GSEIM Users' Manual

Mahesh B. Patil

Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai-400076
e-mail: mbpatil@ee.iitb.ac.in

Contents

Chapter 1

Introduction

GSEIM (General-purpose Simulator with Explicit and Implicit Methods) is meant for simulation of electrical circuits, especially power electronic circuits, and also for numerical solution of ordinary differential equations (ODEs). In terms of basic functionality, GSEIM is similar to commercial packages such as `Simulink` [1], `Simscape` [2], and `PSIM` [3]. An earlier version of GSEIM is described in [4]. Some additional features were subsequently added to the GSEIM solver to make it suitable for Jupyter Notebooks related to power electronic circuits [5].

Fig. 1.1 shows the block diagram of GSEIM. The GSEIM solver takes as input the user's "circuit file", i.e., a description of the circuit to be simulated, and produces output files (as specified in the circuit file). Using the data available in the output files, the user can then plot the various quantities of interest. We will discuss the circuit file structure in Chapter 6.

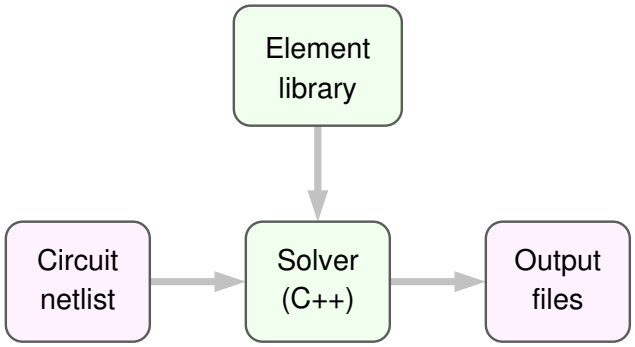


Figure 1.1: Block diagram of GSEIM.

GSEIM Elements: GSEIM allows two types of elements: (a) electrical and (b) directional (flow-graph). In electrical elements – such as resistor, capacitor, voltage source – the element equations involve relationships between node voltages and currents. In addition, charge conservation applies, requiring a KCL equation at each node. In GSEIM, as in most commonly used circuit simulators, the modified nodal analysis (MNA) scheme is used to assemble the circuit equations, as described in Chapter 2.

In directional (flow-graph) elements – such as adder, integrator, limiter – only relationships between the input and output variables of the element are involved, and there are no addition KCL-type constraints to be satisfied. We will refer to a "circuit" or system composed of only directional elements as a "flow graph." Simulation of flow graphs is discussed in Chapter 4.

In GSEIM, each element is described in a "template", i.e., a text file with information about nodes, parameters, and equations related to that element. The elements are collectively shown as the

"element library" in Fig. 1.1. GSEIM currently includes elements which enable simulation of several power electronic circuits. Lists of electrical and directional elements are given in Appendices A and B, respectively.

Analysis types: GSEIM allows different types of analyses to be performed. In the circuit file (see Fig. 1.1), the required type of analysis is specified by the user in the "solve block", as described in Chapter 6. The following options are available.

1. **DC**: In DC analysis, the solution is obtained under the condition that all variables are constant. Time derivatives are set to zero. For example, the capacitor equation, $i = C\,\frac{dV}{dt}$ becomes, in the DC case, $i = 0$.

2. **Transient**: In this case, the time derivatives, such as $\frac{dV}{dt}$ in the capacitor equation, are discretized (see Chapter 4). The solution to the resulting system of equations is obtained at each time point between t_{start} and t_{end}.

3. **Periodic steady state (PSS)**: This situation is of particular interest in power electronic circuits such as the buck converter. Here, we are interested in the steady-state solution for one period. GSEIM obtains the periodic steady-state solution using an iterative process (see Chapter 5).

4. **AC (phasor analysis)**: In this case, sinusoidal steady state solution is obtained using phasors.

5. **Start-up**: This analysis provides the "start-up solution," i.e., the solution obtained for a circuit starting with some known values of the state variables, such as capacitor voltages and inductor currents. In case of computation, the start-up analysis is similar to DC analysis.

This manual covers two broad topics:

1. Mathematical background: In this part, numerical techniques used in GSEIM [6] are described, as listed below.
   (a) Modified Nodal Analysis for systematic assembly of circuit equations (Chapter 2)
   (b) Newton-Raphson method for nonlinear equations (Chapter 3)
   (c) Numerical methods for discretisation of time derivatives (Chapter 4)
   (d) Computation of periodic steady-state solution (Chapter 5)

2. Syntax of the GSEIM circuit file: The overall structure of the circuit file, and the syntax of the statements involved in the circuit file (Chapter 6)

Chapter 2

Modified Nodal Analysis

To find the solution for an electrical circuit, the following constraints need to be satisfied simultaneously: (a) Kirchoff's current law (KCL) at each node, (b) Kirchoff's voltage law (KVL) for each loop, and (c) equation(s) describing the behaviour of each element involved in the circuit (e.g., resistor, capacitor, diode, transistor, switch, transformer). The most common approach employed to solve this set of equations is Modified Nodal Analysis (MNA). As the name suggests, MNA is a modified version of Nodal Analysis (NA) which is based on KCL equations written in terms of node voltages (see [7],[8], for example). In the following, we will describe the NA approach with the help of an example, see why it needs to be modified, and then look at the MNA approach. For now, we will restrict our discussion to linear circuits operating under DC conditions. In later chapters, we will see how the MNA approach can be used for circuits involving nonlinear components and time derivatives.

2.1 Nodal Analysis

In nodal analysis, one of the circuit nodes is taken as the reference node (ground) and is assigned a node voltage of 0 V. All other node voltages are defined with respect to the reference node. The element currents are written in terms of the node voltages, and the sum of the element currents at each node is equated to zero, as required by KCL. The resulting set of equations is then solved for the unknowns – the node voltages. Other quantities of interest such as currents, branch voltages are computed by post-processing the solution vector, i.e., the node voltages. Let us illustrate this process with an example.

Consider the circuit shown in Fig. 2.1. We take one of the nodes (node A) as the reference node. The other nodes (B, C, D) are assigned node voltages V_1, V_2, V_3. We write the various element currents in terms of the node voltages, e.g., $I_1 = G_1(V_1 - V_2)$, $I_3 = G_3(0 - V_3)$, where $G_1 = 1/R_1$, etc. Finally, we substitute the expressions for the currents in the KCL equations at nodes B, C, D, and get the following set of equations.

$$\begin{aligned} &\text{KCL at B}: && -I_0 + I_1 = 0\,, \\ &\text{KCL at C}: && -I_1 - I_2 + I_4 + I_5 = 0\,, \\ &\text{KCL at D}: && -I_3 - I_4 - I_5 = 0\,. \end{aligned} \tag{2.1}$$

In terms of node voltages, we have

$$\begin{aligned} -I_0 + G_1(V_1 - V_2) &= 0\,, \\ -G_1(V_1 - V_2) + G_2V_2 + (G_4 + G_5)(V_2 - V_3) &= 0\,, \\ G_3V_3 - (G_4 + G_5)(V_2 - V_3) &= 0\,. \end{aligned} \tag{2.2}$$

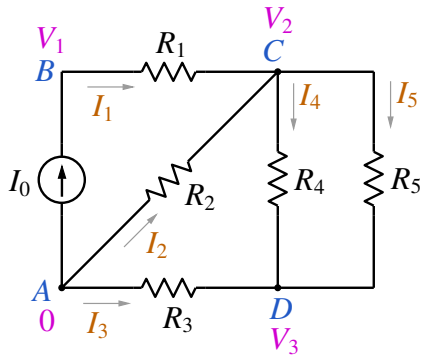


Figure 2.1: Nodal analysis example.

The above equations can be written in a matrix form:

$$\begin{bmatrix} G_1 & -G_1 & 0 \\ -G_1 & G_1+G_2+G_4+G_5 & -G_4-G_5 \\ 0 & -G_4-G_5 & G_3+G_4+G_5 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} I_0 \\ 0 \\ 0 \end{bmatrix}. \tag{2.3}$$

We now have a matrix description of the circuit equations: $\mathbf{YV} = \mathbf{I}_S$. The matrix $\mathbf{Y}$ is called the admittance matrix, $\mathbf{V}$ is the vector of node voltages which we want to obtain, and $\mathbf{I}_S$ is the current source vector, which contains $\pm I_k$, I_k being the current of an independent current source connected at node k. For larger circuits, the admittance matrix is typically sparse, with only 10 to 15% non-zero entries. The sparse nature of the admittance matrix can be exploited to reduce the storage requirement and the number of arithmetic operations (and therefore the CPU time) in solving the linear system.

2.2 Modified Nodal Analysis

If there are voltage sources in the circuit, the NA approach needs to be modified. As an example, consider the circuit of Fig. 2.2. We take A as the reference node and assign V_1, V_2, V_3 to the

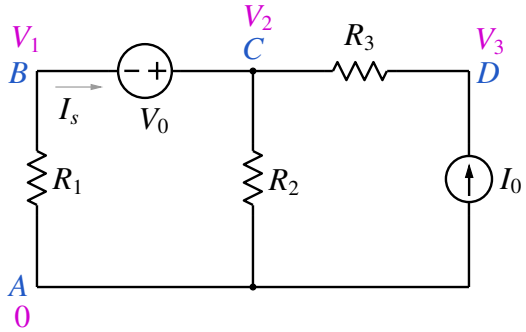


Figure 2.2: Modified Nodal analysis example.

remaining nodes. When we attempt to write KCL at node B or C, we encounter a problem – the current through the voltage source cannot be written in terms of the node voltages V_1 and V_2, and the nodal analysis approach therefore needs to be modified. In the MNA approach, we augment the solution vector (consisting of node voltages) with currents through voltage sources, and the KCL

equations are written in terms of the node voltages as well as these additional variables, i.e., currents through voltage sources[1]. For the circuit of Fig. 2.2, we get

$$\begin{aligned} \text{KCL at B}: &\quad G_1 V_1 + I_s = 0\,, \\ \text{KCL at C}: &\quad -I_s + G_2 V_2 + G_3(V_2 - V_3) = 0\,, \\ \text{KCL at D}: &\quad -I_0 + G_3(V_3 - V_2) = 0\,. \end{aligned} \tag{2.4}$$

We now have four unknowns (V_1, V_2, V_3, I_s) but only three equations. The fourth equation comes from the element equation for the voltage source, viz., $V_2 - V_1 = V_0$. The equations can be written in a matrix form:

$$\begin{bmatrix} G_1 & 0 & 0 & 1 \\ 0 & G_2 + G_3 & -G_3 & -1 \\ 0 & -G_3 & G_3 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ I_s \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ I_0 \\ V_0 \end{bmatrix}. \tag{2.5}$$

We can already guess what a circuit simulator must be doing behind the scenes for a linear circuit under DC conditions:

1. Read the "circuit file", which is a description of the connections in the circuit (the *topology*) and the specification of each element (the *behaviour*). For the circuit of Fig. 2.2, a SPICE-like circuit description[2] may look like

   ```
   R1  A  B  1k
   R2  A  C  0.5k
   R3  C  D  2k
   VS  C  B  5
   IS  D  0  1m
   ```

   where the first string of the statement (e.g., `R1`) gives the type of the element (`R`) and its name. The next two strings (`A` and `B`) specify that it is connected between nodes `A` and `B`. The last string in the statement says that its value is 1 kΩ.

2. Decide "what goes where" in the matrix equation: This step is called "parsing", and as we can imagine, it takes a significant programming effort. However, the basic idea is simple. We need to figure out the following.

   (a) How many variables (unknowns)?

   (b) What does each row of the matrix correspond to? A KCL or the branch equation for one of the voltage sources?

   (c) Where are the non-zero entries in the MNA matrix? How is each entry computed in terms of the circuit parameters?

[1] The currents through independent voltage sources as well as dependent voltage sources (CCVS, VCVS) are added to the solution vector.

[2] In the good old days, one had to write the circuit file using one's favourite editor. In modern times, the user has the luxury of entering the circuit schematic using a GUI which converts the schematic to the circuit file format internally, often without the user's knowledge.

3. Solve the matrix equation: This is the most crucial part of a circuit simulator since it generally takes the largest chunk of the CPU time, particularly for large circuits. The reason is easy to understand: The number of multiplications involved in solving $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $N \times N$ matrix, goes as N^3. By exploiting sparsity, the number of multiplications can be reduced, but the dependence on N remains superlinear. Fortunately, efficient sparse matrix solvers are available in the public domain, and one need not reinvent the wheel.

4. Calculate the quantities of interest by post-processing. Solving the MNA circuit equations yields the node voltages and voltage source currents. These can be used to obtain other quantities simply by post-processing, i.e., without solving any additional equations. For example, the current through R_3 in the circuit of Fig. 2.2 can be obtained as $I_3 = (V_2 - V_3)/R_3$, and the power supplied by the voltage source as $P = (V_2 - V_1) \times I_s$.

Note that we have described the MNA approach for linear circuits in a DC situation only. We will see in Chapter 3 how it can be extended to nonlinear circuits in a DC situation. In Chapter 4, we will go one step further and see how elements involving time derivatives (e.g., capacitors and inductors) can be incorporated within the MNA equations.

Chapter 3

Newton-Raphson Method

Nonlinear equations arise in a wide variety of electronic and power electronic circuits, and they need to be solved using an iterative method. The Newton-Raphson (NR) iterative method is commonly used for solving nonlinear equations because of its excellent convergence properties. GSEIM also employs the NR method, and it would be useful for GSEIM users to be familiar with the functioning of the NR method.

To begin with, let us see where the NR method comes from.

3.1 Single Equation

Consider the equation $f(x) = 0$. Let $x = r$ be the root (assumed to be single and real), i.e., $f(r) = 0$. Let us say that we have some idea of the root in the form of *initial guess* $x^{(0)}$. The goal of the NR method is to iteratively refine this value so that $f(x) = 0$ is satisfied to a higher accuracy with every NR iteration. We denote the successive values of x by $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, $\cdots$

Consider $x = x^{(i)}$. Expanding $f(x)$ around this value, we get

$$f(x^{(i)} + \Delta x^{(i)}) = f(x^{(i)}) + \Delta x^{(i)} \left.\frac{df}{dx}\right|_{x^{(i)}} + \frac{(\Delta x^{(i)})^2}{2!} \left.\frac{d^2 f}{dx^2}\right|_{x^{(i)}} + \cdots \tag{3.1}$$

We seek the value of $\Delta x^{(i)}$ which will satisfy $f(x^{(i)} + \Delta x^{(i)}) = 0$, assuming that the contribution from second- and higher-order terms is small compared to the first term, i.e.,

$$f(x^{(i)} + \Delta x^{(i)}) \approx f(x^{(i)}) + \Delta x^{(i)} \left.\frac{df}{dx}\right|_{x^{(i)}} = 0, \text{ or } \Delta x^{(i)} = -\frac{f(x^{(i)})}{\left.\frac{df}{dx}\right|_{x^{(i)}}}. \tag{3.2}$$

If our assumption (that only the first term in $\Delta x^{(i)}$ is significant) is indeed valid, our job is done: we simply add $\Delta x^{(i)}$ to $x^{(i)}$, and that gives us the solution. If not, we treat $x^{(i)} + \Delta x^{(i)}$ as the next candidate for x (i.e., $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$), perform another NR iteration, and so on. Let us illustrate this procedure with an example.

Consider $f(x)$ given by

$$f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0, \tag{3.3}$$

with a_3=1, a_2=−3.2, a_1=8.7, a_0=−14.3. The equation $f(x) = 0$ has a real root at $x = 2.2$ (see Fig. 3.1).

The following C++ program performs NR iterations to obtain the root.

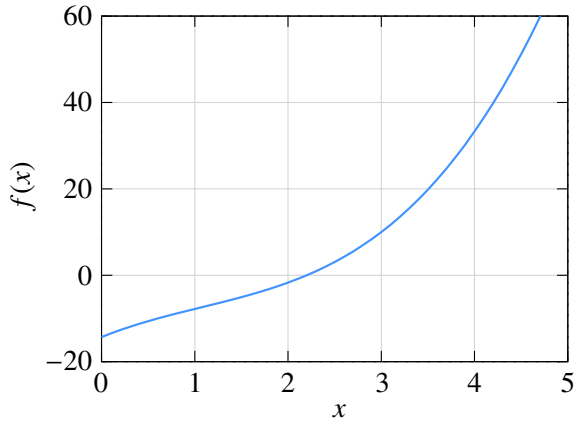


Figure 3.1: Plot of $f(x)$ given by Eq. 3.3.

```
#include <iostream>
#include <iomanip>
#include <math.h>
using namespace std;

int main ()
{
   double x,f,dfdx,delx,tolr,r;
   double a3,a2,a1,a0;
   double x2,x3;

   a3 = 1.0; a2 = -3.2; a1 = 8.7; a0 = -14.3;

   x = 4.0;        // initial guess
   tolr = 1.0e-8;  // tolerance
   r = 2.2;        // actual solution

   for (int i=0; i < 10; i++) {
     x2 = x*x; x3 = x2*x;              // powers of x
     f = a3*x3 + a2*x2 + a1*x + a0;    // function
     dfdx = 3.0*a3*x2 + 2.0*a2*x + a1; // derivative
     delx = -f/dfdx;                   // correction delta_x

     cout << std::setw(2) << i << "  ";
     cout << std::scientific;
     cout << x << "  " << f << "  " << delx << "  " << (x-r) << endl;

     if (fabs(f) < tolr) break;        // tolerance met; exit loop
     x = x + delx;                     // update x
   }
   return 0;
}
```

The output of the program is shown in Table 3.1. Note how quickly the NR process converges to the root. After three iterations, we already have an accuracy of 0.44%. This rapid convergence is the reason for the popularity of the NR method. Near convergence, the "errors" for iterations i and $(i + 1)$ are related by

$$\epsilon^{(i+1)} = k\left[\epsilon^{(i)}\right]^2 , \tag{3.4}$$

i	$x^{(i)}$	$f(x^{(i)})$	$\Delta x^{(i)}$	$(x^{(i)} - r)$	$(x^{(i)} - r)/r$
0	4.000000×10^{0}	3.330000×10^{1}	-1.070740×10^{0}	1.800000×10^{0}	8.181818×10^{-1}
1	2.929260×10^{0}	8.861467×10^{0}	-5.646248×10^{-1}	7.292605×10^{-1}	3.314820×10^{-1}
2	2.364636×10^{0}	1.601388×10^{0}	-1.548606×10^{-1}	1.646356×10^{-1}	7.483436×10^{-2}
3	2.209775×10^{0}	8.966910×10^{-2}	-9.739489×10^{-3}	9.774978×10^{-3}	4.443172×10^{-3}
4	2.200035×10^{0}	3.243738×10^{-4}	-3.548854×10^{-5}	3.548901×10^{-5}	1.613137×10^{-5}
5	2.200000×10^{0}	4.282173×10^{-9}	$-4.685091 \times 10^{-10}$	4.685092×10^{-10}	2.129587×10^{-10}

Table 3.1: The NR process for finding the root of the $f(x) = 0$ where $f(x)$ is given by Eq. 3.3.

where $\epsilon^{(i)} = \left|x^{(i)} - r\right|$ is the deviation of $x^{(i)}$ from the root r. The factor $k \approx g''(r)/2$, i.e., $\left.\frac{1}{2}\frac{d^2 f}{dx^2}\right|_{x=r}$. Eq. 3.4 explains why the error goes down so dramatically as the NR process converges. Because of the second power in Eq. 3.4, the NR process is said to have *quadratic convergence*.

3.2 Extension to set of equations

The NR method can be generalised to a system of N equations in N variables given by

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_N) &= 0, \\ f_2(x_1, x_2, \ldots, x_N) &= 0, \\ &\vdots \\ f_N(x_1, x_2, \ldots, x_N) &= 0. \end{aligned} \tag{3.5}$$

In this case, we define a solution *vector*,

$$\mathbf{x^{(i)}} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ .. \\ \\ x_N^{(i)} \end{bmatrix}. \tag{3.6}$$

To start the NR process, we start with an initial guess for the solution vector[1], i.e., $x_1^{(0)}, x_2^{(0)}, \cdots, x_N^{(0)}$. The correction vector in the i^{th} iteration, $\Delta\mathbf{x}^{(i)}$ is computed as

$$\Delta x^{(i)} = -\left[\mathbf{J}^{(i)}\right]^{-1} \mathbf{f}^{(i)}, \tag{3.7}$$

[1] In practice, it is often difficult to come up with a good initial guess, and in the absence of a better alternative, $x_1^{(i)} = 0$, $x_2^{(i)} = 0, \cdots$ may be used.

where

$$\mathbf{f}^{(i)} = \begin{bmatrix} f_1(\mathbf{x}^{(i)}) \\ f_2(\mathbf{x}^{(i)}) \\ .. \\ f_N(\mathbf{x}^{(i)}) \end{bmatrix}, \quad \mathbf{J}^{(i)} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & .. & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & .. & \frac{\partial f_2}{\partial x_N} \\ .. & .. & .. & \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & .. & \frac{\partial f_N}{\partial x_N} \end{bmatrix}, \tag{3.8}$$

and the functions and derivatives are evaluated at the current values, $x_1^{(i)}, x_2^{(i)}, \cdots, x_N^{(i)}$. The NR procedure is otherwise similar to that for the one variable case (see Fig. 3.2).

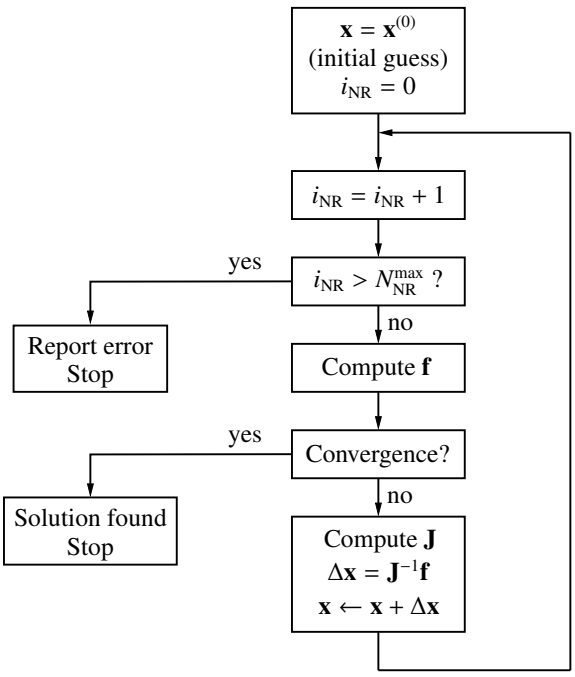


Figure 3.2: Flow chart for the Newton-Raphson procedure.

3.3 Convergence criteria

In the NR method, we need to set a "convergence criterion" to determine when to stop the NR iterations. In the program of Sec. 3.1, for example, the variable `tolr` (tolerance) served this purpose. The following convergence criteria are commonly used.

(a) Norm of $\mathbf{f}$: In this case, we check if the function values are small. Typically, the 2-norm, defined as

$$\| \mathbf{f} \|_2 = \left[\sum_{i=1}^{N} f_i^2 \right]^{1/2}, \tag{3.9}$$

is computed, and the NR iterations are said to converge if $\|\mathbf{f}\|_2 < \epsilon$, a suitable tolerance value. This is an *absolute* convergence criterion since our goal is precisely to solve the set of

equations to get $f_i = 0$ (for each i) which means in practice that $|f_i|$ (or somewhat equivalently, the 2-norm) should be made as small as possible.

(b) Norm of $\Delta\mathbf{x}$: Here, we check if each component of the correction vector is sufficiently small, i.e., $|\Delta x_i| < \epsilon_i$, where ϵ_i may be 0.01 mV for all variables of type voltage, and 1 nA for all variables of type current, for example. This is a *relative* criterion and is based on the fact that, as the NR process converges, Δx_i become smaller and smaller, as seen in the one-variable example earlier (see Table 3.1).

(c) SPICE convergence criterion: In SPICE, a tolerance is computed for each variable as follows:

$$\tau_i = k_{\text{rel}} \times \max\left(|x_i^{(k)}|, |x_i^{(k+1)}|\right) + \tau_{\text{abs}}, \tag{3.10}$$

where k_{rel} (typically 0.001) and τ_{abs} are constants, and $x_i^{(k)}$ denotes the value of x_i in the k^{th} iteration. The first term specifies a *relative* tolerance, specific to the variable x_i, while the second term is an *absolute* tolerance. If x is of type voltage, τ_{abs} may be 0.01 mV, for example. Convergence is said to be attained if

$$|x_i^{(k+1)} - x_i^{(k)}| < \tau_i. \tag{3.11}$$

In a variety of electronic circuits, including oscillatory circuits, the tried and tested SPICE convergence criterion is found to work well.

Why are there so many different convergence criteria? Isn't there a simple "universal" convergence criterion which we can use for all problems? To answer this question, let us take a closer look at convergence of the NR process.

As we have seen earlier, the "error," i.e., the difference between the numerical solution and the actual solution, goes down dramatically with each iteration, as the NR process converges. If our computer had infinite precision, the error can be reduced to arbitrarily small values simply by performing additional NR iterations. In practice, computers have a finite precision. With single-precision (32-bit) numbers, the smallest number that can be represented is about 1.2×10^{-38}, and the largest number is $3.4 \times 10^{+38}$. With double-precision (64-bit) numbers, the smallest and largest numbers are 5.0×10^{-324} and $1.8 \times 10^{+308}$, respectively. Furthermore, because of the finite number of bits used for the mantissa, only a finite number of real numbers can be represented, say, $r_1, r_2, r_3, \cdots$ Any number falling between r_k and r_{k+1} is rounded off to r_k or r_{k+1}, leading to a "round-off error" which is of the order of 10^{-8} for single-precision numbers and 10^{-16} for double-precision numbers.

The round-off error, however small, is finite, and it limits the accuracy that we can achieve with the NR method. If our convergence check is too stringent, convergence will not be attained, and the NR process will get terminated with an error message (although the solution may already be sufficiently accurate). If it is too loose, we end up with the wrong solution. Setting an appropriate convergence criterion is therefore crucial in implementing the NR method, as illustrated in the following example.

Consider the systems of equations,

$$\begin{aligned} f_1(x_1, x_2) &\equiv k \times (x_1 + x_2 - 6\sqrt{3}) = 0, \\ f_2(x_1, x_2) &\equiv 10x_1^2 - x_2^2 + 45 \qquad = 0. \end{aligned} \tag{3.12}$$

We want to solve this system of equations with the initial guess $x_1 = 1$, $x_2 = 1$. With this initial guess, the NR method converges to the solution[2] $x_1 = \sqrt{3}$, $x_2 = 5\sqrt{3}$. The results of applying the NR method to Eq. 3.12 using single- and double-precision numbers are shown in Fig. 3.3 and Table 3.2. We can make the following observations from the results.

(a) For this system of equations, $\|\mathbf{f}\|_2$ does not keep reducing indefinitely with each NR iteration; it saturates at some point.

(b) For the same k, the NR method can achieve higher accuracy (smaller $\|\mathbf{f}\|_2$) when double precision is used.

(c) For the same precision (single or double), higher accuracy can be achieved for a smaller value of k although the actual solution does not depend on k at all (see Eq. 3.12).

(d) Although the lowest achievable value of $\|\mathbf{f}\|_2$ depends on k and on the precision used, the solution is already accurate up to the seventh decimal place at the end of iteration no. 7 in all four cases.

Clearly, an arbitrarily small 2-norm cannot be set as the convergence criterion. For example, with double-precision numbers, a 2-norm of 10^{-12} will work (i.e., the NR process will exit after attaining convergence) for $k = 1$, but not with $k = 10^5$. This means that selection of the convergence criterion must be made differently for different problems! In reality, the situation is not so hopeless. For example, if we are only interested in electronic circuits, the default set of convergence criteria in SPICE (see [9], for example) would generally work well and may need to be tweaked only for a few specific simulations.

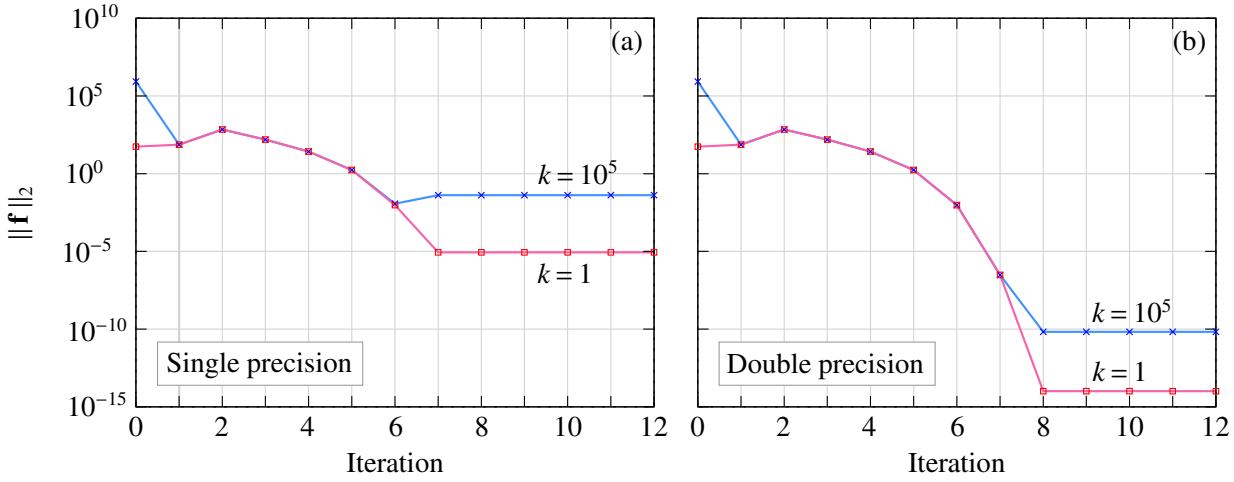


Figure 3.3: $\|\mathbf{f}\|_2$ versus NR iteration number for the system of equations given by Eq. 3.12: (a) single-precision arithmetic, (b) double-precision arithmetic.

[2]The system of equations given by Eq. 3.12 actually has two real roots; but only the root $x_1 = \sqrt{3}$, $x_2 = 5\sqrt{3}$ is relevant for our discussion, considering the initial guess we have used.

	single precision			
	$k = 1$		$k = 10^5$	
i	x_1	x_2	x_1	x_2
0	0.10000000×10^1	0.10000000×10^1	0.10000000×10^1	0.10000000×10^1
1	-0.69160879×10^0	0.11083913×10^2	-0.69160879×10^0	0.11083913×10^2
2	0.80743456×10^1	0.23179598×10^1	0.80743456×10^1	0.23179598×10^1
3	0.39112959×10^1	0.64810090×10^1	0.39112959×10^1	0.64810090×10^1
4	0.22007749×10^1	0.81915302×10^1	0.22007749×10^1	0.81915302×10^1
5	0.17647886×10^1	0.86275158×10^1	0.17647886×10^1	0.86275158×10^1
6	0.17322344×10^1	0.86600704×10^1	0.17322344×10^1	0.86600704×10^1
7	0.17320508×10^1	0.86602545×10^1	0.17320508×10^1	0.86602545×10^1
8	0.17320508×10^1	0.86602545×10^1	0.17320508×10^1	0.86602545×10^1
9	0.17320508×10^1	0.86602545×10^1	0.17320508×10^1	0.86602545×10^1
10	0.17320508×10^1	0.86602545×10^1	0.17320508×10^1	0.86602545×10^1
	double precision			
	$k = 1$		$k = 10^5$	
0	0.10000000×10^1	0.10000000×10^1	0.10000000×10^1	0.10000000×10^1
1	-0.69160865×10^0	0.11083913×10^2	-0.69160865×10^0	0.11083913×10^2
2	0.80743398×10^1	0.23179650×10^1	0.80743398×10^1	0.23179650×10^1
3	0.39112931×10^1	0.64810118×10^1	0.39112931×10^1	0.64810118×10^1
4	0.22007739×10^1	0.81915309×10^1	0.22007739×10^1	0.81915309×10^1
5	0.17647886×10^1	0.86275163×10^1	0.17647886×10^1	0.86275163×10^1
6	0.17322344×10^1	0.86600705×10^1	0.17322344×10^1	0.86600705×10^1
7	0.17320508×10^1	0.86602540×10^1	0.17320508×10^1	0.86602540×10^1
8	0.17320508×10^1	0.86602540×10^1	0.17320508×10^1	0.86602540×10^1
9	0.17320508×10^1	0.86602540×10^1	0.17320508×10^1	0.86602540×10^1
10	0.17320508×10^1	0.86602540×10^1	0.17320508×10^1	0.86602540×10^1

Table 3.2: The NR process for solving the system of equations given by Eq. 3.12 using single- and double-precision numbers.

