1
Function Root-Finding

Examples of this chapter can be found in the `Vol3_Chapter1` directory in the `WileyVol3.zip` file available at www.chem.polimi.it/homes/gbuzzi.

1.1
Introduction

This chapter deals with the problem of finding the value $t = t_s$ that zeroes a function in the one-dimensional space:

$$y(t) = 0 \tag{1.1}$$

We will, for instance, find the value of t_s that zeroes the function

$$y(t) = t^3 - 2t - 5 \tag{1.2}$$

in the interval $[t_A = 2, t_B = 3]$.

Only real problems involving the variable $t \in R$ are considered.

Later, we describe the main iterative methods used to numerically solve this problem: these are referred to as *iterative* because they provide a series of values $\{t_1, t_2, \ldots, t_i\}$ such that the value of t_i for a sufficiently large i approximates the solution t_s with the required precision.

A series $\{t_1, t_2, \ldots, t_i\}$ *converges* to the solution t_s with the *order* $p \geq 1$ if

$$\frac{|t_{i+1} - t_s|}{|t_i - t_s|^p} \leq C, \quad i \geq i_0 \tag{1.3}$$

where C and i_0 are appropriately selected.

The function $y(t)$ could prove very complicated and it is not necessary to have it as an explicit algebraic function. For example, the problem might be to find either the value $t = t_s$ that zeroes the integral

$$y(t) = \int_o^t f(x)dx \tag{1.4}$$

Nonlinear Systems and Optimization for the Chemical Engineer: Solving Numerical Problems,
First Edition. Guido Buzzi-Ferraris and Flavio Manenti.

with $t_s \neq 0$ or the value $t = t_s$ for which a combination of the elements of the vector $\mathbf{y}$ is zero during the integration of a differential system

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t), \quad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{1.5}$$

When the function is particularly simple, however, certain special algorithms can be used.

Only general algorithms are proposed here and the potential peculiarities of the function are not exploited. This means, for instance, that special methods designed to find the roots of a n-degree polynomial are not considered.

It is important to distinguish between the following three situations:

1) The function assumes values with opposite signs at the interval boundaries $[t_A, t_B]$. Furthermore, the function is continuous within the interval where a single solution exists or, if there is more than one solution, the program need find only one of them.

The interval $[t_A, t_B]$, in which function continuity and opposite signs at the boundaries are both present, is called the *interval of uncertainty*.

2) The interval of uncertainty is unknown but the function is monotone.
3) Neither the interval of uncertainty nor the function monotonicity is known. The function may have several solutions with the objective being to find all of them. Alternatively, it may have some discontinuities or may be zero in several points without changing the sign locally.

When zeroing a function, there is a considerable difference between the first two situations and the third one.

This chapter introduces the algorithms that solve the first two families of problems. The third family of problems, however, demands more robust algorithms (see Chapter 6).

When the *interval of uncertainty* $[t_A, t_B]$ for a function $y(t)$ is known or the function is monotone, special algorithms can be used to guarantee the following appealing features:

1) A solution with a *finite number* of calculations.
2) A solution with a *predictable number* of calculations for every required accuracy.
3) High performance.

It is also necessary to distinguish between two different situations in this discussion:

- The computer is either a monoprocessor or parallel computing is unsuitable or cannot be exploited. Specifically, it is unsuitable when the function is very

simple and it cannot be exploited when root-finding is part of a calculation in which parallel computations have already been used.
- It is possible – and helpful – to exploit parallel computing for function root-finding.

The algorithms used to find roots of functions can be grouped into three families:

1) Substitution algorithms.
2) Bolzano algorithms.
3) Algorithms that approximate the function with a simpler function.

Analyzing simple problems and algorithms like this is particularly instructive as it illustrates the various ways of solving numerical problems: by using manual methods and classical analysis (without *round-off errors*) or, alternatively, a computer program (in which *round-off errors* play an important role) (Vol. 1 – Buzzi-Ferraris and Manenti, 2010a).

Anyone working on numerical problems may benefit from writing their own general root-finding function.

In the following, we will refer to the function evaluated in t_i ($y(t_i)$) simply as y_i. We will also refer to the derivative $y'(t_i)$ as y'_i.

Before proceeding, however, we feel it is useful to reiterate an important statement from Vol. 1 (Buzzi-Ferraris and Manenti, 2010a).

When a problem is numerically solved on a computer, the *round-off errors* may become so relevant that even certain modifications to traditional classical analysis thought appear necessary.

Some methods for the collocation of a new point t_{i+1} given a series of values $\{t_1, t_2, \ldots, t_i\}$ demand function monotonicity.

A function, which is theoretically monotone in classical analysis (without round-off errors), can be constant in numerical analysis with respect to two points, if they are too close.

For instance, given the function

$$y(t) = \cos(t) \tag{1.6}$$

and two points $t_1 = 1.000000000000000\text{e-}003$ and $t_2 = 1.000000000005000\text{e-}003$, the function value is the same in double precision because of the round-off errors.

The *resolution*, δ, represents the minimum distance required to ensure that a theoretically monotone function is also numerically monotone.

It is interesting to estimate the order of magnitude of δ for the root-finding problem. Let us suppose that the function $y(t)$ can be expanded in Taylor series in

the neighborhood of the solution, t_s:

$$y(t) \approx y_s + y'_s(t - t_s) = y_s + y'_s\delta \tag{1.7}$$

The term $y'_s\delta$ is not negligible with respect to y_s when

$$|y'_s\delta| \geq \varepsilon|y_s| \tag{1.8}$$

In these relations, ε is the macheps of the calculation.
From (1.8), we obtain

$$|\delta| \geq \varepsilon\frac{|y_s|}{|y'_s| + \alpha} \tag{1.9}$$

The *root* of a function can be commonly found with a precision in the order of the macheps.

The previous discussion is valid only if $y'_s \neq 0$; otherwise, the function $y(t)$ intersects the axis of abscissa horizontally and the points must be at least at a distance

$$|\delta| \geq C\sqrt{\varepsilon} \tag{1.10}$$

When $y'_s = 0$ at the solution, the precision of a numerical method is proportional to $\sqrt{\varepsilon}$ rather than to ε. In double precision on 32 bit computers, the error is on the 6–7th significant digit rather than the 14–15th.

1.2 Substitution Algorithms

These algorithms transform the problem to produce expressions such as

$$t = g(t) \tag{1.11}$$

so as to exploit the iterative formulation:

$$t_{i+1} = g(t_i) \tag{1.12}$$

For example, given the function

$$y(t) = t^3 - 2t - 5 = 0 \tag{1.13}$$

we have

$$t = t^3 - t - 5 \tag{1.14}$$

$$t = \frac{2}{t} + \frac{5}{t^2} \tag{1.15}$$

$$t = \frac{t^3 - 5}{2} \tag{1.16}$$

$$t = t - \frac{t^3 - 2t - 5}{10} \tag{1.17}$$

Starting from an initial point $t_0 = 2$, the following series are obtained in correspondence with the functions (1.14)–(1.17), respectively:

- $2, 1, -5, -125, -1.95 \times 10^6$: the series diverges.
- $2, 2.25, 1.876543, 2.485674, 1.6113859, 3.158989, 1.134155$: the series diverges.
- $2, -0.8125, -2.768188, -13.1061, -112.8$: the series diverges.
- $2, 2.1, 2.0939, 2.094627, 2.094543, 2.094553, 2.094551, 2.094552$: the series properly converges to the problem solution.

This approach was often adopted when the calculations were manual as no general programs were available to solve the problem in its *original form*.

However, this kind of procedure also has numerous disadvantages.

The iterative procedure can also *diverge* with very simple functions; for example, it can diverge with a linear equation.

If the iterative formula is linear

$$t_{i+1} = m \cdot t_i + a \tag{1.18}$$

the method converges only if

$$|m| < 1 \tag{1.19}$$

It is useful to plot the two trends to illustrate the reason for this limitation:

$$y = t \tag{1.20}$$

$$y = m \cdot t + a \tag{1.21}$$

and proceed with the iterations. Note that the method diverges if equation (1.21) has a slope $m > 1$ and $m < -1$ (Figure 1.1).

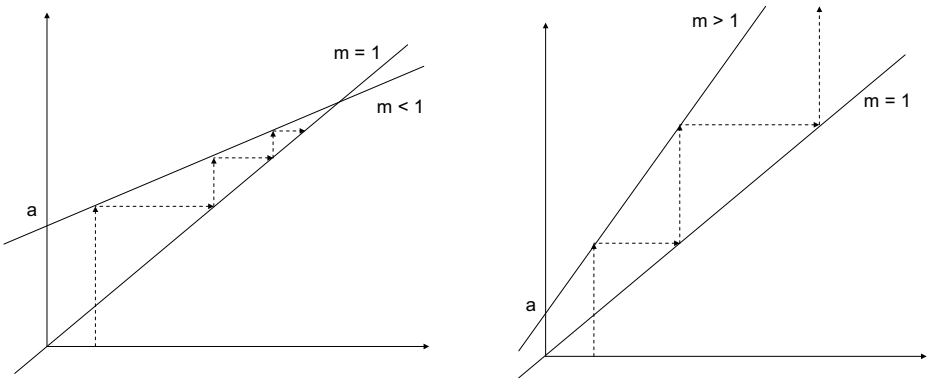

Figure 1.1 Convergence conditions of substitution algorithms.

When the procedure converges, the *convergence is usually slow.*

For example, the problem mentioned above

$$y(t) = t^3 - 2t - 5 = 0 \tag{1.22}$$

is solved using Newton's method, which has quadratic convergence, yielding the following series results: $2, 2.1, 2.094568, 2.094552$.

