SEQUEL Users' Manual: Part 1

Mahesh B. Patil

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

To the child
who brought the King
back to his senses

Contents

Chapter 1

Introduction

SEQUEL is a general-purpose circuit simulation package developed at IIT Bombay [1]. It can be used for simulation of analog circuits, digital circuits, mixed-signal circuits, and power electronic circuits. In this first part of the SEQUEL manual, we will look at numerical techniques which are commonly used in circuit simulation: how the circuit equations are assembled, how nonlinear equations are solved, how ODEs are solved, etc. This background will help in understanding what a circuit simulator does "behind the scenes", why it fails when it does, what is the remedy in that case, etc. In short, it is intended to make the readers more aware about the technical aspects of circuit simulation so that they can use it more effectively. In that sense, it is different than manuals of commercial simulation programs which tell the user how to connect wires, how to set component values, how to view graphs, but disclose only the minimum possible details, if any, about how things are implemented. That is because the vendors need to guard all their secrets – however insignificant they may be – to make sure that no other vendor benefits from their disclosure. One of the famous fairy tales [2] by Hans Christian Andersen is relevant in this context. And what a lovely story.

1.1 The Emperor's New Clothes

Many years ago there lived an emperor who loved beautiful new clothes so much that he spent all his money on being finely dressed. His only interest was in going to the theater or in riding about in his carriage where he could show off his new clothes. He had a different costume for every hour of the day. Indeed, where it was said of other kings that they were at court, it could only be said of him that he was in his dressing room!

One day two swindlers came to the emperor's city. They said that they were weavers, claiming that they knew how to make the finest cloth imaginable. Not only were the colors and the patterns extraordinarily beautiful, but in addition, this material had the amazing property that it was to be invisible to anyone who was incompetent or stupid.

"It would be wonderful to have clothes made from that cloth," thought the emperor. "Then I would know which of my men are unfit for their positions, and I'd also be able to tell clever people from stupid ones." So he immediately gave the two swindlers a great sum of money to weave their cloth for him.

They set up their looms and pretended to go to work, although there was nothing at all on the looms. They asked for the finest silk and the purest gold, all of which they hid away, continuing to work on the empty looms, often late into the night.

"I would really like to know how they are coming with the cloth!" thought the emperor, but he was a bit uneasy when he recalled that anyone who was unfit for his position or stupid would not be able to see the material. Of course, he himself had nothing to fear, but still he decided to send someone else to see how the work was progressing.

"I'll send my honest old minister to the weavers," thought the emperor. He's the best one to see how the material is coming. He is very sensible, and no one is more worthy of his position than he.

So the good old minister went into the hall where the two swindlers sat working at their empty looms. "Goodness!" thought the old minister, opening his eyes wide. "I cannot see a thing!" But he did not say so.

The two swindlers invited him to step closer, asking him if it wasn't a beautiful design and if the colors weren't magnificent. They pointed to the empty loom, and the poor old minister opened his eyes wider and wider. He still could see nothing, for nothing was there. "Gracious" he thought. "Is it possible that I am stupid? I have never thought so. Am I unfit for my position? No one must know this. No, it will never do for me to say that I was unable to see the material."

"You aren't saying anything!" said one of the weavers.

"Oh, it is magnificent! The very best!" said the old minister, peering through his glasses. "This pattern and these colors! Yes, I'll tell the emperor that I am very satisfied with it!" (see Fig. 1.1)

"That makes us happy!" said the two weavers, and they called the colors and the unusual pattern by name. The old minister listened closely so that he would be able say the same things when he reported back to the emperor, and that is exactly what he did.

Figure 1.1: The weaver and the minister, the weaver displaying his fabric which the minister is trying hard to appreciate [3].

The swindlers now asked for more money, more silk, and more gold, all of which they hid away. Then they continued to weave away as before on the empty looms.

The emperor sent other officials as well to observe the weavers' progress. They too were startled when they saw nothing, and they too reported back to him how wonderful the material was, advising him to have it made into clothes that he could wear in a grand procession. The entire city was alive in praise of the cloth. "Magnifique! Nysseligt!

Excellent!" they said, in all languages. The emperor awarded the swindlers with medals of honor, bestowing on each of them the title Lord Weaver.