3.4 Graphical interpretation of the NR process

In the one-variable case, the NR process has a useful graphical interpretation. The correction $\Delta x^{(i)}$ in the i^{th} NR iteration is given by

$$\Delta x^{(i)} = -\frac{f(x^{(i)})}{\left.\frac{df}{dx}\right|_{x^{(i)}}}. \tag{3.13}$$

Since $\left.\frac{df}{dx}\right|_{x^{(i)}}$ is the slope of the $f(x)$ curve at $x = x^{(i)}$, the magnitude of $\Delta x^{(i)}$ is given by drawing a tangent at the point $(x^{(i)}, f(x^{(i)}))$ and extending it to the x-axis, as shown in Fig. 3.4. $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$ is then obtained by simply going from $x^{(i)}$ in the negative x-direction if the sign of $\left.\frac{df}{dx}\right|_{x^{(i)}}$ is positive (and *vice versa*) a distance of $\Delta x^{(i)}$. This leads to the following interpretation of the NR process.

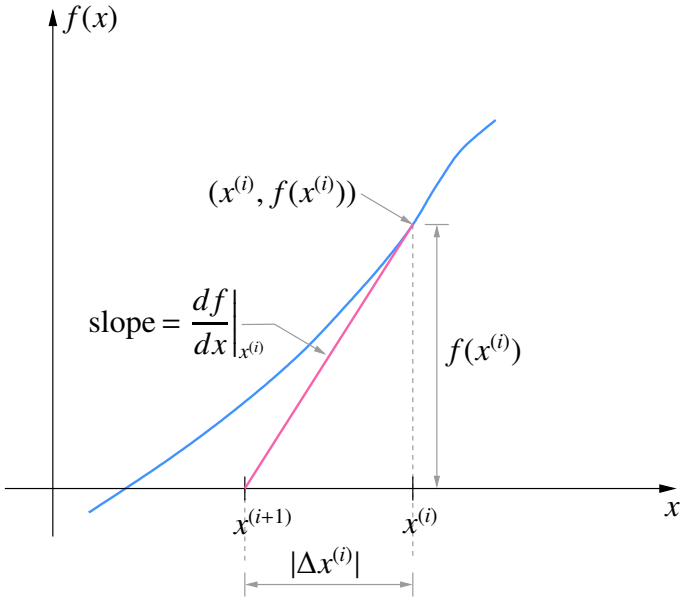


Figure 3.4: Graphical interpretation of the NR process.

1. Draw a tangent at $(x^{(i)}, f(x^{(i)}))$.
2. Extend the tangent to the x-axis.
3. The point of intersection of the tangent with the x-axis gives the next values of x, i.e., $x^{(i+1)}$.

Fig. 3.5 illustrates the NR process for Eq. 3.3. It is easy to see that, if $f(x)$ is linear (i.e., $f(x) = k_1 x + k_2$), the NR process will converge in exactly one iteration.

3.5 Convergence issues

We have seen that the NR method has the desirable property of rapid convergence. The big question is whether it will always converge. Unfortunately, convergence of the NR method is guaranteed only

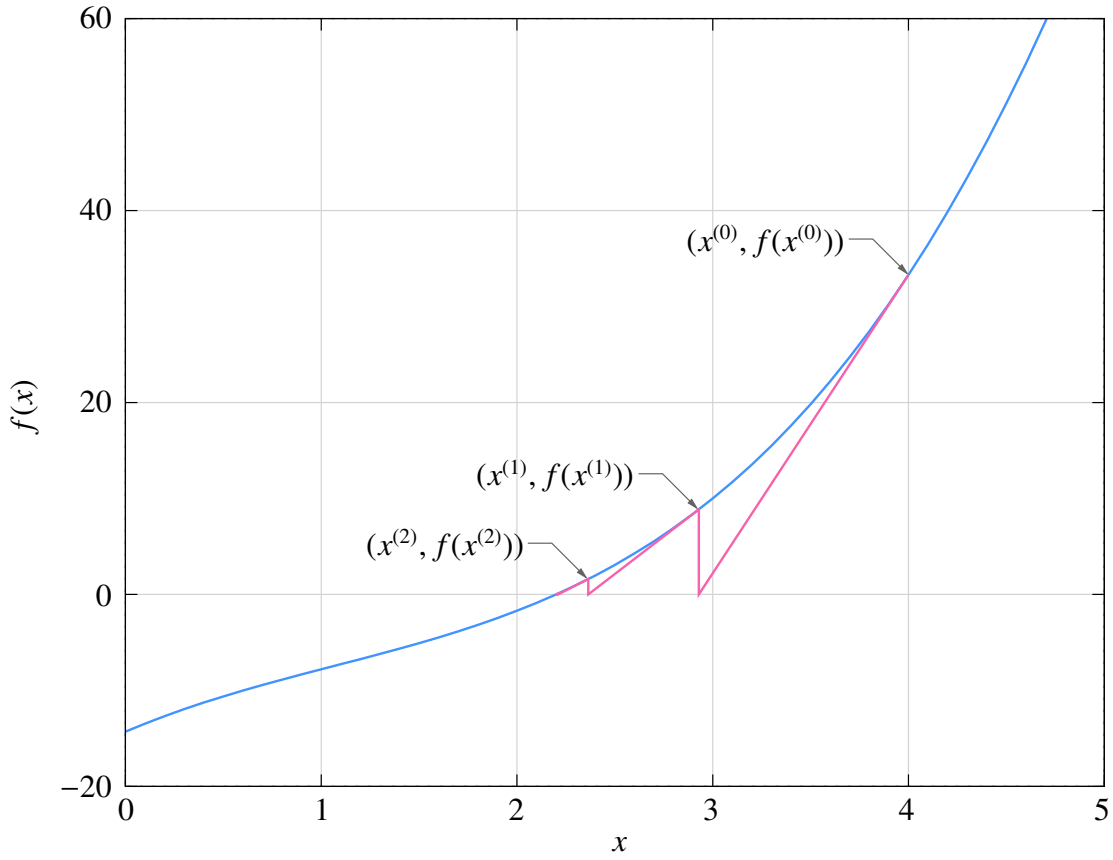


Figure 3.5: Graphical interpretation of NR process with $f(x)$ given by Eq. 3.3 and $x^{(0)} = 4.0$.

if the initial guess is sufficiently close to the solution (root). In the one-variable case, it can be shown that, if

$$\frac{|f(x)f''(x)|}{(f'(x))^2} < 1 \tag{3.14}$$

for some interval (x_1, x_2) containing the root r, the NR method will converge for an initial guess $x^{(0)}$ lying in that interval. If not, the NR process may not converge.

As an example, consider

$$f(x) = \tan^{-1}(x - a). \tag{3.15}$$

For this function,

$$f_1(x) \equiv \frac{f(x)f''(x)}{(f'(x))^2} = -2(x - a)\tan^{-1}(x - a). \tag{3.16}$$

Figs. 3.6 (a) and 3.6 (b) show plots of $f(x)$ and $f_1(x)$, respectively, for $a = 1.5$. For $|f_1(x)| < 1$, we need $0.735 < x < 2.265$. If the initial guess is within this range, the NR process for $f(x)$ is guaranteed to converge (see Fig. 3.7, for example); otherwise, it may not converge (see Fig. 3.8, for example). An equally catastrophic situation, in which the NR process oscillates around the root, is shown in Fig. 3.9.

Failure of the NR procedure is not a hypothetical calamity; it is a very real possibility in circuit simulation. Fortunately, some clever ways have been devised to nudge the NR process toward convergence as discussed in the following.

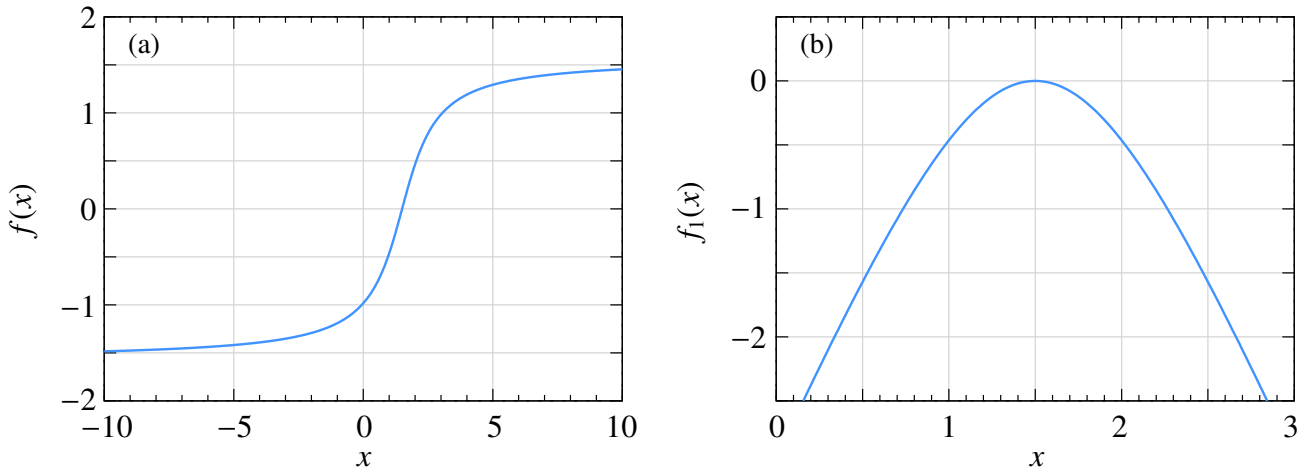


Figure 3.6: (a) $f(x)$ (Eq. 3.15) and (b) $f_1(x)$ (Eq. 3.16) versus x

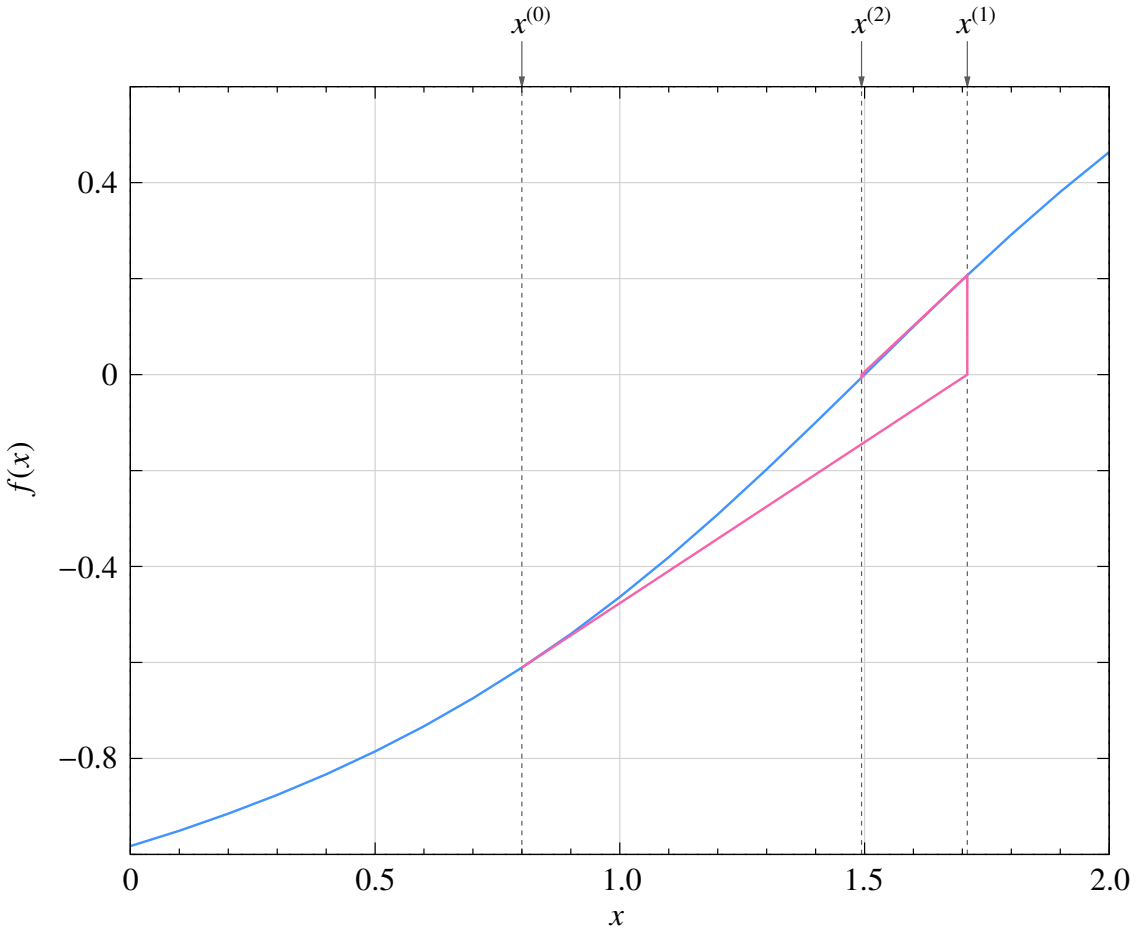


Figure 3.7: NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 0.8$.

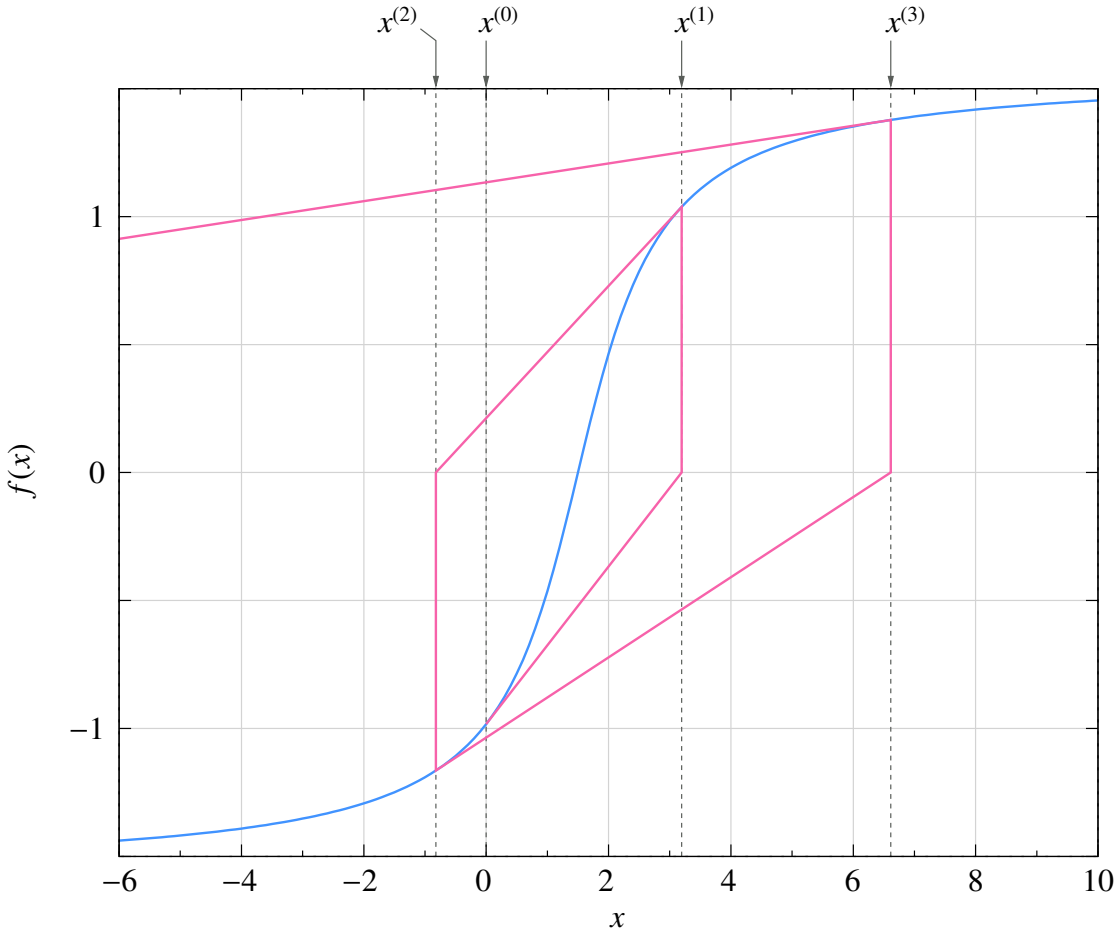


Figure 3.8: NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 0$.

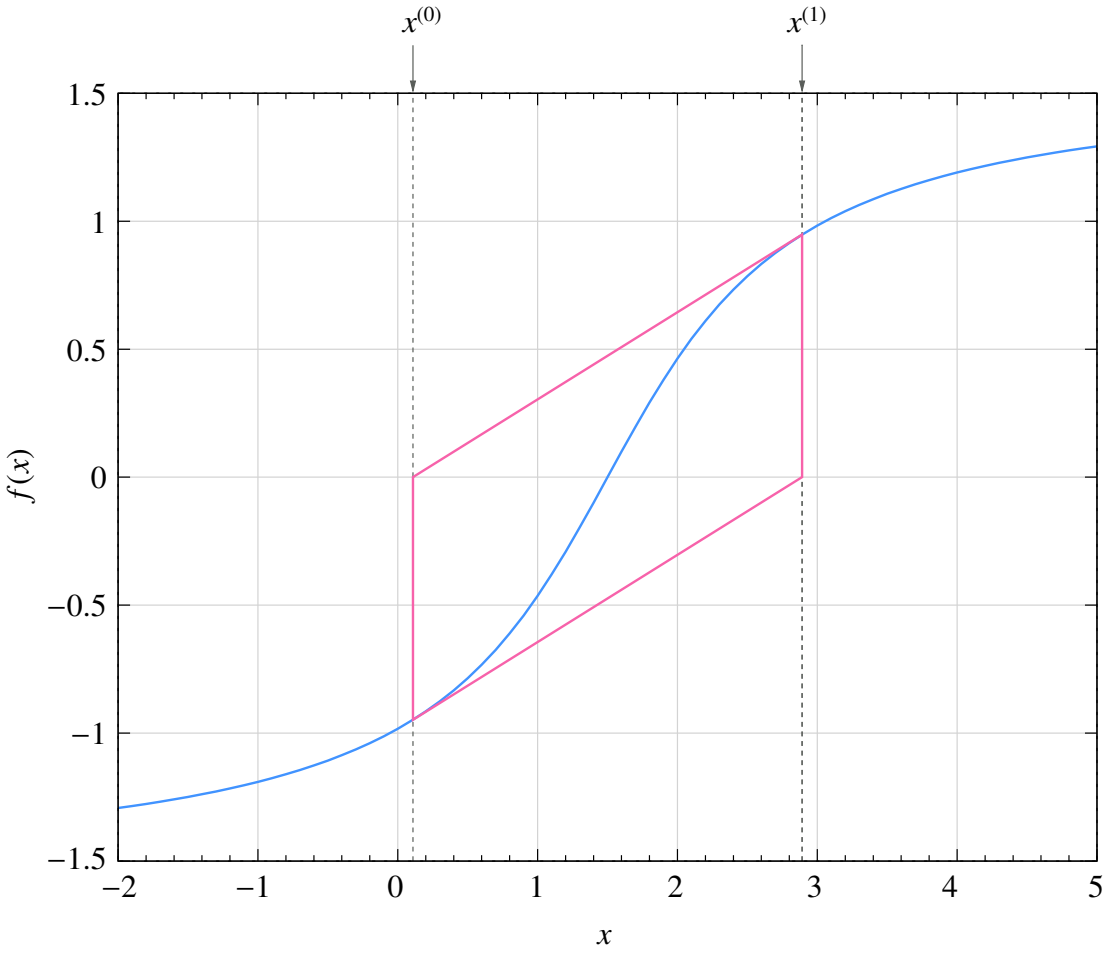


Figure 3.9: NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 2.89175$.

3.5.1 Damping of the NR iterations

Consider the one variable case. As we have seen earlier, the NR method is related to the Taylor series of a function around the current value $x^{(i)}$:

$$f(x^{(i)} + \Delta x^{(i)}) = f(x^{(i)}) + \Delta x^{(i)} \left.\frac{df}{dx}\right|_{x^{(i)}} + \text{higher-order terms.} \tag{3.17}$$

If the higher-order terms are small, the NR method is expected to work well. Convergence problems can arise when they are not small. To be specific, let us look at the example of Fig. 3.8 in which the NR process diverges. The slope at $(x^{(i)}, f(x^{(i)}))$ corresponds to the first term of the Taylor series, and the curvature is due to the higher-order terms. We note that the slope does take us in the correct *direction*[3] (i.e., toward the root), but because of the curvature, we end up going too far in that direction. The idea behind damping of the NR process is to play safe and go only part of the way.

In the standard NR process, the correction vector is computed as $\Delta x^{(i)} = -\left[\mathbf{J}^{(i)}\right]^{-1} \mathbf{f}^{(i)}$ and is added to the current solution vector to obtain the next guess:

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \Delta \mathbf{x}^{(i)}. \tag{3.18}$$

We can dampen or slow down the NR process by adding only a fraction of the correction vector, i.e.,

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + k \times \Delta \mathbf{x}^{(i)}, \quad (0 < k < 1), \tag{3.19}$$

where k is the "damping factor."

Fig. 3.10 shows the effect of damping for the example shown Fig. 3.8 with the same initial guess, viz., $x^{(0)} = 0$. In each iteration, we draw a tangent at $(x^{(i)}, f(x^{(i)}))$ as before, but instead of going all the way to the intercept with the x-axis (the dashed line), we go only a fraction of the way to obtain the next iterate $x^{(i+1)}$. The NR process is now seen to converge to the solution.

If damping it so effective, should we always use it? Not really. Although damping improves the chances of convergence, it slows down the NR process. Damping should therefore be used only if the standard NR process fails to converge. Fig. 3.11 shows the effect of k for $f(x) = \tan^{-1}(x)$ with $x^{(0)} = 1.5$. In this case, the standard NR method fails, and therefore damping is useful. When k is small, the convergence is slower. An excellent strategy is to use damping only in the first few NR iterations and then use the standard NR process (i.e., make $k = 1$) thereafter. In this way, we get convergence and also retain the quadratic convergence property of the NR process when damping is lifted. An example is shown in the same figure.

[3] In the one variable case, the direction is simply the positive or negative x-direction.

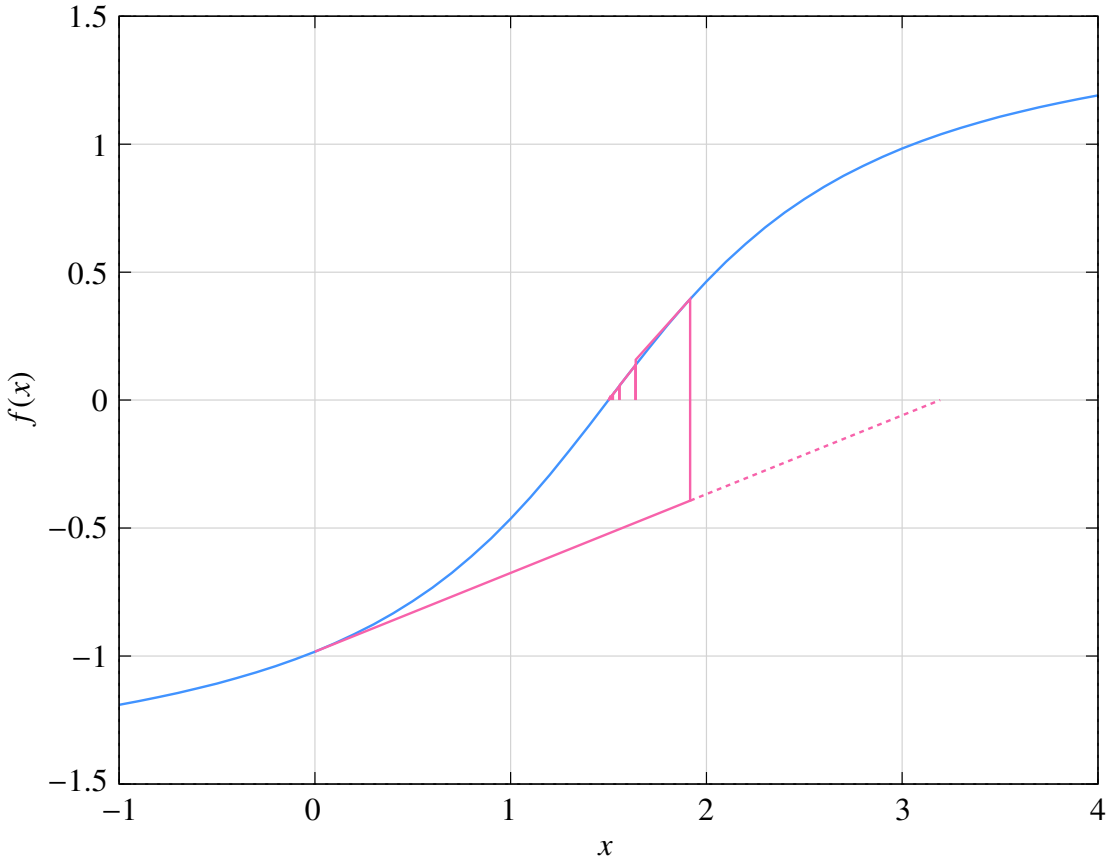


Figure 3.10: Damped NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 0$, and $k = 0.6$.

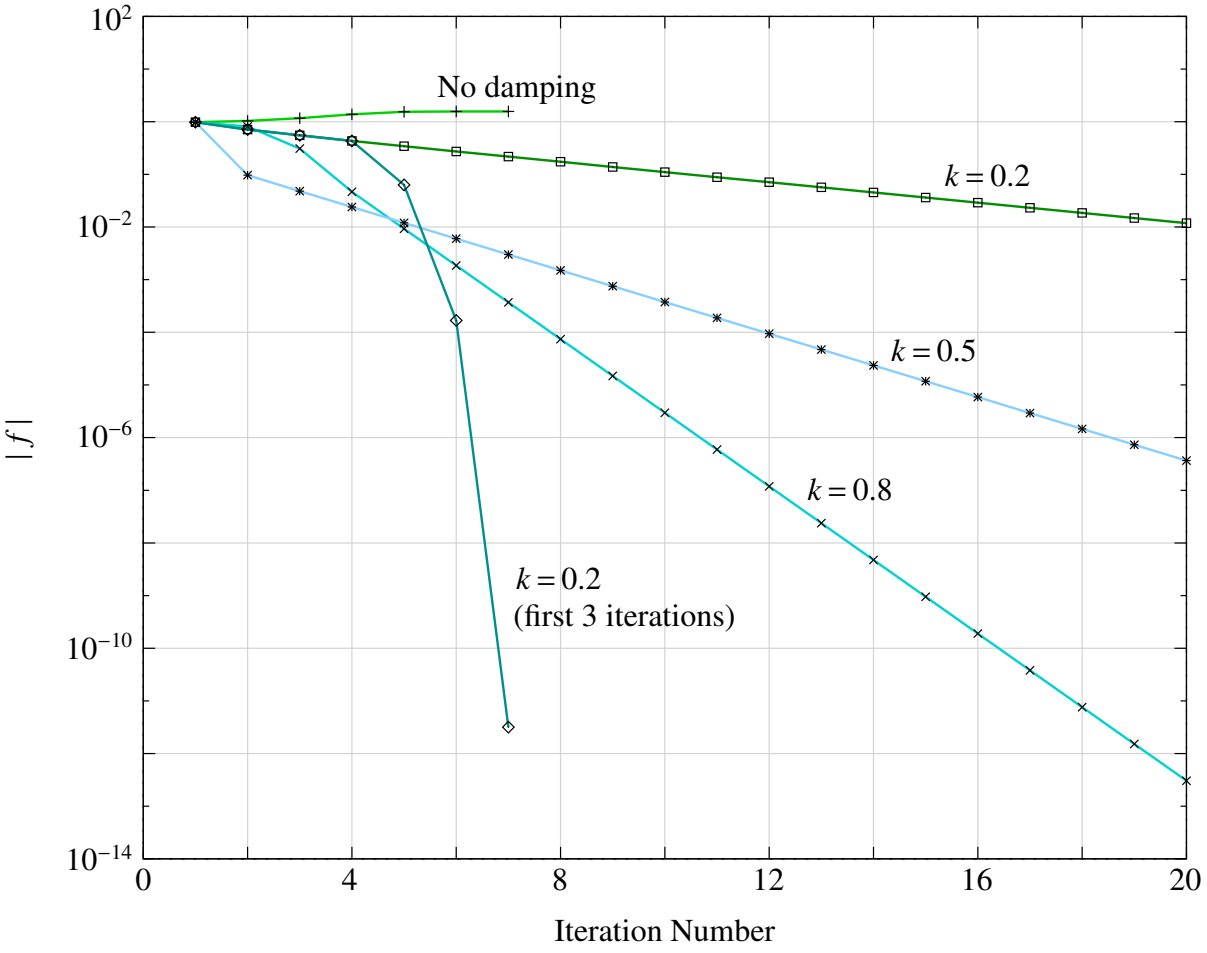


Figure 3.11: $|f|$ versus NR iteration number for different values of k for $f(x) = \tan^{-1}(x)$ with $x^{(0)} = 1.5$.

3.5.2 Parameter stepping

Suppose we try to solve $f(x) = 0$ with an initial guess $x^{(0)}$, and find that the standard NR method fails to converge. We can then construct another function $h(x) = f(x) + g(x)$, where $g(x)$ is a suitable "auxiliary" function. To be specific, let us consider $g(x) = kx$. To begin with, k is made sufficiently large (call it $k^{(0)}$), $h(x)$ then takes an approximately linear form $h^{(0)}(x) \approx k^{(0)}x$, and the NR method can be used effectively to solve $h^{(0)}(x) = 0$ without any convergence issue. Let us denote the solution obtained for $h^{(0)}(x) = 0$ by $r^{(0)}$. Next, we relax the parameter k in $g(x)$ to a smaller value $k^{(1)}$ and solve $h^{(1)}(x) \equiv f(x) + k^{(1)}x = 0$, using $r^{(0)}$ as the initial guess. Once again, the NR process is likely to converge if $k^{(1)}$ is sufficiently close to $k^{(0)}$. We repeat this process, making k progressively smaller. Finally, when k is negligibly small, $h(x) = f(x) + g(x) \approx f(x)$, and we have got the solution for our original problem, $f(x) = 0$.

