.10 \tag{4.1}$$

as shown in Figure 4.4. The best line fit is called a *regression* line. Note that this line does not pass through the data points.

Using this model you can calculate the amount of water necessary to make 3 and 300 portions. Table 4.3 shows the values calculated using model (4.1).

portions	0	1	2	3	4	300
water (dl)	0.10	2.33	4.56	6.79	9.01	668.67

Table 4.3: *Portions required for 3 and 300 persons using linear regression.*

4.3 Exercise

It is known that the tensile strength of a plastic increases as a function of the time it is heat-treated. Some observed data is collected in Table 4.4.

(a) Fit a straight line to this data and use the equation to determine the tensile strength at a time of 32 min.

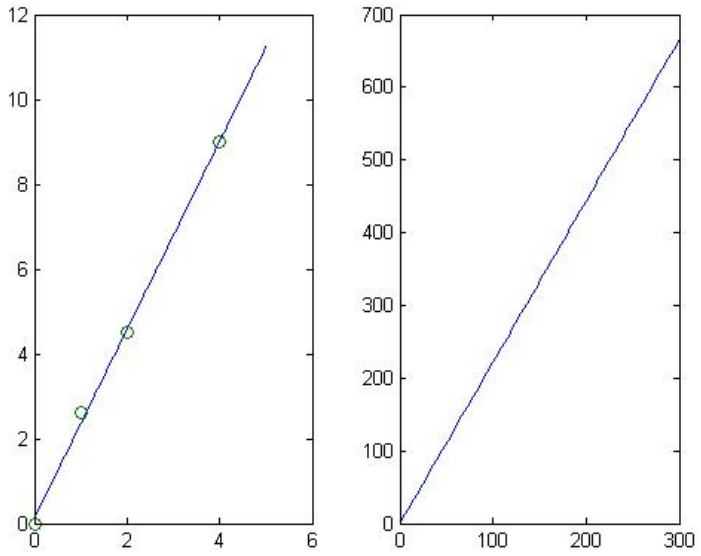

Figure 4.4: *Plot of the regression line for the oat porridge problem. The second plot shows the line used to extrapolate at 300.*

(b) Repeat the analysis but for a straight line that passes through (0,0).

Time	10	15	20	25	40	50	55	60	75
Tensile strength	5	20	18	40	33	54	70	60	78

Table 4.4: *Tensile strength of heated plastic.*

Correct results: (a) 34.7049, (b) 29.0840

Chapter 5

Interpolation

5.1 Motivation

Perry's Chemical Engineer's Handbook [8] gives the following values for the heat capacity at constant pressure, c_p, of an aqueous solution of methyl alcohol as a function of the alcohol mole percentage, ϕ:

ϕ(%)	5.88	12.3	27.3	45.8	69.6	100.0
c_p(cal/g°C)	0.995	0.98	0.92	0.83	0.726	0.617

Table 5.1: *Heat capacity of a solution of methyl alcohol.*

All data is provided at $T = 40°\text{C}$ and at atmospheric pressure.

But what if you need to know the heat capacity for a 20% solution? With the information at hand, you might want to find out a good estimate for that value. Or you might want to construct a table that lists the heat capacity for $\phi = 6, 8, 10, \ldots, 100\%$ or any other sequence of values between 5.88 and 100.

More generally, given the value of a certain unknown function at several points, how can you make a good guess of its value at another point?

5.2 Polynomial interpolation

If you only know some values of an otherwise unknown function, one possibility is to construct *some* function passing through the given points and use it as a substitute of the unknown function. For instance, you could use polynomial functions

$$P(x) = p_1 x^n + p_2 x^{n-1} + \cdots + p_{n-1} x^2 + p_n x + p_{n+1} \tag{5.1}$$

because they are continuous, easy to evaluate, and have continuous derivatives. Fig. 5.1 shows a polynomial passing through seven given points.

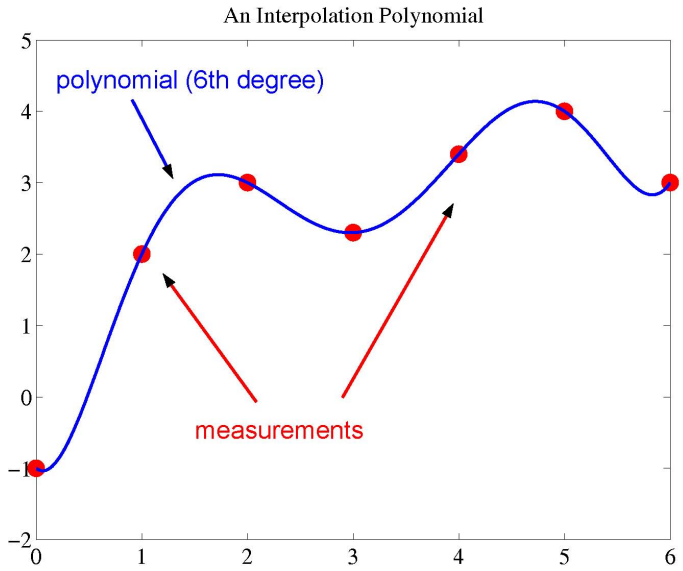

Figure 5.1: *Polynomial interpolation of 7 points.*

5.2.1 Data Interpolation

Suppose there are n+1 distinct (all abscissa points x_j are different) data points (x_j, y_j), $j = 1, \ldots, n+1$. The function $f(x)$ is said to *interpolate* these points if

$$f(x_j) = y_j \qquad 1 \le j \le n+1. \tag{5.2}$$

There can be many different functions (curves) that interpolate a given set of points. For example, the unique straight line (polynomial of degree 1) that passes through two points *interpolates* those points, but there are infinitely many curves (not straight lines, though) that you can draw between the two points.

Interpolation Conditions

If you want the data points to be on the polynomial curve (5.1), you can set up one equation for each point,

$$p_1 x_j^n + \cdots + p_{n-1} x_j^2 + p_n x_j + p_{n+1} = y_j \qquad 1 \le j \le n+1. \tag{5.3}$$

In matrix form these equations are

$$\begin{pmatrix} x_1^n & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & \cdots & x_2^2 & x_2 & 1 \\ & & \vdots & & \\ x_{n+1}^n & \cdots & x_{n+1}^2 & x_{n+1} & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_{n+1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{pmatrix}$$

or $Vp = y$, where p is the vector containing the coefficients of the interpolating polynomial.

A matrix V having this particular structure is called a *Vandermonde* matrix. Square Vandermonde matrices are always invertible, and this implies that the interpolation polynomial defined by (5.3) is unique. Thus, there is a unique polynomial of degree n (or less) that interpolates $n + 1$ distinct points.

In MATLAB

The built-in function `polyfit` sets up V and solves for p

```
>> p = polyfit(x, y, n)
```

It is important to observe that the degree of the polynomial, n, is an input to `polyfit`. Vector p contains the coefficients of the interpolation polynomial as defined in (5.1).

Alternatively, you can set up your own system and solve it:

```
>> V = vander(x)  % Vandermonde matrix corresponding to vector x
>> p = V\y  % solves for p
```

After you have obtained your interpolating polynomial, you can calculate $p(x)$, the value of the polynomial at the particular point x, using the built-in function `polyval`. You can also evaluate the polynomial at several points with a single command by using a vector as input.

```
>> yv = polyval(p, xv)  % evaluates polynomial p at the values in vector xv
```

Although there are several ways of making the necessary computations in MATLAB , there are some essential steps to generate and plot an interpolation polynomial:

1. Computing the coefficients of the polynomial.

2. Generating x-values for plotting.
3. Evaluating the polynomial at the x-values for plotting.
4. Plotting.

To generate the x-values, the usual way is to construct a vector whose coordinates start with the smallest data point and end with the largest. The vector must have a sufficient number of points (say 100) so that the curve looks smooth. For instance, to generate the points $\{0, 0.1, 0.2, 0.3, \ldots, 9.8, 9.9, 10\}$, you could write

```
>> xplot = 0:0.1:10;
```

or you could also write

```
>> xplot = linspace(0,10,101);
```

5.2.2 Example

Given the data in Table 5.1 calculate the interpolating polynomial of least possible degree and predict $f(10)$. Then you are to construct a table that lists the heat capacity for $\phi = 6, 8, 10, \ldots, 100\%$.

Every person has his or her own style of coding, but there are some important issues that must be addressed in every code. Apart from the fact that it should give the correct numerical results, it should be economical and logical and should have enough comments so that any person reading the code can more or less easily figure out what it does. Here is one possible solution.

```
% constructs an interpolating polynomial given 6 data points.
% evaluates the polynomial at a given point.
% uses polyfit and polyval.
%
% Given data points
x=[5.88  12.3  27.3  45.8  69.6  100.0]';  % column vectors
y=[0.995  0.98  0.92  0.83  0.726  0.617]';
%
% Interpolating polynomial
% Coefficients are found using polyfit
format short g
p=polyfit(x,y,5)   %  alternatively, V=vander(x); p=V\y
%
```

```
% Evaluation at a given point
% The value of the polynomial at 10 is obtained using polyval
v=polyval(p,10)
%
% Plot with title, axes labels and legend
xplot=linspace(5.88,100); % construct 100 plotting points
yplot=polyval(p,xplot);   % evaluate the polynomial at plotting points
plot(10,v,'o',xplot,yplot,x,y,'ro')
xlabel('x')
ylabel('y')
title('Interpolating polynomial, data points and evaluated point')
legend('evaluated pt','polynomial','data pts')
```

The solution generated by this script is the following:

$$\begin{aligned} p(x) &= 3.2644 \times 10^{-11}x^5 - 1.5357 \times 10^{-8}x^4 + 2.5152 \times 10^{-6}x^3 \\ &\quad -0.0001047x^2 + 0.00016375x + 0.99944 \\ p(10) &= 0.9864 \end{aligned}$$