If the substitution method converges, the convergence speed can be improved using the Aitken method (Buzzi-Ferraris and Manenti, 2010a).

Given a linearly convergent series of t_i, the Aitken formula generates a new and more efficient series z_n:

$$z_n = t_i - \frac{(t_i - t_{i-1})^2}{(t_i - t_{i-1}) - (t_{i-1} - t_{i-2})} \tag{1.23}$$

For example, given the series

$$t_{i+1} = t_i - \left(t_i^3 - 2t_i - 5\right)/20 \tag{1.24}$$

linearly converging to the value $t_s = 2.094552$, the series from Table 1.1 is obtained.

A laborious preliminary study is required to convert the problem into a convergent form.

The substitution method is only useful from an educational point of view.

Table 1.1 Series for the iterative method.

t_i	z_n	y_m
2.000 000		
2.050 000		
2.074 244	2.097 064	
2.085 448	2.095 074	
2.090 502	2.094 657	2.094 546
2.092 757	2.094 573	2.094 552
2.094 200	2.094 555	
2.094 396	2.094 553	
2.094 483	2.094 552	
2.094 521		
2.094 538		
2.094 546		
2.094 549		
2.094 551		
2.094 552		

When such calculations were performed manually, a lot of time had to be devoted to search for a *special* method for solving a *specific* problem. We encountered several disadvantages to retaining this approach when developing computer programs:

1) Intrinsically similar problems (e.g., root-finding for different functions) are not solved by the same calculation procedure.
2) Techniques adopted to solve the problem are inextricably linked to the formulation of the problem itself: there is no separation between the *formulation* of the problem and the *numerical method used to solve it*.

1.3 Bolzano's Algorithm

The following conditions must be satisfied to use Bolzano's method.

1) The function $y(t)$ is continuous in the interval $[t_A, t_B]$. Derivability and derivative continuity are not required; function monotonicity is also unnecessary.
2) The function has values with opposite signs at the boundaries t_A and t_B.

If the previous conditions are both satisfied, Bolzano's theorem has demonstrated the existence of at least one solution within the interval $[t_A, t_B]$.

From a numerical point of view, if a single processor is available, the Bolzano method is equivalent to the bisection method.

The algorithm is very simple: the function is evaluated at the central point:

$$t_M = t_A + \frac{t_B - t_A}{2} \tag{1.25}$$

if y_M has the same sign of y_A, t_M replaces the original boundary t_A; otherwise t_M replaces t_B. This procedure allows the original interval to be iteratively halved, preserving the two necessary conditions as above.

Let us denote the width of the initial interval $L_1 = t_B - t_A$. As the interval is halved at each iteration, after n iterations, we have

$$L_n = \frac{L_1}{2^n} \tag{1.26}$$

From (1.26), it is possible either to evaluate a priori the interval of uncertainty after a predetermined amount n of iterations or, given the final interval, to know a priori the number of iterations required to reach it.

Let us select a point different from the center of the interval. If we are lucky, the solution is in the smallest of the two new subintervals; otherwise, the solution is in

the largest interval. Consequently, the maximum interval of uncertainty is larger than Bolzano's interval of uncertainty at each iteration.

In the case of a single processor, Bolzano's method minimizes the maximum interval of uncertainty.

If several processors are available simultaneously, Bolzano's method can be generalized: the function should be simultaneously evaluated in an evenly spaced number of points equal to the number of available processors and within the interval of uncertainty.

Equally, in the general form available for several processors, Bolzano's method retains the advantages of the corresponding method for single-processor machines.

1) The method always converges to a solution.
2) The number of iterations is finite and known a priori for an assigned precision.
3) The method minimizes the maximum interval of uncertainty.

However, it also has one major deficiency:

Bolzano's algorithms do not exploit the peculiarities of the function. While this is an advantage for complicated functions (e.g., nonmonotone or multimodal), it may prove a handicap in other situations where the function is smooth.

For example, if we apply Bolzano's method to the linear function

$$y(t) = 3t - 5 \tag{1.27}$$

or to any other function, the same number of iterations is required.

1.4 Function Approximation

The key point on which this family of algorithms is based is the approximation of the function through a simpler formulation that makes searching for the root easier; the function root is then adopted as the initial guess for the successive iteration.

This kind of algorithm can be implemented using two different strategies.

1) One can use an iterative procedure without controlling the interval of uncertainty.
2) If the interval of uncertainty $[t_A, t_B]$, where the function assumes opposite signs at the boundaries, is known, the intervals that follow will retain this property.

The first strategy is the only one of any use when the *interval of uncertainty* is unknown. If the function is monotone, it is always possible to find an interval of uncertainty and, consequently, to use *ad hoc* techniques to find its root.

Algorithms for function root-finding with a known interval of uncertainty are discussed below, whereas the other situation is considered in Section 1.6.

1.4.1 Newton's Method

If the function $y(t)$ and its derivatives are continuous, the same function can be expanded in Taylor series in the neighborhood of t_i:

$$y(t) = y_i + y_i'(t - t_i) + \frac{y_i''}{2}(t - t_i)^2 + \cdots \tag{1.28}$$

If we stop the Taylor series expansion at the first-degree term, we obtain the equation of the tangent to the curve in t_i, y_i:

$$T(t) = y_i + y_i'(t - t_i) \tag{1.29}$$

The point t_{i+1}, where the tangent intersects the axis of abscissas, can be obtained by imposing the condition

$$T(t_{i+1}) = 0 \tag{1.30}$$

thus

$$t_{i+1} = t_i - \frac{y_i}{y_i'} \tag{1.31}$$

Given an initial guess t_i and evaluating the function y_i and its first derivative y_i' in correspondence with this point, it is possible to get a value for the new point t_{i+1} using (1.31).

For example, to evaluate the square root of 2 using Newton's method, the root of the function

$$y(t) = t^2 - 2 = 0 \tag{1.32}$$

needs to be found. Relation (1.31) becomes

$$t_{i+1} = t_i - \frac{t_i^2 - 2}{2t_i} = \frac{1}{2}\left(t_i + \frac{2}{t_i}\right) \tag{1.33}$$

By choosing $t_0 = 1$ as the initial guess, the series $t_1 = 1.5$, $t_2 = 1.416667$, $t_3 = 1.414216$, and $t_4 = 1.414214$ is obtained, and the final value is already a good approximation of the solution.

If the point t_i is close to the solution t_s and the function can be expanded using a Taylor series

$$y_s = y_i + y_i'(t_s - t_i) + \frac{y''(\xi)}{2}(t_s - t_i)^2 = 0 \tag{1.34}$$

Dividing by y_i'

$$t_s - \left(t_i - \frac{y_i}{y_i'}\right) = t_s - t_{i+1} = -\frac{y''(\xi)}{2y_i'}(t_s - t_i)^2 \tag{1.35}$$

If the method converges

$$|t_{i+1} - t_s| \approx c|t_i - t_s|^2 \tag{1.36}$$

Newton's method has quadratic convergence.

This method too, however, has certain disadvantages.

1) Convergence is not guaranteed also if the function is monotone or if we know the interval of uncertainty (unless opportune modifications are adopted in the latter case).
2) The function $y(t)$ and its first derivative should be continuous within the interval. The function should also be monotone.
3) The first derivatives of the function must be calculated analytically. This can not only be computationally intensive but also a source of errors if the function is particularly complex.
4) The first derivatives of the function should not be calculated numerically as, in this instance, the method is less efficient than the secant method (see Section 1.4.2).
5) When the first derivative is zero, problems arise as the theoretical basis of the model is no longer valid. The convergence speed is no longer quadratic, for instance.

Consider, for example, the function

$$y = \frac{t}{1-t} \tag{1.37}$$

which is equal to zero in $t = 0$. Starting from $t = -2$, the series $4, 16, 256, 6.6 \times 10^5, 4.3 \times 10^9, \ldots$ is obtained and the method diverges.

The divergence of Newton's method with a monotone function is reported in Figure 1.2.

The advantage of Newton's method is in the fast convergence of the cases in which it effectively converges. Convergence is quadratic in these cases.

Newton's method is never used in the ***BzzMath*** library classes dedicated to root-finding.

1.4.2
The Secant Method

In relation (1.31), by approximating the derivative y_i' by the line joining the points y_i, t_i and y_{i-1}, t_{i-1}, the following relation takes place:

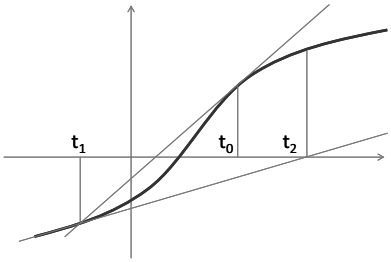

Figure 1.2 Divergence of Newton's method with monotone function.

$$t_{i+1} = t_i - \frac{y_i}{(y_i - y_{i-1})/(t_i - t_{i-1})} \tag{1.38}$$

Two points t_0 and t_1 are needed to initialize the algorithm.

Let us consider again the previous example of evaluating the square root of 2, which corresponds to zero in equation (1.32).

Let $t_0 = 1$ and $t_1 = 2$ be the initial guesses. The series is $t_3 = 1.333333$, $t_4 = 1.400000$, $t_5 = 1.414634$, $t_6 = 1.414211$, and $t_7 = 1.414214$.

The secant method has a convergence speed raised to the power of 1.618. It is slower than Newton's method, but the first derivative does not need to be evaluated. When the computational effort involved in evaluating the derivative is in the order of the computational time required to calculate the function, the secant method performs better numerically than Newton's method.