The above procedure in which the parameter k is changed from a large value to zero (or a negligibly small value) in several steps may be called "parameter stepping." Fig. 3.12 shows an example.

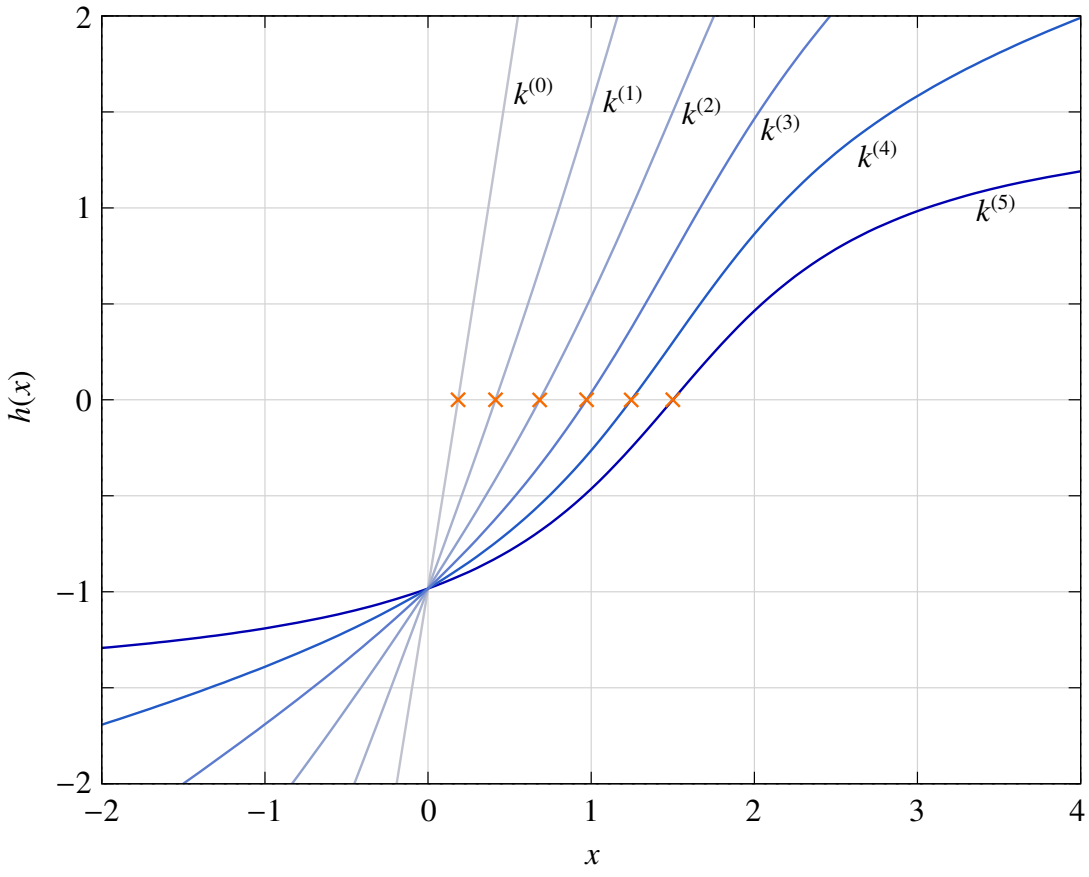


Figure 3.12: NR process for solving $f(x) \equiv \tan^{-1}(x - a) = 0$ with $a = 1.5$, and $x^{(0)} = 0$ as the initial guess. An auxiliary function $g^{(i)}(x) = k^{(i)}\,x$ is used, and $h^{(i)}(x) = f(x) + g^{(i)}(x)$ is solved with the NR method. $x = 0$ is used as the initial guess for solving $h^{(0)} = 0$. Thereafter, the solution $r^{(i-1)}$ for $h^{(i-1)} = 0$ is used as the initial guess for solving $h^{(i)} = 0$. The values of $k^{(i)}$ for $i = 0$ to 5 are 5, 2, 1, 0.5, 0.2, 0, respectively. The roots are denoted by crosses.

In electronic circuits, there are many situations in which no suitable initial guess is available.

Parameter stepping is useful in such cases. It can be carried out in different forms.

(a) $g_{\min}$ **stepping**: In this scheme, a conductance g (i.e., a resistance $1/g$) is added from each circuit node to ground[4], as shown in Fig. 3.13. If g is large (i.e., the resistance is small), the nonlinear devices are essentially bypassed, the circuit reduces to an approximately linear circuit, and the NR method converges easily. Using the solution so obtained as the initial guess, the same circuit with a lower value of g is then solved, and so on. Finally, when g is equal to $g_{\min}$ (a very small value such as 10^{-12} ℧, i.e., a resistance of $10^{12}\,\Omega$), we get the solution for the original circuit since the added resistances are as good as open circuits.

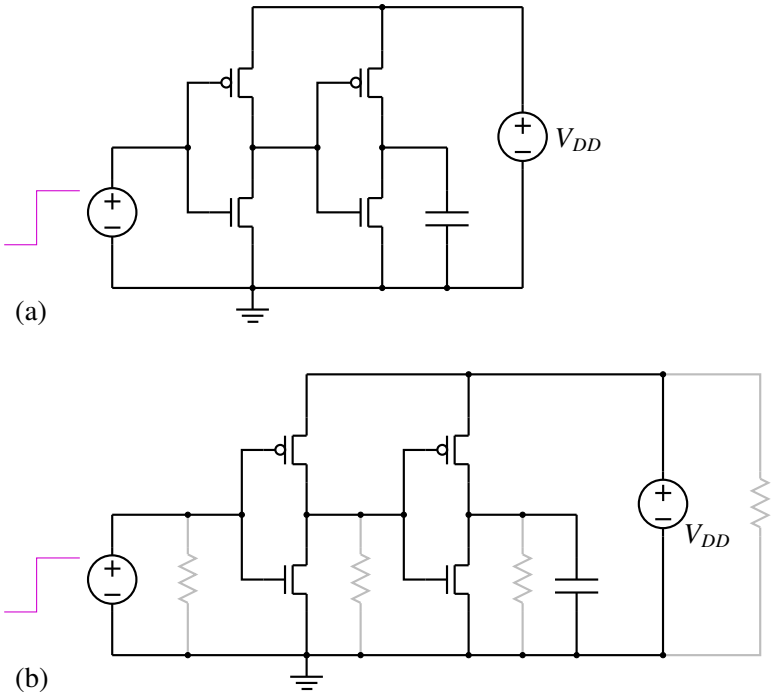


Figure 3.13: Illustration of $g_{\min}$ stepping: (a) original circuit, (b) circuit with a resistor added from each node to ground.

(b) **Source stepping**: In electronic circuits, there is a voltage supply (denoted typically by V_{CC} in BJT circuits and by V_{DD} in FET circuits) which "drives" the circuit. If this source voltage is made zero, all currents and voltages would become zero[5]. This suggests that, with V_{CC} (or V_{DD}) equal to zero, the NR method should have no trouble in converging to the solution with the simple initial guess of zero currents and voltages. Next, we increase V_{CC} by a small amount, say, 0.1 V. Since this situation is not substantially different, we once again expect the NR process to converge easily. Continuing this procedure, we finally obtain the solution for the actual source voltage, typically 5 V in BJT circuits. Since the parameter being stepped is a source voltage, we can refer to this procedure as "source stepping."

A variation of the above approach is "source ramping" in which the source voltage is ramped (in time) from 0 V to its final value in a suitable time interval, taking the solution obtained at a given time point as the initial guess for the next time point.

[4]or between each pair of nodes of the nonlinear devices
[5]There could be signal sources in the circuit (e.g., a BJT amplifier) which should also be made zero.

3.5.3 Limiting junction voltages

Semiconductor devices generally have one or more *p*-*n* junctions. In the actual solution for the circuit under consideration, the voltage across a junction is limited to about 0.8 V which corresponds to a few Amps. However, during the NR process, some of the junction voltages can become larger than the values expected in the solution, causing the current – which is proportional to e^{V/V_T} – to blow up. For example, with $V = 2$ V and $V_T = 26$ mV, e^{V/V_T} is of the order of 10^{33}. When that happens, the NR method comes to a grinding halt because of numerical overflow. It is important therefore to limit the junction voltages in the NR process. The strategy used in SPICE for this purpose is shown in Fig. 3.14. The junction voltages in iterations i and $(i+1)$ are denoted by V_{old} and V_{new}, respectively. The "critical voltage" V_{crit} in the flow chart is a fixed voltage at which the exponential factor e^{V/V_T} becomes impractically large.

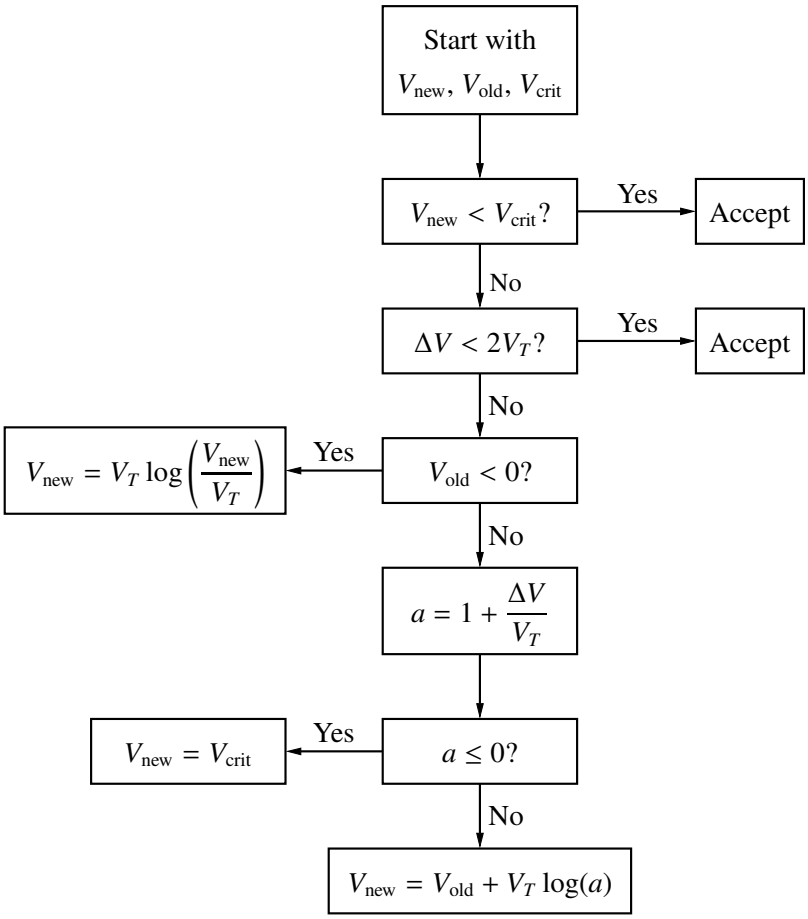


Figure 3.14: Flow chart for limiting junction voltages in SPICE. V_T is the thermal voltage, and $\Delta V = V_D^{(i+1)} - V_D(i)$.

3.5.4 Changing time step

In transient (or "dynamic") simulation, the time axis is discretised (see Fig. 3.15), and the circuit equations are solved at discrete time points $t_0, t_1, t_2, \cdots, t_n, t_{n+1}$, all the way up to the last time point of interest t_{end}. The solution at t_n serves as the initial guess for the NR process at t_{n+1}. If t_{n+1} is

sufficiently close to t_n, we expect the NR process at t_{n+1} to converge easily. If we perform a fixed number of NR iterations and find that the NR process has not converged, we can reduce the time step $\Delta t = t_{n+1} - t_n$, i.e., bring t_{n+1} closer to t_n. In other words, we now look for $\mathbf{x}_{n+1}$ which is closer to $\mathbf{x}_n$, and that improves the chances of convergence. We will visit this topic again in Chapter 4.

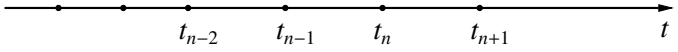


Figure 3.15: Discretisation of the time axis.

3.6 Nonlinear circuits

In combination with the Modified Nodal Analysis (MNA) approach for assembling the circuit equations (see Chapter 2), the NR method can be used to obtain the solution – currents and voltages – for a nonlinear circuit. Consider the circuit shown in Fig. 3.16. The diode current can be written using the Shockley equation as

$$I_D = I_s\left(e^{V_D/V_T} - 1\right) = I_s\left(e^{V_2/V_T} - 1\right) \equiv I_D(V_2), \tag{3.20}$$

where I_s is the reverse saturation current of the diode (typically of the order of pA for low-power diodes), and $V_T = kT/q$ is the thermal voltage (about 25 mV at room temperature). Using the MNA approach, we can assemble the circuit equations as

$$\begin{aligned} \text{KCL at B}: \quad & G_1(V_1 - V_2) + I_s = 0\,, \\ \text{KCL at C}: \quad & G_1(V_2 - V_1) + G_2 V_2 + I_D(V_2) = 0\,, \\ \text{Voltage source equation}: \quad & V_1 - V_0 = 0\,. \end{aligned} \tag{3.21}$$

The above set of equations can be solved with the NR method, starting with a suitable initial guess for the three variables, viz., V_1, V_2, and I_s.

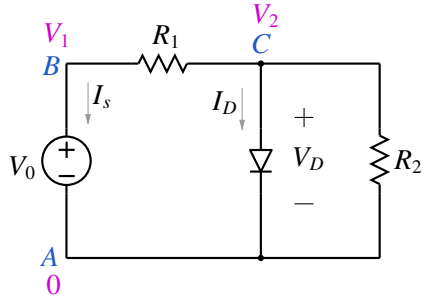


Figure 3.16: A nonlinear circuit example.

In a similar manner, the NR scheme can be used for transient simulation. More about that later, after we cover numerical solution of ODEs.

Chapter 4

Discretisation of Time Derivatives

In transient (dynamic) simulation, we are interested in the behaviour of a circuit or a system in a time interval from t_{start} to t_{end}. To obtain the numerical solution, the interval of interest is divided into sub-intervals (see Fig. 4.1), and the circuit equations are solved at each time point. If the circuit elements do not involve time derivatives, transient simulation is no different than DC simulation. For example, consider the circuit shown in Fig. 4.2. To obtain the solution for this circuit at a given time point t_k, all we need to do is to find $V_s(t_k)$, replace the AC source with a DC source $V_s = V_s(t_k)$, and simulate the circuit using the MNA method discussed in Chapter 2.

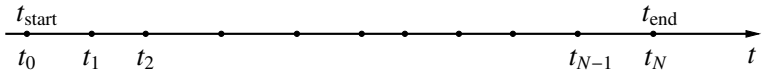


Figure 4.1: Discretisation of the time interval from t_{start} to t_{end}.

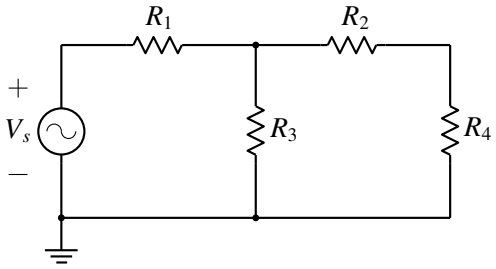


Figure 4.2: A circuit with a time-dependent voltage source.

The situation is different when time derivatives are involved, e.g., in the form of capacitors or inductors. As the first step, we need to replace each of the time derivatives – a continuous function – with a discretised approximation. Two broad sets of methods are available for this purpose: explicit and implicit methods. In the following, we look at these methods in some detail.

4.1 Explicit Methods

We start with a single ODE,

$$\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0. \tag{4.1}$$

We are interested in getting a numerical solution i.e., the *discrete* values $x_1, x_2, \cdots$, corresponding to t_1, t_2, etc. We will denote the exact solution by $x(t)$; e.g., $x(t_1)$ means the exact solution at t_1. On the other hand, we will denote the *numerical* solution at t_1 by x_1.

4.1.1 Forward Euler Method

At $t = t_0$, we start with $x = x_0$ (the initial condition). From Eq. 4.1, we can compute the slope of $x(t)$ at t_0 which is simply $f(t_0, x_0)$. In the Forward Euler (FE) scheme, we make the approximation that this slope applies to the entire interval (t_0, t_1), i.e., from the current time point to the next time point. With this assumption, $x(t_1)$ is given by $x_1 = x_0 + (t_1 - t_0) \times f(t_0, x_0)$, as shown in Fig. 4.3. Similarly, from x_1, we can get x_2, and so on. In general, we have

$$x_{n+1} = x_n + \Delta t_n f(t_n, x_n), \tag{4.2}$$

where $\Delta t_n = t_{n+1} - t_n$.

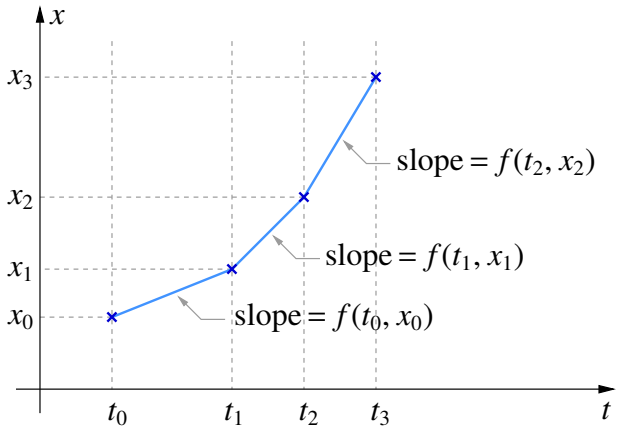


Figure 4.3: Illustration of the Forward Euler method.

Let us apply the FE method to the ODE,

$$\frac{dx}{dt} = a\,(\sin \omega t - x)\,, \quad x(0) = 0, \tag{4.3}$$

which has the analytical (exact) solution,

$$x(t) = \frac{a\omega}{a^2 + \omega^2}\,(e^{-at} - \cos \omega t) + \frac{a^2}{a^2 + \omega^2}\,\sin \omega t. \tag{4.4}$$

The following C program can be used to obtain the numerical solution of Eq. 4.3 with the FE method.

```
#include<stdio.h>
#include<math.h>

int main()
{
  double t,t_start,t_end,h,x,x0,f;
  double a,w;
  FILE *fp;
```

```
  x0=0.0; t_start=0.0; t_end=5.0; h=0.05;
  a=1.0; w=5.0;

  fp=fopen("fe1.dat","w");

  t=t_start; x=x0;
  fprintf(fp,"%13.6e %13.6e\n",t,x);

  while (t <= t_end) {
    f = a*(sin(w*t)-x);
    x = x + h*f;
    t = t + h;
    fprintf(fp,"%13.6e %13.6e\n",t,x);
  }
  fclose(fp);
}
```

Fig. 4.4 shows the numerical solution[1] along with the analytical (exact) solution given by Eq. 4.4. We notice some difference between the two, but it can be made smaller[2] by using a smaller step size h. In general, the accuracy of a numerical method for solving ODEs is described by the "order" of the method which in turn depends on the "Local Truncation Error" (LTE) for that method. The LTE is a measure of the "local" error (i.e., error made in a single time step) and is defined as (see [10]),

$$\text{LTE} = x(t_{n+1}) - u_{n+1}, \tag{4.5}$$

where $x(t_{n+1})$ is the exact solution at t_{n+1}, and u_{n+1} is the solution obtained by the numerical method, starting with the exact solution $x(t_n)$ at $t = t_n$. If our numerical method were perfect, u_{n+1} would be the same as $x(t_{n+1})$, and the LTE would be zero.

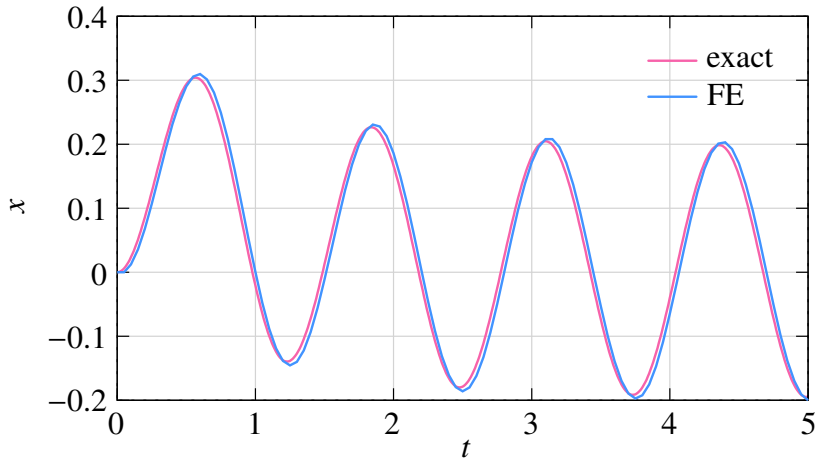


Figure 4.4: Analytical solution (red curve) and numerical solution obtained with the FE method (blue curve) for the ODE given by Eq. 4.3.

[1]The numerical solution appears to be continuous; however, in reality, it consists of discrete points which are generally connected with line segments as a "guide to the eye."

[2]Readers new to numerical analysis should try this out by running the program with a smaller value of h, e.g., $h = 0.02$. In these matters, there is no substitute for hands-on experience.

Let us look at the LTE of the FE method for the test equation[3],

$$\frac{dx}{dt} = -\lambda x, \quad x(0) = 1, \quad \lambda > 0, \tag{4.6}$$

with the exact solution

$$x = e^{-\lambda t}. \tag{4.7}$$

To compute the LTE, we take a specific t_n, say, $t_n = t_0 = 0$, and compute x_{n+1} (i.e., x_1) by performing one step of the FE method, starting with $x_0 = 1$. The exact solution at $t = t_{n+1} = h$ is $x(h) = e^{-\lambda h}$. The LTE is the difference between x_1 and $x(h)$.

Fig. 4.5 shows the LTE (magnitude) as a function of time step h. As h is reduced by a factor of 10 (from 10^{-1} to 10^{-2}, for example), the LTE goes down by two orders of magnitude. In other words, the LTE varies as h^2, and therefore the FE method is said to be of order 1. In general, if the LTE $\sim h^{k+1}$ for a numerical method, then it is said to be of order k.

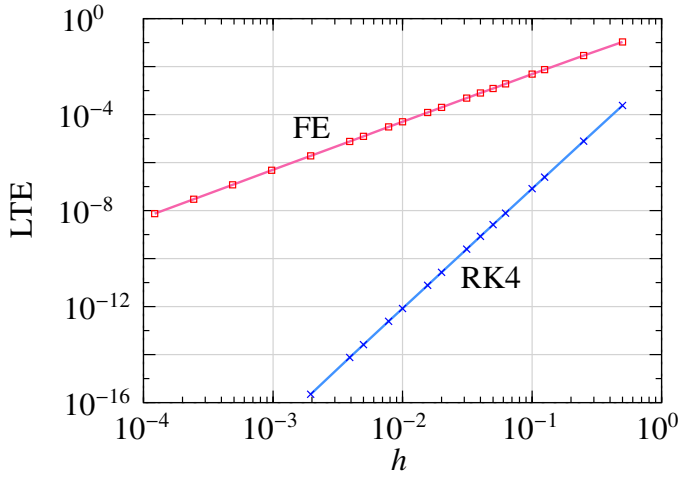


Figure 4.5: LTE versus step size h for the FE and RK4 methods in solving the ODE given by Eq. 4.6.

4.1.2 Runge-Kutta method of order 4

The Runge-Kutta method of order 4 is given by[4]

$$\begin{aligned} f_0 &= f(t_n, x_n), \\ f_1 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2} f_0), \\ f_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2} f_1), \\ f_3 &= f(t_n + h, x_n + h f_2), \\ x_{n+1} &= x_n + h\left(\frac{1}{6} f_0 + \frac{1}{3} f_1 + \frac{1}{3} f_2 + \frac{1}{6} f_3\right). \end{aligned} \tag{4.8}$$

[3]Eq. 4.6 is commonly used to discuss various aspects of numerical methods for solving ODEs.

[4]There are other Runge-Kutta methods of order 4 (see [11]); the method described here is called the "classic" form and is commonly used.

The order of the RK4 method can be made out from Fig. 4.5: if h is reduced by one order of magnitude, the LTE goes down by five orders of magnitude (2.5 divisions, with each division corresponding to two decades), and therefore the order is four.

Since the RK4 method is of a higher order, we expect it to be more accurate than the FE method. Let us check that by comparing the numerical solutions obtained with the two methods for the ODE given by Eq. 4.3. The following program can be used for the RK4 method.

```
#include<stdio.h>
#include<math.h>

int main()
{
  double t,t_start,t_end,h,x,x0;
  double f0,f1,f2,f3;
  double a,w;
  FILE *fp;

  x0=0.0; t_start=0.0; t_end=5.0; h=0.05;
  a=1.0; w=5.0;

  fp=fopen("rk4.dat","w");

  t=t_start; x=x0;
  fprintf(fp,"%13.6e %13.6e\n",t,x);

  while (t <= t_end) {
    f0 = a*(sin(w*t)-x);
    f1 = a*(sin(w*(t+0.5*h))-(x+0.5*h*f0));
    f2 = a*(sin(w*(t+0.5*h))-(x+0.5*h*f1));
    f3 = a*(sin(w*(t+h))-(x+h*f2));

    x = x + (h/6.0)*(f0+f1+f1+f2+f2+f3);
    t = t + h;
    fprintf(fp,"%13.6e %13.6e\n",t,x);
  }
  fclose(fp);
}
```

Fig. 4.6 shows the numerical solutions obtained with the FE and RK4 methods using a step size of $h = 0.05$. Clearly, the RK4 method is more accurate.

Both FE and RK4 are *explicit* methods in the sense that x_{n+1} can be computed simply by evaluating quantities based on the past values of x (see Eqs. 4.2 and 4.8). The RK4 method is more complicated as it involves computation of the function values f_0, f_1, f_2, f_3, but the entire computation can be completed in a step-by-step manner. For example, the computation of f_2 requires only x_n and f_1, which are already available. The simplicity of the computation is reflected in the programs we have seen.

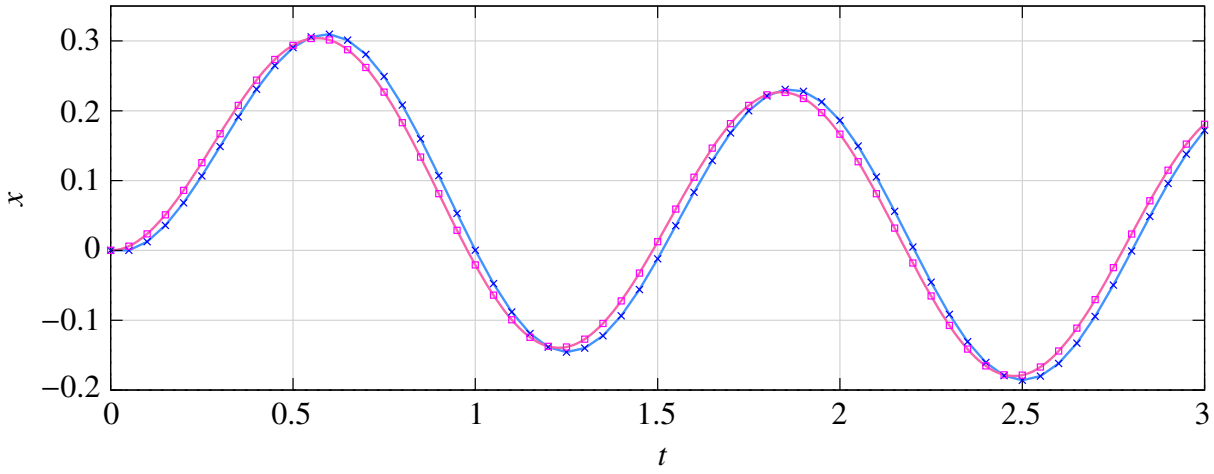


Figure 4.6: Numerical solution of Eq. 4.3 obtained with the FE and RK4 methods (crosses and squares, respectively) with a step size of $h = 0.05$. The exact solution is also shown (red curve).

4.1.3 System of ODEs

The above methods (and other explicit methods) can be easily extended to a set of ODEs of the form

$$\begin{aligned}
\frac{dx_1}{dt} &= f_1(t, x_1, x_2, \cdots, x_N), \\
\frac{dx_2}{dt} &= f_2(t, x_1, x_2, \cdots, x_N), \\
&\vdots \\
\frac{dx_N}{dt} &= f_N(t, x_1, x_2, \cdots, x_N),
\end{aligned} \tag{4.9}$$

with the initial conditions at $t = t_0$ specified as $x_1(t_0) = x_1^{(0)}$, $x_2(t_0) = x_2^{(0)}$, etc. With vector notation, we can write the above set of equations as

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}), \quad \mathbf{x}(t_0) = \mathbf{x}^{(0)}. \tag{4.10}$$

The FE method for this set of ODEs can be written as

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + h\mathbf{f}(t_n, \mathbf{x}^{(n)}), \tag{4.11}$$

and the RK4 method as

$$\begin{aligned}
\mathbf{f}_0 &= \mathbf{f}(t_n, \mathbf{x}^{(n)}), \\
\mathbf{f}_1 &= \mathbf{f}(t_n + \frac{h}{2}, \mathbf{x}^{(n)} + \frac{h}{2}\mathbf{f}_0), \\
\mathbf{f}_2 &= \mathbf{f}(t_n + \frac{h}{2}, \mathbf{x}^{(n)} + \frac{h}{2}\mathbf{f}_1), \\
\mathbf{f}_3 &= \mathbf{f}(t_n + h, \mathbf{x}^{(n)} + h\mathbf{f}_2), \\
\mathbf{x}^{(n+1)} &= \mathbf{x}^{(n)} + h\left(\frac{1}{6}\mathbf{f}_0 + \frac{1}{3}\mathbf{f}_1 + \frac{1}{3}\mathbf{f}_2 + \frac{1}{6}\mathbf{f}_3\right),
\end{aligned} \tag{4.12}$$

where $\mathbf{x}^{(n)}$ denotes the solution vector at time t_n. As in the single-equation case, the evaluations can be performed in a step-by-step manner, enabling a straightforward implementation.

The simplicity of explicit methods makes them very attractive. Furthermore, the implementation remains easy even if the functions f_i are nonlinear. Let us illustrate this point with an example. Consider the system of ODEs given by

$$\begin{aligned} \frac{dx_1}{dt} &= a_1 (\sin \omega t - x_1)^3 - a_2 (x_1 - x_2), \\ \frac{dx_2}{dt} &= a_3 (x_1 - x_2), \end{aligned} \tag{4.13}$$

with the initial condition, $x_1(0) = 0$, $x_2(0) = 0$. If we use the FE method to solve the equations, we get

$$\begin{aligned} x_1^{(n+1)} &= x_1^{(n)} + h\left[a_1 \left(\sin \omega t_n - x_1^{(n)}\right)^3 - a_2 \left(x_1^{(n)} - x_2^{(n)}\right)\right], \\ x_2^{(n+1)} &= x_2^{(n)} + h\left[a_3 \left(x_1^{(n)} - x_2^{(n)}\right)\right]. \end{aligned} \tag{4.14}$$

The following program, which is a straightforward extension of our earlier FE program, can be used to implement the above equations.

```
#include<stdio.h>
#include<math.h>

int main()
{
   double t,t_start,t_end,h;
   double x1,x2,f1,f2,b1;
   double w,a1,a2,a3,f_hz,pi;
   FILE *fp;

   f_hz = 5.0e3; // frequency = 5 kHz
   pi = acos(-1.0);
   w = 2.0*pi*f_hz;

   a1 = 5.0e3;
   a2 = 5.0e3;
   a3 = 5.0e3;

   t_start=0.0;
   t_end=5.0e-3;
   h=0.001e-3; // time step = 0.001 msec

   x1=0.0;
   x2=0.0;

   fp=fopen("fe3.dat","w");

   t=t_start;
   fprintf(fp,"%13.6e %13.6e %13.6e\n",t,x1,x2);

   while (t <= t_end) {
     b1 = sin(w*t)-x1;
```

```
    f1 = a1*b1*b1*b1 - a2*(x1-x2);
    f2 = a3*(x1-x2);
    x1 = x1 + h*f1;
    x2 = x2 + h*f2;
    t = t + h;
    fprintf(fp,"%13.6e %13.6e %13.6e\n",t,x1,x2);
  }
  fclose(fp);
}
```

As simple as that! The numerical solution is shown in Fig. 4.7.