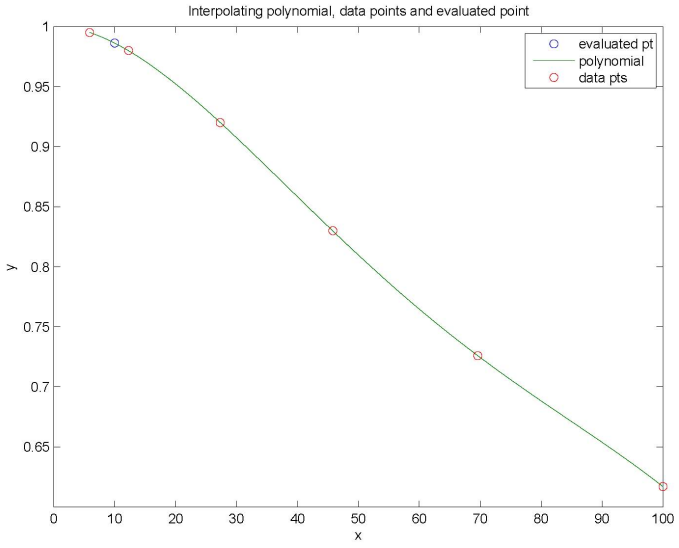

Figure 5.2: *Interpolating polynomial for Table 5.1.*

To construct the table, you might want to use an M-function:

```
function [A,p]=perry(x,y,points)
```

```
% constructs a table and plots interpolation polynomial
% input values: x, y, data points (x ordered from smallest to
%               largest); points, row vector of desired table pts
% output values: A is the table, p contains the polynomial
%                coefficients, starting at the highest degree
%
n = length(x)-1;  % degree of the polynomial
p = polyfit(x,y,n);  % calculate interpolation polynomial
yt = polyval(p,points);  % evaluate polynomial at table points
A = [points' yt'];  % construct table
xplot = linspace(points(1),points(end));  % construct plotting pts
yplot = polyval(p,xplot);
plot(x,y,'o',xplot,yplot)
```

After saving this function in `perry.m`, if you write the following MATLAB commands,

```
>> phi=[5.88 12.3 27.3 45.8 69.6 100];
>> cp=[.995 .98 .92 .83 .726 .617];
>> tablepoints=6:2:100;
>> A=perry(phi,cp,tablepoints)
```

you will get

```
A =
    6.0000    0.9948
    8.0000    0.9911
   10.0000    0.9864
      .          .
      .          .
      .          .
   38.0000    0.8681
   40.0000    0.8582
   42.0000    0.8484
   44.0000    0.8387
      .          .
      .          .
      .          .
   96.0000    0.6312
   98.0000    0.6241
  100.0000    0.6170
```

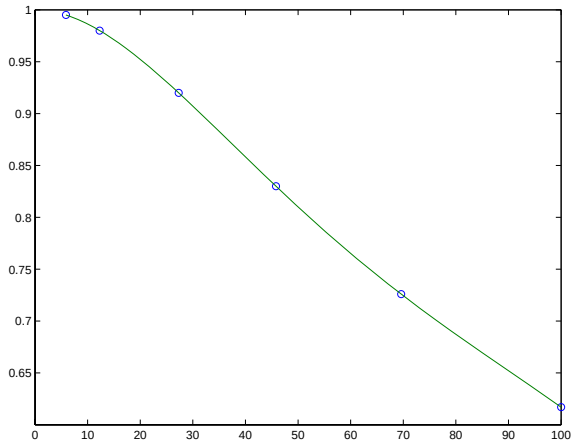

Figure 5.3: *Heat capacity of a solution of methyl alcohol. Interpolation polynomial for Table 5.1.*

5.2.3 Function Interpolation

If you need to work with a function f that is difficult to evaluate, or has some other unattractive property, you may want to substitute f with a polynomial P. Given n+1 distinct values $x_j,\ j = 1, \ldots, n+1$, the polynomial $P(x)$ is said to *interpolate* $f(x)$ at these values if

$$P(x_i) = f(x_i) \qquad 1 \leq i \leq n+1.$$

Function interpolation is also called *function approximation*.

Suppose you want to interpolate the function $f(x) = \sin x + \cos x$ in the interval $[-4, 4]$ by a polynomial. You can decide to *sample* the function at say, 7 equally spaced points, and calculate the value of f at these points

$$f(x_i) = \sin(x_i) + \cos(x_i),\ i = 1, \ldots, 7,$$

producing the values in Table 5.2. You can then construct the 6th degree polynomial that passes through these points, precisely as you did with data interpolation.

Here is a script that samples the given function and then interpolates the sampled points with a polynomial of the appropriate degree. It also plots the original function in blue and the interpolation polynomial in red. The sample points are shown as red circles. The plot has a title and a legend.

```
% Constructs and plots an approximating polynomial
%
f=@(x)sin(x)+cos(x);    % construct the anonymous function
```

x_i	$f(x_i)$
-4	0.10316
-8/3	-1.3466
-4/3	-0.7367
0	1
4/3	1.2072
8/3	-0.43205
4	-1.4104

Table 5.2: *The function* $f(x) = \sin x + \cos x$ *is sampled at seven points.*

```
xsample=linspace(-4,4,7)       % 7 equally spaced sample points
ysample=f(xsample)      % evaluate f at the sample points
n=length(xsample)-1;    % degree of interpolating polynomial
p=polyfit(xsample,ysample,n); % coefficients of polynomial
x=linspace(-4,4);   % get 100 points in the interval
y=f(x);             % evaluate f at the 100 points
yp=polyval(p,x);        % evaluate polynomial at 100 points
plot(x,y,xsample,ysample,'ro',x,yp,'r')
title('The function $f(x)=sin(x)+cos(x)$ approximated by a polynomial of degree 6.')
legend('f(x)','sample',polynomial')
```

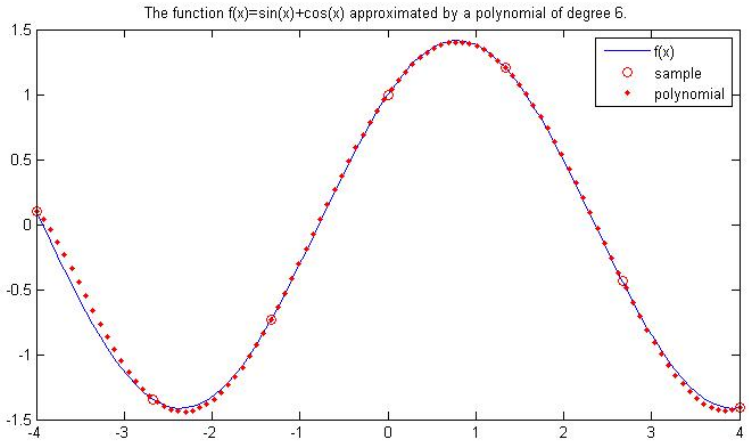

Figure 5.4: *Polynomial approximation of a function sampled at 7 points.*

Note that the approximation looks quite good at *most* points.

5.3 How good is the interpolation? — Errors

Absolute errors

The error between the calculated value and the exact value is defined as

$$error = Y_{computed} - Y_{true}.$$

In the previous problem, the errors at 100 equidistant points are calculated and plotted by adding to the previous script the following commands:

```
error=yp-yplot;
plot(xplot,error)
```

Figure 5.5 shows a plot of the errors.

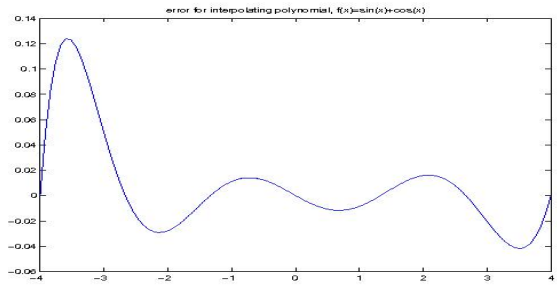

Figure 5.5: *Interpolation error for* $f(x) = \sin x + \cos x$ *with a 6th degree polynomial.*

Is this a small or a large error? It's hard to say, unless you know what the exact solution is supposed to be. In this plot you see that the maximum error is about 0.125, but if the exact value of the function at that point is around 0.1, this would be a very large error. On the other hand, if the value of the function at that point is around 100, the error is not large.

Relative errors

The previous definition of error does not take into account the order of magnitude of the true value. An error of 1 cm is "greater" when measuring a table than when measuring a bridge.

The relative error

$$relative\,error = \frac{Y_{computed} - Y_{true}}{Y_{true}}$$

may be multiplied by 100 to give the error as a percentage of the true value

$$error\,percent = relative\,error \times 100\%.$$

5.4 Interpolation vs Curve Fitting

Both polynomial interpolation and polynomial curve fitting start with a set of data points and construct a polynomial that describes the relation between the points. But there are important differences between the two approaches. In polynomial interpolation, the polynomial curve contains all the data points, while in polynomial curve fitting the polynomial passes as close to the points as possible, but not necessarily through the data points. Table 5.3 enumerates the main differences between the two concepts.

Interpolation	Curve Fitting
polynomial degree = no. points - 1	polynomial degree < no. points - 1
square system of equations	overdetermined system
unique polynomial	user can choose degree of polynomial
solution passes through data pts	solution does not pass through data pts

Table 5.3: *Differences between interpolation and curve fitting.*

5.5 Extrapolation

Estimating the value of $f(x)$ outside the range of the $x_0, x_1, \ldots, x_n$ is risky. Here is a table with the U.S.A. population (in millions) between the years 1920 and 1990.

If the 8-point data is interpolated with a 7th degree polynomial, the population in the year 2000 is estimated as $p(2000) = 173.7$. A close look at the table values

1920	1930	1940	1950	1960	1970	1980	1990
106.5	123.1	132.1	152.3	180.7	205.0	227.2	249.5

Table 5.4: *U.S.A. population in millions.*

indicates this cannot be right. The interpolation polynomial produces a wrong prediction for 2000. The main reason for this is that 2000 is outside the range of the data. The prediction is not an interpolation but an *extrapolation* as 2000 is outside the range of the data points, $[1920, 1990]$.