The secant method has the same pros and cons as Newton's method, except for the need to provide the analytical expression of the first derivative of the function.

The secant method in its pure iterative form is never used in the classes of the ***BzzMath*** library dedicated to root-finding.

1.4.3 Regula Falsi Method

The regula falsi algorithm is very similar to the previous one. The difference is in the support points adopted to linearize the function: the last two values at each iteration are used in the secant method, whereas the boundaries of the interval of uncertainty are adopted in the regula falsi method.

The advantage this has over the previous criteria is that the new iteration falls within the interval of uncertainty and thus method convergence is ensured.

The new prevision replaces the bound where the function assumes the same sign.

Its main disadvantage is that its convergence speed is slower than the secant method.

To calculate the square root of 2 using this method

$$y(t) = t^2 - 2 = 0 \tag{1.39}$$

and with $t_0 = 1$ and $t_1 = 2$ as initial guesses, we obtain the series $t_3 = 1.333333$, $t_4 = 1.400000$, $t_5 = 1.411765$, $t_6 = 1.413793$, $t_7 = 1.414141$, $t_8 = 1,414201$, $t_9 = 1.414421$, $t_{10} = 1.414213$, and $t_{11} = 1.414214$.

This algorithm too is important only from an educational point of view, as it is the simplest algorithm in this family, ensuring convergence when the interval of uncertainty is known.

The efficiency of the regula falsi method can be improved by using the two best values from the previous iterations rather than the boundary values for the linear approximation. This device is implemented by checking the interval of uncertainty.

This device has the advantage of a better convergence speed, similar to the one of secant method. Many programs adopt this strategy as a basic algorithm (often combined with Bolzano's method to guarantee convergence even with complex and/or nonmonotone functions).

Other authors attempt to improve the efficiency of the regula falsi method with another device. Typically the method would not work satisfactorily when the function has very different absolute values at the boundaries of the interval. If we select these boundaries as support points for linearization, we can guarantee convergence to the solution, but the interval reduction might be very small. The device consists of the selection of a smaller value (i.e., dividing by 2) in correspondence with the boundary where the function has its maximum absolute value. A reduction of this kind in the ordinate can also be proposed in successive iterations.

The regula falsi method using the two best values of the previous iterations rather than the boundary values for the linear approximation is implemented in the classes of the ***BzzMath*** library dedicated to root-finding.

1.4.4
Muller's Method or Parabolic Interpolation

This method approximates the function with a parabola. Contrary to the previous methods, three points are needed to estimate the parameters of the parabola.

The algorithm has the following pros and cons.

1) It is efficient at finding the roots of a polynomial with degree higher than 3.
2) It allows real and complex roots to be found.

As the method approximates the function using a parabola, it is necessary to choose a solution from the two possible predictions.

Except in the case of polynomial root-finding and complex roots, there are more efficient and easier-to-implement algorithms.

The parabolic interpolation method is never used in the ***BzzMath*** library classes dedicated to root-finding.

1.4.5 Hyperbolic Interpolation Method

The function can be approximated using the simplest rational function: a hyperbole:

$$y(t) = \frac{a_1 - a_2 t}{1 + a_3 t} \tag{1.40}$$

Given three support points, the three parameters a_1, a_2, and a_3 can be calculated by solving the linear system:

$$\begin{cases} a_1 - a_2 t_1 = y_1(1 + a_3 t_1) \\ a_1 - a_2 t_2 = y_2(1 + a_3 t_2) \\ a_1 - a_2 t_3 = y_3(1 + a_3 t_3) \end{cases} \tag{1.41}$$

Unlike Muller's method, this algorithm has a single solution:

$$t_{i+1} = \frac{a_1}{a_2} = \frac{t_1 y_2 y_3 (t_2 - t_3) - t_2 y_1 y_3 (t_1 - t_3) + t_3 y_1 y_2 (t_1 - t_2)}{y_2 y_3 (t_2 - t_3) - y_1 y_3 (t_1 - t_3) + y_1 y_2 (t_1 - t_2)} \tag{1.42}$$

Moreover, an interpolation with a rational function is usually preferable to using a polynomial function (Vol. 2 – Buzzi-Ferraris and Manenti, 2010b).

The hyperbolic interpolation method is used when the root-finding classes from the ***BzzMath*** library have performed three iterations.

1.4.6 Inverse Polynomial Interpolation Method

Muller's method has to select the new point between the two solutions of the interpolating parabola. This difficulty is enhanced by interpolating the function with a higher degree polynomial.

However, it is possible to overcome it by adopting an inverse interpolation. In other words, a polynomial in the dependent variable y can be developed.

Given $n + 1$ support points, it is possible to calculate the polynomial parameters, approximating t with respect to y:

$$P_n(y) = a_0 + a_1 y + a_2 y^2 + \cdots + a_n y^n \tag{1.43}$$

and estimate the polynomial prediction in $y = 0$.

Since the polynomial is useful to predict the point $t = t_{i+1}$ where $y = 0$, the Neville method, which does not require any parameter estimation, can be adopted (see Buzzi-Ferraris and Manenti, 2010b).

The inverse polynomial interpolation method is never used in the ***BzzMath*** library classes dedicated to root-finding.

1.4.7
Inverse Rational Interpolation Method

An exact polynomial interpolation can deteriorate as the polynomial degree increases (Buzzi-Ferraris and Manenti, 2010b), whereas this problem does not arise with rational functions.

An inverse interpolation can be effectively exploited using a rational function rather than a polynomial and the Bulirsch–Stoer algorithm rather than the Neville method.

It is worth remarking that this method cannot be implemented on its own in a program, rather certain additional (and less efficient) methods must be integrated with it to ensure convergence. These additional algorithms should be used whenever the inverse rational function interpolation encounters any difficulty.

Inverse interpolation with rational functions is very efficient and is the ideal basis for the development of a general function root-finding program, even though it is slightly more complex than the other algorithms.

In the `BzzInterpolation` class, the `Neville` function allows an interpolation with polynomials to be performed, whereas `BulirschStoer` performs an interpolation with rational functions.

These functions also provide an estimation of the improvement obtained by increasing the order of the interpolating polynomial.

Example 1.1

In the root-finding of the function

$$y(t) = \frac{0.1}{t\sqrt{5t + 1}} - 2.2822 \tag{1.44}$$

the first five iterations return the following values:

$$\mathbf{y} = \begin{Bmatrix} 5.101153572647464 \\ -0.5858531627256769 \\ 1.458676912555765 \\ 0.5652831530950460 \\ -0.2310321319973876 \end{Bmatrix}$$

in

$$\mathbf{t} = \begin{Bmatrix} 0.013120451134123 \\ 0.052468291627363 \\ 0.025192134726134 \\ 0.032567142617336 \\ 0.044127138234521 \end{Bmatrix}$$

Calculate the prediction of function zeroing when an inverse polynomial interpolation and an inverse rational interpolation are used. To perform an inverse interpolation is sufficient to invert the independent and dependent variables **t** and **y**. In other words, the objective is to find the value of **t** such that $\mathbf{y} = 0$.

The program is

```
#define BZZ_COMPILER 0 // default
#include "BzzMath.hpp"
void main(void)
   {
   int i;
   BzzVector x(5,.013120451134123,
         .052468291627363,.025192134726134,
         .032567142617336,.044127138234521);
 BzzVector y(5,5.101153572647464e+000,
         -5.858531627256769e-001,
          1.458676912555765e+000,
          5.652831530950460e-001,
         -2.310321319973876e-001);
 BzzInterpolation p(y,x); //y as independent
 BzzVector prev;
 prev = p.Neville(0.); // x for y = 0
 BzzPrint("\nNeville Previsions");
 BzzPrint("\n%d %25.14e",1,prev[1]);
 for(i = 2;i <= 5;i++)
      BzzPrint("\n%d %25.14e %25.14e",
        i,prev[i],prev[i] - prev[i - 1]);
 BzzPrint("\nBulirschStoer Previsions");
 prev = p.BulirschStoer(0.); // x for y = 0
```

```
    BzzPrint("\n%d %25.14e",1,prev[1]);
    for(i = 2;i <= 5;i++)
       BzzPrint("\n%d %25.14e %25.14e",
             i,prev[i],prev[i] - prev[i - 1]);
    }
```

The results are

```
Neville Previsions
1     4.41271382345210e-002
2     4.07732776678377e-002   -3.35386056668333e-003
3     3.97532091300153e-002   -1.02006853782238e-003
4     3.99068238087031e-002    1.53614678687811e-004
5     3.99365948649282e-002    2.97710562250988e-005

BulirschStoer Previsions
1     4.41271382345210e-002
2     4.00070899138442e-002   -4.12004832067678e-003
3     3.99976870459859e-002   -9.40286785836714e-006
4     3.99996250073634e-002    1.93796137758218e-006
5     3.99996279136990e-002    2.90633557498143e-009
```

The solution is `3.999963311389726e-002`. The error in the prediction obtained using all the five points is 0.16% with Neville and 0.000013% with Bulirsch–Stoer.

The inverse rational interpolation method is used as the basic method when more than three iterations of the ***BzzMath*** classes for function root-finding have been performed.

1.5
Use of a Multiprocessor Machine with a Known Interval of Uncertainty

If an interval of uncertainty is known and some processors are available to find the root, Bolzano's method can effectively be coupled with a function approximation method.

There are various strategies when more than a single processor, $n_P > 1$, are available. The following is the one adopted in the ***BzzMath*** library.

Suppose we have n_P available processors. The first one is used to calculate the function in the point selected by the efficient method being used, whereas the remaining $n_P - 1$ processors evaluate the function in the points opportunely placed in the two new subintervals, created by the splitting of the interval of uncertainty.