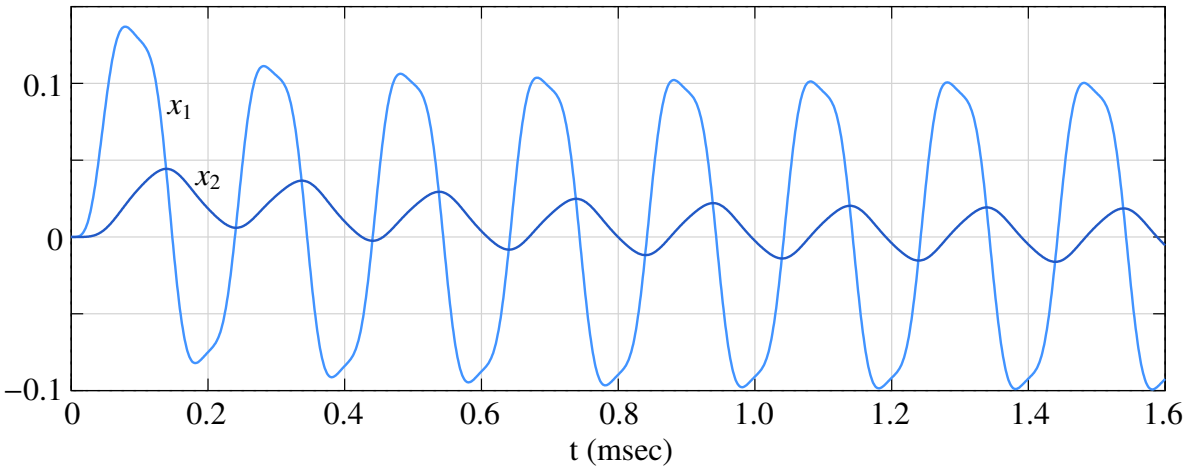


Figure 4.7: Numerical solution of Eq. 4.13 obtained with the FE method with a step size of $h = 1\,\mu$ sec.

4.1.4 Adaptive time step

We have so far considered the time step h to be constant. When the solution has regions of fast and slow variations, it is much more efficient to use adaptive (variable) time steps. When the solution varies rapidly, small time steps are required to capture the transients accurately; at other times, larger time steps can be used. In this way, the total number of time points – and thereby the simulation time – can be made much smaller than using a small, uniform time step.

In order to implement an adaptive time step scheme, we need a mechanism to judge the accuracy of the numerical solution. Ideally, we would like to use the difference between the analytical solution and the numerical solution, i.e., $|x(t_n) - x_n|$. However, since the analytical solution is not available, we need some other means of checking the accuracy. A commonly used method is to estimate the local truncation error by computing x_{n+1} with two numerical methods: one method of order p and the other of order $p+1$. Let $\text{LTE}^{(p)}$ and $\text{LTE}^{(p+1)}$ denote the local truncation errors in going from t_n to t_{n+1} for the two methods, and let x_{n+1} and $\tilde{x}_{n+1}$ be the corresponding numerical solutions. As we have seen, $\text{LTE}^{(p)}$ and $\text{LTE}^{(p+1)}$ are $O(h^{p+1})$ and $O(h^{p+2})$, respectively. If we assume[5] x_n to be equal to the exact solution $x(t_n)$, we have

$$\text{LTE}^{(p)} = x(t_{n+1}) - x_{n+1}, \tag{4.15}$$

$$\text{LTE}^{(p+1)} = x(t_{n+1}) - \tilde{x}_{n+1}. \tag{4.16}$$

[5]In reality, x_n and $x(t_n)$ would not be the same, but Eq. 4.17 remains valid (see [10]).

Subtracting Eq. 4.16 from Eq. 4.15, we get

$$\mathrm{LTE}^{(p)} - \mathrm{LTE}^{(p+1)} = \tilde{x}_{n+1} - x_{n+1}\,. \tag{4.17}$$

Since $\mathrm{LTE}^{(p+1)}$ is expected to be much smaller than $\mathrm{LTE}^{(p)}$, we can ignore it and obtain

$$\mathrm{LTE}^{(p)} \approx \tilde{x}_{n+1} - x_{n+1}\,. \tag{4.18}$$

Having obtained an estimate for the LTE (denoted by $\mathrm{LTE}^{\mathrm{est}}$) resulting from a time step of $h_n = t_{n+1} - t_n$, we can now check if the solution should be accepted or not. If $\mathrm{LTE}^{\mathrm{est}}$ is larger than the specified tolerance, we reject the current step and try a smaller step. If $\mathrm{LTE}^{\mathrm{est}}$ is smaller than the tolerance, we accept the solution obtained at t_{n+1} (i.e., x_{n+1}). In this case, there is a possibility that our current time step h_n is too conservative, and we explore whether the next time step (i.e., $h_{n+1} = t_{n+2} - t_{n+1}$) can be made larger.

Fig. 4.8 shows a flow chart for implementing adaptive time steps based on the above ideas. The tolerance τ specifies the maximum value of the LTE per time step (i.e., LTE/h) that is acceptable. The method of order p is used for actually advancing the solution, and the method of order $p+1$ is used only to compute $\mathrm{LTE}^{\mathrm{est}}$ using Eq. 4.18. Since LTE/h is $O(h^p)$, we can write

$$\frac{\mathrm{LTE}^{(p)}}{h_n} = \frac{|\tilde{x}_{n+1} - x_{n+1}|}{h_n} = K h_n^p. \tag{4.19}$$

Next, we compute the time step ($\equiv \delta \times h_n$) which would result in an LTE per time step equal to τ, using

$$\tau = K(\delta h_n)^p. \tag{4.20}$$

From Eqs. 4.19 and 4.20, we obtain δ as[6]

$$\delta = \left(\frac{\tau h_n}{|\tilde{x}_{n+1} - x_{n+1}|}\right)^{1/p}. \tag{4.21}$$

If the current LTE per time step (in going from t_n to t_{n+1}) is larger than τ (i.e., $\delta < 1$), we reject the current time step and try a new time step $h_n \leftarrow \delta \times h_n$. If it is smaller than τ, we accept the current solution (x_{n+1}). In this case, δ is larger than one, and the next time step is taken to be $h_{n+1} = \delta \times h_n$.

Since drastic changes in the time step are not suitable from the stability perspective (see [10]), the value of δ is generally restricted to $\delta_{\min} \le \delta \le \delta_{\max}$, as shown in the flow chart. Minimum and maximum limits are also imposed on the step size ($h_{\min}$ and $h_{\max}$ in the flow chart).

Different pairs of methods – of orders p and $p+1$ – are available in the literature for implementing the above scheme. In the commonly used Runge-Kutta-Fehlberg (RKF45) pair, which consists of an order-4 method and an order-5 method, the LTE is estimated from [12]

$$\begin{aligned} x_{n+1} &= x_n + h_n\left(\frac{25}{216}f_0 + \frac{1408}{2565}f_2 + \frac{2197}{4104}f_3 - \frac{1}{5}f_4\right), \\ \tilde{x}_{n+1} &= x_n + h_n\left(\frac{16}{135}f_0 + \frac{6656}{12825}f_2 + \frac{28561}{56430}f_3 - \frac{9}{50}f_4 + \frac{2}{55}f_5\right), \end{aligned} \tag{4.22}$$

[6] Typically, δ is made a little smaller than that given by Eq. 4.21, see [12]. Also, note that controlling the LTE (rather than LTE/h) is also possible, and in that case Eq. 4.21 gets replaced by $\delta = \left(\frac{\tau}{|\tilde{x}_{n+1} - x_{n+1}|}\right)^{1/(p+1)}$.

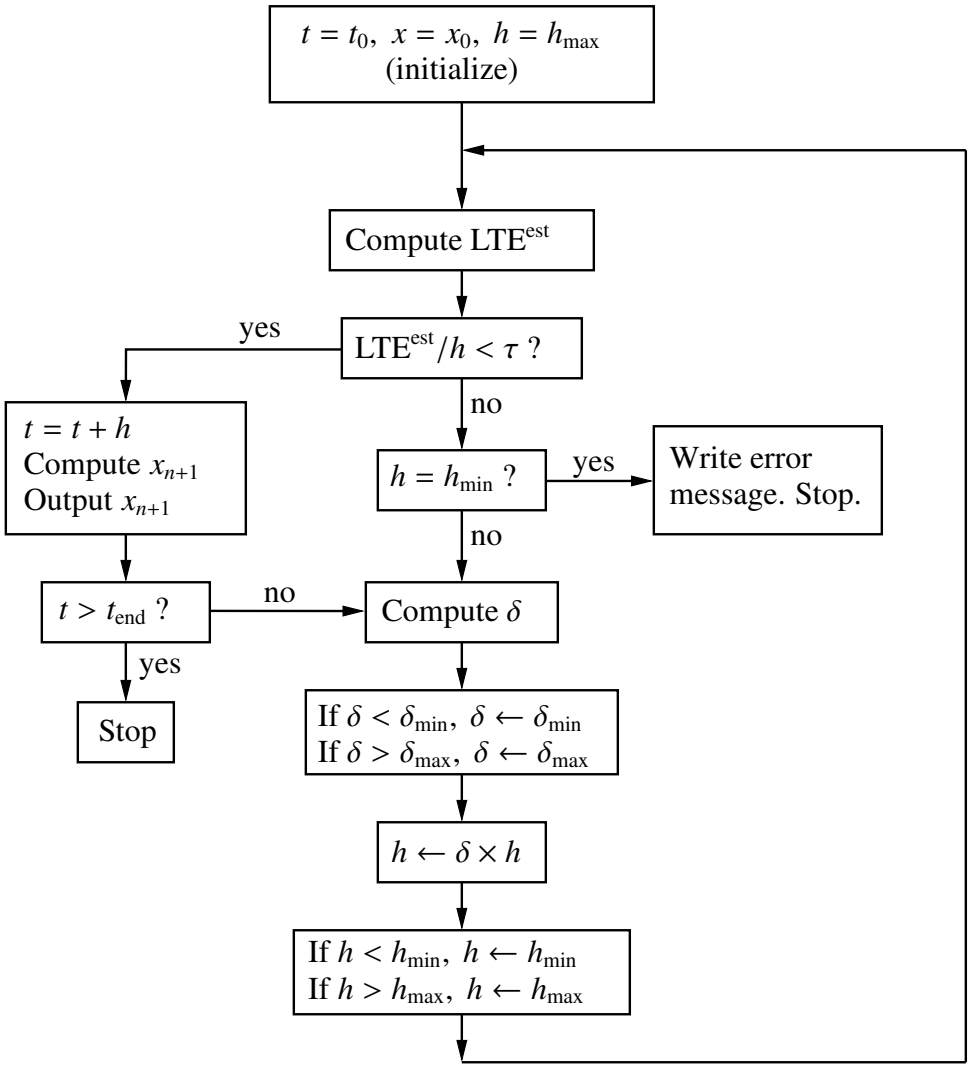


Figure 4.8: Flow chart for adaptive time step selection based on computation of local truncation error.

where

$$
\begin{aligned}
f_0 &= f(t_n, x_n),\\
f_1 &= \left(t_n + \frac{h_n}{4}, x_n + \frac{1}{4} f_0\right),\\
f_2 &= \left(t_n + \frac{3h_n}{8}, x_n + \frac{3}{32} f_0 + \frac{9}{32} f_1\right),\\
f_3 &= \left(t_n + \frac{12h_n}{13}, x_n + \frac{1932}{2197} f_0 - \frac{7200}{2197} f_1 + \frac{7296}{2197} f_2\right),\\
f_4 &= \left(t_n + h_n, x_n + \frac{439}{216} f_0 - 8 f_1 + \frac{3680}{513} f_2 - \frac{845}{4104} f_3\right),\\
f_5 &= \left(t_n + \frac{h_n}{2}, x_n - \frac{8}{27} f_0 + 2 f_1 - \frac{3544}{2565} f_2 + \frac{1859}{4104} f_3 - \frac{11}{40} f_4\right).
\end{aligned}
\tag{4.23}
$$

Note that the order-4 part of this method – the computation of x_{n+1} in Eq. 4.22 – is different from the classic RK4 method we have seen earlier (Eq. 4.8). The order-4 and order-5 methods in the RKF45 scheme are designed such that, with little extra computation (over the order-4 method), we get the order-5 result ($\tilde{x}_{n+1}$ in Eq. 4.22).

Let us illustrate the effectiveness of the RKF45 method with an example. Consider the *RC* circuit shown in Fig. 4.9. The behaviour of this circuit is described by the ODE,

$$\frac{dv_c}{dt} = \frac{1}{\tau}(v_s - v_c), \tag{4.24}$$

where $\tau = RC$, and v_s is a known function of time. In particular, we will consider $v_s(t)$ to be a pulse going from 0 V to 1 V at $t_1 = 0.5$ sec with a rise time of 0.05 sec and from 1 V back to 0 V at $t_2 = 2$ sec with a fall time of 0.05 sec. Fig. 4.10 shows the solution of Eq. 4.24 obtained with the RKF45 method with a tolerance $\tau = 10^{-4}$. As we expect, the time steps are small when the solution is changing rapidly (i.e., near the pulse edges) and large when it is changing slowly.

The tolerance τ needs to be chosen carefully. If it is large, it may not give us a sufficiently accurate solution. For example, with $\tau = 10^{-2}$, the solution differs significantly from that obtained with $\tau = 10^{-4}$, as shown in Fig. 4.11. On the other hand, reducing τ beyond 10^{-4} does not change the solution any more[7] (see Fig. 4.12), but it does add more time points. Fig. 4.13 shows the total number of time points (N_{total}) used by the RKF45 method in covering the time interval from t_{start} to t_{end} (0 to 10 sec) as a function of τ. With a tighter tolerance (smaller τ), the number of time points to be simulated goes up and so does the simulation time. For this specific problem, we would not even notice the difference in the computation time, but for larger problems (with a large number of variables or a large number of time points or both), the difference could be substantial.

Sometimes, there is an "aesthetics" issue. What has aesthetics got to do with science, we may ask. Consider the same *RC* circuit of Fig. 4.9 with a sinusoidal voltage source $V_m \sin \omega t$. Fig. 4.14 shows the results obtained with $\tau = 10^{-5}$ and $\tau = 10^{-6}$. In the former case, the solution is accurate, i.e., the numerical solution agrees closely with the analytical solution. However, it appears discontinuous because of the small number of time points. In the latter case, the RKF45 method forces a larger number of time points (smaller time steps) in order to meet the tolerance requirement, and the solution now appears continuous, more in tune with what we want to see.

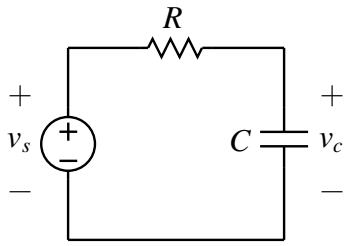


Figure 4.9: *RC* circuit example.

Explicit methods such as RK4 (with a fixed time step) and RKF45 (with variable/auto time steps) are commonly used to solve several engineering problems of interest. They are attractive since the implementation is so simple and straightforward.

[7]We are being somewhat lax about terminology here – When we say that the solution does not change, what we mean is, "I cannot make out the difference, if any" which is often good enough in practice. Looking through the microscope for a change in the fifth decimal place is generally not called for in engineering problems.

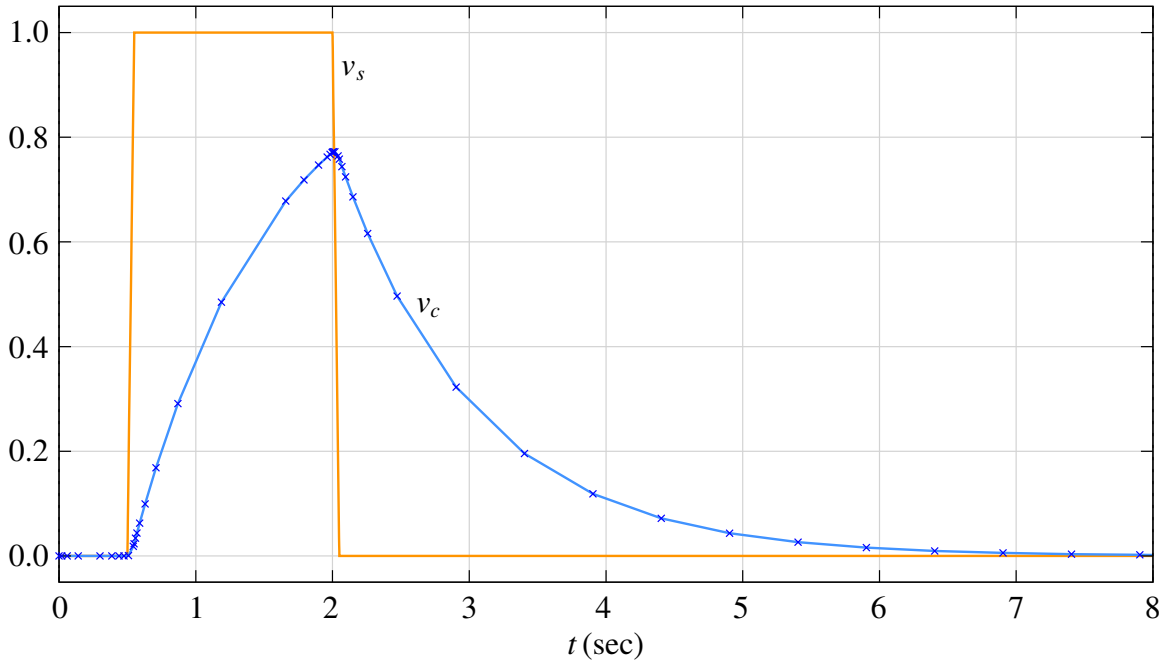


Figure 4.10: Numerical solution obtained for the *RC* circuit of Fig. 4.9 using the RKF45 method with $\tau = 10^{-4}$. The other parameters are $R = 1\,\Omega$, $C = 1\,\text{F}$, $\delta_{\min} = 0.2$, $\delta_{\max} = 2$, $h_{\min} = 10^{-5}$, $h_{\max} = 0.5$.

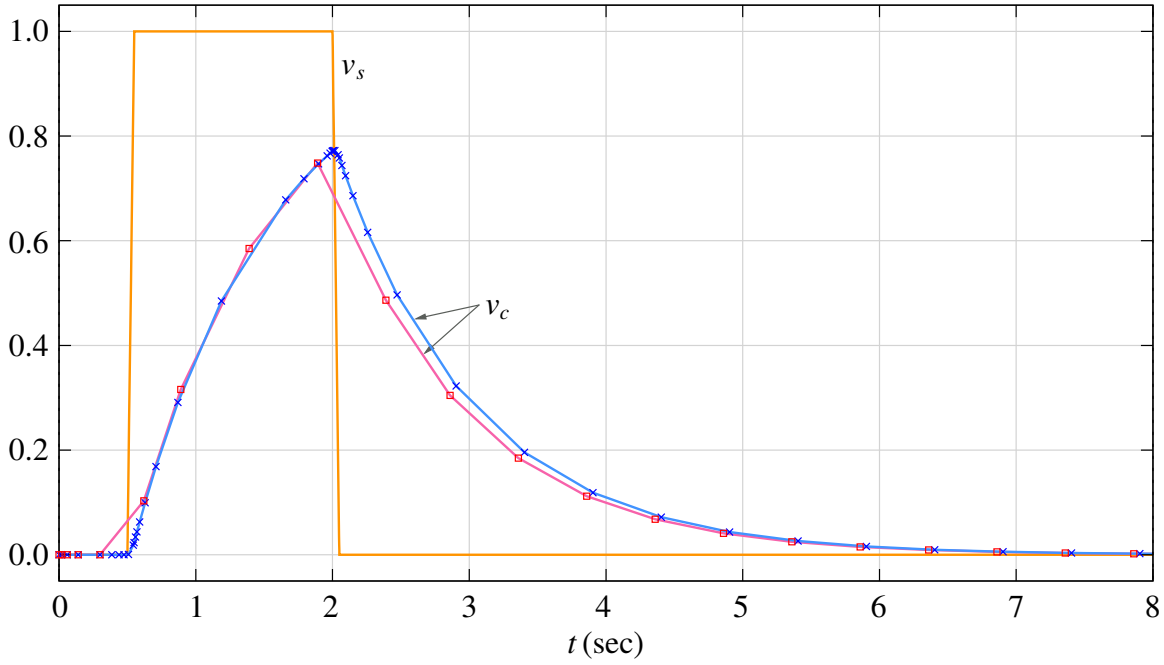


Figure 4.11: Numerical solutions obtained for the *RC* circuit of Fig. 4.9 using the RKF45 method with two values of τ: 10^{-2} (squares) and 10^{-4} (crosses). The other parameters are the same as in Fig. 4.10.

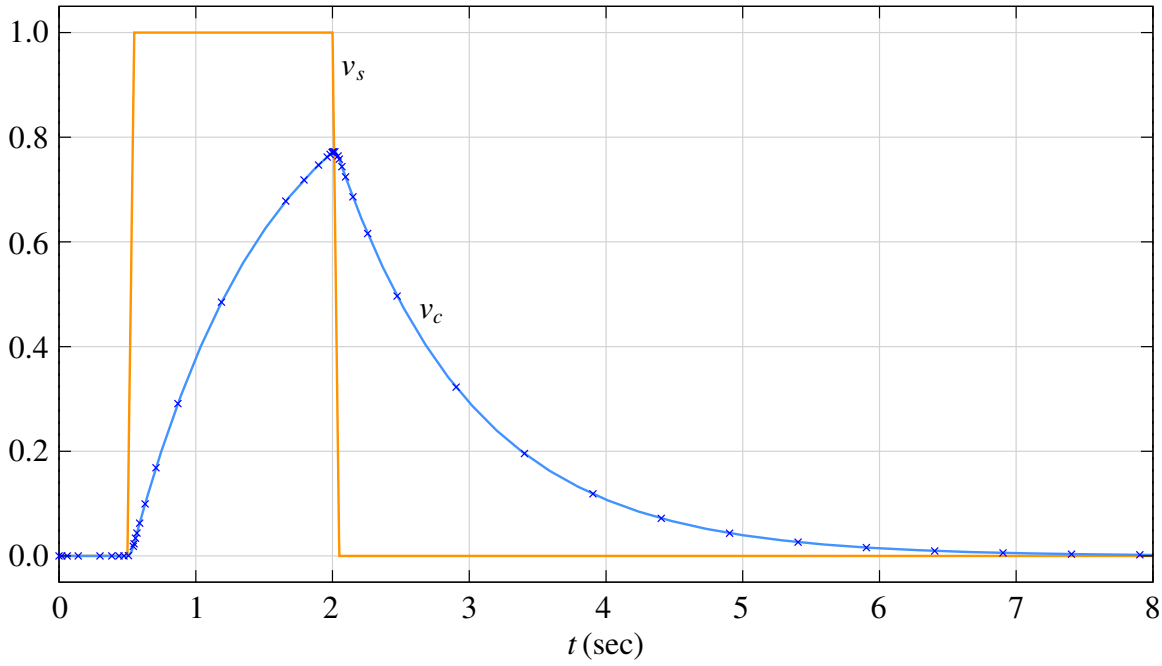


Figure 4.12: Numerical solutions obtained for the *RC* circuit of Fig. 4.9 using the RKF45 method with two values of τ: 10^{-4} (crosses) and 10^{-6} (blue graph). The other parameters are the same as in Fig. 4.10.

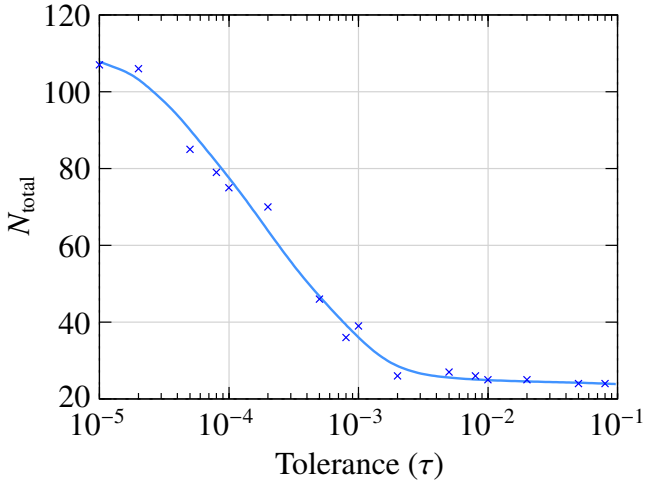


Figure 4.13: Total number of time points N_{total} used by the RKF45 method in computing the numerical solution for the *RC* circuit of Fig. 4.9 as a function of tolerance τ.

4.1.5 Stability

Apart from being sufficiently accurate, a numerical method for solving ODEs must also be *stable*, i.e., its "global error" $|x(t_n) - x_n|$ must remain bounded[8]. Broadly, we can talk of two types of stability.

(a) Stability for small h: Based on our discussion of local truncation error, we would expect that the accuracy of the numerical solution would generally get better if we use smaller step sizes.

[8]We remind ourselves that $x(t_n)$ and x_n are the exact and numerical solutions, respectively, at $t = t_n$.

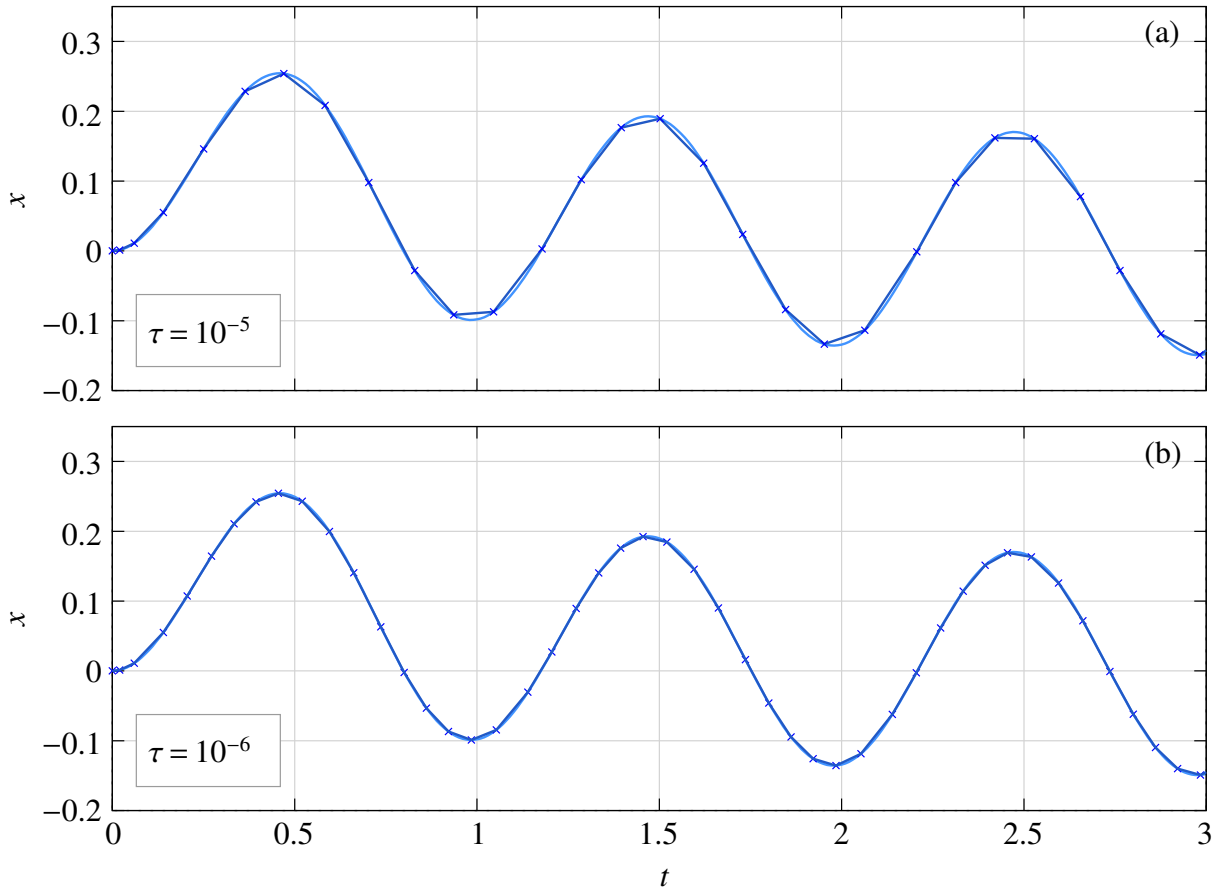


Figure 4.14: Numerical solutions obtained for the *RC* circuit of Fig. 4.9 with $v_s = \sin \omega t$ where $\omega = 2\pi$. Light blue curve: analytical solution, Crosses: numerical solution obtained with the RKF45 method. (a) $\tau = 10^{-5}$, (b) $\tau = 10^{-6}$. The other parameters are the same as in Fig. 4.10.

However, this is not true for all numerical methods. We can have a method which is accurate to a specified order in the local sense but is unstable in the global sense (i.e., the numerical solution "blows up" as time increases) *even if* the time steps are small (see [8] for an example). Such methods are of course of no use in circuit simulation, and we will not discuss them here.

(b) Stability for large h: All commonly used numerical methods for solving ODEs (including the FE and RK4 methods we have seen before) can be expected to work well when the step size h is sufficiently small. When h is increased, we expect the solution to be less accurate, but quite apart from that, the solution may also become unstable, and that is a serious concern. In the following, we will illustrate this point with an example.