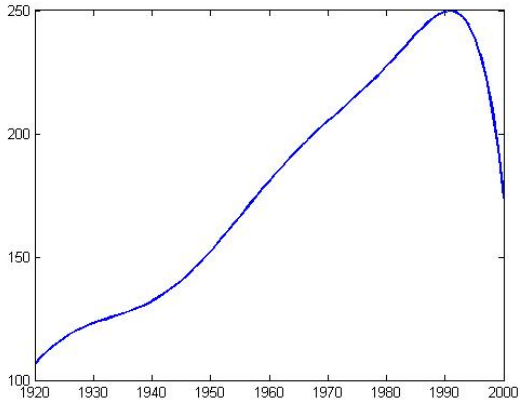

Figure 5.6: *Extrapolation is a dangerous procedure. Prediction of the population in the U.S.A. in 2000.*

5.6 Oscillations

Higher order polynomials tend to oscillate. If we suspect that the underlying function that describes certain data is not oscillatory, it is clear that interpolation with a high degree polynomial can lead to very large errors. Below is a plot of the function

$$f(x) = \frac{1}{1 + 5x^2}$$

interpolated at eleven equally spaced points by a 10th degree polynomial.

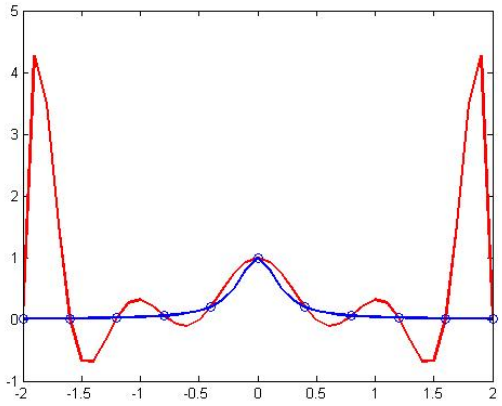

Figure 5.7: *Oscillatory behavior of a high degree polynomial used to interpolate a rational function at 11 equally spaced points.*

5.7 Piecewise Polynomial Interpolation

Piecewise interpolation is an alternative to data interpolation that helps deal with some of the difficulties encountered with interpolation. In polynomial interpolation, a single high-degree polynomial is used to pass through all data points. Piecewise polynomial interpolation uses several low-degree polynomials to interpolate the data.

5.7.1 Piecewise linear interpolation

Suppose you need to use the table we constructed from Perry's handbook to estimate the heat capacity for a 41% solution. You might try to construct the polynomial that interpolates the table points and evaluate it at $\phi = 41$, but there are 48 points, so the polynomial would be of degree 47. Instead, you can interpolate only the two points closest to 41, that is, 40 and 42. Interpolating the table data $(40, 0.8582), (42, 0.8484)$ we get the straight line

$$\phi - 0.8582 = \frac{0.8484 - 0.8582}{42 - 40}(cp - 40).$$

For $cp = 41$ we get $\phi = 0.8533$. If you do the same for every pair of points in the table, you get 47 straight lines.

5.7.2 Piecewise linear interpolation in MATLAB

You can obtain a piecewise linear interpolation by using the built-in function `interp1`. A piecewise linear interpolation of Table 5.1 is obtained with the built-in MATLAB command `interp1`, as shown.

```
>> phi=[5.88 12.3 27.3 45.8 69.6 100];
>> cp=[.995 .98 .92 .83 .726 .617];
>> plin=interp1(phi,cp,'linear','pp')
plin =
      form: 'pp'
    breaks: [5.8800 12.3000 27.3000 45.8000 69.6000 100]
     coefs: [5x2 double]
    pieces: 5
     order: 2
       dim: 1
    orient: 'first'
```

- `pp.coefs` is an $N \times 2$ matrix; N is the number of intervals.
- The rows of this matrix are the polynomial coefficients for the corresponding interval.
- The polynomial for the kth interval has the form

$$\text{p(x)=pp.coefs(k,1)(x-pp.breaks(k))+pp.coefs(k,2)}$$

To visualize the coefficients of each linear polynomial, you need to write

```
>> plin.coefs
ans =
   -0.0023    0.9950
   -0.0040    0.9800
   -0.0049    0.9200
   -0.0044    0.8300
   -0.0036    0.7260
```

From here you read that, for example, the first polynomial (for $x \in [5.88, 12.3]$) is

$$p_1(x) = -0.0023(x - 5.8800) + 0.9950.$$

5.7.3 Evaluating and plotting a linear piecewise polynomial

To evaluate the interpolation polynomial at $phi = 41$, write

```
>> plineval=interp1(phi,cp,41,'linear')
plineval =
    0.8534
```

To plot the linear interpolator, write

```
>> xp=linspace(5.88,100,500);
>> yp=interp1(phi,cp,xp,'linear');
>> plot(xp,yp,phi,cp,'o')
```

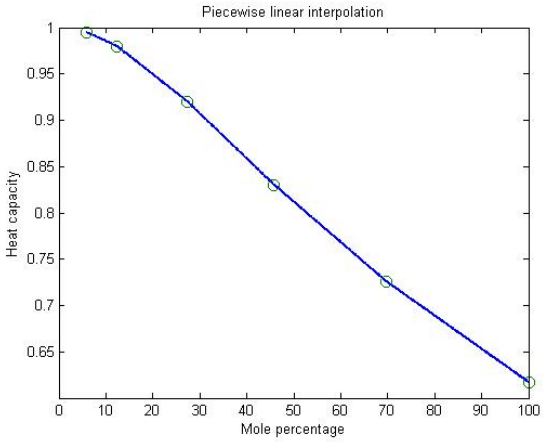

Figure 5.8: *Piecewise linear interpolation.*

5.8 Splines

You can make a similar piecewise interpolation, but instead of straight lines use cubic polynomials between each pair of points. In order to get a nice smooth curve, the first and second derivatives of the polynomials are made continuous at the data points. This is very practical when you have many data points to interpolate, as oscillations are avoided.

Consider the data in Table 5.5. Interpolating with a 9th degree polynomial you get the curve shown in Figure 5.9. Note the range of the polynomial. This

x_i	y_i
0.97447	2.58430
1.1207	0.428980
2.9156	0.058848
2.9989	5.744200
3.3331	7.439000
3.9745	8.068300
6.0971	6.375700
8.3856	2.512700
8.5604	1.443200
9.4423	6.515500

Table 5.5: *Table of data points to be interpolated.*

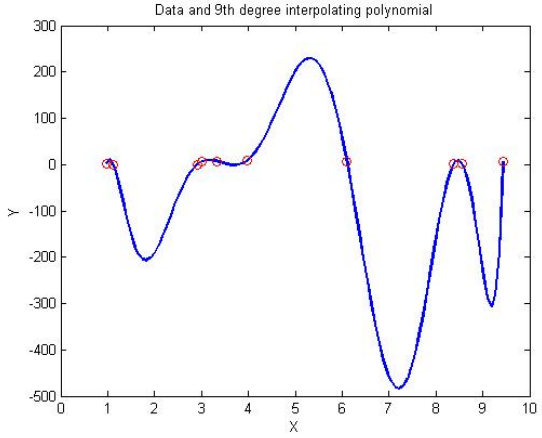

Figure 5.9: *Polynomial interpolation of Table 5.5.*

represents the data **very badly**.

Instead of interpolating with a single polynomial, you can use *splines*:

```
>> x = [0.97447 1.1207 2.9156 2.9989 3.3331 ...
        3.9745 6.0971 8.3856 8.5604 9.4423];
>> y = [2.5843 0.42898 0.058848 5.7442 7.439 ...
        8.0683 6.3757 2.5127 1.4432 6.5155];
>> xp = linspace(x(1),x(end));    % generate 100 x-points
>> yp = spline(x,y,xp);           % generate spline and
                                  % evaluate 100 y-points
>> plot(xp,yp,'r')
```

The resulting spline curve is shown in Figure 5.10. Note, by comparing the y-axes of Figs. 5.9 and 5.10, that the range of the function in the data interval does not diverge as much from the original ordinate data points.

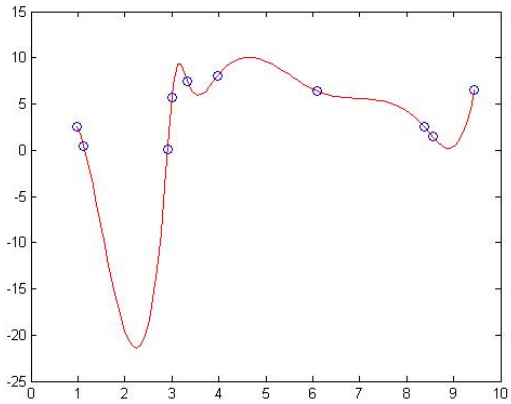

Figure 5.10: *Spline interpolation of Table 5.5.*

The solution of the problem with piecewise linear interpolation is

```
>> x = [0.97447 1.1207 2.9156 2.9989 3.3331 ...
        3.9745 6.0971 8.3856 8.5604 9.4423];
>> y = [2.5843 0.42898 0.058848 5.7442 7.439 ...
        8.0683 6.3757 2.5127 1.4432 6.5155];
>> xp = linspace(x(1),x(end));     % generate 100 x-points
>> ypl = interp1(x,y,xp,'linear');      % generate piecewise linear and
                                % evaluate 100 y-points
>> plot(xp,ypl,'r')
```

The result of linear interpolation is shown in Figure 5.11. If you want the coefficients of each linear piece, write

```
>> s = interp1(x,y,'linear','pp');
>> matrix_coeffs = s.coefs
matrix_coeffs =
       -14.74          2.58
        -0.21          0.43
        68.25          0.06
         5.07          5.74
         0.98          7.44
```

```
-0.80          8.07
-1.69          6.38
-6.12          2.51
 5.75          1.44
```

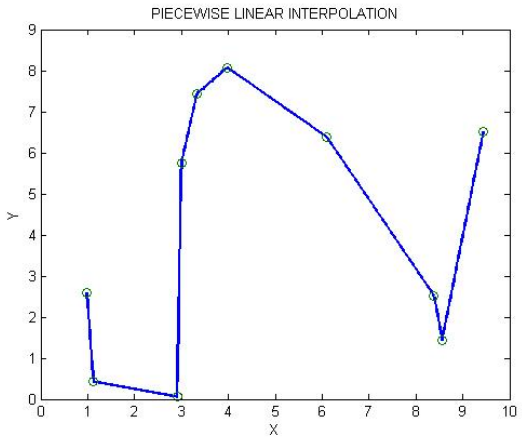

Figure 5.11: *Piecewise linear interpolation of Table 5.5.*

5.8.1 Splines in MATLAB

By typing the command

```
>> sp = spline(x,y)                % this creates the spline structure
```

you obtain the following structure in MATLAB

```
sp =
      form: 'pp'
    breaks: [1x10 double]
     coefs: [9x4 double]
    pieces: 9
     order: 4
       dim: 1
```