1.6 Search for an Interval of Uncertainty

The search for an interval of uncertainty is very simple if it is known a priori that the function is monotone. Specific techniques for the most general case are explained in Chapter 6.

Heuristic techniques or function approximations (i.e., the secant method) can be exploited when the function is monotone.

One widely adopted technique is to move in the steepest descent direction with a step dt, which progressively increases (e.g., with $dt_{i+1} = 2 \cdot dt_i$), until an interval of uncertainty is found.

If necessary, the prediction provided by the secant method can also be exploited on the condition that said prediction is checked: it should be in the order of the current value of dt.

When the interval of uncertainty is identified, a dedicated algorithm capable of exploiting it can be adopted.

Suppose we have n_P available processors: it is useful to place n_P points at a distance that gradually increases up to the detection of a proper interval of uncertainty.

It is worth remarking that, if the function is monotone, it is possible to find a rough interval of uncertainty and, once detected, the solution can be obtained using very efficient methods, although the interval of uncertainty is wide.

1.7 Stop Criteria

In all problems solved using iterative methods, it is important to define a criterion to stop the evaluations either when a reasonable approximation of the solution is achieved or when convergence is impossible.

There are various possibilities where the specific problem of function root-finding is concerned and it is advisable for us to analyze some of them here.

1) A first very simple test (which *must* be introduced in each program involving iterative algorithms) is to check the maximum number of iterations.

   

   When an iterative method is adopted, there is rarely a guarantee of finding the solution. The introduction of an upper bound on the iterations number prevents an endless calculation loop.

2) One reasonable way of testing is to check the variation of the variable t between successive iterations. If

$$|t_{i+1} - t_i| < \varepsilon_t \tag{1.45}$$

   the calculation is stopped.

This criterion is *very dangerous* if applied without precautions: in fact, it does not ensure you will be close to the solution; as it is conceived, it simply denotes that the selected algorithm is unable to modify the value of t. Thus, it does not necessarily mean that the solution is achieved.

Furthermore, a series from the t_i may diverge even though the decrease in the difference between successive terms is monotone. Again, it could happen that the series converges to a specific value of t, which is *not* a solution of the problem, even if it leads to the function decrease.

Let us, for example, consider the following algorithm:

$$t_{i+1} = \frac{1 + t_i}{2} \tag{1.46}$$

with an initial guess $t_0 = 2$.
The series is equal to

$$t_i = 1 + 2^{-i} \tag{1.47}$$

and it converges to $t = 1$. If such a sequence is applied to the equation $y(t) = t^2 = 0$, the function y_i monotonically decreases, while the difference $t_{i+1} - t_i$ approaches zero, but no function root corresponds to $t = 1$.

The test is meaningful only if it is applied to an algorithm where the relation (1.45) represents a necessary condition to achieve the solution.

In Newton's method, the variation between successive iterations is provided by

$$\delta_i = -\frac{y_i}{y_i'} \tag{1.48}$$

If $y_i' \neq 0$, the value of $|\delta_i|$ should be equal to zero if t_i is a solution to the problem.

Therefore, using Newton's method, the difference $|t_{i+1} - t_i| = |\delta_i|$ may be used as an *estimation of the solution's accuracy* when $y_i' \neq 0$.

If at the ith iteration of Newton's method

$$|\delta_i| < \varepsilon_N \tag{1.49}$$

and $y_i' \neq 0$, there is a good probability that the solution is achieved with the given accuracy ε_N.
For example, let us consider the problem of finding the root in the interval $t_A = 1$ and $t_B = 2$ of the function

$$y(t) = t^3 - t - 1 = 0 \tag{1.50}$$

In the point $t_i = 1.3247182$, the function is $y_i = 1.192726 \times 10^{-6}$ and its derivative is $y_i' = 4.264635$.
Hence

$$|\delta_i| = 2.8 \times 10^{-7} \tag{1.51}$$

which can be considered as the estimation of the error of the solution t_i.

The same procedure is approximately valid using the secant rather than the derivative. In this case, δ_i can be estimated as

$$t_{i+1} - t_i = \delta_i = -\frac{y_i(t_i - t_{i-1})}{y_i - y_{i-1}} \tag{1.52}$$

This test is therefore suitable for the Newton and secant families of algorithms.

The test can also be adopted with different algorithms but we must change our original point of view: it can be adopted only if the iterations t_{i-1} and t_i are used to estimate the solution error.

For example, let us suppose that a specific algorithm used to find the roots of the function

$$y(t) = t^3 - t - 1 = 0 \tag{1.53}$$

leads to the following two iterations: $t_{i-1} = 1.3248182$ and $t_i = 1.3247182$, where the function $y_{i-1} = 4.277668 \times 10^{-4}$ and $y_i = 1.192726 \times 10^{-6}$.
It is possible to estimate the error using the two iterations, supposing that they come from the secant method. Through equation (1.52), it results in

$$|\delta_i| = \left|\frac{1.035262 \times 10^{-6}(1.3247182 - 1.3247282)}{1.035262 \times 10^{-6} - 4.368201 \times 10^{-5}}\right| = 2.427529 \times 10^{-7} \tag{1.54}$$

Therefore, the basic idea of this test viewed from a different perspective is that it *cannot be considered a convergence check* when two successive iterations are almost the same, rather *as a criterion to estimate the error* of the best iteration with respect to the solution.

For example, in the series $t_i = 1 + 2^{-i}$ above, the difference $t_{i+1} - t_i$ approaches to zero, but the value of δ_i for the function $y(t) = t^2 = 0$ calculated with equation (1.52) tends to 0.5. It is thus possible to detect the unsatisfactory accuracy of the solution.

In this context, the value of δ_i calculated using (1.52) assumes the meaning of an error estimation of the solution also for methods other than the secant method. The test $|t_{i+1} - t_i| < \varepsilon_t$ is therefore replaced with $|\delta_i| < \varepsilon_N$ independent of the method adopted.

Nevertheless, checking the error with its absolute value only is not the correct approach either as certain problems may arise.

For example, if the solution of a problem is $t_s = 10^5$ and one selects $\varepsilon_N = 10^{-7}$, the iterative method may be always unsatisfactory for the test.
It is preferable to use a relation such as

$$|\delta| = \left|\frac{|y_i(t_i - t_{i-1})|}{|y_i - y_{i-1}| + \alpha}\right| \leq \varepsilon(|t_i| + |t_{i-1}|) + \varepsilon_A \tag{1.55}$$

where ε_A is again related to the acceptable absolute error. ε is either related to computer accuracy (and therefore equal to the macheps or to its multiples) or

provided by the user. The small α value is necessary to avoid numerical problems.

This test must only be used if we know a priori that the function is monotone and at the solution $y'_s \neq 0$.

This test must be used only in programs that do not guarantee the solution will be kept within the interval of uncertainty. However, other very reliable tests will be covered later.

3) If an inverse interpolation method is used and the Bulirsch–Stoer algorithm in particular is applied to rational functions, the differences between the predictions of different rational functions can be calculated in order to check whether they tend to zero as well as to estimate the error arising in using the differences between the various previsions.

Using inverse approximation of rational functions has the additional advantage of calculating error estimation in this way while the function is zeroed.

This should not be considered as a convergence test, but merely useful information regarding the program's convergence speed.

4) Another possible test consists of controlling the absolute value of the function during the iterations. The iterations are stopped if the function's absolute value is smaller than an assigned value:

$$|y_i| < \varepsilon_2 \tag{1.56}$$

This criterion must be used with caution, however, as it can be overly restrictive in certain circumstances and overly optimistic in others.

By way of example, let us consider the function $y(t) = t^8$. In $t_i = 0.1$, the function is equal to 10^{-8}, but the solution is still distant.

Ideally, this criterion should either not be used or only be adopted optionally, specifically when the order of magnitude, for which it is reasonable to consider the problem solved, is known.

5) In the special case of a known interval of uncertainty, there is a test that provides a maximum guarantee of reliability. If the algorithm checks the interval of uncertainty $[t_A, t_B]$ and the best point between t_A and t_B is selected as the solution, the error on the solution is

$$\delta_{AB} = |t_B - t_A| \tag{1.57}$$

Analogous to (1.55), a similar test can be applied to this error estimation:

$$|\delta_{AB}| \leq \varepsilon_{Rel}(|t_A| + |t_B|) + \varepsilon_{Abs} \tag{1.58}$$

This is the criterion that ensures maximum reliability in deeming the accuracy of the solution achieved, when the *interval of uncertainty* is given.

1.8 Classes for Function Root-Finding

In the ***BzzMath*** library, the following classes are implemented for function root-finding:

```
BzzFunctionRoot
BzzFunctionRootMP
BzzFunctionRootRobust
```

The `BzzFunctionRoot` class is designed to search for the root of a function when an interval of uncertainty is given and when the user wants to find one solution within that interval.

The function must be continuous within the interval. The `BzzFunctionRoot` class will not benefit from several processors.

Conversely, if multiprocessor computers and openMP directives are used, the `BzzFunctionRootMP` class should be adopted under the same above-mentioned condition. It is worthwhile remarking that if we use objects from the `BzzFunctionRootMP` class, we must also use a compiler that supports openMP directives and activate them.

The `BzzFunctionRootRobust` class is designed to search for root functions in the following cases:

- When the interval of uncertainty is unknown and the function is not monotone.
- When the function has many roots and all of them need to be found.
- When the function is not evaluable in certain regions of the domain.
- When the function does not change sign in the neighborhood of certain roots.