Consider the *RC* circuit shown in Fig. 4.15 (a) which can be described by the ODEs

$$\begin{aligned} \frac{dV_1}{dt} &= \frac{1}{R_1 C_1}(V_s - V_1) - \frac{1}{R_2 C_1}(V_1 - V_2), \\ \frac{dV_2}{dt} &= \frac{1}{R_2 C_2}(V_1 - V_2). \end{aligned} \tag{4.25}$$

With a sinusoidal input, $V_s = V_m \sin \omega t$, we can use phasors (see Fig. 4.15 (b)) to estimate $V_1(t)$ in steady state. In particular, let $R_1 = 1\,\text{k}\Omega$, $R_2 = 2\,\text{k}\Omega$, $C_1 = 540\,\text{nF}$, $C_2 = 1\,\text{mF}$. With these component

values, and with a frequency of 50 Hz, we have $\mathbf{Z}_1 = -j5.9\,\text{k}\Omega$, $\mathbf{Z}_2 = -j3.2\,\Omega$. Since $\mathbf{Z}_1$ is relatively large, we can replace it with an open circuit. Similarly, since $\mathbf{Z}_2$ is small, we can replace it with a short circuit. The approximate solution for $\mathbf{V}_1$ is then simply

$$\mathbf{V}_1 \approx \mathbf{V}_s \frac{R_2}{R_1 + R_2}. \tag{4.26}$$

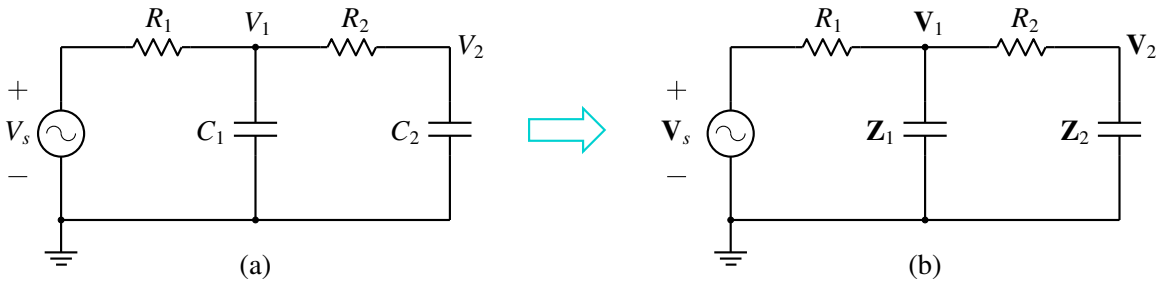


Figure 4.15: *RC* circuit with two time constants: (a) time-domain circuit, (b) frequency-domain circuit.

The numerical solution obtained with the RK4 method with a time step of $h = 1$ msec is shown in Fig. 4.16 (the light blue curve). Its amplitude and phase are in agreement with Eq. 4.26.

Do we expect any changes if C_1 is changed from 540 nF to 535 nF? Nothing, really. This change is not going to significantly affect the value of $\mathbf{Z}_1$, and we do not expect the solution to change noticeably. However, as seen in Fig. 4.16, the numerical solution (obtained with the same RK4 method and with the same step size) is dramatically different. It starts off being similar, but then takes off toward infinity.

Why does the value of C_1 make such a dramatic difference? The answer has to do with stability of the RK4 method. Table 4.1 gives the condition for stability of a few explicit RK methods when applied to the ODE $\frac{dx}{dt} = -\frac{x}{\tau}$. We see that the RK4 method is unstable when the step size exceeds about $2.8\,\tau$ ($2.785\,\tau$ to be more precise). If there are several time constants governing the set of ODEs being solved, this limit still holds with τ replaced by the *smallest* time constant.

Method	Maximum step size
RK-1 (same as FE)	$2\,\tau$
RK-2	$2\,\tau$
RK-3	$2.5\,\tau$
RK-4	$2.8\,\tau$

Table 4.1: Maximum step size allowed for stability of explicit Runge-Kutta methods when used for solving $\frac{dx}{dt} = -\frac{x}{\tau}$.

Let us see the implications of the above limit in the context of our *RC* example described by Eq. 4.25. Fig. 4.17 shows the variation of the time constants with C_1. The time constant marked τ_2 is the smaller of the two, and it is 0.356645 msec and 0.359978 msec for $C_1 = 535$ nF and 540 nF,

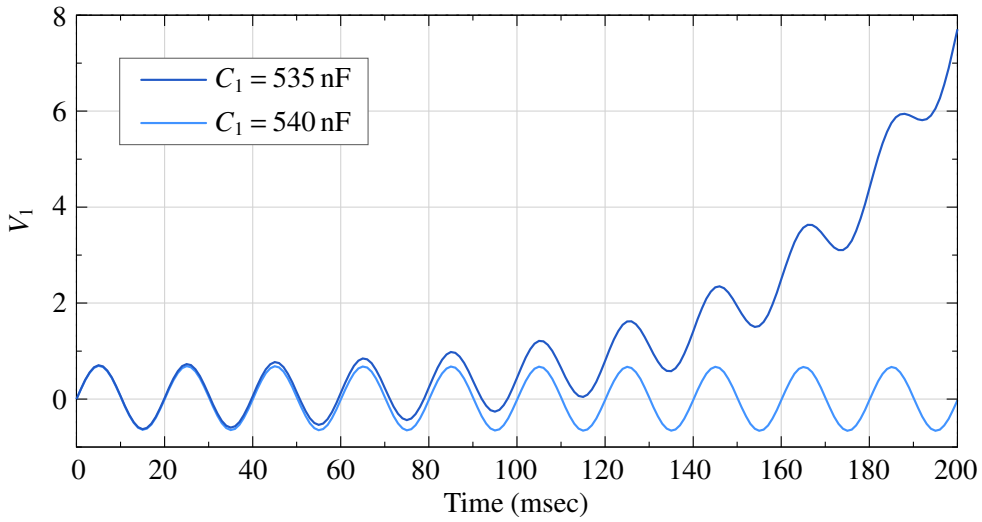


Figure 4.16: Numerical solution obtained with the RK4 method for the circuit of Fig. 4.15 for two values of C_1. The other parameters are $R_1 = 1\,\text{k}\Omega$, $R_2 = 2\,\text{k}\Omega$, $C_2 = 1\,\text{mF}$, $V_s = V_m \sin \omega t$, with $V_m = 1\,\text{V}$, $f = 50\,\text{Hz}$. The time step is $h = 1\,\text{msec}$.

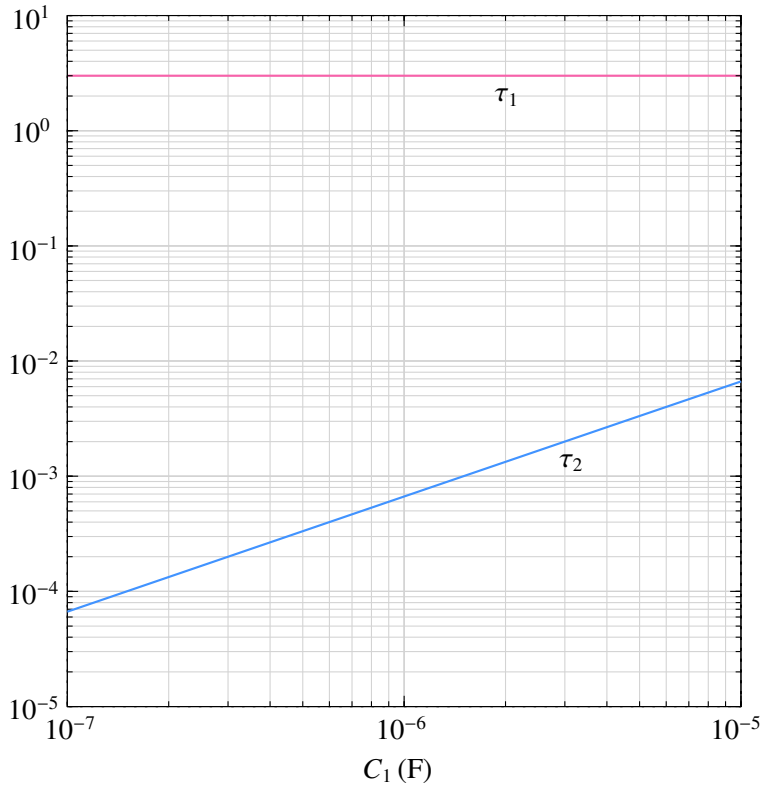


Figure 4.17: Time constants for the circuit of Fig. 4.15 (in seconds) as a function of C_1 with $R_1 = 1\,\text{k}\Omega$, $R_2 = 2\,\text{k}\Omega$, $C_2 = 1\,\text{mF}$.

respectively. For each of these, the maximum time step h_{max} to guarantee stability of the RK4 method can be obtained as $2.785\,\tau_2$. For $C_1 = 540$ nF, h_{max} is 1.0025 msec. The actual time step used for the numerical solution (1 msec) is smaller than this value, and the solution is stable. For $C_1 = 535$ nF, h_{max} is 0.9933 msec; the actual time step (1 msec) is now larger, leading to instability.

We can handle the instability problem by reducing the RK4 time step. However, we generally do not have a good idea of the time constants in a given circuit, and we would then need to carry out a cumbersome trial-and-error process of choosing a time step, checking if it works, reducing it if it does not, and so on. In these situations, the adaptive (auto) time step methods – such as the RKF45 method discussed in Sec. 4.1.4 – can be used to advantage. When a given time step leads to instability, the LTE also goes up, and the adaptive step method automatically reduces the time step suitably *without* any intervention from the user. This is indeed an attractive choice. Let us illustrate it with an example, again the *RC* circuit of Fig. 4.15 but with different component values, viz., $R_1 = 1$ kΩ, $R_2 = 1$ kΩ, $C_1 = 1\,\mu$F, $f = 1$ kHz. As C_2 is varied, the time constants change (see Fig. 4.19). When the RKF45 method is used to solve the ODEs (Eq. 4.25), the time step is automatically adjusted to meet the specified tolerance requirement, and a stable solution (not shown) is obtained. As C_2 is made smaller, the smaller of the two time constants becomes smaller, and the step sizes employed by the RKF45 algorithms also become smaller (see Fig. 4.20) as we would expect.

Sounds good, but there is a flip side. Take for example $C_2 = 10^{-10}$ F. The impedance $\mathbf{Z}_2$ is then (with $f = 1$ kHz) $-j64$ MΩ and is an open circuit for all practical purposes. The circuit reduces to a series combination of R_1 and C_1, and the solution is independent of C_2 (as long as it is small enough). We now have an unfortunate situation in which C_2 has no effect on the solution, yet it forces small time steps because of stability considerations.

The above situation is an example of "stiff" circuits in which the time constants are vastly different. We are often not interested in tracking the solution of a stiff circuit on the scale of the smallest time constant, but because of the stability constraints imposed by the numerical method, we are forced to use small time steps. This is a drawback of explicit methods since, like the RK4 method, they are *conditionally* stable.

4.1.6 Application to flow graphs

We now describe how GSEIM handles a flow graph when an explicit numerical scheme (such as forward Euler, RK4, RKF45) is used. We will consider a simple flow graph (see Fig. 4.18) for the purpose of illustration. The signals have been named as x_1, x_2, etc. and the elements as A, B, etc. The flow graph corresponds to the ODE: $\dfrac{dx_6}{dt} = k_1 x_1 + k_2 x_2$, where $x_1 = A_1 \sin \omega_1 t$, $x_2 = A_2 \sin \omega_2 t$ (for example).

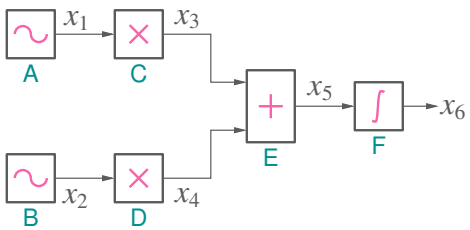


Figure 4.18: Flow graph example.

The flow graph has the following elements:

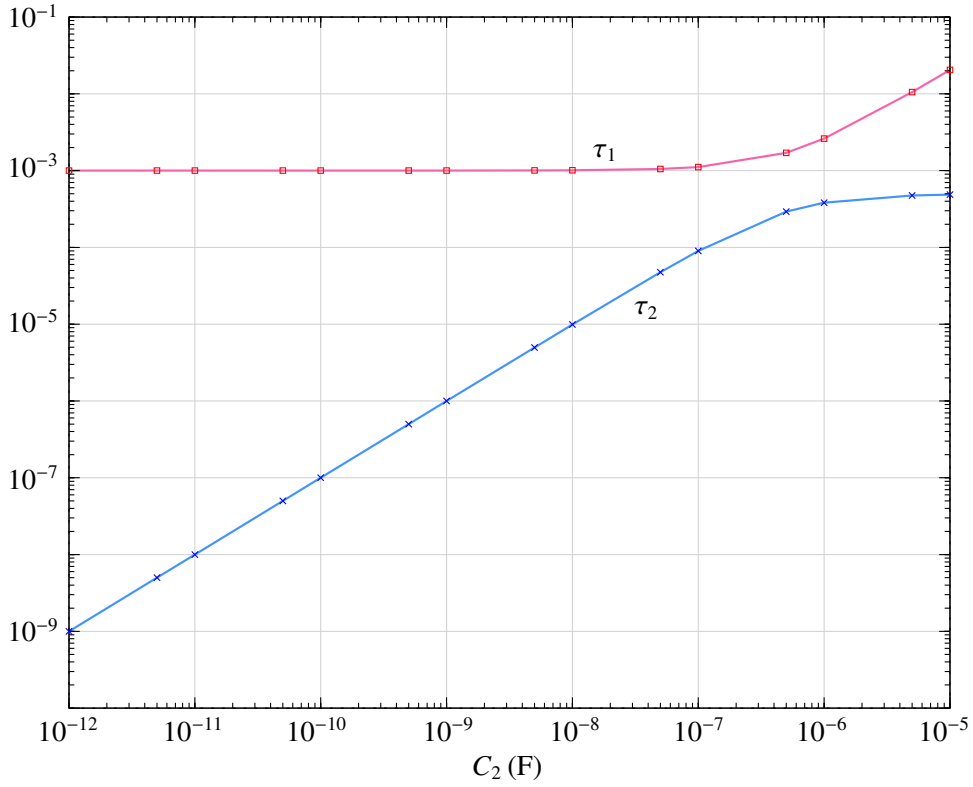


Figure 4.19: Time constants for the circuit of Fig. 4.15 (in seconds) as a function of C_2 with $R_1 = 1\,\text{k}\Omega$, $R_2 = 1\,\text{k}\Omega$, $C_1 = 1\,\mu\text{F}$.

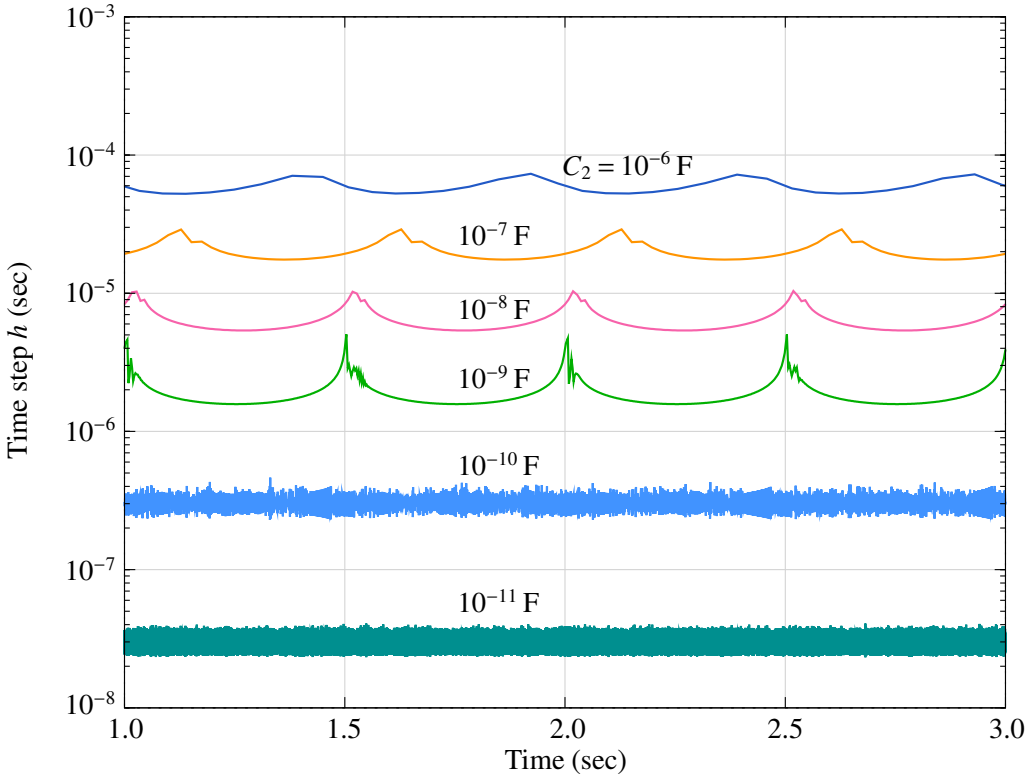


Figure 4.20: Time step (h) used by the RKF45 method versus elapsed time (t) in solving the set of ODEs for the circuit of Fig. 4.15 (Eq. 4.25) for different values of C_2. The other circuit parameters are $R_1 = 1\,\text{k}\Omega$, $R_2 = 1\,\text{k}\Omega$, $C_1 = 1\,\mu\text{F}$, $f = 1\,\text{kHz}$. The following algorithmic parameters were used: $\tau = 10^{-3}$, $\delta_{\min} = 0.2$, $\delta_{\max} = 2$, $h_{\min} = 10^{-5}$, $h_{\max} = 0.5$.

(a) signal sources (A, B)

(b) scalar multipliers (C, D)

(c) summer (E)

(d) integrator (F)

The integrator equation is $x_6 = \int x_5 dt$, or equivalently, $\frac{dx_6}{dt} = x_5$. For simplicity, we will use the forward Euler method to handle the time derivative involved in the integrator equation. The remaining elements can be described by algebraic equations, not involving time derivatives. Denoting the numerical solution for x_1 at t_n as x_1^n, etc., we have the following equations for the elements.

* Element A: $x_1^{n+1} = A_1 \sin \omega_1 t_{n+1}$
* Element B: $x_2^{n+1} = A_2 \sin \omega_2 t_{n+1}$
* Element C: $x_3^{n+1} = k_1 x_1^{n+1}$
* Element D: $x_4^{n+1} = k_2 x_2^{n+1}$
* Element E: $x_5^{n+1} = x_3^{n+1} + x_4^{n+1}$
* Element F: $x_6^{n+1} = x_6^n + \Delta t \times x_5^n$

Simple enough. The question is: In what order should we visit these equations? The order obviously depends on how the elements are connected in the flow graph. For example, it makes sense to evaluate the summer output x_5^{n+1} only after its inputs x_3^{n+1} x_4^{n+1} have been updated.

The following observations would help in deciding on the order in which the elements should be processed.

1. The integrator equation involves only the *past* values on the right hand side; it can be evaluated independently of the other elements, i.e., we can process the integrator first and then the remaining elements, keeping in mind the flow graph connections.
2. Source elements do no have inputs. We can therefore process them independently of the other elements.
3. The remaining elements should be processed in an order which ensures that the inputs of a given element are updated before updating its outputs.

Using the above considerations, we can come up with the following options for processing the elements:

* F → A → B → C → D → E
* F → A → B → D → C → E
* F → B → A → C → D → E

* F → B → A → D → C → E
* F → A → C → B → D → E

Note that all of these options are equivalent; the simulator can pick any of them and use it in each time step.

4.1.7 Algebraic Loops

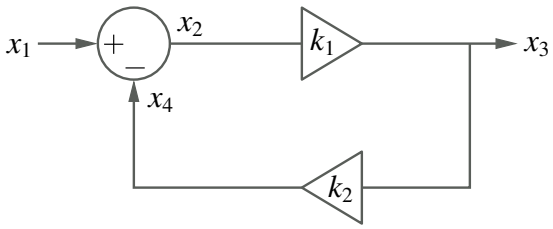


Figure 4.21: Flow graph with an algebraic loop.

Consider the feedback loop shown in Fig. 4.21. We can write the following equation for this system:

$$x_3 = k_1\, x_2 = k_1\,(x_1 - x_4) = k_1\,(x_1 - k_2\, x_3)\,, \tag{4.27}$$

giving

$$x_3 = \frac{k_1}{1 + k_1 k_2}\, x_1\,. \tag{4.28}$$

This result is valid at all times, and we can obtain $x_3(t_n)$ in terms of $x_1(t_n)$ simply by evaluating the above formula. Let us see if we can arrive at the same result by writing out the equations for the respective elements, as we did earlier. We get the following equations.

$$x_2^{n+1} = x_1^{n+1} - x_4^{n+1}\,, \tag{4.29}$$
$$x_4^{n+1} = k_2\, x_3^{n+1}\,, \tag{4.30}$$
$$x_3^{n+1} = k_1\, x_2^{n+1}\,. \tag{4.31}$$

As before, we wish to evaluate these formulas *one at a time*, but this leads to a conflict. Eqs. 4.29 to 4.31 are supposed to be valid *simultaneously*. For example, the value of x_3^{n+1} in Eqs. 4.30 and 4.31 is supposed to be the same. However, since Eq. 4.31 is evaluated after Eq. 4.30, the two values are clearly different. This type of conflict occurs when there is an "algebraic loop" in the system, i.e., there is a loop in which the variables are related through purely *algebraic* equations[9], not involving time derivatives.

For an explicit method, an algebraic loop presents a roadblock. We have the following options for a system with one or more algebraic loops.

(a) If possible, avoid algebraic loops. For example, instead of the system shown in Fig. 4.21, we could use a single element, $x_3 = k\, x_1$ with $k = \dfrac{k_1}{1 + k_1 k_2}$ pre-computed.

[9]Simulink manual uses the term "Direct feedthrough" which sounds a little more exotic.

(b) Insert a delay element to "break" an algebraic loop. For example, we can replace the block diagram of Fig. 4.21 with that of Fig. 4.22 in which a "delay" element is added in the loop. How does this help?

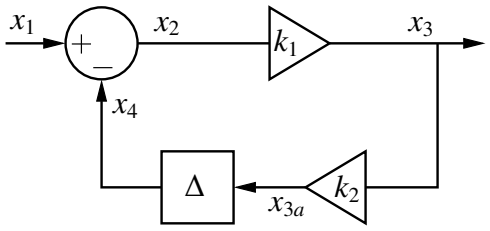


Figure 4.22: Flow graph to show breaking of an algebraic loop with a delay element.

The system equations can now be written as

$$\begin{aligned} x_4^{n+1} &= x_{3a}^{n}\,, \\ x_2^{n+1} &= x_1^{n+1} + x_4^{n+1}\,, \\ x_3^{n+1} &= k_1\, x_2^{n+1}\,, \\ x_{3a}^{n+1} &= k_2\, x_3^{n+1}\,. \end{aligned}$$

We can see that the variables have now got "decoupled," allowing a sequential evaluation of the formulas, not requiring any iterative procedure. If the delay is small compared to the time constants in the overall system, this approach – however crude it may sound – gives acceptable accuracy although it does change the original problem to some extent.

(c) Treat the integrator block(s) first (since their outputs depend only on the past values), compute their outputs, and with those held constant, solve the system of equations representing the rest of the system. Let us illustrate this process with the example shown in Fig. 4.23.

We first update the integrator output as

$$x_2^{n+1} = x_2^n + h\, x_1^n\,, \tag{4.32}$$

where h is the time step. Now, we treat x_2^{n+1} as a constant (say, c), and assemble the equations as follows.

$$\begin{aligned} x_2^{n+1} &= c, \\ x_1^{n+1} &= A_1 \sin \omega\, t_{n+1}, \\ x_3^{n+1} &= x_2^{n+1} + x_5^{n+1}, \\ x_4^{n+1} &= k_1\, x_3^{n+1}, \\ x_5^{n+1} &= k_2\, x_4^{n+1}. \end{aligned}$$

We have now obtained a system of equations which can be solved to get updated values of x_1, x_3, x_4, x_5. Note that, in this example, the system of equations happens to be linear, but in the general case, it could be nonlinear.

(d) Use an implicit method; in that case, algebraic loops do not need any special treatment.

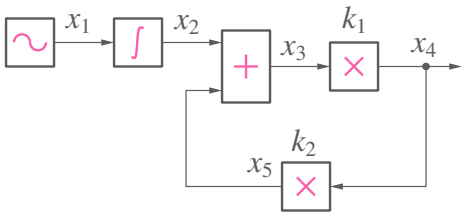


Figure 4.23: Algebraic loop example.

4.2 Implicit Methods

We have discussed in Sec. 4.1 some of the explicit methods – Forward Euler (FE), Runge-Kutta order 4 (RK4), and Runge-Kutta-Fehlberg (RKF45) – for solving the ODE

$$\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0. \tag{4.33}$$

We have also seen how the RKF45 method – a combination of an RK4 method and an RK5 method – can be used to control the time step in order to maintain a given accuracy (tolerance). There are several other explicit methods available in the literature (see [11], for example). We can summarise the salient features of explicit methods as follows.

(a) Explicit methods are easy to implement since they only involve *evaluation* of quantities rather than *solution* of equations. The computational effort per time point is therefore relatively small.

(b) Explicit methods can be extended to a system of ODEs in a straightforward manner. If there are N ODEs, the computational effort would increase by a factor of N as compared to solving a single ODE (assuming all ODEs to be of similar complexity).

(c) Explicit methods are conditionally stable – if the time step is not sufficiently small (of the same order as the smallest time constant), the numerical solution can "blow up" (i.e., become unbounded).

(d) In some problems, it may be difficult to know the various time constants involved. In such cases, the "auto" (automatic or adaptive) time step methods such as RKF45 are convenient. These methods can adjust the time step automatically to ensure a certain accuracy (in terms of the local truncation error estimate) and in the process keep the numerical solution from blowing up.

(e) Explicit methods are not suitable for stiff problems in which there are vastly different time constants involved. This is because the stability constraints imposed by an explicit method force small time steps *even when* the transients on the scale of the smaller time constants have vanished.

(f) When the system or circuit of interest involves nonlinear algebraic equations, it may not be possible to describe it with a set of ODEs. In such cases, explicit methods cannot be used.

Given the above limitations, it is clear that an alternative must be found for explicit methods. Some of the implicit[10] methods provide that alternative.

[10]The meaning of the term "implicit" will soon become clear.

4.2.1 Backward Euler and trapezoidal methods

Let us look at the most commonly used implicit methods, viz., the backward Euler and trapezoidal methods. Systematic approaches are available to derive these methods [8]; here, we will present a simplistic intuitive picture. Suppose the solution $x(t)$ of Eq. 4.33 is given by the curve shown in Fig. 4.24. We are interested in obtaining a numerical solution at discrete time points $t_1, t_2, \cdots$.

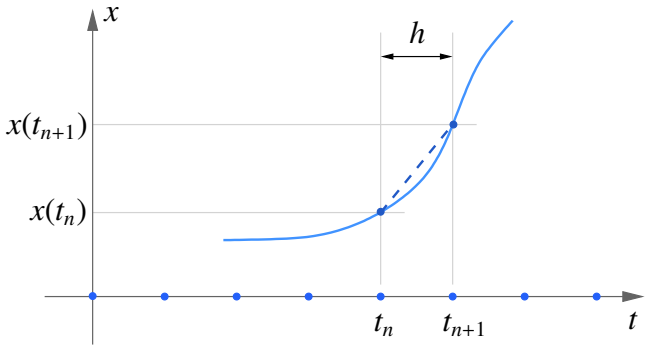


Figure 4.24: An intuitive view of the FE, BE, TRZ methods.

Consider the slope of the line joining the points $(t_n, x(t_n))$ and $(t_{n+1}, x(t_{n+1}))$, i.e., $m = \dfrac{x(t_{n+1}) - x(t_n)}{h}$. The forward Euler (FE) method (Eq. 4.2) results if we approximate the slope as

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \frac{x_{n+1} - x_n}{h} \approx \left.\frac{dx}{dt}\right|_{(t_n, x_n)} = f(t_n, x_n), \tag{4.34}$$

where x_n and x_{n+1} are the numerical solutions at t_n and t_{n+1}, respectively.

The backward Euler (BE) method results if the slope m is equated to the slope at (t_{n+1}, x_{n+1}), i.e., $\left.\dfrac{dx}{dt}\right|_{(t_{n+1}, x_{n+1})}$ rather than $\left.\dfrac{dx}{dt}\right|_{(t_n, x_n)}$. In that case, we get

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \frac{x_{n+1} - x_n}{h} \approx \left.\frac{dx}{dt}\right|_{(t_{n+1}, x_{n+1})} = f(t_{n+1}, x_{n+1}), \tag{4.35}$$

leading to

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}). \tag{4.36}$$

In case of the trapezoidal (TRZ) method, the slope m is equated to the average of the two slopes (at (t_n, x_n) and (t_{n+1}, x_{n+1})), i.e.,

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \frac{x_{n+1} - x_n}{h} \approx \frac{1}{2}\left(\left.\frac{dx}{dt}\right|_{(t_n, x_n)} + \left.\frac{dx}{dt}\right|_{(t_{n+1}, x_{n+1})}\right) = \frac{1}{2}\left[f(t_n, x_n) + f(t_{n+1}, x_{n+1})\right], \tag{4.37}$$

leading to

$$x_{n+1} = x_n + \frac{h}{2}\left[f(t_n, x_n) + f(t_{n+1}, x_{n+1})\right]. \tag{4.38}$$

Note that we have shown the time steps to be uniform in Fig. 4.24, but the above equations can also be used for non-uniform time steps by replacing h with $h_n \equiv t_{n+1} - t_n$.

The BE and TRZ methods are *implicit* in nature since x_{n+1} is involved in the right-hand sides of Eqs. 4.36 and 4.38. In other words, x_{n+1} cannot be simply *evaluated* in terms of known quantities[11]; instead, it needs to be obtained by *solving* Eq. 4.36 or 4.38. As an example, consider

$$\frac{dx}{dt} \equiv f(t,x) = \cos x, \quad x(0) = 0. \tag{4.39}$$

The FE and BE methods give the discretised equations,

$$\begin{aligned} \text{FE}: &\quad x_{n+1} = x_n + h\,\cos(x_n), \\ \text{BE}: &\quad x_{n+1} = x_n + h\,\cos(x_{n+1}). \end{aligned} \tag{4.40}$$

There is a fundamental difference between the two equations in terms of computational effort. If we use the FE method, x_{n+1} can be obtained by simply evaluating the right-hand side. With the BE method, the task is far more complex because the presence of x_{n+1} on the RHS has made the equation nonlinear, thus requiring an iterative solution process. If we use the NR method, for example, then each iteration will involve evaluation of the function, its derivative, and the correction. Clearly, the work involved per time point is much more when the BE method is used. The following C program shows the implementation of the FE and BE methods for solving Eq. 4.39. The results are shown in Fig. 4.25.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main()
{
   double t,x,f,t_start,t_end,h;
   double x_old;
   double f_nr,dfdx_nr,delx_nr;
   int i_nr,nmax_nr=10;
   FILE *fp;

   t_start = 0.0;
   t_end = 8.0;
   h = 0.05;

// Forward Euler:
   fp=fopen("fe.dat","w");

   t=t_start;
   x = 0.0; // initial condition
   fprintf(fp,"%13.6e %13.6e\n",t,x);

   while (t <= t_end) {
     f = cos(x);
     x = x + h*f;
     t = t + h;
     fprintf(fp,"%13.6e %13.6e\n",t,x);
```

[11] except in some special cases such as (a) $f(t_{n+1}, x_{n+1})$ is linear in x_{n+1}, (b) $f(t_{n+1}, x_{n+1})$ does not involve x_{n+1} at all, e.g., $f(t,x) = \cos \omega t$.