This states that there are 9 cubic polynomials, and their coefficients can be obtained with the command `unmkpp`:

```
>> [breaks,coeff,L,K] = unmkpp(sp)   % "unmakes" (shows) structure
breaks =
  Columns 1 through 4
         0.97          1.12          2.92          3.00
  Columns 5 through 8
         3.33          3.97          6.10          8.39
  Columns 9 through 10
         8.56          9.44
coeff =
        16.67        -27.30        -11.10          2.58
        16.67        -19.99        -18.02          0.43
     -1276.96         69.75         71.30          0.06
       287.09       -249.36         56.34          5.74
       -23.26         38.48        -14.13          7.44
         1.33         -6.27          6.53          8.07
        -0.87          2.19         -2.13          6.38
        12.21         -3.81         -5.83          2.51
        12.21          2.60         -6.04          1.44
L =
     9
K =
     4
```

Here you read that the sixth cubic polynomial in your spline is `coeff(6,:)`,

$$1.33(x-3.97)^3 - 6.27(x-3.97)^2 + 6.53(x-3.97) + 8.07.$$

5.8.2 Evaluating and plotting a spline

To calculate the value of the spline at a certain point, say at $x = 8.5$, you need only to write

```
>> cpval = spline(x,y,8.5)
cpval = 1.81
```

Notice that MATLAB chooses the appropriate piece, in this case, the eighth piece. You can plot the previous spline with

```
>> xp = linspace(x(1),x(end),500);
>> yp = spline(px,y,xp);
>> plot(xp,yp,x,y,'o')
```

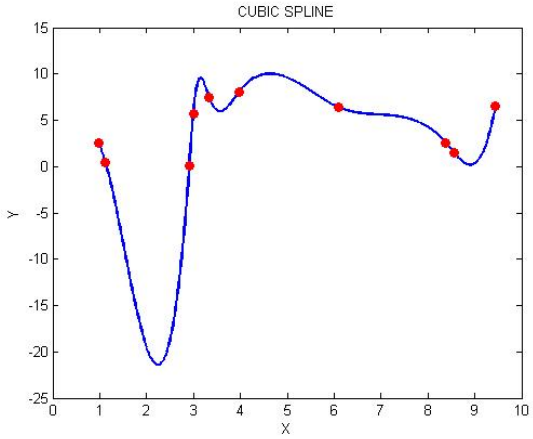

Figure 5.12: *Spline interpolation.*

The extrapolation problem to estimate the U.S.A. population in 2000 can be solved using splines,

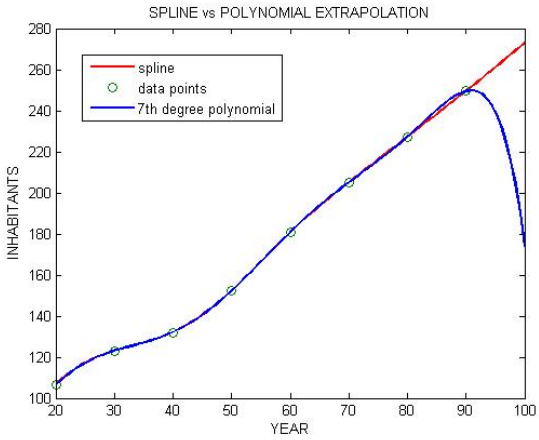

Figure 5.13: *Extrapolation with splines. Estimation of future population in the U.S.A.*

```
>> x = 20:10:90;
>> y = [106.5 123.1 132.1 152.3 ...
       180.7 205 227.2 249.5];
>> pp = interp1(x,y,100,'spline')
pp =
       273.05
```

and it is clear that this solution is acceptable and certainly much better that the solution obtained through interpolation with a single polynomial, as shown in Figure 5.13.

5.9 Exercise

The variation of heat transfer per unit area (q) during the boiling of water under pressure (p) has been found to be as follows:

q (MW/m^2)	1.1	2.4	3.4	3.9	4.0	3.8	3.0	1.2
p (MPa)	0	1	2	4	6	10	15	20

Table 5.6: *Variation of heat transfer.*

Develop a suitable polynomial relation between q and p by trying out curve fitting, polynomial interpolation and spline interpolation.

Correct result: a fifth degree polynomial or a cubic spline.

Chapter 6

Integration

Many definitions are based on integrals. Some of these are

- Area under a curve: $\int_a^b f(x)dx$
- Average value of a function: $f(\xi) = \dfrac{1}{b-a}\int_a^b f(x)dx$
- Distance as a function of velocity: $d = \int_a^b v(t)dt$
- Mass as a function of concentration: $M = \int\int\int c(x,y,z)\,dx\,dy\,dz$
- Heat transfer: $\int\int f(x,y)dx\,dy$ (f is flux in $cal/(cm^2 s)$)

6.1 Motivation

Unsteady Flow: Controlled Drug Release

A drug is encapsulated in a polymer and then released slowly into the bloodstream of patients. 100 μg of a drug in a controlled release capsule is injected into a patient. The drug is released at a rate of $8e^{-0.1t}$ μg/h, where t is hours after injection. What

fraction of the drug remains in the capsule after 24h?

$$\begin{aligned} m_f - m_0 &= -\int_{t_0}^{t_f} \dot{m}(t)\,dt \\ m_f &= 100 - \int_{t_0}^{t_f} 8e^{-0.1t}\,dt \\ m_f &= 100 - \frac{8}{0.1}(e^{-2.4} - e^0) = 100 - 72.7 = 27.3\mu g \end{aligned}$$

Now suppose you have the same problem with a release rate of $8e^{-0.1t^2}$ μg/h. The fraction remaining in the capsule is

$$m_f = 100 - \int_{t_0}^{t_f} 8e^{-0.1t^2}\,dt$$

but the integral cannot be calculated analytically! Instead, you need to approximate it numerically.

Mass flow

The amount of mass transported via a pipe over a period of time is

$$M = \int_{t_1}^{t_2} Q(t)c(t)\,dt$$

where $Q(t) =$ flow rate (m^3/min), $c(t) =$ concentration (mg/m^3). To compute the mass transported between $t_1 = 2$ and $t_2 = 8$ when

$$\begin{aligned} Q(t) &= 9 + 4\cos^2(0.4t) \\ c(t) &= 5e^{-0.5t} + 2e^{0.15t} \end{aligned}$$

you must calculate

$$\int_{t_1}^{t_2} (9 + 4\cos^2(0.4t))(5e^{-0.5t} + 2e^{0.15t})\,dt$$

but it is more convenient to approximate this integral numerically instead of trying to find its primitive.

6.2 Numerical Integration

Numerical *quadrature* is the approximation of definite integrals,

$$I(f) = \int_a^b f(x)dx.$$

If f is a function that does not change sign on the interval $[a, b]$, $I(f)$ may be interpreted as the area between the curve $y = f(x)$ and the x-axis. Areas above the x-axis are positive; those below the x-axis are negative. To approximate the integral as an area, three basic steps may be taken:

1. The interval $[a, b]$ is divided into subintervals.
2. The area on each subinterval is approximated.
3. The areas of all subintervals are added up.

There are many numerical methods designed to accomplish the second step. In general, they consist of two main parts (see Fig. 6.1):

1. Generate a table of discrete values for the integrand or function to be integrated.
2. Estimate the integral numerically using those values.

6.3 Quadrature Rules

Approximating the area under a curve can be done by substituting the original *integrand* (function to be integrated) f with a polynomial. As the formula for calculating the integral of any polynomial is well-known, you can thus construct formulas that approximate the integral of f. This can be accomplished in the following steps:

- Choose the n sample points or *nodes* $x_i \in [a, b]$.
- Sample the integrand f at these points: $f(x_i)$.
- Interpolate f by a polynomial passing through $(x_i, f(x_i))$.
- Integrate the polynomial.

These steps result in a *quadrature rule*

$$Q(f) = \sum_{i=1}^{n} w_i f(x_i)$$

where the quantities w_i are called the *weights*. The exact value of the integral is $I(f) = Q(f) + E(f)$, and $E(f)$ is called the *truncation error*.

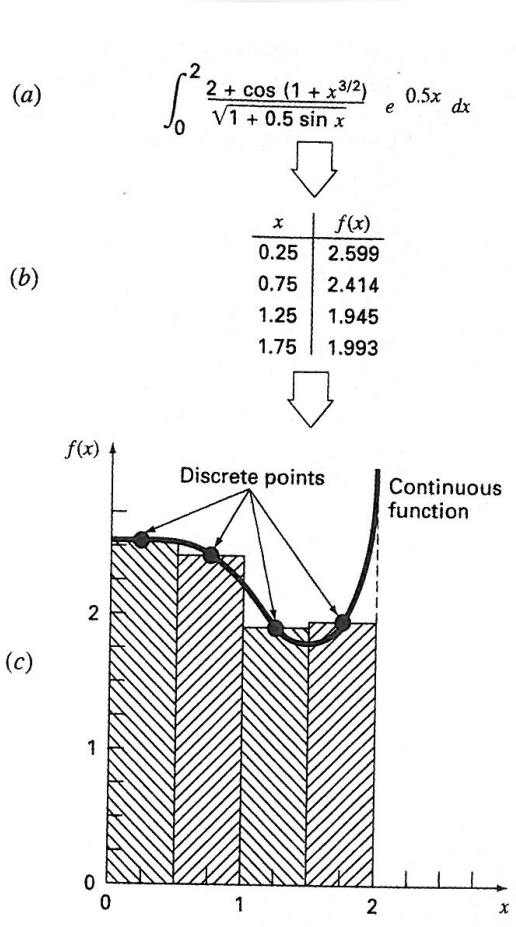

Figure 6.1: *Approximating the area on each subinterval.*

6.3.1 The Trapezoidal Rule

If you take two nodes, $x_0 = a, x_1 = b$, and sample f at these nodes, you can construct the interpolation polynomial passing through $(a, f(a))$ and $(b, f(b))$, the straight line

$$p(x) = f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a}.$$

Performing the integration $\int_a^b p(x)\,dx = \frac{b-a}{2}(f(a)+f(b))$ results in the quadrature formula

$$\int_a^b f(x)dx \approx Q_T(f) = \frac{b-a}{2}(f(a) + f(b)),$$

with truncation error

$$E(f) = -\frac{(b-a)^3}{12} f''(\xi), \quad \xi \in [a, b].$$

This formula is called the *trapezoidal rule* and it approximates the integral below the curve as the area of a trapezoid as shown in Fig. 6.2