Only the `BzzFunctionRoot` and `BzzFunctionRootMP` classes are described in this chapter, whereas the `BzzFunctionRootRobust` class is explained in Chapter 6.

The `BzzFunctionRoot` class has several constructors. The simplest ones are

```
BzzFunctionRoot z1;
BzzFunctionRoot z2(tA,tB,BzzFunctionRootExample);
BzzFunctionRoot z4(tA,tB,BzzFunctionRootExample,
      yA,yB);
```

The first is the default one and no data are needed. The second requires the boundaries t_A and t_B of the interval of uncertainty.

If the values of the function at the two boundaries have the same sign, the calculation is stopped immediately and a warning message is printed out.

The third constructor requires the function values at the interval boundaries, y_A and y_B.

As the object is defined, it is possible to use the operator `()` to initialize it either with different values for the interval boundaries and/or with a different function.

Example 1.2

Use the constructors from the `BzzFunctionRoot` class to zero the function

$$y(t) = t - \exp(-t) \tag{1.59}$$

within the interval $[t_A = 0.; t_B = 10.]$:

```
#define BZZ_COMPILER 0 // default Visual C++ 6
#include "BzzMath.hpp"
double BzzFunctionRootExample(double t);
void main(void)
   {
   double tA = 0.;
   double tB = 10.;
   BzzFunctionRoot z1; // default constructor
   z1(tA,tB,BzzFunctionRootExample);
   z1();
   BzzFunctionRoot z2(tA,tB,
      BzzFunctionRootExample);
   z2();
   double yA = BzzFunctionRootExample(tA);
   double yB = BzzFunctionRootExample(tB);
   BzzFunctionRoot z3; // default constructor
   z3(tA,tB,BzzFunctionRootExample,yA,yB);
   z3();
   BzzFunctionRoot z4(tA,tB,
      BzzFunctionRootExample,yA,yB);
   z4();
   BzzPrint("Results");
   z1.BzzPrint("z1");
   z2.BzzPrint("z2");
   z3.BzzPrint("z3");
   z4.BzzPrint("z4");
   }

double BzzFunctionRootExample(double t)
   {
   return t - exp(-t);
   }
```

The search for the solution is performed by the overloaded operator `()`. Two different versions of the overloaded operator `()` are implemented:

```
char operator()(void);
char operator()(int niter);
```

For example

```
BzzFunctionRoot z(tA,tB,FunZero);
z();
```

Using this operator, the search proceeds until the required accuracy is obtained. On the other hand, using the operator

```
z(niter);
```

the root-finding process is stopped when the number of calculations is equal to `niter`, which is an `int` that limits the number of calculations. In this instance, it is worth stressing the following feature.

If the function is iteratively called, the object accounts for the previous calculations and executes the calculations as if an overall single call were carried out.

In other words, invoking $z(6)$ or six times $z(1)$ is the same from a numerical point of view. This property allows the user to manage the object and check the root-finding of the same while the search is proceeding without compromising efficiency. Moreover, it should be stressed that the object manages *its own* data, which are masked to the user inside the operator `()` and therefore there is no risk of modifying them unintentionally, for example, by using several objects simultaneously. As the object is aware of previous calculations, it does not execute any new calculations when it has solved the problem with the required accuracy even though it is requested to.

It is possible to adjust the accuracy of the root-finding process using the following functions: `SetTolRel`, `SetTolAbs`, and `SetTolY`. The former two functions allow the default value to be modified in (1.58). The default values are $\varepsilon_{Rel} = 1 \times 10^{-15}$ and $\varepsilon_{Abs} = 1 \times 10^{-30}$, respectively. The function `SetTolY` allows the value for test (1.56) to be modified; the default value is 1×10^{-30}.

For example, the tolerances of the object `z` are changed in the following code sample:

```
z.SetTolAbs(1.e-10);
z.SetTolRel(1.e-5);
z.SetTolY(1.e-15);
```

If the function is iteratively called, the tolerances can be modified anywhere. For example, accuracy can be progressively increased. The object preserves its history in this case too and exploits it as much as possible also.

Example 1.3

The present example shows how to use the object `z(1)` iteratively until the required precision is achieved.

```
#include "BzzMath.hpp"
double BzzFunctionRootExample(double t);
void main(void)
   {
   BzzFunctionRoot z(0.,10.,BzzFunctionRootExample);
   char control;
   for(int i = 1; i <= 100; i++)
      {
      control = z(1);
      if(control != 1)break;
      // user controls
      }
   z.BzzPrint("\nResults");
   }

double BzzFunctionRootExample(double t)
   {
   return t - exp(-t);
   }
```

In the operator `()`, the `return` is a `char` and indicates the reason for which the search was stopped. It can assume the following values:

- `-2`: the values of the function at the two boundaries have the same sign.
- `-1`: the program is stopped by the user through the global variable `bzzStop = 1`.
- `0`: the program encountered certain numerical problems or niter is less than 1.
- `1`: the number of iterations is equal to niter in this specific call of the function.
- `2`: the test on the interval of uncertainty (1.58) is satisfactory:

```
fabs(tRight-tLeft) <= tTolAbs + tTolRel*(fabs(tLeft)+fabs
(tRight));
```

- `3`: the absolute value of the function is smaller than `yTolAbs`.

The user can use the following interfaces:

- Total amount of iterations: `IterationCounter()`.
- Value of t at the solution: `TSolution()`.
- Value of y at the solution: `YSolution()`.

- Value of t at the left boundary of the interval of uncertainty: `TLeft()`.
- Value of y at the left boundary of the interval of uncertainty: `YLeft()`.
- Value of t at the right boundary of the interval of uncertainty: `TRight()`.
- Value of y at the right boundary of the interval of uncertainty: `YRight()`.

Moreover, it is possible to use the `BzzPrint` and `BzzMessage` functions, for each class implemented (Buzzi-Ferraris and Manenti, 2010a), as they summarize all the above information. For example

```
BzzFunctionRoot z(.6,.7,yzero1);
char control;
for(int i = 1;i <= 10;i++)
   {
   control = z(1);
   BzzPrint("\ncontrol = %d",control);
   BzzPrint("\nsolution t = %.8e",z.TSolution());
   BzzPrint("\nsolution y = %.8e",z.YSolution());
   BzzPrint("\nleft t = %.8e y = %.8e",
         z.TLeft(),z.YLeft());
   BzzPrint("\nright t = %.8e y = %.8e",
         z.TRight(),z.YRight());
   double delta = fabs(z.TRight()-z.TLeft());
   BzzPrint("\ndelta %.8e", delta);
   BzzPrint("\nIteration count %.d",
         z.IterationCount());
   if(control != 1) break;
   }
z.BzzPrint("\nResults:");
```

The function root-finding is performed by exploiting one of the following algorithms depending on the situation: the inverse rational interpolation with the Bulirsch–Stoer prevision; the regula falsi method using the two best values of the previous iterations; the hyperbolic method with three points; Bolzano's method. The former is the basic method. The regula falsi method is adopted either when only two points are available or when there are problems with the Bulirsch–Stoer prediction, that is, when the error between the predictions does not decrease with the degree of rational functions. The hyperbolic method is adopted when only three points are available. Bolzano's methods are adopted when the first three methods do not improve the objective function significantly; it is also used when the interval of uncertainty does not decrease satisfactorily in the iterations of the three previous methods. A check is performed for each new prediction to ensure that it falls within the search interval and is also significantly distant from the points already analyzed.

The `BzzFunctionRootMP class` has the same constructors and functions as the `BzzFunctionRoot` class. The difference between them lies in the fact that the former requires more processors to be available in a shared memory

architecture. Consequently, anyone availing of it must use a compiler that accepts openMP libraries and the right version of the ***BzzMath*** library. Moreover, the selected compiler must be pointed out by means of an *ad hoc* `define` command (see the readme.pdf file in the tutorial).

If we are using the INTEL C++ 11 compiler for an MS Windows environment, for example, we need to start the program as follows:

```
#define BZZ_COMPILER 11 // Intel 11 Compiler
#include "BzzMath.hpp"
```

1.9 Case Studies

This section illustrates a set of case studies in which root-finding plays an important role in chemical engineering including the calculation of the volume of a nonideal gas, bubble point, and zero-crossing. However, these scenarios also crop up in several other areas. For instance, the calculation of the volume of a nonideal gas is a typical problem in fluid dynamics, whereas the zero-crossing problem is very common in all disciplines involving differential and differential-algebraic systems as convolutions models, such as the optimal control for electrical and electronic purposes.

1.9.1 Calculation of the Volume of a Nonideal Gas

The calculation of the volume of a nonideal gas is a root-finding problem. Let us consider, for example, Rice's problem (Rice, 1993) where the Beattie–Bridgeman equation of state joins the temperature $T(\mathrm{K})$, the pressure $P(\mathrm{atm})$, and the molar volume $V(\mathrm{l/mol})$ through the formula

$$P = \frac{\alpha}{V} + \frac{\beta}{V^2} + \frac{\gamma}{V^3} + \frac{\delta}{V^4} \tag{1.60}$$

where $R = 0.08206$ $(\mathrm{l\,atm/K})$ is the gas constant and

$$\alpha = RT \tag{1.61}$$

$$\beta = RTB_0 - A_0 - Rc/T^2 \tag{1.62}$$

$$\gamma = -RTB_0 b + A_0 a - RcB_0/T^2 \tag{1.63}$$

$$\delta = RB_0 bc/T^2 \tag{1.64}$$

Example 1.4

Consider a stage of the compression section of an air separation unit (Zhu et al., 2010): the selected gas is the air and the following numerical values must be used as parameters: $A_0 = 5.0065$, $B_0 = 0.10476$, $a = 0.07132$, $b = 0.07235$, and $c = 660\,000$. Calculate the volume V for the air at 273.15 K and 2 atm.