```
   }
   fclose(fp);

// Backward Euler:
   fp=fopen("be.dat","w");

   t=t_start;
   x = 0.0; // initial condition
   x_old = x;
   fprintf(fp,"%13.6e %13.6e\n",t,x);

   while (t <= t_end) {
//   Newton-Raphson loop
     for (i_nr=0; i_nr < (nmax_nr+1); i_nr++) {
       if (i_nr == nmax_nr) {
         printf("N-R did not converge.\n");
         exit(0);
       }
       f_nr = x - h*cos(x) - x_old;
       if (fabs(f_nr) < 1.0e-8) break;
       dfdx_nr = 1.0 + h*sin(x);
       delx_nr = -f_nr/dfdx_nr;
       x = x + delx_nr;
     }
     t = t + h;
     fprintf(fp,"%13.6e %13.6e\n",t,x);
     x_old = x;
   }
   fclose(fp);
}
```

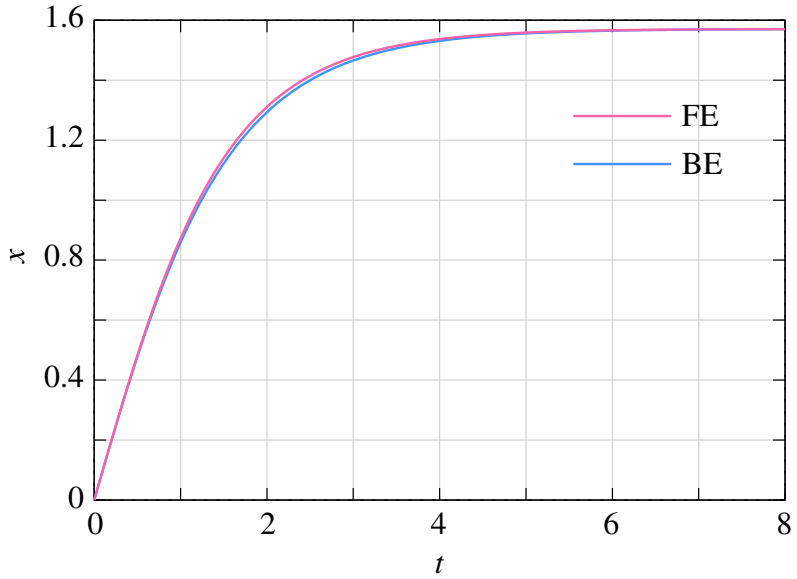


Figure 4.25: Numerical solutions of Eq. 4.39 obtained with the FE and BE using a step size of $h = 0.05$.

The BE and TRZ methods can be extended to solve a set of ODEs (see Eqs. 4.9, 4.10):

$$\begin{aligned} \text{BE:} \quad & \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + h\,\mathbf{f}(t_{n+1}, \mathbf{x}^{(n+1)}), \\ \text{TRZ:} \quad & \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \frac{h}{2}\left(\mathbf{f}(t_n, \mathbf{x}^{(n)}) + \mathbf{f}(t_{n+1}, \mathbf{x}^{(n+1)})\right), \end{aligned} \tag{4.41}$$

where $\mathbf{x}^{(n)}$ stands for the numerical solution at t_n, and so on. As an illustration, let us consider the set of two ODEs seen earlier (Eq. 4.13) and reproduced here:

$$\begin{aligned} \frac{dx_1}{dt} &= a_1\,(\sin\omega t - x_1)^3 - a_2\,(x_1 - x_2)\,, \\ \frac{dx_2}{dt} &= a_3\,(x_1 - x_2)\,, \end{aligned} \tag{4.42}$$

with the initial condition, $x_1(0) = 0$, $x_2(0) = 0$. The BE equations are given by

$$\begin{aligned} x_1^{(n+1)} &= x_1^{(n)} + h\left[a_1\left(\sin\omega t_{n+1} - x_1^{(n+1)}\right)^3 - a_2\left(x_1^{(n+1)} - x_2^{(n+1)}\right)\right], \\ x_2^{(n+1)} &= x_2^{(n)} + h\left[a_3\left(x_1^{(n+1)} - x_2^{(n+1)}\right)\right]. \end{aligned} \tag{4.43}$$

We now have a set of nonlinear equations which must be solved at each time point. The NR method is commonly used for this purpose, and it entails computation of the Jacobian matrix and the function vector, and solution of the Jacobian equation (of the form $\mathbf{Ax} = \mathbf{b}$) in each iteration of the NR loop. As N (the number of ODEs) increases, the computational cost of solving the Jacobian equation increases superlinearly ($O(N^3)$ for a dense matrix). On the other hand, with an explicit method, we do not require to solve a system of equations, and the computational effort can be expected to grow linearly with N (see the FE equations given by Eq. 4.14, for example). With respect to computational effort, explicit methods are therefore clearly superior to implicit methods if the number of time points to be simulated is the same in both cases. That is a big if, as we will soon discover.

4.2.2 Stability

As we have seen in Sec. 4.1.5, the explicit methods we have discussed, viz., FE and RK4, are conditionally stable. This puts an upper limit $h_{\max}$ on the step size while solving the test equation,

$$\frac{dx}{dt} = -\frac{t}{\tau}, \quad x(0) = 1. \tag{4.44}$$

For the FE method, $h_{\max}$ is $2\,\tau$, and for the RK4 method, it is $2.8\,\tau$. Other explicit methods are also conditionally stable.

The FE method is a member of the Adams-Bashforth (AB) family of explicit linear multi-step methods (see [8], for example). The regions of stability of the AB methods with respect to Eq. 4.44 are shown in Fig. 4.26 (a). For the AB1 (FE) method, we require $h/\tau < 2$, as we have already seen. As the order increases, the AB methods become more accurate, but the region of stability shrinks.

The BE and TRZ methods belong to the Adams-Moulton (AM) family of implicit linear multi-step methods. The BE method is of order 1 while the TRZ method is of order 2. The stability properties of the AM methods with respect to Eq. 4.44 are shown in Fig. 4.26 (b). The BE and TRZ methods are *unconditionally* stable while higher-order AM methods have finite regions of stability which shrink as the order increases.

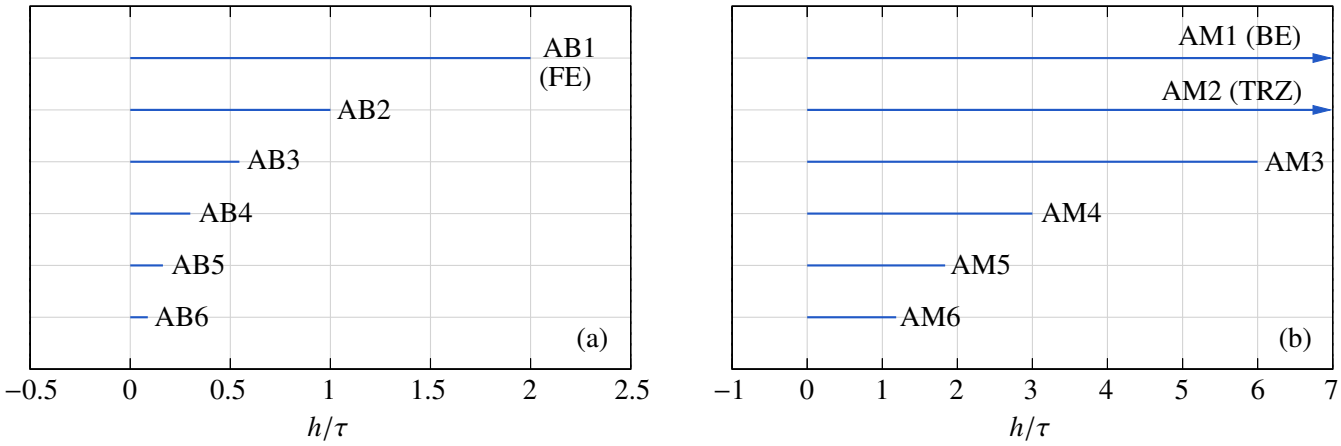


Figure 4.26: Regions of stability for Adams-Bashforth and Adams-Moulton methods of various orders with respect to the test equation $\dfrac{dx}{dt} = -t/\tau$.

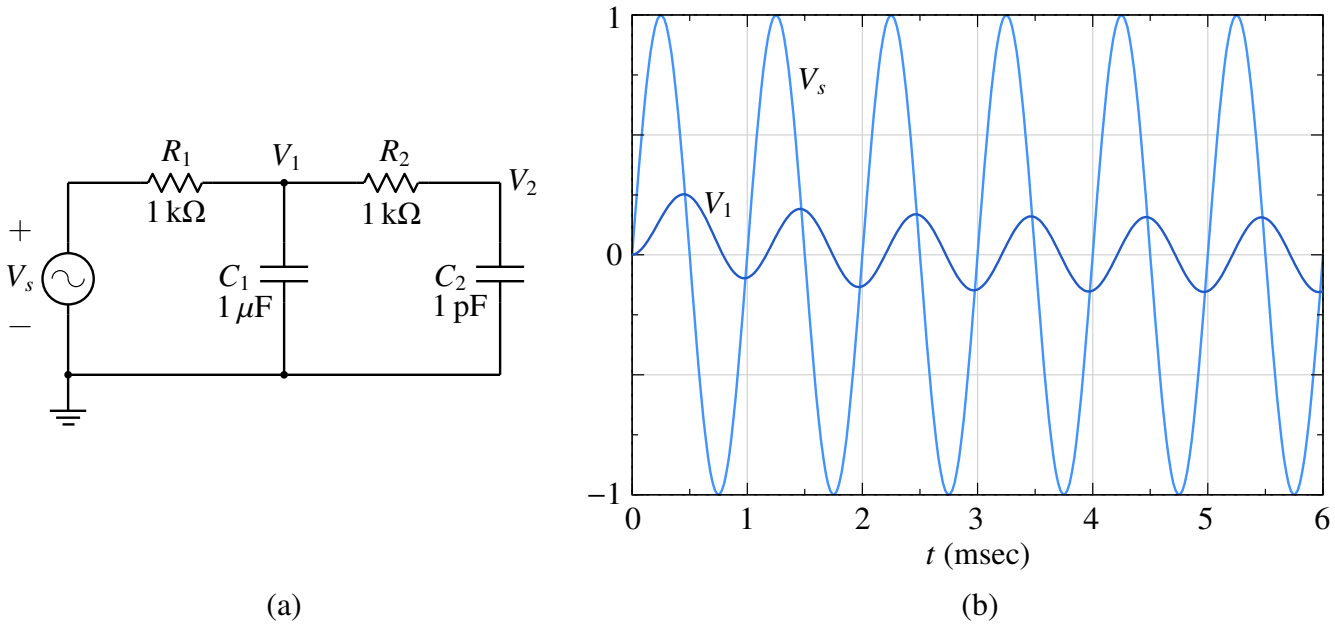
The unconditional stability of the BE and TRZ methods allows us to break free from the stability constraints which arise in stiff circuits (see Sec. 4.1.5). This is a *huge* benefit; it means that, if we use the BE or TRZ method, the only restriction on the time step comes from accuracy of the numerical solution and not from stability. For example, let us re-visit the stiff circuit of Fig. 4.27 (a). We recall that, for this circuit, the RKF45 method was forced to use very small time steps (of the order of nanoseconds, see Fig. 4.20) because of stability considerations. The BE method is not constrained by stability issues, and therefore a much larger time step ($h = 0.02$ msec) can be used to obtain the numerical solution, as shown in Fig. 4.27 (b).

As another example, consider the half-wave rectifier circuit of Fig. 4.28. The diode is represented with the R_{on}/R_{off} model, with a turn-on voltage equal to 0 V and $R_{off} = 1\,\text{M}\Omega$. The following ODE describes the circuit behaviour.

$$\frac{dV_o}{dt} = \frac{1}{C}\left[\frac{V_s - V_o}{R_D} - \frac{V_o}{R}\right], \tag{4.45}$$

where R_D is the diode resistance (R_{on} if $V_s > V_o$; R_{off} otherwise). Fig. 4.28 shows the numerical solution of Eq. 4.45 obtained with the RKF45 method. The charging and discharging intervals can be clearly identified – charging takes place when the diode current is non-zero. In the discharging interval, the capacitor discharges through the load resistor. When the diode conducts, the circuit time constant is $\tau_1 = (R_{on} \parallel R)\,C$ (approximately $R_{on}C$); otherwise it is $\tau_2 = R\,C$. The RKF45 method – like other conditionally stable methods – requires that the time step be of the order of the circuit time constant. Since $\tau_1 \ll \tau_2$, the time step used by the RKF45 algorithm is much smaller in the charging phase compared to the discharging phase, as seen in the figure. If we use a smaller value of R_{on}, τ_1 becomes smaller, and smaller time steps get forced.

On the other hand, the BE or TRZ method would not be constrained by stability considerations at all, and for these methods, a much larger time step can be selected as long as it gives sufficient accuracy. For the half-wave rectifier example, 20 μsec is suitable, *irrespective* of the value of R_{on}.



(a) (b)

Figure 4.27: (a) A stiff circuit with two time constants (see Fig. 4.19), (b) Numerical solution obtained with the BE method. $V_s = V_m \sin \omega t$, with $V_m = 1$ V, $f = 1$ kHz. The time step is $h = 0.02$ msec.

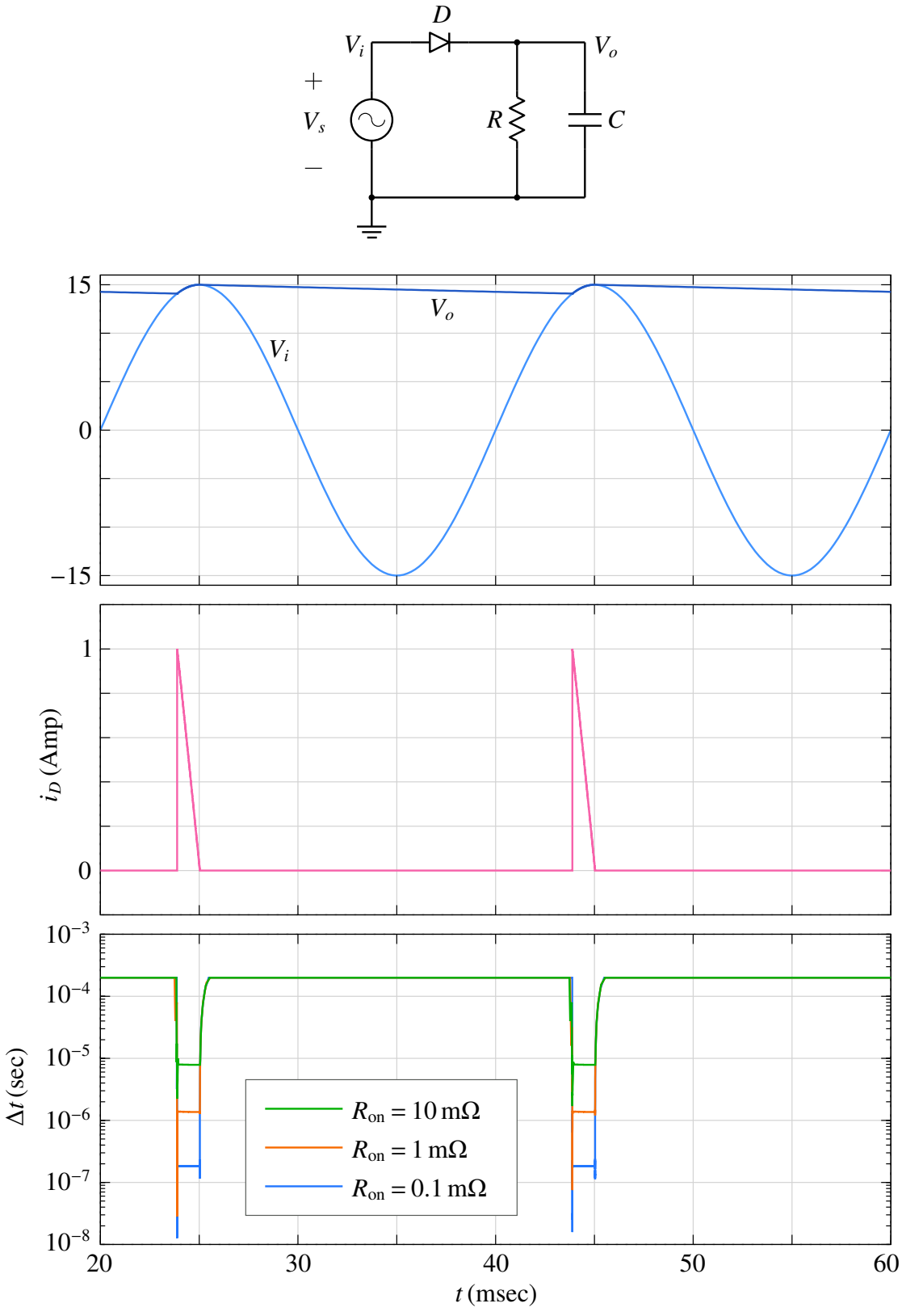


Figure 4.28: RKF45 results for a half-wave rectifier. (a) $V_i(t)$ and $V_o(t)$ in Volts, (b) diode current versus time, (c) time step used by the RKF45 method versus time. $V_s = V_m \sin \omega t$, with $V_m = 1\,\text{V}$, $f = 50\,\text{Hz}$, $R = 500\,\Omega$, $C = 600\,\mu\text{F}$. The R_{on}/R_{off} model is used for the diode, and the turn-on voltage is taken as 0 V. R_{off} is kept constant at 1 MΩ.

4.2.3 Some practical issues

We have already covered the most important aspects of numerical methods for solving ODEs: (a) comparison of explicit and implicit methods with respect to computation time, (b) accuracy (order) of a method, (c) constraints imposed on the time step by stability considerations of a method. In addition, we need to understand some specific situations which arise in circuit simulation.

4.2.3.1 Oscillatory circuits

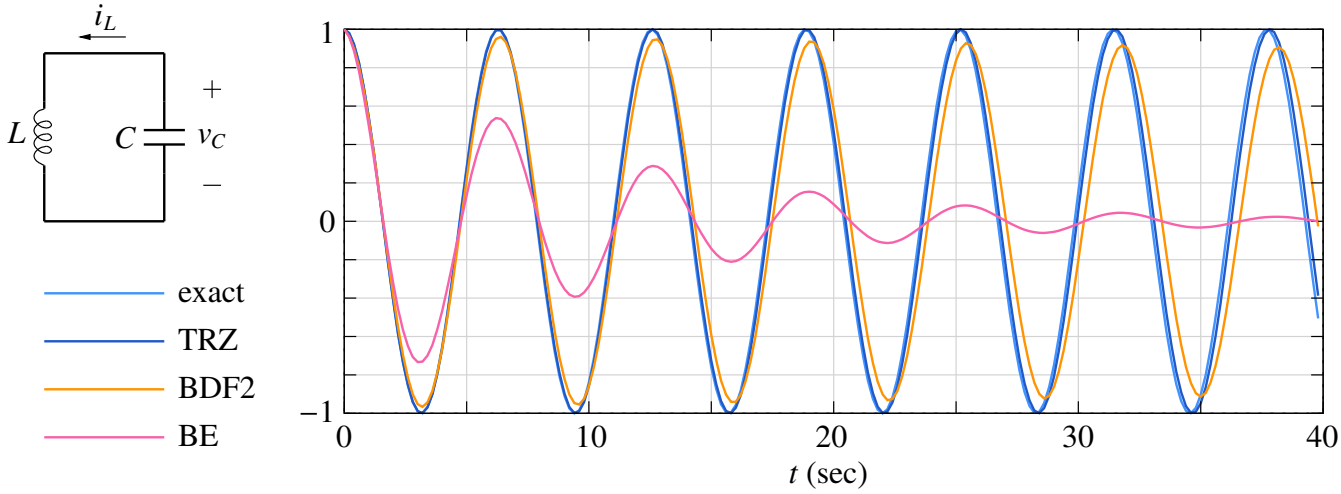


Figure 4.29: $v_C(t)$ obtained for an LC circuit with Backward Euler, Trapezoidal, and second-order Gear (BDF2) methods. $L = 1$ H, $C = 1$ F, and h (time step)=0.2 sec in all cases. The exact solution is also shown for comparison.

Consider an LC circuit without any resistance (see Fig. 4.29) and with the initial conditions, $v_C(0) = 1$ V, $i_L(0) = 0$ A. The circuit equations can be described by the following set of ODEs.

$$\begin{aligned} \frac{dv_C}{dt} &= -\frac{1}{C}\, i_L\,, \\ \frac{di_L}{dt} &= \frac{1}{L}\, v_C\,, \end{aligned} \tag{4.46}$$

with the analytic solution given by

$$\begin{aligned} v_C &= \cos(\omega t)\,, \\ i_L &= \sin(\omega t)\,, \end{aligned} \tag{4.47}$$

with $\omega = 1/\sqrt{LC}$. We will consider $L = 1$ H, $C = 1$ F which gives the frequency of oscillation $f_0 = 1/2\pi$ Hz, i.e., a period $T = 2\pi$ sec. Fig. 4.29 shows the numerical results obtained with the BE, TRZ, and BDF2 methods along with the analytic (exact) solution. The TRZ method maintains the amplitude of $v_C(t)$ constant whereas the BE and BDF2 methods lead to an artificial reduction (damping) of the amplitude with time. Clearly, the BE and BDF2 methods are not suitable for purely oscillatory circuits or for circuits with small amount of natural damping[12]. In such cases, the TRZ method should be used.

[12]Some of the passive filters fall in this category.

Although the TRZ method does not result in an amplitude error, it does give a phase error, i.e., the phase of $v_C(t)$ differs from the expected phase as time increases, as seen in the figure[13]. The phase error can be reduced by selecting a smaller time step.

4.2.3.2 Ringing

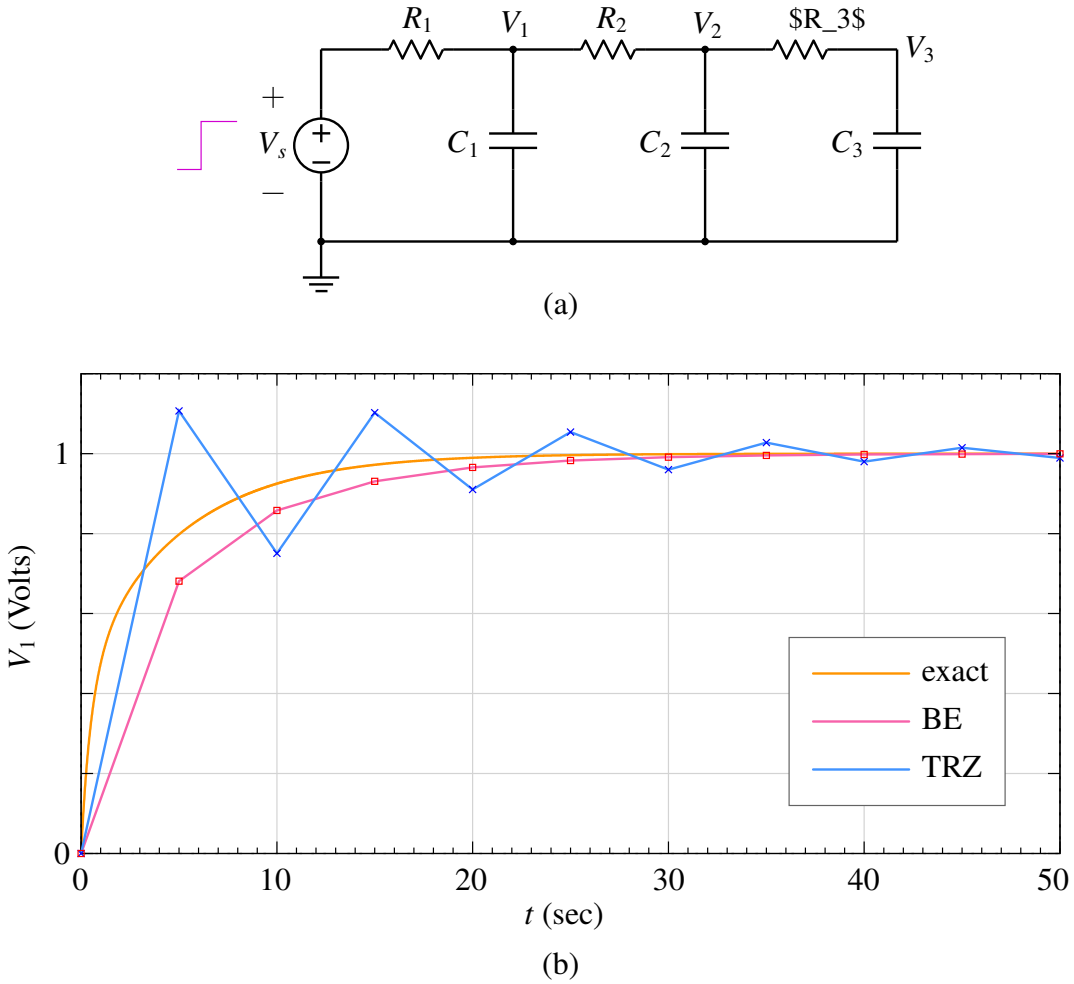


Figure 4.30: (a) *RC* circuit example, (b) Simulation results for V_1, with $R_1 = R_2 = R_3 = 1\,\Omega$, $C_1 = C_2 = C_3 = 1\,\text{F}$, $V_C(0) = 0\,\text{V}$ for all capacitors, and a step input going from 0 V to 1 V applied at $t = 0$.

Consider the *RC* circuit shown in Fig. 4.30. The time constants for this circuit are 0.31, 0.64, and 5.05 seconds, the smallest being $\tau_{\text{min}} = 0.31$ sec. Fig. 4.30 shows the BE and TRZ results with a relatively large time step. Sine the time step (5 sec) is much larger than τ_{min}, neither of the two methods can track closely the expected solution. However, there is a substantial difference in the nature of the deviation of the numerical solution from the exact solution – With the TRZ method, the numerical solution *overshoots* the exact solution, hovers around it in an oscillatory manner, and finally returns to the expected value when the transients have vanished. This phenomenon, which is

[13]To observe the phase error, expand the plot (zoom in), and look at the exact and TRZ results; you will see the phase error growing with time.

specifically associated with the TRZ method, is called "ringing." The BE result, in contrast, follows the exact solution without overshooting.

Is ringing relevant in practice? Are we ever going to use a time step which is much larger than $\tau_{\min}$? Yes, the situation does arise in practice, and we should therefore be watchful. For example, in a typical power electronic circuit, there is frequent switching activity. Whenever a switch closes, we can expect transients. Since the switch resistance is small, $\tau_{\min}$ is also small, typically much smaller than the time scale on which we want to resolve the transient. In other words, in such cases, the time step would be often much larger than $\tau_{\min}$, and we can expect ringing to occur if the TRZ method is used. If there is a good reason for using the TRZ method for the specific simulation of interest, the time step should be suitably reduced in order to avoid ringing.

4.2.4 Systematic assembly of circuit equations

A circuit simulator like SPICE must be able to handle any general circuit topology, and it should get the solution in an efficient manner. In this section, we will see how a general circuit involving time derivatives (e.g., in the form of capacitors and inductors) is treated in a circuit simulator.

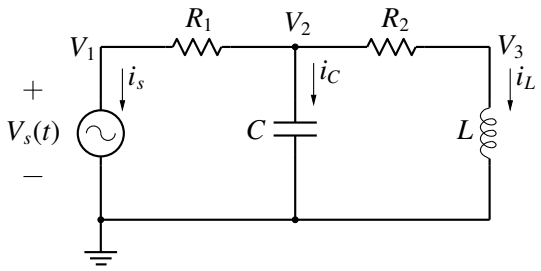


Figure 4.31: Circuit to illustrate equation assembly.

Consider the circuit in Fig. 4.31. The following equations describe the circuit behaviour, taking into account the relationship to be satisfied for each element in the circuit.

$$C\frac{dv_C}{dt} = G_1(V_s - V_2) - i_L\,, \tag{4.48}$$

$$L\frac{di_L}{dt} = V_3\,, \tag{4.49}$$

$$i_L = G_2(V_2 - V_3)\,, \tag{4.50}$$

where $G_1 = 1/R_1$. $G_2 = 1/R_2$. Note that the above system of equation is a "mixed" system – Eq. 4.50 is an algebraic equation whereas Eqs. 4.48 and 4.49 are ODEs. Such a set of equations is called "Differential-algebraic equations" (DAEs). In some cases, it may be possible to manipulate the equations to eliminate the algebraic equations and reduce the original set to an equivalent set of ODEs. However, that is not possible in general, especially when nonlinear terms are involved.

How did we come up Eqs. 4.48-4.50? We knew that the equations for the capacitor current and inductor voltage must be included somewhere. We looked at the circuit and found that there is a KCL which involves the capacitor current. Similarly, there is a node voltage which is the same as the inductor voltage. We then invoked the circuit topology and wrote the equations such that they describe the circuit completely. In other words, we used our *intuition* about circuits. Unfortunately, this is not a *systematic* approach and is therefore of no particular use in writing a general-purpose

circuit simulator. Instead of an *ad hoc* approach, we can use a systematic approach we have already seen before – the MNA approach – and write the circuit equations as

$$i_s + G_1(V_1 - V_2) = 0\,, \tag{4.51}$$
$$G_1(V_2 - V_1) + G_2(V_2 - V_3) + i_C = 0\,, \tag{4.52}$$
$$G_2(V_3 - V_2) + i_L = 0\,, \tag{4.53}$$
$$V_1 = V_s(t)\,, \tag{4.54}$$

where the first three are KCL equations, and the last equation is the element equation for the voltage source. Our interest is in obtaining the solution of the above equations for t_{n+1} from that available for t_n. Let us write the above equations specifically for $t = t_{n+1}$:

$$i_s^{n+1} + G_1(V_1^{n+1} - V_2^{n+1}) = 0\,, \tag{4.55}$$
$$G_1(V_2^{n+1} - V_1^{n+1}) + G_2(V_2^{n+1} - V_3^{n+1}) + i_C^{n+1} = 0\,, \tag{4.56}$$
$$G_2(V_3^{n+1} - V_2^{n+1}) + i_L^{n+1} = 0\,, \tag{4.57}$$
$$V_1^{n+1} = V_s(t_{n+1})\,. \tag{4.58}$$

Next, we select a method for discretisation of the time derivatives. As we have seen earlier, stability constraints restrict the choice to the BE, TRZ, and BDF2 methods[14], all of them implicit in nature. Suppose we choose the BE method. The capacitor and inductor currents (i_C^{n+1}, i_L^{n+1}) are then obtained as

$$\frac{dV_2}{dt} = \frac{1}{C}\,i_C \;\rightarrow\; \frac{V_2^{n+1} - V_2^n}{h} = \frac{i_C^{n+1}}{C} \;\rightarrow\; i_C^{n+1} = \frac{C}{h}\,(V_2^{n+1} - V_2^n)\,, \tag{4.59}$$
$$\frac{di_L}{dt} = \frac{1}{L}\,V_3 \;\rightarrow\; \frac{i_L^{n+1} - i_L^n}{h} = \frac{V_3^{n+1}}{L} \;\rightarrow\; i_L^{n+1} = i_L^n + \frac{h}{L}\,(V_3^{n+1})\,, \tag{4.60}$$

where $h = t_{n+1} - t_n$ is the time step. Substituting for i_C^{n+1} and i_L^{n+1} in Eqs. 4.56 and 4.57, we get

$$G_1 V_1^{n+1} - G_1 V_2^{n+1} + i_s^{n+1} = 0\,, \tag{4.61}$$
$$-G_1 V_1^{n+1} + \left(G_1 + G_2 + \frac{C}{h}\right) V_2^{n+1} - G_2 V_3^{n+1} = \frac{C}{h}\,V_2^n\,, \tag{4.62}$$
$$-G_2 V_2^{n+1} + \left(G_2 + \frac{h}{L}\right) V_3^{n+1} = -i_L^n\,, \tag{4.63}$$
$$V_1^{n+1} = V_s(t_{n+1})\,, \tag{4.64}$$

a linear system of four equations in four variables (V_1^{n+1}, V_2^{n+1}, V_3^{n+1}, i_s^{n+1}). It is now a simple matter[15] of solving this $\mathbf{Ax} = \mathbf{b}$ type problem to obtain the numerical solution at t_{n+1}.

What if there are nonlinear elements in the circuit? Let us consider the circuit shown in Fig. 4.32. The diode is described by

$$I_D = I_s\left(e^{V_D/V_T} - 1\right). \tag{4.65}$$

[14]Implicit Runge-Kutta methods are also unconditionally stable, but they are suitable only when the circuit equations can be written as a set of ODEs.

[15]Solving $\mathbf{Ax} = \mathbf{b}$ is conceptually simple but can take a significant amount of computation time when the system is large.

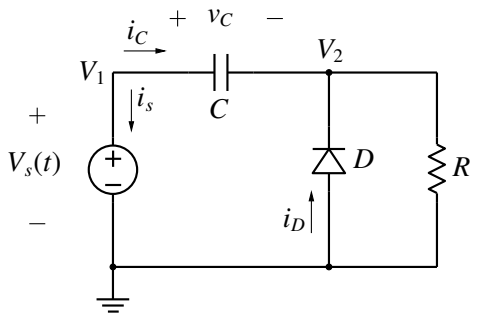


Figure 4.32: A nonlinear circuit.

We can write the MNA equations for this circuit as

$$i_s + i_C = 0, \tag{4.66}$$

$$-i_C - i_D + GV_2 = 0 \;\rightarrow\; -i_C - I_s\left(e^{-V_2/V_T} - 1\right) + GV_2 = 0, \tag{4.67}$$

$$V_1 = V_s(t), \tag{4.68}$$

with $G = 1/R$. At $t = t_{n+1}$, the equations are

$$i_s^{n+1} + i_C^{n+1} = 0, \tag{4.69}$$

$$-i_C^{n+1} - I_s\left(e^{-V_2^{n+1}/V_T} - 1\right) + GV_2^{n+1} = 0, \tag{4.70}$$

$$V_1^{n+1} = V_s(t_{n+1})\,. \tag{4.71}$$

The next step is to obtain i_C^{n+1} using the BE approximation for the capacitor equation:

$$\frac{dv_C}{dt} = \frac{1}{C}\,i_C \;\rightarrow\; \frac{v_C^{n+1} - v_C^n}{h} = \frac{i_C^{n+1}}{C} \;\rightarrow\; i_C^{n+1} = \frac{C}{h}\left[(V_1^{n+1} - V_2^{n+1}) - (V_1^n - V_2^n)\right], \tag{4.72}$$

Finally, substituting for i_C^{n+1} in Eqs. 4.69 and 4.70, we get

$$i_s^{n+1} + \frac{C}{h}V_1^{n+1} - \frac{C}{h}V_2^{n+1} - \frac{C}{h}(V_1^n - V_2^n) = 0, \tag{4.73}$$

$$-\frac{C}{h}V_1^{n+1} + \frac{C}{h}V_2^{n+1} - I_s\left(e^{-V_2^{n+1}/V_T} - 1\right) + GV_2^{n+1} + \frac{C}{h}(V_1^n - V_2^n) = 0, \tag{4.74}$$

$$V_1^{n+1} - V_s(t_{n+1}) = 0. \tag{4.75}$$

We now have a nonlinear set of three equations in three variables (V_1^{n+1}, V_2^{n+1}, i_s^{n+1}). Generally, circuit simulators would use the NR method to solve these equations because of the attractive convergence properties of the NR method seen earlier. Note that the NR loop needs to be executed in *each* time step, i.e., the Jacobian equation $\mathbf{J}\Delta\mathbf{x} = -\mathbf{f}$ must be solved several times in each time step until the NR process converges. Obviously, this is an expensive computation. Some tricks may be employed to make the NR process more efficient, e.g., $\mathbf{J}^{-1}$ from the previous NR iteration can be used if $\mathbf{J}$ has not changed significantly. In some circuit simulators, nonlinear functions are approximated with piecewise linear functions, thus avoiding an NR loop in each time step and leading to a significant speed advantage.

The benefits of the above approach (we will refer to it as the "DAE approach") outweigh the computational complexity:

(a) Since unconditionally stable methods (BE, TRZ, BDF2) are used, stiff circuits can be handled without very small time steps.

(b) "Algebraic loops" (see Sec. 4.1.7) do not pose any special problem since the set of differential-algebraic equations is solved directly, without any approximations.

Clearly, for any serious circuit simulation work, we should use a circuit simulator based on the DAE approach.

4.2.5 Adaptive time steps using NR convergence

As we have seen in Chapter 3, the initial guess plays an important role in deciding whether the NR process will converge for a given nonlinear problem. This feature can be used to control the time step in transient simulation. The solution obtained at t_n serves as the initial guess for solving the circuit equations at t_{n+1}. If $h \equiv t_{n+1} - t_n$ is sufficiently small, we expect the initial guess to work well, i.e., we expect the NR process to converge. If h is large, the NR process may not converge, or may take a larger number of iterations to converge. By monitoring the NR convergence process, it is possible control the time step.

The flow chart for auto (adaptive) time steps is shown in Fig. 4.33. The basic idea is to allow only a certain maximum number $N_{\text{NR}}^{\text{max}}$ of NR iterations at each time point. If the NR process does not converge, it means that the time step being taken is too large, and it needs to be reduced (by a factor k_{down}). However, if the NR process consistently converges in less than $N_{\text{NR}}^{\text{max}}$ iterations, it means that a small time step is not necessary any longer, and we then increase it (by a factor k_{up}). In this manner, the step size is made small when required but is allowed to become larger when convergence is easier. In practice, this means that small time steps are forced when the solution is varying rapidly, and large time steps are used when the solution is varying gradually.

Fig. 4.34 [8] shows the application of the above procedure to an oscillator circuit. The output voltage V_4 controls the switch – when V_4 is high, the switch turns on; otherwise, it is off. Note that, when V_4 changes from low to high (or high to low), small time steps get forced. As V_4 settles down, the time steps become progressively larger, capped finally by h_{max}.

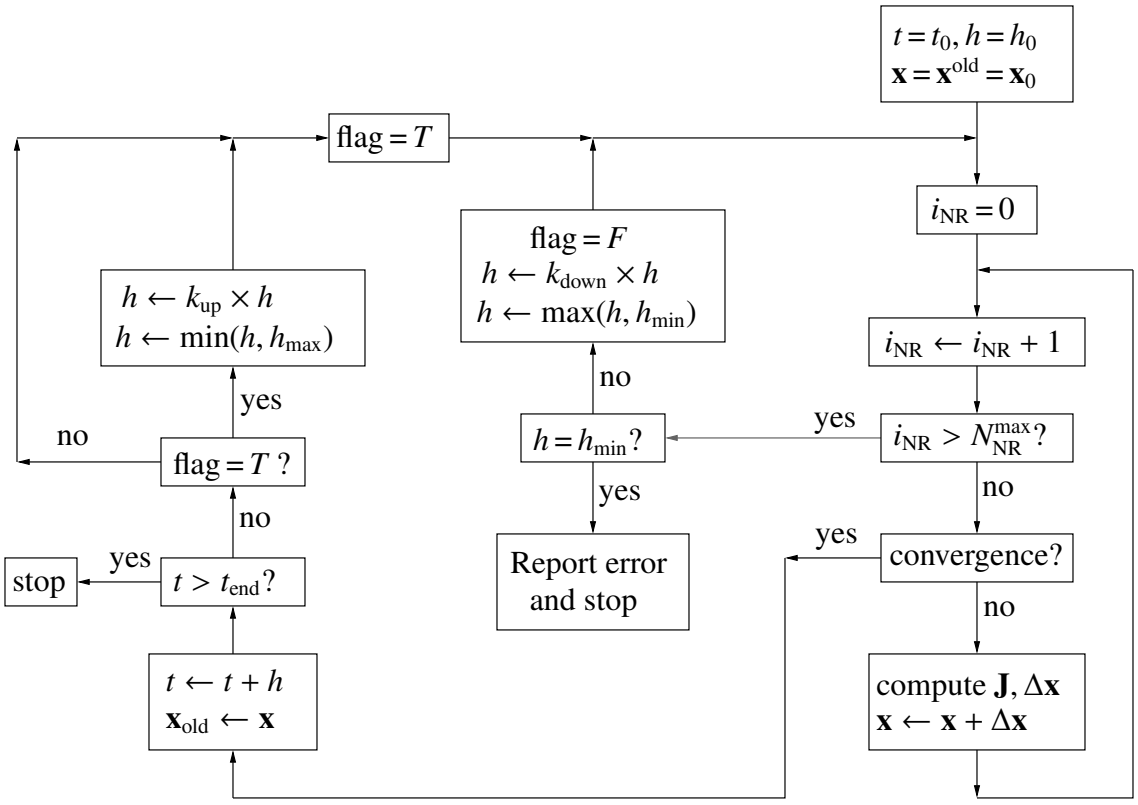


Figure 4.33: Flow chart for automatic time step adjustment using convergence of the NR process.

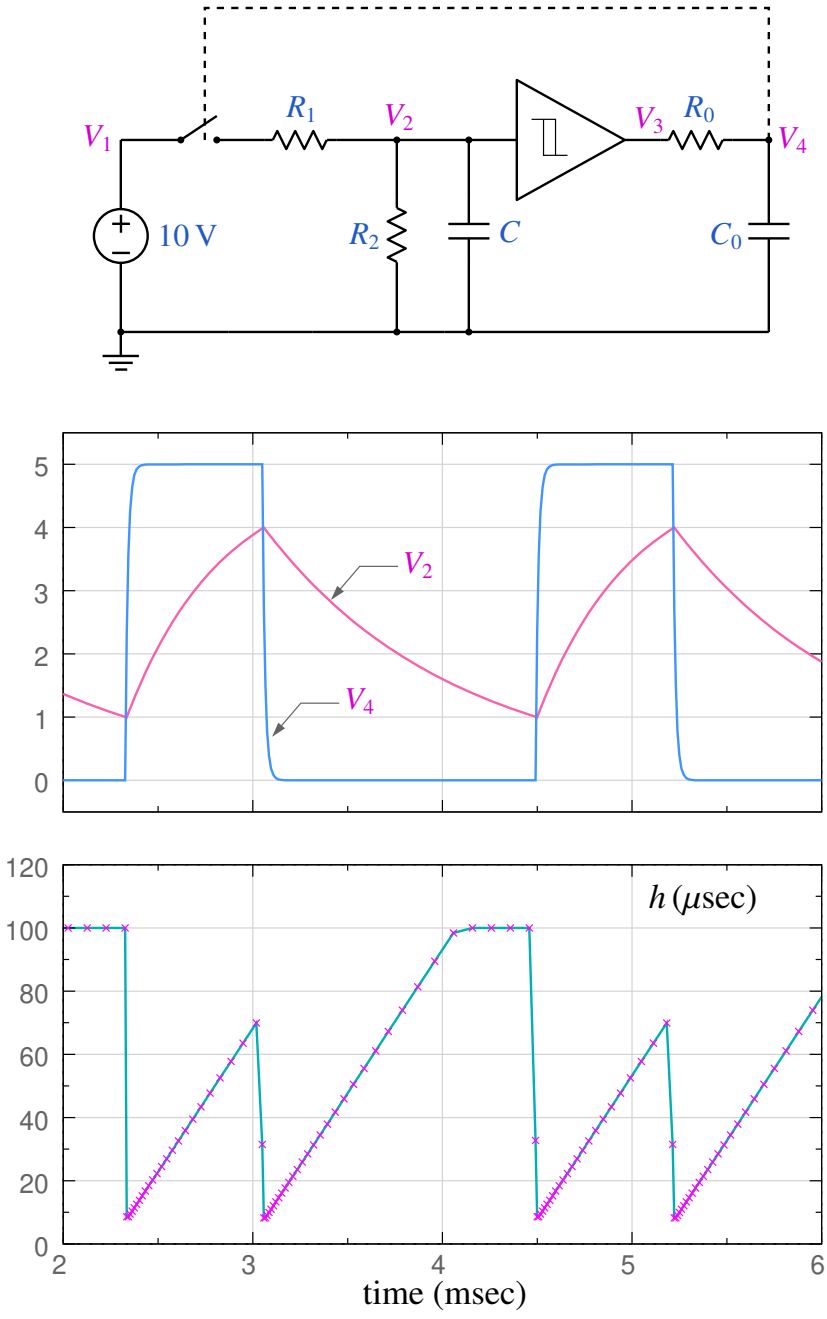


Figure 4.34: Example of automatic time step adjustment using convergence of the NR process. Parameter values are $R_1 = R_2 = 1\,\text{k}\Omega$, $R_0 = 100\,\Omega$, $C = 1\,\mu\text{F}$, $C_0 = 0.1\,\mu\text{F}$, $V_{IL} = 1\,\text{V}$, $V_{IH} = 4\,\text{V}$, $V_{OL} = 0\,\text{V}$, $V_{OH} = 5\,\text{V}$, $h_{\min} = 10^{-9}$ sec, $h_{\max} = 10^{-4}$ sec, $k_{\text{up}} = 1.1$, $k_{\text{down}} = 0.8$, $N_{\text{NR}}^{\max} = 10$.

Chapter 5

Periodic Steady-State (PSS) Analysis

Consider the boost converter circuit shown in Fig. 5.1 (a), with a clock frequency of 25 kHz (i.e., a period of 40 μs) and a duty cycle of 0.8. We are interested in the source current as a function of time.

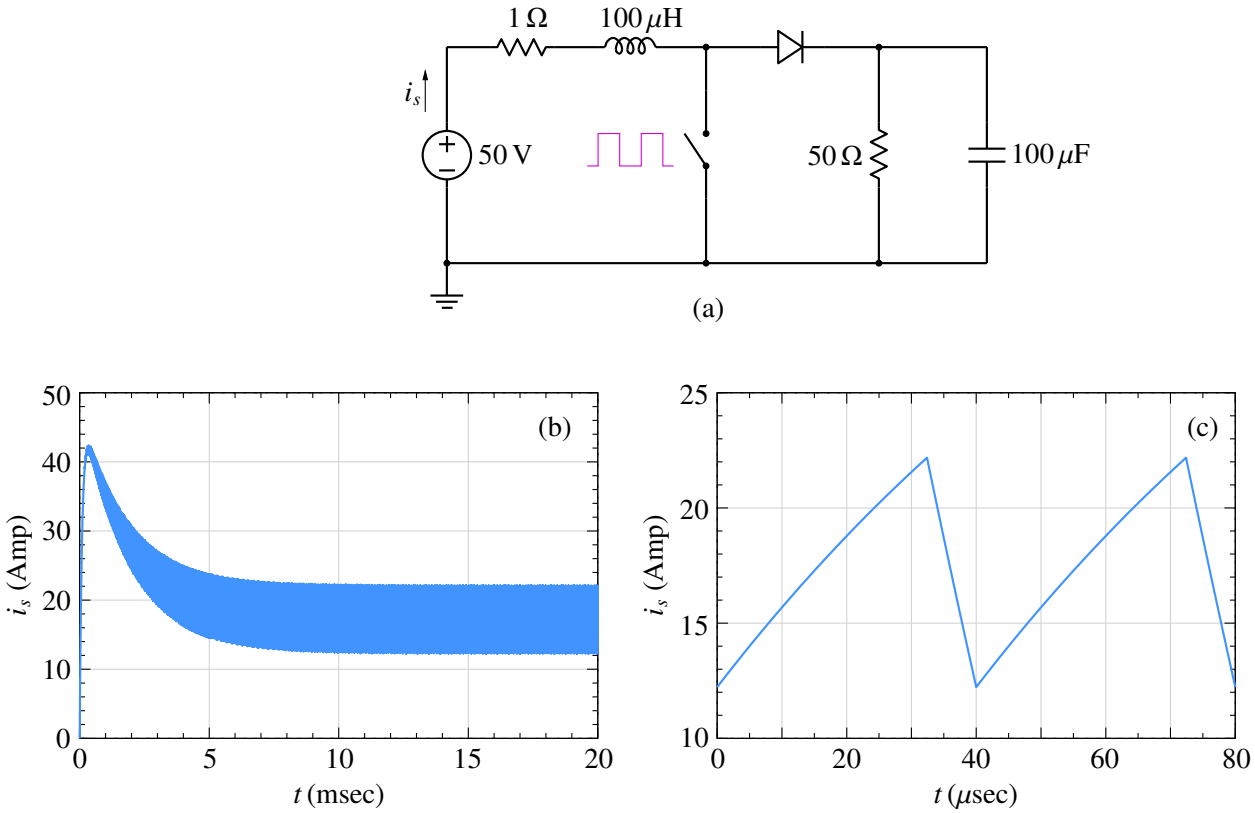


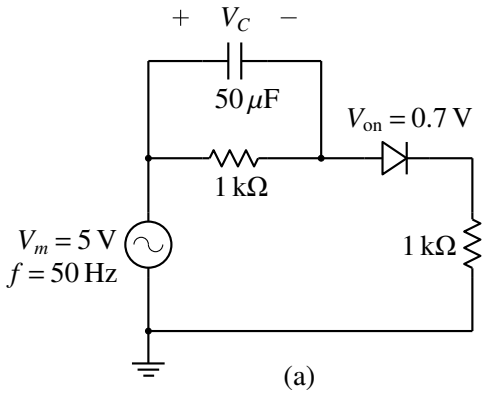
Figure 5.1: (a) Boost converter, (b) Source current obtained with transient simulation, (c) Steady-state source current.

Fig. 5.1 (b) shows the source current $i_s(t)$ obtained by transient simulation with a time step of 0.1 μsec, starting with zero initial conditions. The circuit goes through a relatively long transient and finally reaches the periodic steady state (PSS) at about 10 msec. Our interest is in the steady-state behaviour of the circuit[1], i.e., just *one* period in the steady state comprising a time interval of $T = 40\,\mu$sec (see Fig. 5.1 (c)). To get to that one cycle in the steady state, we have ended up

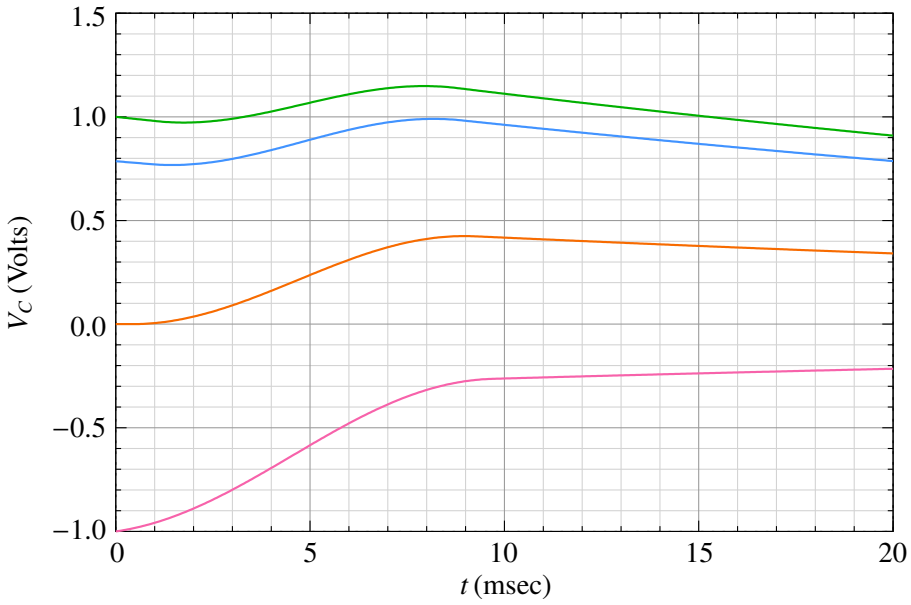
[1]This is also true about several other converter circuits.

simulating 10 msec/40 μsec or 250 cycles! Things get worse when the circuit time constants are larger.

Apart from being inefficient in such cases, transient simulation presents another practical difficulty because the time taken to reach the steady state is generally not known in advance. In that situation, we would end up following a trial-and-error approach – simulate the circuit for a certain time, check if the steady state has been reached; if not, increase the simulation time, and check again. That is cumbersome. Clearly, it is desirable to have some means of computing the steady-state waveform (SSW) *directly* rather than going through a long transient.



(a)



(b)

Figure 5.2: (a) Circuit to illustrate the SSW problem, (b) V_C versus time for different initial values.

The SSW problem has been addressed relatively early in the history of circuit simulation (see [13],[14])[2]. The basic idea behind SSW computation is illustrated in Fig. 5.2 with an example. In this circuit, the capacitor voltage V_C is the only state variable. If $V_C(t_0)$ is known, then the behaviour of the circuit for $t > t_0$ can be uniquely determined[3]. Fig. 5.2 (b) shows the results obtained with

[2]It is also possible to use a frequency-domain approach to compute the SSW solution (see [15], for example); we will describe only the time-domain approach here.

[3]In most cases of practical interest, the circuit response cannot be computed analytically, and we have to then employ transient simulation.

various values of $V_C(t_0)$ (where $t_0 = 0$) by performing transient simulation for one period of the source voltage, i.e., $T = 20$ msec. For example, consider $V_C(0) = 0$ V. In this case, we get $V_C(T) = 0.34$ V. This solution cannot be the periodic steady-state solution since $V_C(T) \neq V_C(0)$. We can try other values of $V_C(0)$ and check if the condition of periodicity is satisfied. As seen in the figure, $V_C(0) = 1$ V or -1 V also does not work. The correct value of $V_C(0)$ turns out to be 0.786 V (the blue curve).

If there is only one state variable x_s, we may be able to use a trial-and-error approach to find $x_s(0)$ such that $x_s(T) = x_s(0)$, but it is surely not a satisfactory approach. If the number of state variables increases, it would quickly become unmanageable.

Aprille and Trick [13] presented a systematic Newton-Raphson approach to compute the initial values of the state variables such that the condition $\mathbf{x}_s(T) = \mathbf{x}_s(0)$ is satisfied[4]. Fig. 5.3 shows the basic idea. For simplicity, the figure is drawn for the case of one state variable; however, the same procedure applies if the system has several state variables.

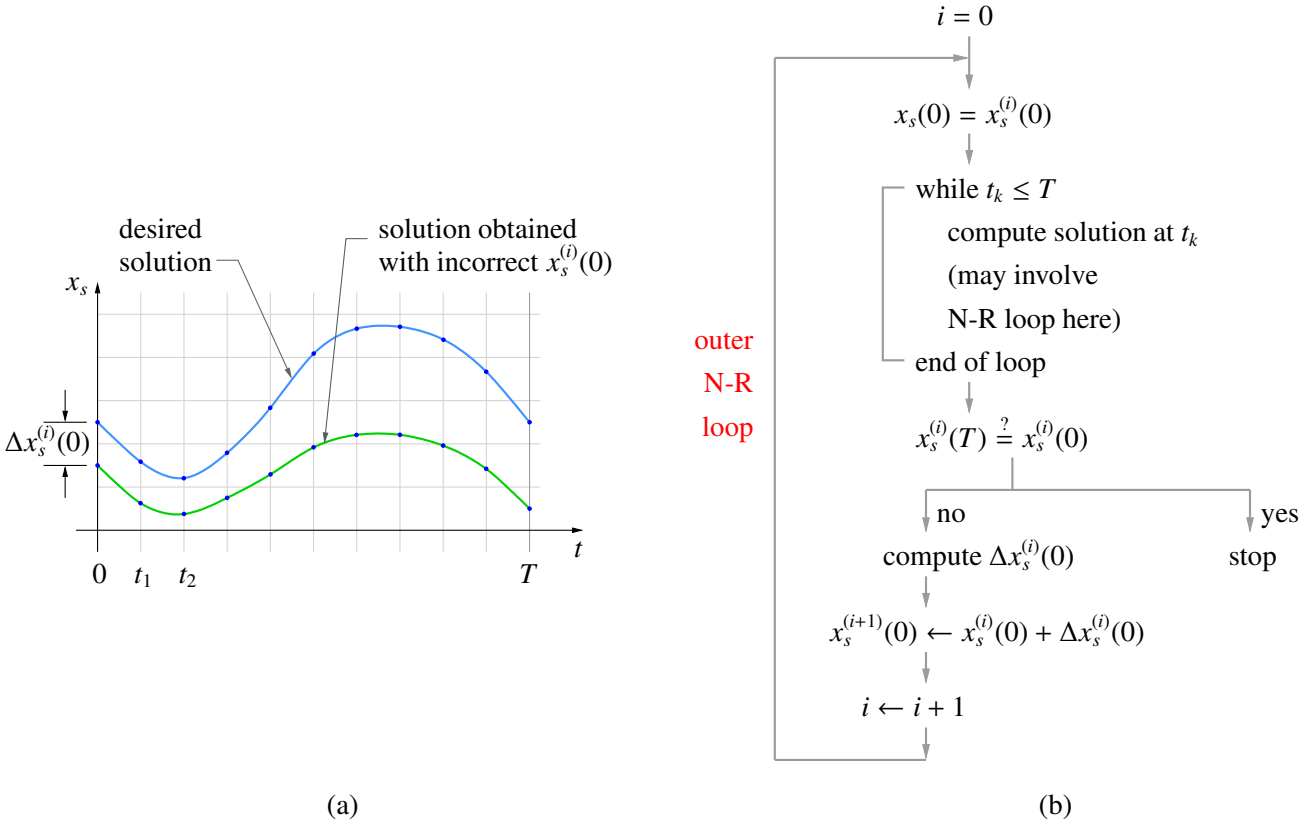


Figure 5.3: (a) Illustration of Newton-Raphson approach for SSW computation, (b) flow chart.

The SSW "outer loop" (see the flow chart in Fig. 5.3 (b)) is a Newton-Raphson loop for computing the state variable value at $t = 0$, i.e., $x_s(0)$. The integer i denotes the outer loop index. The value of $x_s(0)$ in the i^{th} outer loop iteration is denoted by $x_s^{(i)}(0)$. At the beginning of each outer loop, the state variable value is set to $x_s^{(i)}(0)$, and the system response is computed for one period. This computation involves several time points, as shown in Fig. 5.3 (a). Furthermore, at each time point, there may be an *inner* NR loop if the system is nonlinear.

[4]We have used $\mathbf{x}_s$ to denote the vector of the state variables.

We then check if $x_s(T)$ is equal to the starting value $x_s(0)$ (within a tolerance). If it is, our job is done; we have found the PSS solution. If not, the NR correction for $x_s(0)$ and the next iterate $x_s^{(i+1)}(0)$ are computed (in the outer NR loop), and the process is repeated. The Jacobian matrix for the outer NR loop is computed along with the response of the system with some extra computation, as explained in [13].

As we have seen earlier, convergence of the NR process depends on the initial guess, and that is true for the SSW NR loop (the outer NR loop in the flow chart of Fig. 5.3) as well. In our experience, convergence is generally not an issue for power electronic converter circuits – the SSW NR loop would converge starting with the zero initial condition, i.e., zero capacitor voltages and zero inductor currents. However, for some circuits, it may be required to perform transient simulation for a few cycles and then use the solution obtained as the initial guess for the SSW computation.

Chapter 6

GSEIM Circuit File

The block diagram of GSEIM, seen earlier in Chapter 1, is reproduced in Fig. 6.1. The circuit file contains the netlist of the circuit to be simulated, the connections, and parameter values for the elements. It also includes details such as simulation type, variables to be stored in the output files, parameters related to the simulation algorithm, etc. In Sec. 6.1, circuit files related to a few simple circuits are presented. A detailed description of the various statements of the circuit file are given in Sec. 6.2 and Sec. 6.3.

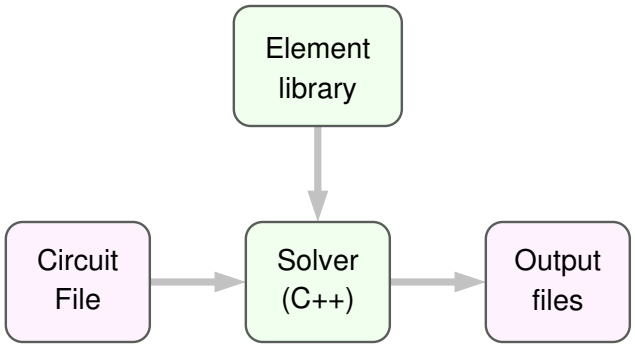


Figure 6.1: Block diagram of GSEIM.

6.1 Circuit File Examples

The circuit file is made up of two types of blocks. The "circuit block", enclosed between `begin_circuit` and `end_circuit`, describes the circuit netlist. The "solve block", enclosed between `begin_solve` and `end_solve`, is used to convey to the simulator the type of solution to be computed, algorithm parameter values, variables to be stored, names of output files to be created, etc. The following examples illustrate the syntax of the circuit file.

6.1.1 Example 1

The circuit file given below can be used to simulate the circuit in Fig. 6.2.