The swindlers stayed up the entire night before the procession was to take place, burning more than sixteen candles. Everyone could see that they were in a great rush to finish the emperor's new clothes. They pretended to take the material from the looms. They cut in the air with large scissors. They sewed with needles but without any thread. Finally they announced, "Behold! The clothes are finished!"

The emperor came to them with his most distinguished cavaliers. The two swindlers raised their arms as though they were holding something and said, "Just look at these trousers! Here is the jacket! This is the cloak!" and so forth. "They are as light as spider webs! You might think that you didn't have a thing on, but that is the good thing about them."

"Yes," said the cavaliers, but they couldn't see a thing, for nothing was there.

"Would his imperial majesty, if it please his grace, kindly remove his clothes." said the swindlers. "Then we will fit you with the new ones, here in front of the large mirror."

The emperor took off all his clothes, and the swindlers pretended to dress him, piece by piece, with the new ones that were to be fitted. They took hold of his waist and pretended to tie something about him. It was the train. Then the emperor turned and looked into the mirror.

"Goodness, they suit you well! What a wonderful fit!" they all said. "What a pattern! What colors! Such luxurious clothes!"

"The canopy to be carried above your majesty awaits outside," said the grandmaster of ceremonies.

"Yes, I am ready!" said the emperor. "Don't they fit well?" He turned once again toward the mirror, because it had to appear as though he were admiring himself in all his glory.

The chamberlains who were to carry the train held their hands just above the floor as if they were picking up the train. As they walked they pretended to hold the train high, for they could not let anyone notice that they could see nothing.

The emperor walked beneath the beautiful canopy in the procession, and all the people in the street and in their windows said, "Goodness, the emperor's new clothes are incomparable! What a beautiful train on his jacket. What a perfect fit!" No one wanted it to be noticed that he could see nothing, for then it would be said that he was unfit for his position or that he was stupid. None of the emperor's clothes had ever before received such praise.

"But he doesn't have anything on!" said a small child.

"Good Lord, let us hear the voice of an innocent child!" said the father, and whispered to another what the child had said.

"A small child said that he doesn't have anything on!"

Finally everyone was saying, "He doesn't have anything on!"

The emperor shuddered, for he knew that they were right, but he thought, "The procession must go on!" He carried himself even more proudly, and the chamberlains walked along behind carrying the train that wasn't there.

1.2 Weavers and their tricks

Vendors of commercial simulators are somewhat like the weavers in our story. They need to glorify their product so that the user remains interested (to the extent that he or she will pay for it). This exercise leads to interesting consequences – mundane stuff appears exotic, limitations sound like features, common sense gets packaged as patents, bug fixes get sold as updates, benchmarking becomes benchmarketing, people with PhDs start talking like salespersons, and so on. The weavers[1] would want us to believe that there are layers and layers of secrets in their product while in reality there are none.

The fact of the matter is that the science behind circuit simulation has not really changed all that much after the SPICE program was written and made available in the public domain in 1973 (see [4]). The "machinery" for assembling and solving circuit equations has been around for decades, and the Newton-Raphson method for handling nonlinear equations dates back to the 17^{th} century! This means that the core of all circuit simulation packages must be necessarily the same. The difference may be in the bells and whistles, e.g., in the way a circuit schematic is entered or how the output plots are shown to the user or which components are made available to the user. The weavers may tell us anything, but there *isn't* any new fabric in this business; what may be new is the dress you can make with the same old fabric. That is what eventually matters to the user of course, but we should not confuse a new dress with a new fabric. If we had enough time (and a bit of programming skill), we could make the dress ourselves!

In the following, we list some of the most interesting – and sometimes entertaining – statements made by weavers in the area of circuit simulation. We will also present alternative interpretations of each statement. There are three players in the game.

(a) **cleverVendor**: the equivalent of the weaver in our story. He has certain goods, and his primary goal is to sell them at as high a price as the market allows. If he has a grey item to sell, he may describe it as grey or black or white, whichever is more convenient.

(b) **averageUser**: the equivalent of the average spectator watching the King's procession. He has too many day-to-day worries, and no time to question the weavers. He wants to get a simulator which will do his job at a price he can afford, and get on with life.