The required program is

```
#define BZZ_COMPILER 0 //Visual C++ 6 Compiler
#include "BzzMath.hpp"
double BeattieBridgeman(double t);
void main(void)
   {
   BzzPrint("\n\n\n* Beattie-Bridgeman Example *");
   BzzFunctionRoot z(.01,1000.,BeattieBridgeman);
   z();
   z.BzzPrint("Results");
   }

double BeattieBridgeman(double t)
   {
   static double R = .08206;
   static double A0 = 5.0065;
   static double B0 = .10476;
   static double a = .07132;
   static double b = .07235;
   static double c = 660000.;
   static double T = 273.15;
   static double P = 2.;
   static double alfa = R * T;
   static double beta = R * T * B0 - A0 - R *
       c / (T * T);
   static double gamma = -R * T * B0 * b + A0 * a
       - R * c * B0 / (T * T);
   static double delta = R * B0 * b * c / (T * T);
   double ut = 1. / t;
   double y = P - ut * (alfa + ut * (beta + ut *
       (gamma + ut * delta)));
   return y;
   }
```

The result is

```
tSolution 11.05473417141489
ySolution 0.00000000000000
```

1.9.2 Calculation of the Bubble Point of Vapor–Liquid Equilibrium

The calculation of vapor–liquid equilibrium is a classic problem usually included in the introductions to books on numerical analysis. It is, in fact, conceptually and mathematically very simple, and since it does not require any deep knowledge in the chemical engineering, it is also very useful to a broader readership especially as a dedicated code has not been used for this specific problem.

Chemical and industrial engineering both provide a series of examples involving nonideal vapor–liquid equilibrium calculation. For example, the equations governing a flash drum separator with molar fractions $z_1, z_2, \ldots, z_C$, flow rate F, and enthalpy H are

- component mass balance: $Fz_i - Vy_i - Lx_i = 0 \qquad (i = 1, \ldots, C)$
- overall mass balance: $F - V - L = 0$
- equilibrium: $y_i - k_i(P, T, \mathbf{x}, \mathbf{y}) \cdot x_i = 0 \qquad (i = 1, \ldots, C)$
- stoichiometry: $\sum_{m=1}^{C} y_i - 1 = 0$
- enthalpy balance: $FH_F + Q - VH - Lh = 0$

where V, L, and F are the vapor, liquid, and inlet flow rates; H, h, and H_F are their corresponding enthalpies; y_i and x_i are the molar fractions; Q is the duty provided to the flash; and k_i are the equilibrium constants. In the nonideal case, $k_i = k_i(T, P, \mathbf{x}, \mathbf{y})$. The system consists of $2C + 3$ equations, where C is the number of components. If we assign F, H_F, $\mathbf{z}$, P, and Q and the functions $\mathbf{k}(T, P, \mathbf{x}, \mathbf{y})$, $H(T, P, \mathbf{y})$, and $h(T, P, \mathbf{x})$ are known, the system is solved in its $2C + 3$ equations to get the variables $\mathbf{x}$, $\mathbf{y}$, T, V, and L.

Nevertheless, there are several variants of the problem that may be of interest. It is possible to assign the value for some of the previous variables and consider an equal number of the remaining ones as unknown.

Some combinations of given-unknown variables are not valid since they lead to ill-posed physical problems and therefore to unfeasible or unreliable numerical solutions.

Two important problems are the bubble and dew point calculations, obtained by zeroing V and L, respectively, while the duty Q is unknown.

Example 1.5

Consider the following binary mixture consisting of 0.3 mol of toluene and 0.7 mol of 1-butanol at 760 mmHg. The partial pressure of the two components is calculated as follows (Henley and Rosen, 1969):

$$\text{Toluene}: \quad \log_{10} P_1 = 6.95508 - \frac{1345.087}{219.516 + T}$$

$$\text{1-Butanol}: \quad \log_{10} P_2 = 8.19659 - \frac{1781.719}{217.675 + T}$$

where $P_i = P_i$ (mmHg) and $T = T$ (°C).

k_i of the two components are obtained from the relations

$$k_i = \frac{\gamma_i P_i}{P} \tag{1.65}$$

and γ_i from

$$\log_{10}\gamma_1 = \frac{0.38969x_2^2}{((0.38969/0.55954)x_1 + x_2)^2} \tag{1.66}$$

$$\log_{10}\gamma_2 = \frac{0.55954x_1^2}{(x_1 + (0.55954/0.38969)x_2)^2} \tag{1.67}$$

If all k_i do not depend on vapor composition, as is usually the case, the bubble point calculation can be performed by searching for the root of the single equation

$$\sum\nolimits_i k_i x_i - 1 = \sum\nolimits_i k_i z_i - 1 = 0 \tag{1.68}$$

with respect to the temperature.

The program is

```
#define BZZ_COMPILER 0 // Visual C++ 6 Compiler
#include "BzzMath.hpp"
double Bubble(double T);
double x1,x2,g1,g2,P;
void main(void)
   {
   double a = .38969;
   double b = .55954;
   P = 760.;
   x1 = .3;
   x2 = .7;
   g1 = pow(10.,a*x2*x2/BzzPow2(a/b*x1 + x2));
   g2 = pow(10.,b*x1*x1/BzzPow2(b/a*x2 + x1));
   BzzFunctionRoot z(90.,120.,Bubble);
   z();
   z.BzzPrint("Results");
   }

double Bubble(double T)
   {
   double p1 = pow(10.,6.95508 -
       1345.087/(219.516 + T));
   double p2 = pow(10.,8.19659 -
       1781.719/(217.675 + T));
   return (g1*p1*x1 + g2*p2*x2)/P - 1.;
   }
```

It results in

```
Number of iterations in the last call 7
Total number of iterations 9
tLeft 1.080560412549669e+002
yLeft -5.551115123125783e-016
tRight 1.080560412549671e+002
yRight 7.549516567451065e-015
tSolution 1.080560412549669e+002
ySolution-5.551115123125783e-016
Cause of exit:
(tRight - tLeft) <= tTolAbs + EPS*(fabs(tLeft) +
      fabs(tRight))
```

1.9.3 Zero-Crossing Problem

The zero-crossing problem is function root-finding during the integration of a differential system. In other words, during the integration of ODE/DAE systems, there may be a need to calculate at which value of the independent variable t a certain function of the dependent variables $\mathbf{y}$ is zeroed.

If the function to be zeroed has a single root and changes sign during the integration of the system, an object from the `BzzFunctionRoot` class can be used. The root-finding classes can be combined with classes for differential systems based on multivalue algorithms (see Vol. 4 – Buzzi-Ferraris and Manenti, in press).

The objects from these classes automatically change the integration step and the order of the algorithm adopted by adapting them to the problem features and requirements. Moreover, the integration step is disjointed from the user's need to know the values at certain specific points. Thus, if the function to be zeroed has a single root and it changes sign during the integration of the system, it is sufficient to monitor its value in the mesh points during the integration and, when it changes sign, to use an object from the `BzzFunctionRoot` class to refine the root. In the `BzzOdeStiff` class, adopted hereafter to integrate the differential system, the functions `GetTimeInMeshPoint` and `GetYInMeshPoint` allow the values of t and y to be obtained at each mesh point used in the integration procedure.

Example 1.6

Consider the following ordinary differential equation (ODE) system:

$$\begin{aligned}\frac{dy_1}{dt} &= -r_1 + r_2 \\ \frac{dy_2}{dt} &= r_1 - r_2 - r_3 \\ \frac{dy_3}{dt} &= r_3\end{aligned} \tag{1.69}$$

with

$$
\begin{aligned}
r_1 &= 0.4 \cdot y_1 \\
r_2 &= 10. \cdot y_2 \cdot y_3 \\
r_3 &= 100. \cdot y_2^2
\end{aligned}
\tag{1.70}
$$

generated by a mass balance in a batch reactor for the reaction mechanism: $A \rightarrow B \rightarrow C$. Suppose that we want to find the value of t where the variable $y_1 = 0.7$. The starting point is $y_0 = \{1, 0, 0\}$.

The program is

```
#define BZZ_COMPILER 0 // Visual C++ 6 Compiler
#include "BzzMath.hpp"
BzzOdeStiff oFindRoot;
void BzzOdeRobertsonVariant(BzzVector &y,
    double t,BzzVector &f);
double BzzFunctionRootRobertsonVariant(double t);
BzzVector yR;

void main(void)
   {
   BzzVector y0(3,1.,0.,0.);
   double t0 = 0.,tOut;
   BzzVector y;
   oFindRoot(y0,t0,BzzOdeRobertsonVariant);
   BzzVector yMin(3);
   oFindRoot.SetMinimumConstraints(&yMin);
   tOut = 1.e-15;
   double tNextMesh;
   BzzVector yNextMesh;
   while(1)
      {
      y = oFindRoot(tOut);
      tNextMesh = oFindRoot.GetTimeInMeshPoint();
      yNextMesh = oFindRoot.GetYInMeshPoint();
      BzzPrint("\nt %e y %e",
            tNextMesh,yNextMesh[1]);
      if(yNextMesh[1] < .7)
         break;
    tOut = tNextMesh + 1.e-10;
    if(tOut > 100.)
       {
       BzzWarning("No solution y1 == .7");
       break;
      }
    y0 = yNextMesh;
    t0 = tNextMesh;
```

```
  }
BzzPrint("\nt0 %e %e tNext %e %e",
      t0,y0[1],tNextMesh,yNextMesh[1]);
BzzFunctionRoot z(t0,tNextMesh,
      BzzFunctionRootRobertsonVariant,
      y0[1] - .7,
      yNextMesh[1] - .7);
z();
z.BzzPrint("Results");
y = oFindRoot(z.TSolution());
y.BzzPrint("\n y in %e", z.TSolution());
}

void BzzOdeRobertsonVariant(BzzVector &y,
       double t,BzzVector &f)
{
double r1 = .4 * y[1];
double r2 = 10. * y[2] * y[3];
double r3 = 100. * y[2] * y[2];
f[1] = -r1 + r2;
f[2] = r1 - r2 - r3;
f[3] = r3;
}

double BzzFunctionRootRobertsonVariant(double t)
   {
   yR = oFindRoot(t);
   return yR[1] - .7;
   }
```

Please note that the differential system is iteratively integrated until the variable $y_1 \geq 0.7$. At each integration step, the mesh point and the corresponding value of y_1 are both collected. As the value of $y_1 < 0.7$, the integration is stopped and the object `z` from the `BzzFunctionRoot` class is initialized on the left with the values collected in the last but one mesh point, and on the right with the values collected in the last mesh point (where the function `yR[1] - .7` changes sign).