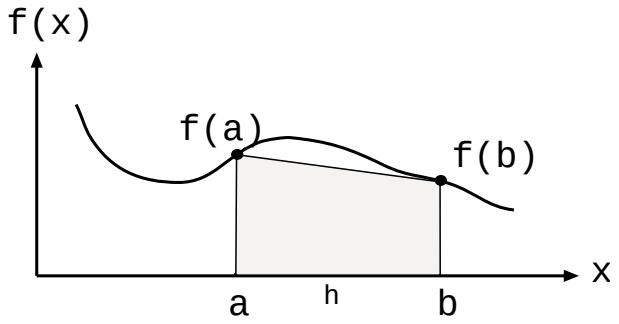

Figure 6.2: *The trapezoidal rule.*

The composite trapezoidal rule

If the interval $[a, b]$ is large, the error of the trapezoidal rule applied to the interval will be large. One way to reduce this error is to divide $[a, b]$ into subintervals,

$$a = x_0 < x_1 < \cdots < x_n = b,$$

$$h = (b-a)/n, \qquad x_i = a + ih.$$

Applying the trapezoidal rule on each subinterval and adding up the results

$$\int_a^b f(x)dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)dx \approx \frac{1}{2} \sum_{i=1}^{n} h(f(x_{i-1}) + f(x_i))$$

yields the *composite trapezoidal rule*

$$\int_a^b f(x)dx \approx T(f) = \frac{h}{2} \left(f(a) + 2 \sum_{i=1}^{n-1} f(a+ih) + f(b) \right).$$

The truncation error for this formula can be calculated by adding the truncation errors for each subinterval,

$$\begin{aligned} E_T(f) &= \sum_{i=1}^{n} -\frac{1}{12}h^3 f''(c_i), \quad c_i \in (x_{i-1}, x_i) \\ & \quad -\frac{1}{12}nh^3 f''(c) = -\frac{1}{12}(nh)h^2 f''(c) \\ &= -\frac{1}{12}(b-a)h^2 f''(c), \quad c \in [a,b]. \end{aligned}$$

The truncation error still depends on the length of the interval, $b-a$, but it also depends on the length of the subintervals, h, so by taking a small h you can reduce the truncation error. In fact, the truncation error is $\mathcal{O}(h^2)$, meaning that it behaves like the function $e(h) = ch^2$, with c constant. Thus, if the length of the subintervals, h, is halved, the truncation error is reduced by one fourth ($(1/2)^2 = 1/4$).

6.3.2 Simpson's rule

Take three nodes in $[a,b]$, a, $(a+b)/2$, and b. Interpolate with a quadratic polynomial and integrate to get the quadrature formula

$$\int_a^b f(x)dx \approx Q_S(f) = \frac{h}{3}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right)$$

where $h = (b-a)/2$. This is called Simpson's rule and its truncation error is

$$E(f) = -\frac{h^5}{90}f^{(4)}(\xi), \qquad \xi \in [a,b].$$

Composite Simpson's rule

Divide the interval into $N = 2m$ subintervals. The step size is $h = (b-a)/N$, and the nodes are $x_i = a + ih$ for $i = 0, 1, \ldots, N$. Adding the approximations on each subinterval you obtain

$$S(f) = \frac{h}{3}\left(f(x_0) + 2\sum_{j=1}^{m-1} f(x_{2j}) + 4\sum_{j=0}^{m-1} f(x_{2j+1}) + f(x_{2m})\right)$$

with truncation error

$$E_S(f) = -\frac{1}{180}(b-a)h^4 f^{(4)}(\xi), \xi \in [a,b].$$

6.3.3 Comparison between Simpson's and Trapezoidal Rules

By comparing the truncation errors of the Trapezoidal rule and Simpson's rule, you can see that

$$\begin{aligned} E_T(f) &= -\frac{1}{12}(b-a)h^2 f''(\xi_1), \quad \xi_1 \in [a,b]; \\ E_S(f) &= -\frac{1}{180}(b-a)h^4 f^{(4)}(\xi_2), \quad \xi_2 \in [a,b]; \\ E_S(f) &= h^2 \frac{f^{(4)}(\xi_2)}{15 f''(\xi_1)} E_T(f). \end{aligned}$$

When h is small Simpson's rule has a smaller truncation error than the Trapezoidal rule. Two conclusions can be reached from this assertion:

- For the same step size h, Simpson's rule is more accurate than the Trapezoidal rule.
- To get the same accuracy, Simpson's rule needs less computations than the Trapezoidal rule.

6.4 Integrating Tabulated Data

Sometimes one does not have a function to integrate, but only a table of data representing discreet values of some underlying function. When this is the case, you are not at liberty to decide what the lengths of the subintervals will be.

Suppose you want to determine the distance traveled by a certain vehicle from the following velocity data:

t	1	2	3.25	4.5	6	7	8	8.5	9.3	10
v	5	6	5.5	7	8.5	6	6	7	7	5

Table 6.1: *Table of velocity vs. time.*

You can apply the trapezoidal rule on each subinterval to approximate the distance traveled by the vehicle by

$$\begin{aligned} \int_1^{10} v(t)\,dt &\approx \frac{2-1}{2}(5+6) + \frac{3.25-2}{2}(6+5.5) + \frac{4.5-3.25}{2}(5.5+7) \\ &\quad + \frac{6-4.5}{2}(7+8.5) + \frac{7-6}{2}(8.5+6) + \frac{8-7}{2}(6+6) \\ &\quad + \frac{8.5-8}{2}(6+7) + \frac{9.3-8.5}{2}(7+7) + \frac{10-9.3}{2}(7+5) \\ &= 58.4250. \end{aligned}$$

Trapezoidal Rule in MATLAB

The built-in function `trapz` can be used to approximate an integral when a data table with the points (X,Y) is given. The command to be used is

```
Z = trapz(X,Y)
```

For instance, to integrate the tabulated data in Table 6.1 you write

```
>> t=[1 2 3.25 4.5 6 7 8 8.5 9.3 10];
>> v=[5 6 5.5 7 8.5 6 6 7 7 5];
>> d=trapz(t,v)
```

6.5 Adaptive quadrature methods

Adaptive methods automatically take into account the behavior of f. To use an adaptive quadrature method the user supplies the integrand f, the integration interval $[a, b]$ and the *tolerance* EPS. This last quantity is the desired upper limit of the truncation error.

6.5.1 Adaptive Simpson

Simpson's rule in $[a, b]$ is

$$S(a,b) = \frac{b-a}{3}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right),$$

with a truncation error of

$$E = -\frac{1}{180}(b-a)^5 f^{(4)}(\xi).$$

Suppose you are able to estimate he value of E. If $E >$ EPS, divide the interval in two and apply Simpson's rule to each subinterval.

$$\int_a^b f(x)dx = \sum_{i=1}^{2} S_i + \sum_{i=1}^{2} E_i$$

where S_1 approximates the integral on $[a, a+h]$ and S_2 on $[a+h, b]$, with $h = (b-a)/2$. The sum of the errors must be less than EPS, so you need to make sure that

$$|E_1| \leq \frac{1}{2}\text{EPS}, \quad |E_2| \leq \frac{1}{2}\text{EPS}.$$

If the error in a subinterval is too big, divide that subinterval in two, apply Simpson's rule to each of these subintervals and check if the errors are less than $\frac{1}{4}$EPS. Continue this process until the sum of all errors is less than EPS.

Example

Adaptive Simpson was used to approximate $\int_0^1 \frac{1}{1+100x^2}dx$ with EPS=10^{-6}. Note

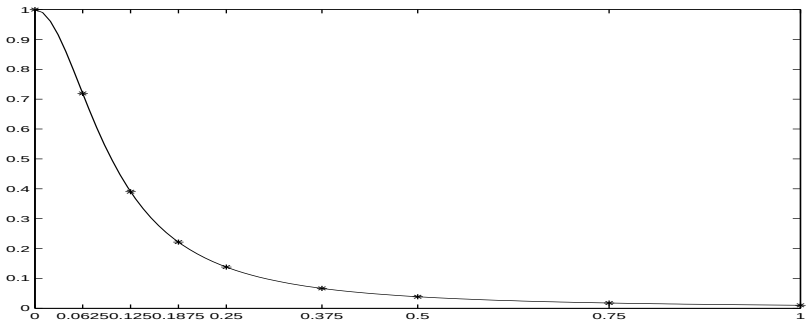

Figure 6.3: *Adaptive integration.*

the unequally spaced nodes; they are placed so that they are denser where the integrand changes faster.

6.5.2 Adaptive Simpson's Rule in MATLAB

The built-in MATLAB function `quad` summons the adaptive Simpson's rule.

`q = quad(fun,a,b)` approximates the integral of function `fun` from `a` to `b` to within an error of 10^{-6}.

`q = quad(fun,a,b,eps)` uses an error tolerance `eps` instead of the default tolerance 10^{-6}.

`[q,fnce] = quad(fun,a,b,eps)` also shows the number of function evaluations, `fnce`. This quantity gives a measure of the computational cost of the numerical integration.

The approximation of $\int_0^{24} 8e^{-0.1t^2}dx$ with tolerances of 10^{-3}, 10^{-6} and 10^{-9}, gives the following results:

```
>> f=@(t) 8*exp(-0.1*t.^2);
>> [m,fnce]=quad(f,0,24,1e-3)
m =
  22.42002755915631
fnce =
    25
>> [m,fnce]=quad(f,0,24,1e-6)
m =
  22.41996524229996
fnce =
    89
>> [m,fnce]=quad(f,0,24,1e-9)
m =
  22.41996486570505
fnce =
   325
```

Note that a smaller tolerance results in more computations (longer time).