(c) **whistleBlower**: the equivalent of the child who finally called a spade a spade. He is relatively free and can afford to look at things critically. If he finds something interesting or odd, he feels obliged to share it with everyone around.

Vendor statements/claims

* **cleverVendor**: `Xxxxx` accepts alphanumeric names as well as numbers to represent nodes. There is no limit to the number of characters allowed in a node name.

  averageUser: Wow! That means I can have a variable with a hundred characters now.

  (Pauses, thinks for ten minutes). Ah, here is an example:

  `this_variable_represents_the_collector_current_of_transistor_Q1`

  Perhaps, I could have used `Ic_Q1` instead, but if I pay for a feature, I must use it.

[1] We will use the terms "weavers" and "vendors of circuit simulation packages" interchangeably.

whistleBlower: Hmm, the vendor has probably run out of ideas for improving his sales performance.

* **cleverVendor**: In `Xxxxxxx` models, algebraic loops are algebraic constraints. Models with algebraic loops define a system of differential algebraic equations. `Xxxxxxx` does not solve DAEs directly. `Xxxxxxxx` solves the algebraic equations (the algebraic loop) numerically for x_a at each step of the ODE solver.
  ...
  Use `Yyyyyyyy`™ to model systems that span mechanical, electrical, hydraulic, and other physical domains as physical networks.

  averageUser: How considerate! If there is a problem with one of his products, the vendor offers another.

  whistleBlower: Are there better options? Maybe cheaper as well?

* **cleverVendor**: `XXX` is an analog electronic circuit simulator working with ideal and piecewise-linear components. ··· ··· `XXX` perfectly fits the needs of all users, regardless of their experience, interests, and expectations.

  averageUser: Great! I must check this out.

  whistleBlower: Piecewise-linear analog circuit simulator good enough for everyone? Really? Can you simulate a common-emitter amplifier and give us the base-emitter voltage accurate up to three decimal places? How about a Wilson current mirror or a CMOS amplifier or a sample-and-hold circuit?

* **cleverVendor**: Computing a discrete state requires knowing the relationship between its value at the current time step and its value at the previous time step. This is referred to this relationship[2] as the state's update function. A discrete state depends not only on its value at the previous time step but also on the values of a model's inputs. Modeling a discrete state thus entails modeling the state's dependency on the systems' inputs at the previous time step. `Xxxxxxxx` block diagrams use specific types of blocks, called discrete blocks, to specify update functions and chains of blocks connected to the inputs of discrete blocks to model the dependency of a system's discrete states on its inputs.

  averageUser: ?! (dazed and breathless)

  whistleBlower: ?! (dazed)

* **cleverVendor**: Whenever you start a simulation, enable display of port data types, or refresh the port data type display, the `Xxxxxxxx` software performs a processing step called data type propagation. This step involves determining the types of signals whose type is not otherwise specified and checking the types of signals and input ports to ensure that they do not conflict. If type conflicts arise, an error dialog is displayed that specifies the signal and port whose data types conflict. The signal path that creates the type conflict is also highlighted.

  averageUser: Sounds so complex! The product must be really quite good.

[2]This paragraph (and others) have been taken verbatim from the manual pages. It looks like the weavers have not found enough time to proofread what they wrote!

> **whistleBlower**: Why burden the user with jargon? Can you not make it transparent to the user with some clever programming? Also, if my problem has only real numbers, do I still need to pay for all data types?

There are many, many more of these, but we can stop here. In the rest of this manual, we will ignore what the weavers tell us and focus on what actually happens inside a circuit simulator. It is much simpler to understand that since there is no hidden agenda, no ulterior motive, no fudging, no beating of drums, no tall claims, no desire to obfuscate and conquer; just a plain technical description of things, where black is black, white is white, and grey is grey.

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)[1]. 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 [5],[6], 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.

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

[1]Some weavers try to pass off (at an exorbitant price) an ODE solver as a circuit simulator, but its limitations – which the weavers have cleverly pushed under the rug – surface as soon as the program is put through some moderately difficult tests.