Within the function `BzzFunctionRootRobertsonVariant` to be zeroed, the object `z` uses the object `oFindRoot` from the class, which has integrated the differential system. oFindRoot allows the vector **y** to be calculated in each point within the two mesh points `t0` and `tNextMesh` of the last integration step.

It results in

```
Number of iterations in the last call 4
Total number of iterations 4
tLeft 1.111197715376305e+000
```

```
yLeft 1.110223024625157e-016
tRight 1.111197715376310e+000
yRight -8.881784197001252e-016
tSolution 1.111197715376305e+000
ySolution 1.110223024625157e-016

Cause of exit:
(tRight - tLeft) <= tTolAbs + EPS*(fabs(tLeft)
   + fabs(tRight))

y in 1.111198e+000
    1      7.00000000000000e-001
    2      4.28812294979364e-002
    3      2.57118770502064e-001
```

1.9.4 Stationary Condition in a Gravity-Flow Tank

A classic control problem (Luyben, 1990; Dash *et al.*, 2003) is the buffer tank with a pipe connected at the bottom of the tank and with one end open to the atmosphere. The pipe is equipped with a valve. The Figure 1.3 shows a scheme of the system. A proportional–integral (PI) controller automatically adjusts the valve position to achieve the set point for the tank level based on the measurement of the tank level.

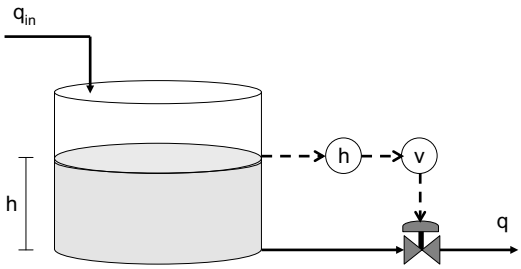

Figure 1.3 Scheme of the buffer tank.

Example 1.7

The main equation that governs the tank system is the mass balance over the tank. It results in an ODE, which relates the tank level h to the inflow q_{in} and outflow q rates as follows:

$$\frac{dh(t)}{dt} = \frac{q_{in}(t)}{A_{tank}} - \frac{q(t)}{A_{tank}} \tag{1.71}$$

In addition to this, an algebraic equation follows on from the pressure balance, equating the hydrostatic pressure at the bottom of the tank with the pressure losses over the valve and the overall tube. This equation reads as follows:

$$\rho g h(t) = 2\frac{f\rho L_{tube}}{\pi^2 D_{tube}^5}(q(t))^2 + \frac{\rho g}{C_V(v(t))^2}(q(t))^2 = \alpha(q(t))^2 + \frac{\beta}{C_V(v(t))^2}(q(t))^2 \tag{1.72}$$

where $\alpha = 2((f\rho L_{tube})/(\pi^2 D_{tube}^5))$, $\beta = \rho g$, and $C_V(v(t)) = (C_{V,MAX}/\tau)e^{-v(t)\ln(1/\tau)}$.

When a turbulent flow regime is assumed, the friction factor is constant. Thus, the algebraic equation can be solved analytically in $q(h(t), v(t))$:

$$q(t) = \sqrt{\frac{\rho g h(t)}{\alpha + \beta(C_V(v(t)))^{-2}}} \tag{1.73}$$

Substituting $q(t)$ in (1.71), it results in

$$\frac{dh(t)}{dt} = \frac{q_{\text{in}}(t)}{A_{\text{tank}}} - \frac{1}{A_{\text{tank}}}\sqrt{\frac{\rho g h(t)}{\alpha + \beta(C_V(v(t)))^{-2}}} \tag{1.74}$$

We want to calculate the diameter of the tube that leads to the steady state with level $h = 10$ m and a valve open position $v = 0.5$. At the steady state

$$\frac{dh(t)}{dt} = 0 \tag{1.75}$$

Hence, the problem is brought back to the search for the diameter D_{tube} that gives

$$\frac{q_{in}(t)}{A_{\text{tank}}} - \frac{1}{A_{\text{tank}}}\sqrt{\frac{\rho g h(t)}{\alpha + \beta(C_V(v(t)))^{-2}}} = 0 \tag{1.76}$$

while the other variables are assigned (numerical values can be acquired directly from the program below).

The program is

```
#define BZZ_COMPILER 0 // Visual C++ 6 Compiler
#include "BzzMath.hpp"
double BzzFunctionRootVessel(double dTube);
void main(void)
   {
   BzzPrint("\n\nStationary Vessel");
   BzzFunctionRoot z;
   z(.01,1.,BzzFunctionRootVessel);
   z();
```

```
    z.BzzPrint("Results");
    }

double BzzFunctionRootVessel(double dTube)
    {
    static double v = .5;
    static double h = 10.;
    static double aTank = 5.;
    static double lTube = 5.;
    static double g = 9.81;
    static double quin = 5.36;
    static double f = 0.0125;
    double alfa = 2. * f * lTube /
          (3.14 * 3.14 * BzzPow5(dTube));
    static double beta = 9.81;
    static double cvmax = 20.;
    static double tau = 50.;
    static double lnta = log(1./50.);
    static double Cv = cvmax / tau * exp(-v*lnta);
    double y = (quin - sqrt((g * h)/
          (alfa + beta /BzzPow2(Cv))))/aTank;
    return y;
    }
```

It results in

```
Stationary Vessel
Number of iterations in the last call 9
Total number of iterations 11
tLeft 3.569331740818127e-001
yLeft 2.309263891220326e-015
tRight 3.569331740818135e-001
yRight -1.421085471520201e-015
tSolution 3.569331740818135e-001
ySolution -1.421085471520201e-015
```

The required value for D_{tube} is `tSolution`.

1.10 Tests for BzzFunctionRoot and BzzFunctionRootMP Classes

The following tests have already been proposed by many different authors (Atkinson, 1989; Brent, 1973; Chapra and Canale, 1988; Conte and De Boor, 1980;

Forsythe *et al.*, 1977; Rice, 1993; Schwarz, 1989) and are adopted to test the `BzzFunctionRoot` and `BzzFunctionRootMP` classes:

- $y_1 = t(t^2 - 2) - 5 \qquad t_A = 2.; t_B = 3$
- $y_2 = t^3 - t - 1 \qquad t_A = 1.; t_B = 2$
- $\begin{cases} 0, & \text{if } |t| < 10^{-6} \\ y_3 = t\exp(-1./t^2) & \text{otherwise} \end{cases} \qquad t_A = -1.; t_B = 4$
- $\begin{cases} y_4 = \exp(t), & \text{if } t > -10 \\ y_4 = \exp(-10) - (t+10)^2 & \text{otherwise} \end{cases} \qquad t_A = -10.012.; t_B = 0$
- $y_5 = -2 \cdot \sum_{i=1}^{20} \frac{(2i-5)^2}{\left(t - i^2\right)^3} \qquad t_A = 11^2 + 0.001; t_B = 12^2 - 0.001$
- $y_6 = \sinh(t) - 2 \qquad t_A = 1.3; t_B = 1.6$
- $y_7 = t - \exp(-t) \qquad t_A = 0.5; t_B = 0.69$
- $y_8 = 1 + \cos(t)\cosh(t) \qquad t_A = 1.8; t_B = 1.9$
- $y_9 = 3t + \sin(t) - \exp(t) \qquad t_A = 0; t_B = 1$
- $y_{10} = t^6 - t - 1 \qquad t_A = 2; t_B = 1$
- $y_{11} = (t-1)\left(1 + (t-1)^2\right) \qquad t_A = 0; t_B = 3$
- $y_{12} = t^2 - 1 \qquad t_A = 0; t_B = 3$
- $y_{13} = -1 + t(3 + t(-3 + t)) \qquad t_A = 0; t_B = 3$
- $y_{14} = t^{10} - 1 \qquad t_A = .5; t_B = 10$
- $\begin{cases} y_{15} = \sin(t) - 1.5, & \text{if } t < 30 \\ y_{15} = -2.8 \cdot (t - 30) & \text{if } 30 \le t < 35 \\ y_{15} = \sin(t) + 1.5 & \text{if } t > 35 \end{cases} \qquad t_A = 0; t_B = 100$
- $\begin{cases} y_{16} = \dfrac{|t - 8.4317|^4}{1 + t^2} & \text{if } t \ge 8.4317 \\ y_{16} = -\dfrac{|t - 8.4317|^4}{1 + t^2} & \text{if } t < 8.4317 \end{cases} \qquad t_A = 8; t_B = 8.45$
- $\begin{array}{l} y_{17} = (z - 4.)(z + 2.)(z + 41.) \\ z \quad = 10^8 \cdot \left(t - 1.01 \times 10^{-9}\right) \end{array} \qquad t_A = 0; t_B = 1 \times 10^{-6}$
- $y_{18} = \exp(t + 1.00202) - e \qquad t_A = -1; t_B = 1$
- $y_{19} = t - 0.327 \cdot (0.06 - 161 \cdot t)^{0.804} \cdot \exp\left(\dfrac{-5230.}{1.987\left(373. + 1.84 \times 10^6 \cdot t\right)}\right) \qquad t_A = 0.; t_B = \frac{0.06}{161.}$
- $y_{20} = \exp(-1./(10000 \cdot t)) + \exp(-t) - 1.0001 \qquad t_A = 0.01; t_B = 10$
- $y_{21} = \frac{1.}{t - 0.1} \qquad t_A = -1; t_B = 1$
- $y_{22} = \exp(-t^2)(t^2 - 17. \cdot t + 71.) \quad t_A = 8.; t_B = 10$
- $y_{23} = \dfrac{20}{(t+1)^{15}} + \dfrac{36}{(t+1)^{25}} + \dfrac{40}{(t+1)^{33}} + \dfrac{475}{(t+1)^{40}} - \dfrac{1.12\left((t+1)^{40} - 1\right)}{t \cdot (t+1)^{40}} - \dfrac{6}{(t+1)^4} - \dfrac{3}{(t+1)^8} - 4.5$