```
title: dc circuit analysis
begin_circuit
  eelement name=Vs type=vsrc_dc p=A n=0 vdc=5
```

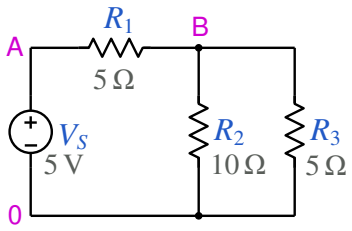


Figure 6.2: Circuit diagram for Sec. 6.1.1.

```
  eelement name=R1 type=r       p=A n=B r=5
  eelement name=R2 type=r       p=B n=0 r=10
  eelement name=R3 type=r       p=B n=0 r=5
  ref_node=0
  outvar:
+   VB=nodev_of_B
+   i_R1=i_of_R1
end_circuit
begin_solve
  solve_type=dc
  initial_sol initialize
  begin_output
    filename=ex_dc_1.dat
    variables: VB i_R1
  end_output
end_solve
end_cf
```

In the circuit block, the statements starting with `eelement` are used to assign nodes and parameter values to V_s, R_1, R_2, R_3. The `ref_node` statement specifies the reference node, i.e., the node with potential 0 V (see Chapter 2). The variables of interest, viz., the node voltage V_B and the current through R_1, are specified by the `outvar` statement.

In the solve block, the solution type is specified as `dc`. In the output block, enclosed between `begin_output` and `end_output`, the name of the output file to be created and the variables to be stored in that file are specified.

6.1.2 Example 2

Consider the circuit shown in Fig. 6.3. We are interested in the frequency response of this circuit; in particular, in the variation of $|\mathbf{V_o}|$ with frequency. The circuit file for this example is given below.

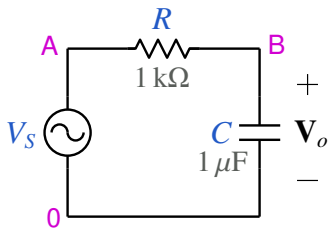


Figure 6.3: Circuit diagram for Sec. 6.1.2.