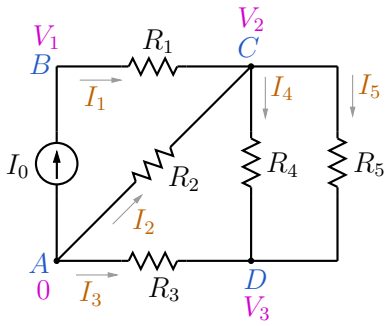


Figure 2.1: Nodal analysis example.

In terms of node voltages, we have

$$\begin{aligned} -I_0 + G_1(V_1 - V_2) &= 0\,, \\ -G_1(V_1 - V_2) + G_2 V_2 + (G_4 + G_5)(V_2 - V_3) &= 0\,, \\ G_3 V_3 - (G_4 + G_5)(V_2 - V_3) &= 0\,. \end{aligned} \tag{2.2}$$

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,

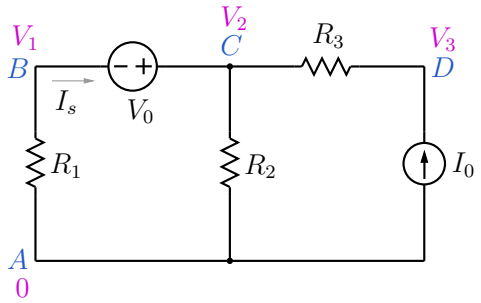


Figure 2.2: Modified Nodal analysis example.

V_3 to the 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[2]. 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 equations comes from the element equation for the voltage source, viz., $V_2 - 0 = V_0$. The equations can be written in a matrix form:

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

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[3] 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?

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

[3]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.

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

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 (or pay large sums to any weaver for this purpose).

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 shortly see how it can be extended to nonlinear circuits in a DC situation. In a later chapter, 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) method is most commonly used because of its excellent convergence properties. 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[1], i.e., $f(r)=0$. Let us say that we have some idea of the root in the form of an *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

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

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

$$f(x) = a_3x^3 + a_2x^2 + a_1x + a_0, \quad \text{with } a_3{=}1,\ a_2{=}{-}3.2,\ a_1{=}8.7,\ a_0{=}{-}14.3\,. \tag{3.3}$$

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.

[1]For simplicity, we will assume that the equation has a single real root.