  $t_A = 0.0001; t_B = 0.1$
- $y_{24} = (\gamma - 4)(\gamma + 2)(\gamma + 4)$ with $\gamma = 10^8\left(t - 1.01 \times 10^{-9}\right) \qquad t_A = 0.; t_B = 1$
- $y_{25} = \cos^2(2 \cdot t) - t^2 \qquad t_A = 0.; t_B = 1.5$

- $y_{26} = (t^2 - 1)^6 \log(t) \qquad t_A = 0.5; t_B = 1.5$
- $y_{27} = \frac{t}{8}\left(63t^4 - 70t^2 + 15\right) \qquad t_A = 0.6; t_B = 1$
- $y_{28} = e^{-t} - 10^{-9} \qquad t_A = 10.; t_B = 30$

The tests can be found in the directory

```
BzzMath/Examples/BzzMathAdvanced/FunctionRoot/
FunctionRootTests
```

and

```
BzzMath/Examples/BzzMathAdvanced/FunctionRoot/
FunctionRootMPTests
```

in `BzzMath7.zip` file available at www.chem.polimi.it/homes/gbuzzi.

The results obtained using the `BzzFunctionRoot` class are

Test	ni	tLeft	tRight	ySolution
1	6	2.058824e+000	2.094551e+000	1.376677e-013
2	7	1.166667e+000	1.324718e+000	1.583178e-013
3	11	-3.408015e-001	1.510560e-001	1.399855e-020
4	11	-1.000674e+001	-1.000637e+001	-6.148104e-017
5	12	1.320386e+002	1.320406e+002	1.211948e-015
6	5	1.443635e+000	1.443643e+000	-3.192896e-011
7	5	5.671433e-001	5.671436e-001	-3.375078e-014
8	5	1.873698e+000	1.875104e+000	-2.695622e-012
9	8	3.604217e-001	3.615823e-001	-1.386447e-012
10	8	1.134724e+000	1.134724e+000	-8.881784e-016
11	10	9.997512e-001	1.000000e+000	4.440892e-016
12	9	9.310345e-001	1.000000e+000	4.004352e-011
13	25	9.989004e-001	1.000168e+000	4.742651e-012
14	21	9.984530e-001	1.000000e+000	4.642953e-012
15	9	3.237578e+001	3.250000e+001	0.000000e+000
16	43	8.431700e+000	8.431700e+000	-1.745483e-008
17	16	4.101000e-008	4.101000e-008	-1.199041e-013
18	7	-4.638343e-001	-2.020000e-003	1.110223e-014
19	10	3.406054e-004	3.406157e-004	-1.269421e-011
20	7	9.106147e+000	9.106634e+000	6.067147e-012
21	100	1.000000e-001	1.000000e-001	-5.542892e+015
22	16	9.618034e+000	9.618034e+000	0.000000e+000
23	8	9.864724e-002	9.864724e-002	-7.993606e-015
24	59	4.101000e-008	4.101000e-008	0.000000e+000
25	12	5.149333e-001	5.149333e-001	-5.551115e-017
26	8	1.000000e+000	1.000000e+000	0.000000e+000

```
27 11  9.061798e-001   9.061798e-001  -2.012123e-016
28 16  2.072327e+001   2.072327e+001   1.240771e-024
```

The results obtained using the `BzzFunctionRootMP` class and a dual-core processors machine are

```
Test ni     tLeft          tRight          ySolution
  1   5  2.094551e+000   2.094552e+000  -3.381295e-012
  2   6  1.324718e+000   1.324718e+000  -1.286748e-013
  3   4 -4.463862e-001   7.636580e-002   2.582394e-076
  4   6 -1.000680e+001  -1.000674e+001   1.271614e-011
  5   7  1.320406e+002   1.321958e+002  -1.593170e-013
  6   4  1.443635e+000   1.443639e+000  -9.192180e-011
  7   4  5.503575e-001   5.671433e-001   3.651524e-013
  8   5  1.875104e+000   1.875104e+000   2.220446e-016
  9   6  3.604217e-001   3.604217e-001   4.440892e-016
 10   7  1.134724e+000   1.134724e+000   1.110223e-015
 11   7  1.000000e+000   1.016659e+000  -9.320322e-012
 12   7  1.000000e+000   1.000000e+000  -9.992007e-015
 13  11  9.995625e-001   1.000223e+000   1.111022e-011
 14   9  9.999998e-001   1.000000e+000   2.100320e-011
 15   5  3.128829e+001   3.250000e+001   0.000000e+000
 16  18  8.431700e+000   8.431700e+000  -3.219405e-007
 17  11  4.101000e-008   4.101000e-008   0.000000e+000
 18   6 -2.045756e-003  -2.020000e-003   0.000000e+000
 19   6  3.406054e-004   3.406151e-004  -2.079496e-011
 20   6  9.106147e+000   9.106294e+000   4.924173e-012
 21  36  1.000000e-001   1.000000e-001   8.590199e+010
 22  11  9.618034e+000   9.618034e+000  -9.495633e-055
 23  12  9.864724e-002   9.864724e-002  -1.110223e-015
 24  33  4.101000e-008   4.101000e-008   0.000000e+000
 25  10  5.149333e-001   5.149333e-001   0.000000e+000
 26   8  1.000000e+000   1.000000e+000   0.000000e+000
 27  11  9.061798e-001   9.061798e-001  -2.012123e-016
 28  16  2.072327e+001   2.072327e+001   6.203855e-025
```

The results obtained using the `BzzFunctionRootMP` class and a quad-core processors machine are

```
Test ni      tLeft          tRight         ySolution
 1   4   2.094551e+000  2.094553e+000 -6.369927e-011
 2   4   1.324718e+000  1.324718e+000 -2.545741e-013
 3   2  -2.545275e-001  1.182087e-001  9.824951e-033
 4   7  -1.000674e+001 -1.000674e+001  5.366284e-013
 5   5   1.320381e+002  1.320406e+002  9.572343e-013
```

```
 6    4   1.443635e+000  1.443635e+000  0.000000e+000
 7    3   5.671433e-001  5.680733e-001 -5.143663e-013
 8    4   1.875085e+000  1.875104e+000 -6.661338e-016
 9    4   3.604214e-001  3.604217e-001  5.286882e-013
10    5   1.134724e+000  1.134724e+000  6.645795e-013
11    2   8.750000e-001  1.000000e+000  0.000000e+000
12    2   8.750000e-001  1.000000e+000  0.000000e+000
13    2   8.750000e-001  1.000000e+000  0.000000e+000
14    6   1.000000e+000  1.000000e+000  2.220446e-015
15    6   3.242825e+001  3.250000e+001  0.000000e+000
16   11   8.431700e+000  8.431700e+000  2.686326e-006
17    7   4.101000e-008  4.101000e-008  0.000000e+000
18    5  -2.020000e-003 -2.020000e-003  0.000000e+000
19    4   3.405094e-004  3.406054e-004  7.773405e-012
20    4   9.104740e+000  9.106147e+000 -3.694319e-011
21   30   1.000000e-001  1.000000e-001  6.443085e+010
22    7   9.618033e+000  9.618034e+000  0.000000e+000
23    9   9.864724e-002  9.864724e-002 -2.420286e-014
24   14   4.101000e-008  4.101000e-008 -2.131628e-014
25   10   5.149333e-001  5.149333e-001  0.000000e+000
26    8   1.000000e+000  1.000000e+000  0.000000e+000
27   11   9.061798e-001  9.061798e-001 -2.012123e-016
28   16   2.072327e+001  2.072327e+001  6.203855e-025
```

1.11 Some Caveats

Before ending this chapter, we should emphasize several points crucial to the function root-finding problem.

Caveat no. 1
Programs based on a single algorithm can only be used in exceptional cases.

It is usually necessary to work with a program that contains at least two algorithms: one robust and one efficient.

Caveat no. 2
It is necessary to mind the stop criteria.

The two most common criteria are verifying that the difference $|t_{i+1} - t_i|$ is small and/or the value of $|y_i|$ is small. Both of these criteria are often unreliable.