Integrating a Function

$$\int_0^{24} 8e^{-0.1t^2}dt$$

```
>> f=@(t) 8*exp(-0.1*t.^2);
>> x=0:4:24;
>> m=trapz(x,f(x))
m = 22.5139
```

- Anonymous function: af = @(x) sin(x) + 2 * cos(x)

$$\int_a^b (\sin(t) + 2\cos(t))\,dt \approx \texttt{quad(af,a,b)}$$

- M-file function: a file called `mf.m` containing

```
function f=mf(y)
f=sin(y)+2*cos(y);
```

$$\int_a^b (\sin(t) + 2\cos(t))\, dt \approx \texttt{quad(@mf,a,b)}$$

6.6 Precision of quadrature rules

Suppose the heat capacity of a material is given by

$$c(T) = 0.132 + 1.56 \times 10^{-4} T + 2.64 \times 10^{-7} T^2.$$

To compute the heat required to raise 1000 g of this material from -100 to 200 degrees (C) you can use the equation

$$\Delta H = m \int_{T_1}^{T_2} c(T)\, dT$$

1) analytically, 2) with Simpson's rule and 3) with the Trapezoidal rule.

1. Analytical integration

$$\begin{aligned} c(T) &= 0.132 + 1.56 \times 10^{-4} T + 2.64 \times 10^{-7} T^2 \\ \Delta H &= m \int_{T_1}^{T_2} c(T)\, dT \\ \Delta H &= 1000 \int_{-100}^{200} (0.132 + 1.56 \times 10^{-4} T + 2.64 \times 10^{-7} T^2)\, dT \\ \Delta H &= 1000(0.132\, T + 1.56 \times 10^{-4} T/2 + 2.64 \times 10^{-7} T^3/3))|_{-100}^{200} \\ &= 42732 \text{ cal} \end{aligned}$$

2. Numerical integration with Simpson's rule

```
>> c=@(t)1000*(0.132+1.56e-4*t+2.64e-7*t.^2)
>> qs=quad(c,-100,200,1e-6)
qs =
       42732
```

Notice that even if we have a very large `tol`

```
>> qs=quad(c,-100,200,1e+2)
qs =
    42732
```

the result is exact. This occurs because the function c is a quadratic polynomial, and Simpson's rule is exact for all polynomials of degree 3 or less.

3. Numerical integration with the Trapezoidal rule If you take 5, 50 and 500 nodes in the interval $[-100, 200]$,

```
>> t=linspace(-100,200,5);
>> qt=trapz(t,c(t))
qt =
    42806.25000000000
>> t=linspace(-100,200,50);
>> qt=trapz(t,c(t))
qt =
    42732.49479383590
>> t=linspace(-100,200,500);
>> qt=trapz(t,c(t))
qt =
    42732.00477106517
```

you can observe that taking more nodes will increase the accuracy of the approximation, in accordance with the fact that the truncation error for this quadrature formula is $\mathcal{O}(h^2)$. Unlike Simpson's rule, integrating a quadratic polynomial did not result in an exact solution. The Trapezoidal rule is exact only for polynomials of degree 1 or less.

6.7 Exercise

The velocity profile of a falling film is

$$v_z = \frac{\rho g \delta^2 \cos\beta}{2\mu}\left(1 - (\frac{x}{\delta})^2\right)$$

with the following constants:

$$\rho = 800 \quad g = 9.81 \quad \delta = 2.5 \times 10^{-3} \quad \beta = \pi/4 \quad \mu = 2.5 \times 10^{-7}$$

Calculate the volume flow per meter

$$q_m = \int_0^\delta v_z dx$$

with Simpson's rule and with the trapezoidal rule. Is any of these approximations actually the exact value? Why?

Correct result: q (quad) =115.6120, q (trapz, 100 pts) = 115.6090

Chapter 7

Ordinary Differential Equations

Equations composed of unknown functions and their derivatives are called *differential equations*. They play a fundamental role in chemical engineering and biotechnology because many physical phenomena are formulated in terms of such equations. Some examples, using different but commonly used notations, are the following:

$$\begin{aligned}
\frac{dc}{dx} &= -D/J \\
\frac{dy}{dx} &= -2x^3 + 12x^2 - 20x + 8.5 \\
y' &= 4e^{0.8x} - 0.5y \\
\frac{dz}{dt} &= -2zt \\
\begin{cases} \dot{c}_1 &= 0.15c_1 - 0.15c_2 \\ \dot{c}_2 &= 0.1c_1 + 0.2c_2 - 0.4 \end{cases}
\end{aligned}$$

In an equation such as

$$\frac{dz}{dt} = -2zt$$

the quantity being differentiated, z, is called the *dependent variable*. The quantity with respect to which z is differentiated, t, is called the *independent variable*. When there is only one independent variable, the equation is called an *ordinary differential equation* (ODE), in contrast to *partial differential equations* that involve more than one independent variable. Differential equations are also classified according to their order. A *first-order differential equation* contains only first derivatives. A first-order differential equation is said to be in *explicit form* if it is written as

$$\frac{dy}{dx} = f(x, y).$$

All the examples above are in explicit form.

7.1 Motivation

Transient Response of a Reactor

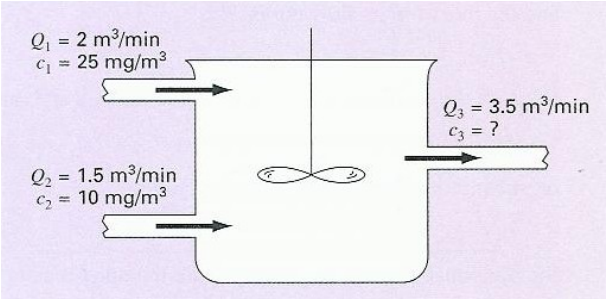

Figure 7.1: *A single reactor with two inflows and one outflow. The accumulation of mass in the reactor is a function of time.*

According to the conservation of mass principle, if the reactor in Fig. 7.1 has constant volume,

$$\text{Accumulation} = \text{inputs - outputs.}$$

Accumulation represents the change in mass in the reaction per change in time,

$$\text{Accumulation } = V\frac{dc}{dt}$$

where V is the volume and c is the concentration.

During the *transient state*,

$$Q_1c_1 + Q_2c_2 - Q_3c_3 = V\,\frac{dc_3}{dt}.$$

After some time, the inputs are the same as the outputs and the mass remains constant. The reactor has reached *steady-state*:

$$\text{Accumulation } = 0 \Rightarrow Q_1c_1 + Q_2c_2 - Q_3c_3 = 0.$$

7.2 The solution of an ODE

Consider the differential equation

$$\frac{dy}{dx} = x^2 + 3$$

and solve it by integrating

$$y = \int (x^2 + 3)\, dx$$

to get

$$y = x^3/3 + 3x + C$$

In order to determine the integration constant C we need an extra condition. An *initial value* or *initial condition*, $y(x_0) = y_0$, will determine the solution on an interval $[x_0, x_f]$. For instance, if $y(0) = 6$, the solution is $y = x^3/3 + 3x + 6$. An ODE with an initial value is called an ***initial value problem*** (IVP).

7.3 The numerical solution of an IVP

In contrast to an analytic solution of an initial value problem, a numerical method finds a *discrete* solution, that is, it finds the solution at *a finite number of points*. The numerical solution can be interpreted as an approximate sample of the true analytical solution, which is a continuous function. One might want to join the solution points to get an approximation to the continuous solution.

7.4 Euler's method

To approximate $y'(t) = f(t, y)$ we approximate the derivative as the slope of the tangent line

$$y'(t) \approx \frac{y(t+h) - y(t)}{h}.$$

Call this approximation $u(t)$, and you have

$$\frac{u(t+h) - u(t)}{h} = f(t, u(t)).$$

A new value can be predicted from the initial condition, $y(t_0) = y_0$,

$$u_1 = u_0 + hf(t_0, u_0)$$

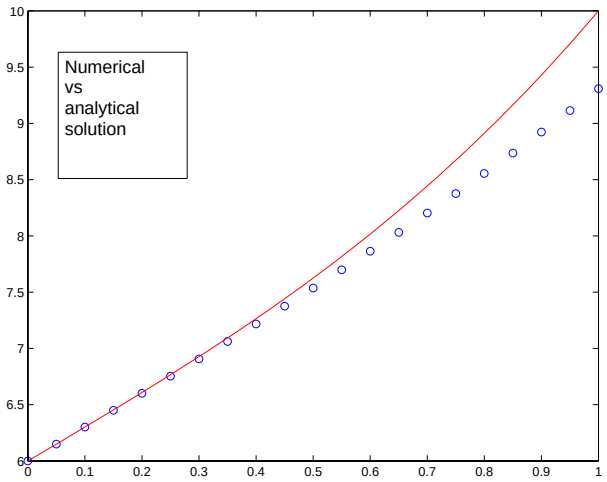

Figure 7.2: *The numerical solution of an initial value problem is a discrete set of points.*

where $u_0 = y_0$. The quantity h is called the *step size*, and u_1 is an approximation to $y(t_0 + h)$. The numerical method

$$\begin{aligned} u_0 &= y_0 \\ u_{n+1} &= u_n + hf(t_n, u_n) \end{aligned}$$

approximates $y(t_{n+1})$ by u_{n+1} with $t_{n+1} = t_n + h$. This formula is called *Euler's method*.

7.4.1 Example

Consider the reactor depicted in Fig. 7.3, where $c_{in} = 50mg/m^3$, $Q = 5m^3/min$, $V = 100m^3$, $c_0 = 10mg/m^3$.