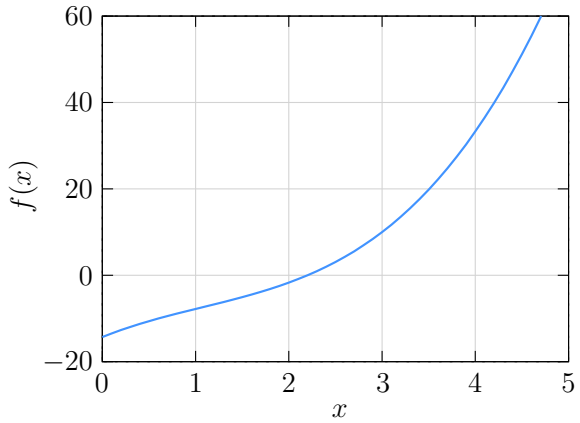


Figure 3.1: Plot of $f(x)$ given by Eq. 3.3.

```
#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

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}	8.181818e-01
1	2.929260×10^{0}	8.861467×10^{0}	-5.646248×10^{-1}	7.292605×10^{-1}	3.314820e-01
2	2.364636×10^{0}	1.601388×10^{0}	-1.548606×10^{-1}	1.646356×10^{-1}	7.483436e-02
3	2.209775×10^{0}	8.966910×10^{-2}	-9.739489×10^{-3}	9.774978×10^{-3}	4.443172e-03
4	2.200035×10^{0}	3.243738×10^{-4}	-3.548854×10^{-5}	3.548901×10^{-5}	1.613137e-05
5	2.200000×10^{0}	4.282173×10^{-9}	$-4.685091 \times 10^{-10}$	4.685092×10^{-10}	2.129587e-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.

"errors" for iterations i and $(i+1)$ are related by

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

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., $\frac{1}{2}\left.\frac{d^2 f}{dx^2}\right|_{x=r}$. 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[2], 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

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