```
title: ac circuit analysis
begin_circuit
  eelement name=Vs type=vsrc_ac p=A n=0 a=1
  eelement name=R  type=r       p=A n=B r=1k
  eelement name=C  type=c       p=B n=0 c=1u
  ref_node=0
  outvar:
+   Vo=nodev_ac_of_B
end_circuit
begin_solve
  solve_type=ac
  initial_sol=initialize
  vary_freq from 1 to 10k type=log n_points=100
  begin_output
    filename=ex_ac_1.dat
    variables: mag_of_Vo
  end_output
end_solve
end_cf
```

In the `outvar` statement of the circuit block, the keyword `nodev_ac` is used to indicate that the phasor $\mathbf{V_o}$ is of interest. In the solve block, the solution type is specified as `ac`, meaning phasor analysis of the circuit. The `vary_freq` statement is used to specify the frequency values of interest – in this case, 100 values from 1 Hz to 10 kHz. The assignment `type=log` indicates that the successive frequency values should satisfy $f_{n+1} = k\, f_n$. In the output block, `mag_of_Vo` is given since we are interested in $|\mathbf{V_o}|$. When the simulation is completed, the output file `ex_ac_cct_1.dat` would be created with two columns: frequency and $|\mathbf{V_o}|$.

6.1.3 Example 3

A buck converter is shown in Fig. 6.4, with the associated circuit file given below.

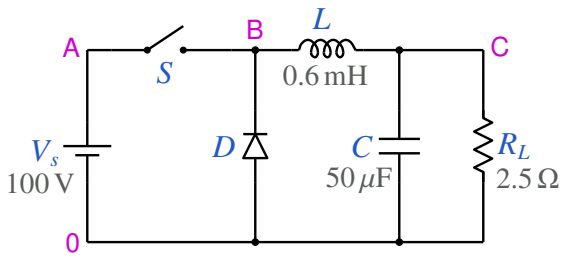


Figure 6.4: Circuit diagram for Sec. 6.1.3.

```
title: buck converter
begin_circuit
   eelement name=VS type=vsrc_dc  p=A  n=0  vdc=100
   eelement name=S  type=switch_1 p=A  n=B  x=g1 r_off=10M
   eelement name=D  type=diode_r  p=0  n=B  r_off=10M r_on=1m
   eelement name=L  type=l        p=B  n=C  l=0.6m
   eelement name=C  type=c        p=C  n=0  c=50u
   eelement name=R  type=r        p=C  n=0  r=2.5
```

```
   xelement name=clock1 type=clock_1 y=g1 D=0.4 f_hz=10k
+     delta1=0.01u delta2=0.01u

   ref_node=0
   outvar: IL=i_of_L
   outvar: IS=i_of_S
   outvar: ID=i_of_D
   outvar: IC=i_of_C
   outvar: v_out=v_of_R
   outvar: clock=y_of_clock1
end_circuit
begin_solve
   solve_type=trns
   initial_sol=initialize
   method: algorithm_trns=backward_euler
   method: nr_check_spice=yes
   method: t_start=0
   method: t_end=2m
   method: delt=5u
   begin_output
     filename=buck1.dat
     variables: IL IS ID IC v_out clock
   end_output
end_solve
end_cf
```

Switch S in Fig. 6.4 is driven by a clock signal with a certain frequency and duty ratio. This element (of type `clock_1`) is a non-electrical element, described by the `xelement` statement in the circuit block. Its output – specified by the `y=g1` assignment – is connected to variable `x` of the switch. In the solve section, the solution type is specified as `trns` (transient). The simulation algorithm to be used is backward Euler (see Chapter 4). The method cards `t_start`, `t_end`, `delt`, are used to specify the starting time, ending time, and time step, respectively, for the simulation. Since this is a nonlinear problem, the simulator will use the Newton-Raphson (NR) method at each time point. The method card with keyword `nr_check_spice` is used to convey that the SPICE convergence criteria should be used to check convergence of the NR iterations (see Chapter 3).

6.1.4 Example 4

In the buck converter of Sec. 6.1.3, the entire transient from $t = 0$ to $t = 2$ msec is simulated. With the clock frequency of 10 kHz, this interval corresponds to 20 cycles. In practice, our interest is often limited to the steady-state behaviour, i.e., just one cycle in the steady state. We can use the method described in Chapter 5 to get the steady-state behaviour directly. For this purpose, the following solve block can be used.

```
begin_solve
  solve_type=ssw
  initial_sol=initialize
  method: algorithm_trns=backward_euler
  method: nr_check_spice=yes
  method: ss_frequency=10k
  method: ss_period_mult_plot=2
  method: ss_ndiv=500
```

```
  method: t_start=0
  begin_output
    filename=buck2.dat
    variables: IL IS ID IC v_out clock
  end_output
end_solve
```

The method parameter `ss_frequency` is used by the simulator to compute the period. The period is divided by `ss_ndiv` to get the time step. The number of cycles to be plotted is given by `ss_period_mult_plot`.

6.2 Circuit Block Statements

As seen in Sec. 6.1, the purpose of the circuit block is to convey to the simulator the following information.

(a) Which elements are involved?

(b) How are the elements connected?

(c) What are the parameter values for the elements?

(d) Which node is to be treated as the reference node?

(e) Which variables are to be stored?

The following statements in the circuit block convey this information. (We will use the string `xx` to mean a value – integer, real, or string, depending on the context – to be assigned by the user.)

* `eelement`: specifies an electrical element. Its general syntax is given below.

  `eelement name=xx type=xx` *p1*`=xx` *p2*`=xx ...`

  The assignments have the following meaning.

  - `type`: type of the element (see Appendix A for the list of available elements)
  - `name`: name assigned by the user. GSEIM assigns a default name if it is not specified.
  - *p1*, *p2*, etc. are the various attributes of the element, such as
    - electrical node
    - non-electrical node
    - real parameter
    - integer parameter
    - start-up parameter

* `xelement`: specifies a non-electrical element. Its general syntax is given below.

  `xelement name=xx type=xx` *p1*`=xx` *p2*`=xx ...`

  The assignments have the following meaning.

  - `type`: type of the element (see Appendix B for the list of available elements)

- ◇ `name`: name assigned by the user. GSEIM assigns a default name if it is not specified.
- ◇ *p1*, *p2*, etc. are the various attributes of the element, such as
  - ▷ non-electrical node
  - ▷ real parameter
  - ▷ integer parameter
  - ▷ start-up parameter

* `ref_node`: specifies the node to be treated as the reference node ("ground"). It will be assigned a node voltage of 0 V (see Chapter 2).

* `outvar`: specifies variables of interest. The general syntax is

  `outvar: zz=xx_of_yy`

  where the string `_of_` is required. `zz` is the name of the output variable. The strings `xx` and `yy` take on different meanings, as listed below.

  - ◇ `xx`: `nodev`, `yy`: electrical node name
  - ◇ `xx`: `nodev_ac`, `yy`: electrical node name
  - ◇ `xx`: `xvar`, `yy`: non-electrical node name
  - ◇ `xx`: an output variable name of one of the circuit elements, `yy`: name of that element

In addition to the information covered above, the circuit block provides a facility to establish a correspondence between a non-electrical variable and a real parameter of one of the circuit elements. As an example, suppose we want a resistance value to be 10 Ω for $t < 1$ sec and 2 Ω for $t > 1$ sec. This can be accomplished with the following statements:

```
eelement name=R1 type=r p=A n=B r=5
xelement name=PWL1 type=pwl20 y=y1 n=2 t1=1 v1=10 t2=1.01 v2=2
cctrprm: R=r_of_R1
cctxvr: R0=y_of_PWL1
set_rparm R=R0
```

The element `PWL1` is used to create the step function `y1` such that $y_1 = 10$ for $t < 1$, $y_1 = 2$ for $t > 1.01$. The correspondence between y_1 and the parameter `r` of `R1` is obtained as follows.

(a) The `cctrprm` statement is used to define `R` as `r` of `R1`.

(b) The `cctxvr` statement is used to define `R0` as `y` of `PWL1` (i.e., y_1).

(c) Finally, the `set_rparm` statement establishes the required correspondence.

6.3 Solve Block Statements

While the circuit block defines the configuration of the circuit, the solve block defines the *action* that the simulator takes on the circuit. In particular, it is used to provide details such as

(a) What kind of solution is desired?

(b) How many output files are to be created? Which variables should be saved in those files?

(c) What are the values of parameters related to the simulation type being requested? For example, if transient simulation is requested, what is the time interval of interest?

In the following, we describe the solve block statements. The `method` statement and output blocks – which are also solve block attributes – are discussed separately.

* `solve_type=xx` specifies the type of analysis. The options are as follows (see Chapter 1).
  - ◇ `dc`: DC analysis. Time derivatives, if any, are set to zero.
  - ◇ `ac`: phasor analysis.
  - ◇ `trns`: transient simulation.
  - ◇ `startup`: start-up analysis.
  - ◇ `ssw`: periodic steady-state computation.
  - ◇ `sss`: periodic steady-state computation.
  - ◇ `ssf_rk4`: periodic steady-state computation.

  For the `ssw` option, the method described in [13], [14] is used. The `sss` option is similar to `ssw` except that the outer Jacobian entries (see Fig. 5.3) are computed using normal transient simulation by perturbing the initial state variable values. The `ssf_rk4` option uses the RK4 method (see Chapter 4) for transient simulation; it is restricted to pure flow graphs (i.e., systems with no electrical elements).

* `initial_sol=xx` specifies how the initial solution (which serves as the "initial guess" in the Newton-Raphson method) should be obtained. The options are
  - ◇ `initialize`: variables are initialized, as specified in the element templates.
  - ◇ `previous`: the solution from the previous solve block is used as the initial solution.

* `set_parm xx_of_yy=zz` is used to assign the value `zz` to parameter `xx` of element `yy`. It can be used only in dc simulation.

* `vary_parm xx_of_yy from p1 to p2 type=linear/log n_points=N`

  This statement is used to vary parameter `xx` of element `yy` from `p1` to `p2`. The number of values is given by `N`. If `type` is `linear`, the successive values are related as $p_{n+1} = p_n + \Delta$. If `type` is `log`, the successive values are related as $p_{n+1} = k \times p_n$.

  `vary_parm` can be used only in dc simulation.

* `set_freq val=xx` is used to set the frequency to the given value in ac simulation.

* `vary_freq from p1 to p2 type=linear/log n_points=N`

  This statement is used to vary the frequency in ac simulation from `p1` to `p2`. The number of values is given by `N`. If `type` is `linear`, the successive values are related as $f_{n+1} = f_n + \Delta$. If `type` is `log`, the successive values are related as $f_{n+1} = k \times f_n$.

6.3.1 Output block

An output block – which appears within a solve block – is enclosed in the `begin_output` and `end_output` statements. It is composed of the following statements.

* `filename=xx append=yes/no limit_lines=xx`

  specifies the output file name to be created. `append` and `limit_lines` are optional assignments. If `append=yes` is used, the file is assumed to be existing (as an output of a previous solve block), and the simulation results are appended at the end of the file. `limit_lines` is a safety feature meant to stop simulation if the number of lines in the file exceeds the given number. Its default value is `1000000` (one million).

* `variables: xx xx ...`

  specifies the variables to be written to the output file. In ac simulation, `xx` must start with either `mag_of_` or `phase_of_` (see Sec. 6.1.2).

* `control: yy=xx yy=xx ...`

  specifies parameters that control how the output information is written.

  For transient and periodic steady-state analysis, the following assignments can be used.

  - ◇ `fixed_interval=xx` specifies that the variables specified in the output block should be written at fixed intervals. If this option is not specified, the variables are written at each time point.
  - ◇ `out_tstart=xx` is used to start writing the variables only after the specified time. The default is the starting time of the simulation.
  - ◇ `out_tend=xx` is used to write the variables only up to the specified time. The default is the end time of the simulation.

  In ac analysis, phase angles are restricted to $-180°$ to $180°$. In a phase versus frequency plot, this can lead to a step change from $-180°$ to $180°$ (or *vice versa*). The purpose of the following assignments is to prevent these steps, and it is achieved by adding suitable multiples of $360°$.

  - ◇ `min_phase=xx`: If `min_phase` is specified, the phase data is written to the output file such that the phase angle is always larger than the specified angle.
  - ◇ `max_phase=xx`: If `max_phase` is specified, the phase data is written to the output file such that the phase angle is always smaller than the specified angle.

  Either `min_phase` or `max_phase` can be specified, but not both.

6.3.2 `method` statement

The `method` statement is used to specify parameters related to the solution type and the algorithms to be used. In the following, we group them in some broad categories.

6.3.2.1 Algorithm type

- ◇ `algorithm_trns=xx` specifies the algorithm to be used for transient and periodic steady-state simulation. The options are (see Chapter 4):
  - ▷ `RK4`: Runge-Kutta method of order 4
  - ▷ `RKF45`: Runge-Kutta-Fehlberg method
  - ▷ `backward_euler`: backward Euler method
  - ▷ `backward_euler_auto`: backward Euler method with variable time step
  - ▷ `trz`: trapezoidal method
  - ▷ `trz_auto`: trapezoidal method with variable time step

  The `RK4` and `RKF45` options can be used only for pure flow graphs (i.e., circuits with only non-electrical elements).

- ◇ `algorithm_startup=xx` specifies the algorithm to be used for start-up simulation for a pure flow graph. There are two options.
  - ▷ `explicit`: to obtain the solution by visiting the elements one by one (see Sec. 4.1.6)
  - ▷ `implicit`: to obtain the solution by solving the element equations simultaneously

6.3.2.2 Newton-Raphson procedure

The Newton-Raphson (NR) method, described in Chapter 3, is used in a variety of situations: DC, transient, periodic steady-state (PSS), and start-up. The NR process is the same in all of these situations. Most of the NR method parameters apply therefore in general.

- ◇ `nr_itermax` (integer): maximum number of NR iterations, default: 500. Typically, the NR method converges in less than ten iterations. A large default value is assigned to `nr_itermax` take care of special cases for which convergence is very slow (e.g., when there are exponential functions in the circuit/system being simulated, and the initial guess is poor).

  In transient simulation, when the `backward_euler_auto` or `trz_auto` option is used, `nr_itermax` should be set to a much smaller number, say, 5.

- ◇ `nr_dmp`: (`yes`/`no`) decides whether damping (see Eq. 3.19) should be used, default: `no`.

- ◇ `nr_dmp_k` (real number): damping factor k where $0 < k < 1$ (see Eq. 3.19, default: 0.2). Not relevant when `nr_dmp` is set to `no`.

- ◇ `nr_dmp_itermax` (integer): number of NR iterations for which damping is applied, default: 50. Not relevant when `nr_dmp` is set to `no`.

- ◇ `nr_spice_reltol` (real number): k_{rel} in Eq. 3.10, default: 10^{-3}.

- ◇ `nr_spice_vntol` (real number): τ_{abs} for node voltages (see Eq. 3.10, default (in Volts): 10^{-4}.

- ◇ `nr_spice_abstol` (real number): τ_{abs} for currents, default (in Amps): 10^{-6}.

- ◇ `nr_eps_rhs` (real number): tolerance value for the 2-norm (see Eq. 3.9), default: 10^{-6}.
- ◇ `nr_check_rhs2`: (`yes`/`no`) decides whether the 2-norm (see Eq. 3.9) should be used to check for convergence. default: `no`
- ◇ `nr_check_spice`: (`yes`/`no`) decides whether the SPICE convergence criteria (see Sec. 3.3) should be used to check for convergence. default: `no`

As discussed in Chapter 3, convergence of the NR process depends on the initial guess. Convergence at the very first time point in transient simulation is more difficult because we may not have a good initial guess to start the NR process. For subsequent time points, the solution obtained at the previous time point generally serves as an excellent initial guess, and convergence is easier. For this reason, NR parameters for the first solution are made available separately, as given below.

- ◇ `nr_itermax0` (integer): maximum number of NR iterations for the first solution, default: 500.
- ◇ `nr_dmp0`: (`yes`/`no`) decides whether damping should be used for the first solution, default: `no`
- ◇ `nr_dmp_k0` (real number): damping factor k ($0 < k < 1$) for the first solution, default: 0.1. Not relevant when `nr_dmp0` is set to `no`.
- ◇ `nr_dmp_itermax0` (integer): number of NR iterations for which damping is applied for the first solution, default: 50. Not relevant when `nr_dmp0` is set to `no`.

As shown in Fig. 5.3, steady-state analysis involves an outer Newton-Raphson (NR) loop. NR parameters related to this loop are described below.

- ◇ `ss_nr_itermax` (integer): maximum number of NR iterations allowed (default: 50).
- ◇ `ss_nr_dmp` (`yes`/`no`): decides whether damping (see Eq. 3.19) should be used (default: `no`)
- ◇ `ss_nr_dmp_k` (real number): damping factor k ($0 < k < 1$) (see Eq. 3.19), default: 0.6.
- ◇ `ss_nr_dmp_itermax` (integer): number of NR iterations for which damping is to be applied, default: 10.
- ◇ `ss_nr_eps_rhs` (real number): specifies the tolerance to check convergence, default: 10^{-10}

6.3.2.3 Transient simulation

In the following, we describe parameters related to transient simulation. Some of these parameters would also apply to periodic steady state (PSS) simulation since transient simulation is involved in computing the PSS solution.

- ◇ `t_start` (real number): starting time for transient simulation
- ◇ `t_end` (real number): ending time for transient simulation
- ◇ `itmax_trns` (integer): maximum number of time points (default: 1000000). This is a "safety feature." If the user by mistake creates conditions – e.g., a very small time step – requiring a larger number of time points, the program will get terminated with an error message. If there is a genuine requirement, the user should increase `itmax_trns` suitably.

- ◇ `delt` (real number): serves as the constant time step for constant time step methods (`backward_euler`, `trz`, `RK4`), and as the first time step for methods with adaptive time steps (`backward_euler_auto`, `trz_auto`, `RKF45`).
- ◇ `delt_min` (real number): smallest time step allowed (default: $0.01 \times$ `delt`)
- ◇ `delt_max` (real number): largest time step allowed (default: $10 \times$ `delt`)
- ◇ `rkf45_tolr_abs` (real): absolute tolerance in checking convergence of the RKF45 algorithm, default: 10^{-5}
- ◇ `rkf45_tolr_rel` (real): relative tolerance in checking convergence of the RKF45 algorithm, default: 10^{-5}
- ◇ `rkf45_fctr_min` (real): minimum factor by which time step can be reduced, default: 0.8
- ◇ `rkf45_fctr_max` (real): maximum factor by which time step can be increased, default: 1.1
- ◇ `factor_step_increase` (real): factor by which time step is increased in `backward_euler_auto`, `trz_auto` algorithms, default: 1.5
- ◇ `factor_step_decrease` (real): factor by which time step is decreased in `backward_euler_auto`, `trz_auto` algorithms, default: 0.6

6.3.2.4 Periodic steady-state simulation

- ◇ `ss_period` (real number): waveform period (T in Fig. 5.3)
- ◇ `ss_frequency` (real number): waveform frequency

  Either `ss_period` or `ss_frequency` must be specified.
- ◇ `ss_period_mult_compute` (integer): number of periods used in computation of the periodic steady state solution, default: 2
- ◇ `ss_period_mult_plot` (integer): number of periods for which variables of interest are to be saved in the output file(s), default: 2
- ◇ `ss_ndiv` (integer): number of divisions in one period, default: 200

Appendix A

List of Electrical Elements

In the following table, we list the electrical elements available in the GSEIM library. The detailed descriptions of the elements can be found in the GSEIM distribution in the form of `pdf` files. For example, the element `ammeter` is described in `be_ammeter.pdf` (`be` stands for "basic element").

Element	Function
`ammeter`	current measurement
`ammeter_fb`	current measurement with current made available as a variable
`battery_c`	used in modelling of a battery with SOC
`battery_r`	used in modelling of a battery with SOC
`battery_vsrc`	used in modelling of a battery with SOC
`c`	capacitor
`cccs`	current-controlled current source
`ccvs`	current-controlled voltage source
`dcmc`	dc machine model
`diode_r`	diode model with V_{on}, R_{on}, R_{off}
`diode_spice`	diode model used in SPICE
`igbt_1`	simple model of IGBT
`indmc1_2`	3-phase induction machine model
`isrc_ac`	ac current source
`isrc_dc`	dc current source
`isrc_x`	current source with current given by input variable x
`l`	inductor
`r`	resistor
`solar_module_1`	solar module model (together with `solar_module_rs`)
`solar_module_rs`	solar module model (together with `solar_module_1`)

Element	Function
solar_module_2	solar module behavioral model
switch_1	R_{on}/R_{off} switch controlled by an input variable
thyristor	simple thyristor model
vccs	voltage-controlled current source
vcvs	voltage-controlled voltage source
voltmeter	voltage measurement
voltmeter_fb	voltage measurement with voltage made available as a variable
vsrc_ac	ac voltage source
vsrc_clock	clock voltage source
vsrc_dc	dc voltage source
vsrc_pulse10	voltage source with up to 10 pulses
vsrc_x	voltage source with voltage given by input variable x
xfmr_l1l2	basic single-phase transformer model without coil resistances and leakage inductances
xfmr_level0_1ph	ideal single-phase transformer model without self and mutual coil inductances, coil resistances, and leakage inductances
xfmr_level0_1ph_1_2	ideal transformer with one primary and two secondary windings
xfmr_level0_1ph_2_2	ideal transformer with two primary and two secondary windings
xfmr_level0_1ph_1_3	ideal transformer with one primary and three secondary windings
z	impedance specified as $R + jX$ (AC simulation only)
z_polar	impedance specified in polar form (AC simulation only)

Appendix B

List of Directional (Flow-graph) Elements

In the following table, we list the directional (flow-graph) elements available in the GSEIM library. The detailed descriptions of the elements can be found in the GSEIM distribution in the form of `pdf` files. For example, the element `sum_2` is described in `be_sum_2.pdf` (`be` stands for "basic element").

Element	Function
`abc_to_alphabeta_1`	(a, b, c) to (α, β) transformation
`abc_to_dq`	(a, b, c) to (d, q) transformation
`abc_to_dq0_2`	(a, b, c) to (d, q) transformation
`abc_to_dq_2`	(a, b, c) to (d, q) transformation
`abs`	$y = \|x\|$
`abs_2`	$y = \sqrt{x_1^2 + x_2^2}$
`alphabeta_to_abc_1`	(α, β) to (a, b, c) transformation
`alphabeta_to_dq_1`	(α, β) to (d, q) transformation
`alphabeta_to_dq_2`	(α, β) to (d, q) transformation
`and_2`	`AND` of `x1`, `x2`
`and_3`	`AND` of `x1`, `x2`, `x3`
`atan2_rad`	$\theta = \tan^{-1}(y/x)$
`average_mv_1`	compute the moving average (`y`) of the input signal (`x`). The averaging is performed between two active edges of the clock signal (`clk`).
`average_mv_2`	compute the moving average (`y`) of the input signal (`x`) over a time interval `T`
`clock`	square wave source
`clock_1`	square wave source
`clock_1a`	square wave source
`clock_1b`	square wave source

Element	Function
clock_3	square wave source
clock_3ph	square wave source
clock_thyr	square wave source, useful particularly for thyristor circuits
clock_1_dual	source with square wave and its complement
cmpr_1_1	$y = y_{\text{high}}$ if $x > x_0$; else, y_{low}.
cmpr_1_2	$y_1 = y_{\text{high}}$ if $x > x_0$; else, y_{low}. y_2 is the complement of y_1.
cmpr_2_1	$y = y_{\text{high}}$ if $x_1 > x_2$; else, y_{low}.
cmpr_2_2	$y_1 = y_{\text{high}}$ if $x_1 > x_2$; else, y_{low}. y_2 is the complement of y_1.
cmpr_band_1_1	$y = y_{\text{high}}$ if $x_{\text{low}} < x < x_{\text{high}}$; else, y_{low}.
cmpr_simple_2_1	$y = y_{\text{high}}$ if $x_1 > x_2$; else, y_{low}. This element should be used only if the simulation time steps are small.
cmpr_simple_2_2	$y_1 = y_{\text{high}}$ if $x_1 > x_2$; else, y_{low}. y_2 is the complement of y_1. This element should be used only if the simulation time steps are small.
cmprh_1_1	comparator with hysterisis, with input x and output y.
cmprh_2_1	comparator with hysterisis, with inputs x1, x2, and output y.
cmprh_2_2	comparator with hysterisis, with inputs x1, x2, and outputs y1, y2 (which are complementary).
const	$y = k$
cos	$y = \cos(x)$
dead_zone	$y = x - x_{\text{min}}$ if $x < x_{\text{min}}$, $x - x_{\text{max}}$ if $x > x_{\text{max}}$, 0 otherwise.
decoder_2_4	a mapping between inputs x0, x1 and outputs y0, y1, y2, y3, where x0, x1 are 0 or 1.
diff	$y = x_1 - x_2$
div	$y = k\, x_1 / x_2$
dq_to_abc	(d, q) to (a, b, c) transformation
dq_to_abc_2	(d, q) to (a, b, c) transformation
dq_to_alphabeta_1	(d, q) to (α, β) transformation
dq_to_alphabeta_2	(d, q) to (α, β) transformation
edge_delay	shift a clock or PWM-type signal by a delay interval
edge_delay_1	shift a clock signal by a delay interval
jkff	behavioral model of JK flip-flop
indmc1	induction manchine model

Element	Function
`integrator`	$y = k \int x\,dt$
`integrator_1`	$y = \frac{k}{\tau} \int x\,dt$
`integrator_modulo_twopi`	$y = k \int x\,dt$, with $-2\pi < y < 2\pi$
`lag_1`	make output `y` lag input `x`
`lag_2`	make output `y` lag input `x`
`limiter`	$y = x_{\min}$ if $x < x_{\min}$, $x_{\max}$ if $x > x_{\max}$, x otherwise.
`linear`	$\frac{a_1}{a_2} y = \frac{b_1}{b_2} x + \frac{c_1}{c_2}$
`max`	$y = \max(x_1, x_2)$
`min`	$y = \min(x_1, x_2)$
`min_max_3`	$y_{\min} = \min(x_1, x_2, x_3)$, $y_{\max} = \max(x_1, x_2, x_3)$.
`modulo`	restrict y if $x > x_2$ or $x < x_1$
`modulo_twopi`	restrict y if $x > \pi$ or $x < -\pi$
`monostable_1`	generate a pulse of width T when an active edge is detected at intput `x`
`multscl`	$y = k\,x$
`mult_2`	$y = k\,x_1\,x_2$
`nand_2`	`NAND` of `x1`, `x2`
`nor_2`	`NOR` of `x1`, `x2`
`not`	`NOT` of `x`
`or_2`	`OR` of `x1`, `x2`
`or_3`	`OR` of `x1`, `x2`, `x3`
`pole_complex_order_1`	$y = \left[\frac{a + j\,b}{s - (\alpha + j\beta)} + \frac{a - j\,b}{s - (\alpha - j\beta)} \right] x.$
`pole_complex_order_2`	$y = \left[\frac{a + j\,b}{(s - (\alpha + j\beta))^2} + \frac{a - j\,b}{(s - (\alpha - j\beta))^2} \right] x.$
`pole_real_order_1`	$y = \frac{a}{s - \alpha}\, x.$
`pole_real_order_2`	$y = \frac{a}{(s - \alpha)^2}\, x.$
`pole_real_order_3`	$y = \frac{a}{(s - \alpha)^3}\, x.$

Element	Function
pole_real_order_4	$y = \frac{a}{(s-\alpha)^4}\, x.$
pole_real_order_5	$y = \frac{a}{(s-\alpha)^5}\, x.$
pulse10	source with up to 10 pulses
pwl10_xy	generate piecewise linear function $y = f(x)$ with up to 10 break points
pwl20	generate piecewise linear function $y = f(t)$ with up to 20 break points
pwm20_1	generate up to 10 pulses which repeat with a specified frequency
sampler	sample a signal at uniform intervals
sampler_1	sample a signal at uniform intervals
sampler_edge	sample x and make it available as output y when an active edge is detected at the clock input.
signal_switch	$y = x_1$ if s is high; else, $x2$.
signum	$y = 1$ if $x > 0$, -1 if $x < 0$, 0 if $x = 0$.
sin	$y = \sin(x)$
sincos	$s = \sin(x)$, $c = \cos(x)$.
sincos2	$s = \sin(x)$, $c = \cos(x)$, $s_2 = \sin(2\,x)$, $c_2 = \cos(2\,x)$.
xx	xx
xx	xx
sop_3_6	sum-of-products function y in terms of x_1, x_2, x_3.
sop_4_6	sum-of-products function y in terms of x_1, x_2, x_3, x_4.
sop_6_6	sum-of-products function y in terms of x_1, x_2, x_3, x_4, x_5, x_6.
sqr	$y = x^2$
sqrt	$y = \sqrt{x}$
src_ac	$y(t) = A\ \sin(2\pi f(t - t_0) + \phi) + x_0\,.$
srff_nand	behavioral model of SR flip-flop with NAND gates
srff_nor	behavioral model of SR flip-flop with NOR gates
sum_2	$y = k_1 x_1 + k_2 x_2$
sum_3	$y = k_1 x_1 + k_2 x_2 + k_3 x_3$
triangle_1	triangle wave source
triangle_2	symmetric triangle wave source

Element	Function
`triangle_3`	symmetric triangle wave source
`triangle_4`	triangle wave source
`vsi_3ph_1`	behavioral model of ideal 3-phase inverter
`xfer_fn`	$y(s) = \dfrac{a_0 + a_1\, s + a_2\, s^2 + a_3\, s^3 + a_4\, s^4 + a_5\, s^5}{b_0 + b_1\, s + b_2\, s^2 + b_3\, s^3 + b_4\, s^4 + b_5\, s^5}\, x(s)\,.$
`xor_2`	`XOR` of `x1`, `x2`

Bibliography

[1] https://in.mathworks.com/products/simulink.html.

[2] https://in.mathworks.com/products/simscape.html.

[3] https://powersimtech.com/products/psim/capabilities-applications/.

[4] M.B. Patil, Ruchita Korgaonkar, and Kumar Appaiah, "GSEIM: a general-purpose simulator with explicit and implicit methods," *Sādhanā*, vol. 46, no. 4, pp. 1–13, 2021.

[5] https://www.ee.iitb.ac.in/~sequel/nb_gseim.html.

[6] M.B. Patil. SEQUEL Users' Manual. [Online]. Available: http://www.ee.iitb.ac.in/~microel/faculty/mbp/sequel1.html

[7] W.J. McCalla, *Fundamentals of Computer-Aided Circuit Simulation*. Boston: Kluwer Academic Publishers, 1987.

[8] M.B. Patil, V. Ramanarayanan, and V.T. Ranganathan, *Simulation of power electronic circuits*. New Delhi: Narosa, 2009.

[9] T. Tuma and A. Bűrmen, *Circuit Simulation with SPICE OPUS*. Boston: Birkhäuser, 2009.

[10] L.F. Shampine, *Numerical Solution of Ordinary Differential Equations*. New York: Chapman and Hall, 1994.

[11] L. Lapidus and J.H. Seinfeld, *Numerical Solution of Ordinary Differential Equations*. New York: Academic Press, 1971.

[12] R.L. Burden and J.D. Faires, *Numerical Analysis*. Singapore: Thomson, 2001.

[13] T.J. Aprille and T.N. Trick, "Steady-state analysis of nonlinear circuits with periodic inputs," *Proc. IEEE*, vol. 60, pp. 108–114, 1972.

[14] ——, "A computer algorithm to determine the steady-state response of nonlinear oscillators," *IEEE Trans. Circuit Theory*, vol. 19, pp. 352–360, 1972.

[15] M.S. Nakhla and F.H. Branin, "Determining the periodic response of nonlinear systems by a gradient method," *Int. J. Circuit Theory Appl.*, vol. 5, pp. 255–273, 1977.