Applying the conservation of mass,

$$\text{Accumulation} = \text{input -output}$$

you get the differential equation

$$V\frac{dc}{dt} = Qc_{in} - Qc.$$

You must first write the equation in explicit form,

$$\frac{dc}{dt} = \frac{1}{V}(Qc_{in} - Qc)$$

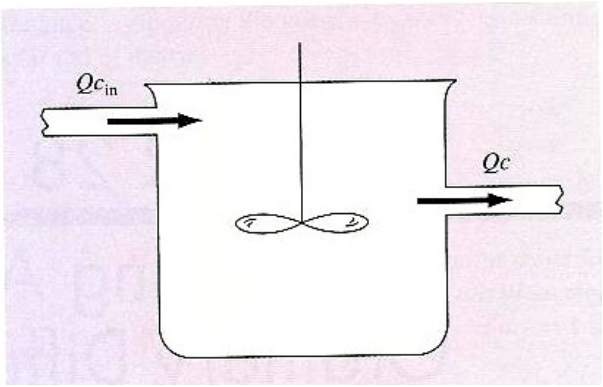

Figure 7.3: *A single well-mixed reactor with one inflow and one outflow.*

and then identify the independent variable, t, the dependent variable, c, the right-hand side function

$$f(t,c) = \frac{1}{V}(Qc_{in} - Qc) = \frac{1}{100}(5 \cdot 50 - 5c) = 2.5 - 0.05c$$

and the initial value

$$c(0) = 10$$

Now you can decide the value of h and set up Euler's equation

$$\begin{aligned} h &= 0.5 \\ f(t_n, u_n) &= 2.5 - 0.05u_n \\ u_0 &= 10 \\ u_{n+1} &= u_n + hf(t_n, u_n) \end{aligned}$$

MATLAB code for Euler's method

```
function [t,Y]=myeuler(f,interval,y0,M)
% solves y'=f(x,y), one dimension
% Input  - f  entered as @f (if M-file) or f (if anonymous funct)
%        - interval is a vector [a,b]
%        - y0 is the initial condition y(a)
%        - M is the number of steps
% Output - t is the vector of abscissas and
%          Y is the solution vector
```

```
h=(interval(2)-interval(1))/M;  % step size
Y=zeros(M+1,1); % solution will be column vector
t=(interval(1):h:interval(2))'; % column vector independent variable
Y(1)=y0;        % initial value
for j=1:M
   Y(j+1)=Y(j)+h*f(t(j),Y(j));  % Euler's equation
end
```

In order to run this MATLAB code you must have defined the right-hand side of your differential equation. This can be done with an M-file or as an anonymous function.

Solution with an M-file function

The code for the right-hand side function is

```
function y = myf(t,x)
y = 2.5 - 0.05*x;
```

Running the code with 10 steps or $h = 0.5$ gives

```
>> [X,Y] = myeuler(@myf,[0,5],10,10);
>> E = [X Y]
E =
         0   10.0000
    0.5000   11.0000
    1.0000   11.9750
    1.5000   12.9256
    2.0000   13.8525
    2.5000   14.7562
    3.0000   15.6373
    3.5000   16.4963
    4.0000   17.3339
    4.5000   18.1506
    5.0000   18.9468
```

Solution with an anonymous function

You can also define the function as an anonymous function,

```
>> f = @(t,x)2.5 - 0.05*x;
```

```
>> [X,Y] = myeuler(f,[0,5],10,10)
>> E = [X Y]
E =
         0   10.0000
    0.5000   11.0000
    1.0000   11.9750
    1.5000   12.9256
    2.0000   13.8525
    2.5000   14.7562
    3.0000   15.6373
    3.5000   16.4963
    4.0000   17.3339
    4.5000   18.1506
    5.0000   18.9468
```

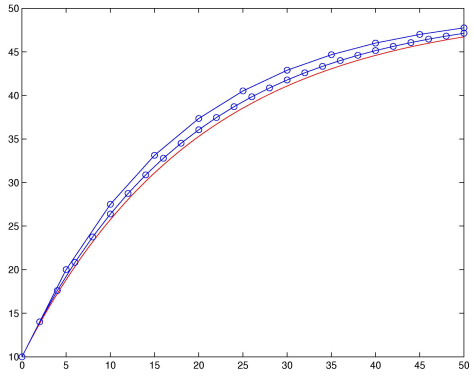

Figure 7.4: *Numerical solution with Euler's method. The top solution was obtained using h=0.5 (N=10), the middle one was obtained using h=0.2 (N=25), and the lower curve is the exact solution obtained analytically.*

7.5 MATLAB Solvers

There are several built-in solvers for first-order ordinary differential equations in explicit form with initial values. The general syntax of these solvers is

```
[T,Y] = solver(odefun,tspan,y0)
```

- *solver* is the name of the method
- *odefun* is the actual name of the function f in $y' = f(t, y)$, defined as an M-function
- *tspan* is the interval or vector of points where the solution is desired
- *y0* is the initial value

A general-use adaptive solver is `ode45`. If the solver is called as

```
[t,Y] = ode45(@myf,[0,5],10)
```

it gives the solution at the points it chooses to solve. If called as

```
[t,Y] = ode45(@myf,0:.05:6,10)
```

it only gives the solution at the points indicated by the user, although it uses exactly the same points as the previous syntax to solve the IVP.

```
>> [t,Y] = ode45(@myf,0:.5:5,10)
t =
                         0
    5.000000000000000e-001
    1.000000000000000e+000
    1.500000000000000e+000
    2.000000000000000e+000
    2.500000000000000e+000
    3.000000000000000e+000
    3.500000000000000e+000
    4.000000000000000e+000
    4.500000000000000e+000
    5.000000000000000e+000

Y =
    1.000000000000000e+001
    1.098760351886393e+001
    1.195082301996605e+001
    1.289026054685001e+001
    1.380650327855137e+001
    1.470012389660369e+001
    1.557168094298307e+001
```

```
1.642171916921507e+001
1.725076987686218e+001
1.805935124960457e+001
1.884796867712176e+001
```

7.6 Systems of equations

The system

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \ldots, y_n) \\ y_2' &= f_2(x, y_1, y_2, \ldots, y_n) \\ &\cdot \\ &\cdot \\ y_n' &= f_n(x, y_1, y_2, \ldots, y_n) \end{aligned}$$

has one independent variable, x, and n dependent variables. It requires n initial values at the starting value of x.

7.6.1 Examples

Reaction dynamics

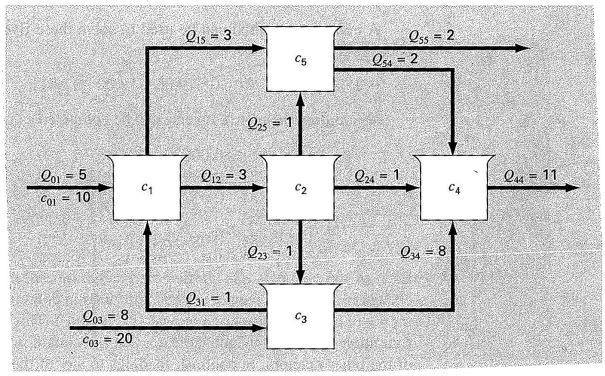

Figure 7.5: *Conservation of mass in a system of reactors with constant volumes.*

The volumes of the reactors (in m^3) are $V_1 = 50, V_2 = 20, V_3 = 40, V_4 = 80, V_5 = 100$. The flow rates Q are in $\mathrm{m}^3/\mathrm{min}$, and concentrations c in ml/m^3. At $t = 0$ concentrations are 0. How will concentrations increase over the next hour?

In the transient state, the mass balance for each reactor is

Reactor 1: $V_1\dot{c}_1 = Q_{01}c_{01} - Q_{12}c_1 - Q_{15}c_1 + Q_{31}c_3$
Reactor 2: $V_2\dot{c}_2 = Q_{12}c_1 - Q_{23}c_2 - Q_{24}c_2 - Q_{25}c_2$
Reactor 3: $V_3\dot{c}_3 = Q_{03}c_{03} + Q_{23}c_2 - Q_{31}c_3 - Q_{34}c_3$
Reactor 4: $V_4\dot{c}_4 = Q_{24}c_2 + Q_{34}c_3 - Q_{44}c_4 + Q_{54}c_5$
Reactor 5: $V_5\dot{c}_5 = Q_{15}c_1 + Q_{25}c_2 - Q_{55}c_5 - Q_{54}c_5$

and substituting the data values, you get the IVP

$$\frac{d}{dt}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{pmatrix} = \begin{pmatrix} -0.12c_1 + 0.02c_3 + 1 \\ 0.15c_1 - 0.15c_2 \\ 0.025c_2 - 0.225c_3 + 4 \\ 0.0125c_2 + 0.1c_3 - 0.1375c_4 + 0.025c_5 \\ 0.03c_1 + 0.01c_2 - 0.04c_5 \end{pmatrix}, \qquad \begin{pmatrix} c_1(0) \\ c_2(0) \\ c_3(0) \\ c_4(0) \\ c_5(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

or $\dot{C} = f(t, C)$, $C(0) = 0$.

Independent variable: time, t (min)

Dependent variable: concentrations, 5×1 vector C (mg/m^3)

Right-hand-side: function, 5×1 vector $f(t, C)$.

Initial value: 5×1 vector $C_0 = 0$

Euler's formula is

$$c_{n+1} = c_n + hf(t_n, c_n),$$

where 5×1 vector c_n approximates C(t_n). To solve the problem with `ode45`, you must define $f(t, C)$ in an M-file.

```
function cdot=myreaf(t,c)
% evaluates a function of several variables
% input t is one dimensional; c is 5-by-1 column vector
% output cdot is a 5-by-1 vector, the right-hand side of the differential equation
cdot = zeros(5,1); % defines cdot as a column vector with 5 components
cdot(1) = -0.12*c(1)+0.02*c(3)+1;
cdot(2) = 0.15*c(1)-0.15*c(2);
cdot(3) = 0.025*c(2)-0.225*c(3)+4;
cdot(4) = 0.0125*c(2)+0.1*c(3)-0.1375*c(4)+0.025*c(5);
cdot(5) = 0.03*c(1)+0.01*c(2)-0.04*c(5);
```

Calling the solver and plotting the solution,

```
>> [t,Y] = ode45(@myreaf,[0,60],zeros(5,1));
>> subplot(3,2,1),plot(t,Y(:,1),'.')
>> title('Concentration in reactor 1')
>> subplot(3,2,2),plot(t,Y(:,2),'.')
>> title('Concentration in reactor 2')
>> subplot(3,2,3),plot(t,Y(:,3),'.')
>> title('Concentration in reactor 3')
>> subplot(3,2,4),plot(t,Y(:,4),'.')
>> title('Concentration in reactor 4')
>> subplot(3,2,5),plot(t,Y(:,5),'.')
>> title('Concentration in reactor 5')
```

you get Fig. 7.6.

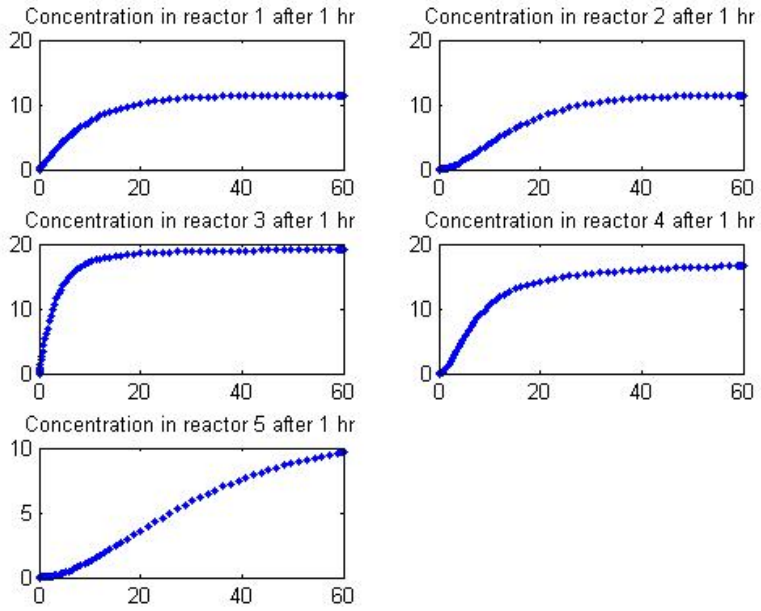

Figure 7.6: *Concentrations in each reactor as a function of time.*

Nonisothermal Batch Reactor

A nonisothermal batch reactor can be described by the following equations:

$$\begin{aligned} \frac{dC}{dt} &= -e^{-10/(T+273)}\, C \\ \frac{dT}{dt} &= 1000\, e^{-10/(T+273)}\, C - 10(T-20) \end{aligned}$$

where C is the concentration of the reactant and T is the temperature of the reactor. Initially the reactor is at 25°C and has a concentration of reactant C of 1.0 gmol/L. Find the concentration and temperature of the reactor as a function of time (min).

Independent variable: time, t (min)

Dependent variable: Y=[C;T], 2×1 vector (gmol/L; °C)

Right-hand-side: function, 2×1 vector $f = (Cdot; Tdot)$, where $Cdot = -e^{-10/(T+273)}\, C$ and $Tdot = 1000\, e^{-10/(T+273)}\, C - 10(T - 20)$

Initial values: 2×1 vector $[1; 25]$

Euler's formula for this problem is

$$u_{n+1} = u_n + hf(t_n, u_n)$$

where the 2×1 vector u_n approximates [C(t_n);T(t_n)].

7.6.2 Modifying the Euler code (for systems)

To solve these systems with Euler's method, you need to modify the Euler code to allow it to work with vectors. The dependent variable and the initial value are now vectors of the same dimension as the number of equations.

```
function [t,Y]=myeulers(f,interval,y0,M)
% solves y'=f(x,y), several dimensions
% Input  - f is the function entered as @f, defined in M-file
%        - interval is vector [a,b] with left and right endpts
%        - y0 is the initial condition y(a), column vector
%        - M is the number of steps
% Output - t: vector of abscissas,Y: matrix of solution vectors
h=(interval(2)-interval(1))/M;   % step size
s=length(y0);   % number of equations
Y=zeros(M+1,s); % solution will be M+1 vectors of dimension s
t=(interval(1):h:interval(2))';  % column vector independ. var.
Y(1,:)=y0;      % initial value
for j=1:M
   Y(j+1,:)=Y(j,:)+h*f(t(j),Y(j,:));  % Euler's equation
end
```

To solve the nonisothermal batch reactor problem stated above, you need to start by constructing the M-file that defines the right-hand side of the equation.

```
function fprime=myodef(t,v)
% defines right-hand-side of ODE
% output fprime is a row vector
z=zeros(1,2);
C=v(1);
T=v(2);
fprime(1)= - exp(-10./(T+273)).* C;
fprime(2)=1000*exp(-10./(T+273)).* C-10*(T-20);
```

Calling the solver with $h = 0.01 \Rightarrow$, or 600 steps,

```
[T,Y]=myeulers(@myodef,[0,6],[1,25],600)
```

you get the solution shown in Fig. 7.7.

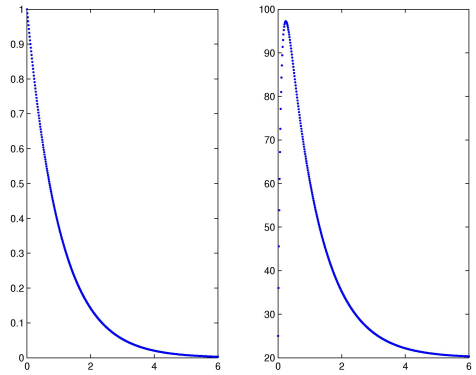

Figure 7.7: *The nonisothermal batch reactor problem. Concentration and temperature as functions of time. The system was solved with Euler's method using 600 steps.*

Effect of the Step Size Choice on Euler's Method

If instead of 600 steps you try to solve the system using only 30 steps or $h = 0.2$,

```
[T,Y]=myeulers(@myodef,[0,6],[1,25],30)
```

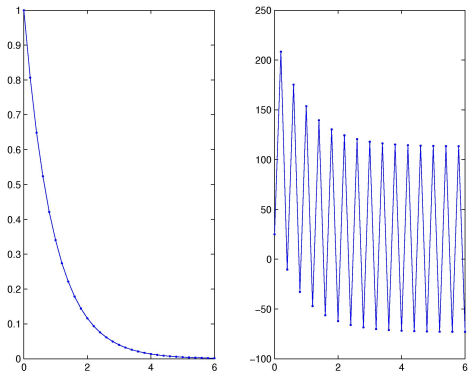

Figure 7.8: *The nonisothermal batch reactor problem. Unstable solution with Euler's method when solved with 30 steps.*

The solution is *unstable* as shown in Fig. 7.8.

7.7 Stiff systems

A *stiff* system involves rapidly changing components together with slowly changing ones. The rapidly carrying components are transients, and soon the slowly varying components dominate the solution. When the solution is rapidly changing, the step size needs to be small, but when the solution varies slowly, a larger step size should suffice. Nevertheless, for many numerical methods the transients dictate the time step for the entire solution. The step size is forced to be very small throughout the entire interval and not just where it is needed. If the step size is increased during the non-transient phase, the solution becomes *unstable* (see Fig. 7.8). Special methods can be used to solve these problems. These methods can solve stiff problems without having to make the step size very small throughout the entire interval.

The MATLAB `ode45` solver is based on a method that is not adequate for stiff problems. For stiff problems you can use `ode15s`. The syntaxis for this solver is identical to that for `ode45`. The nonisothermal batch reactor problem is stiff and can be used as an example of how the two MATLAB solvers behave. If you call ode15s on a large interval,

```
[t,Y]=ode15s(@myodef,[0,10080],[1,25])
```

the solver takes 97 steps to solve the problem. Using `ode45` with the same parameters takes 121508 steps.

Consider a 30-day development of the system of reactors 7.5. This is also a stiff problem. To solve the problem in a stable way, the non-adaptive Euler's method needs about 11000 steps. Euler's method is not appropriate for stiff problems. The adaptive methods used in ode45 and ode15s need 11084 and 77 steps respectively.

There is no easy way to know if a problem is stiff or not before solving it. The usual strategy in MATLAB is to try to solve an IVP with `ode45`. If the solution takes too long to complete, change the solver to `ode15s`. Non-stiff solvers like `ode45` are generally faster and more accurate than stiff solvers like `ode15s`. For a small IVP, it pays off to choose a stiff solver anyway.

7.8 Exercise

A biological process involves the growth of biomass from substrate. The material balances on this batch process yield

$$\begin{aligned} \frac{dB}{dt} &= \frac{kBS}{(K+S)} \\ \frac{dS}{dt} &= -\frac{0.75kBS}{(K+S)} \end{aligned}$$

where B and S are the respective biomass and substrate concentrations. The reaction kinetics are such that $k = 0.3$ and $K = 10^{-6}$ in consistent units. Solve this set of differential equations starting at $t_0 = 0$, when $S = 5$ and $B = 0.05$, to a final time given by $T_f = 20$. Assume consistent units. Plot S and B vs. time. After what time does the reaction stop?

Correct result: the reaction stops at $t \approx 16.3033$

Bibliography

[1] Cavers, I. "An Introductory Guide to MATLAB." Department of Computer Science, University of British Columbia. 4 Dec 1998. 26 Feb 2009. <**http://www.cs.ubc.ca/spider/cavers/MatlabGuide/guide.html**>.

[2] Chapra, S.C. **Applied Numerical Methods with MATLAB,** 2nd ed., McGraw-Hill, Singapore, 2008.

[3] Chapra, S. C. and R.P. Canale. **Numerical Methods for Engineers,** 4th ed., McGraw-Hill, New York, 1986.

[4] Cutlip, M. B. and M. Shacham. **Problem Solving in Chemical and Biochemical Engineering with POLYMATH^{TM}, Excel, and MATLAB®,** 2nd ed., Prentice Hall PTR, 2008.

[5] Naoum, A. and Führer, C. **Introduction to Chemical Engineering and Biological Technology: Computational Methods,** Matematikcentrum, LTH, Lund, 2003.

[6] Grimsberg, M.. **Börja med Matlab,** 2:a upplagan, Inst för Kemiteknik, LTH, Lund, 2006.

[7] "MATLAB Tutorial." Mathworks. 26 Feb 2009 <**http://www.mathworks.com/academia/student_center/tutorials/launchpad.html**>.

[8] Perry, R.H. and Green, D.W. **Perry's Chemical Engineers' Handbook,** 7th Edition, McGraw-Hill., 1997.

[9] Rao, S. S. **Applied Numerical Methods for Engineers ans Scientists,** Prentice Hall, 2002.

[10] "Matlab Tutorial." 22 Oct 2005. Department of Mathematics, University of Utah. 26 Feb 2009 <**http://www.math.utah.edu/lab/ms/matlab/matlab.html**>.

Index