[2]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).

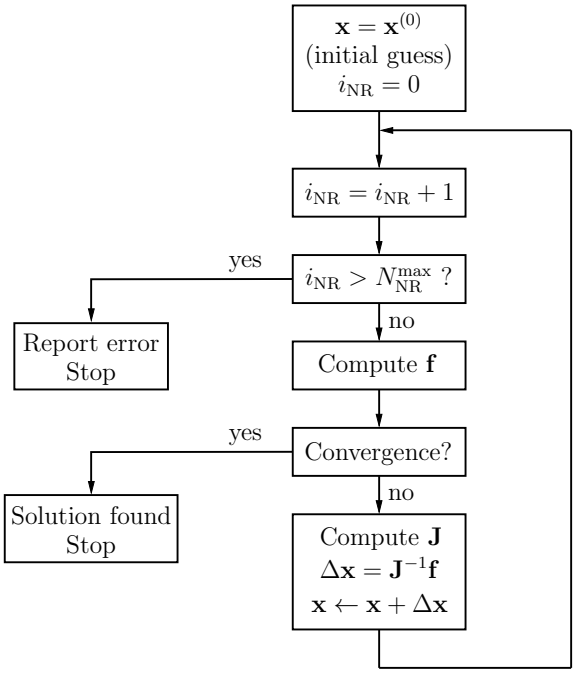


Figure 3.2: Flow chart for the Newton-Raphson procedure.

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[3] $x_1 = \sqrt{3}$, $x_2 = 5\sqrt{3}$. 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 [7], for example) would generally work well and may need to be tweaked only for a few specific simulations.

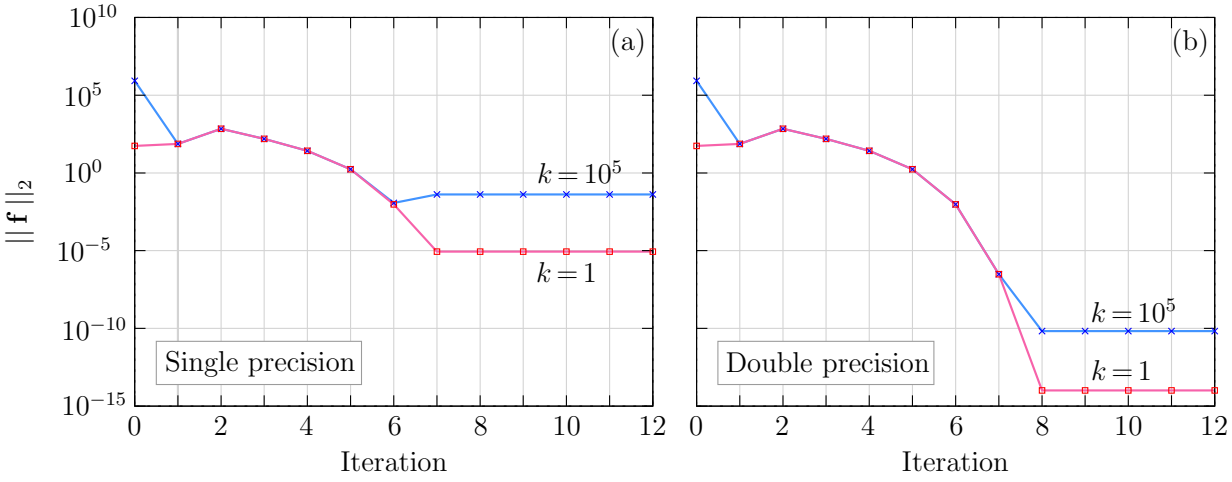


Figure 3.3: $||\,\mathbf{f}\,||_2$ versus NR iteration number for the system of equations given by Eq. 3.12: (a) single-precision arithmetic, (b) double-precision arithmetic.

[3]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.

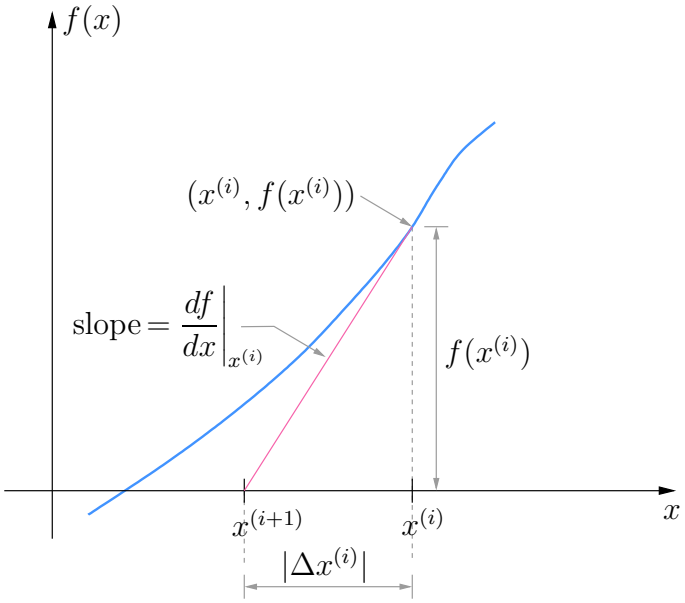


Figure 3.4: Graphical interpretation of the NR process.

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

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

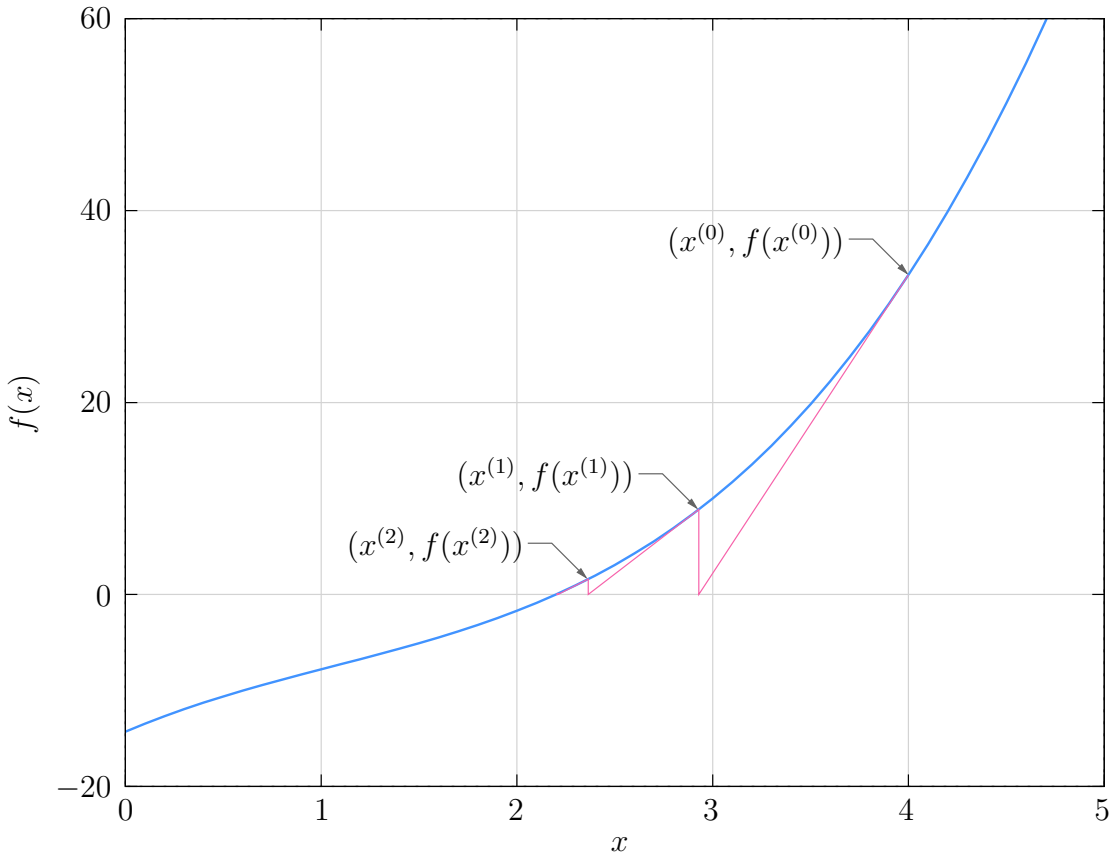


Figure 3.5: Graphical interpretation of NR process with $f(x)$ given by Eq. 3.3 and $x^{(0)} = 4.0$.

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.

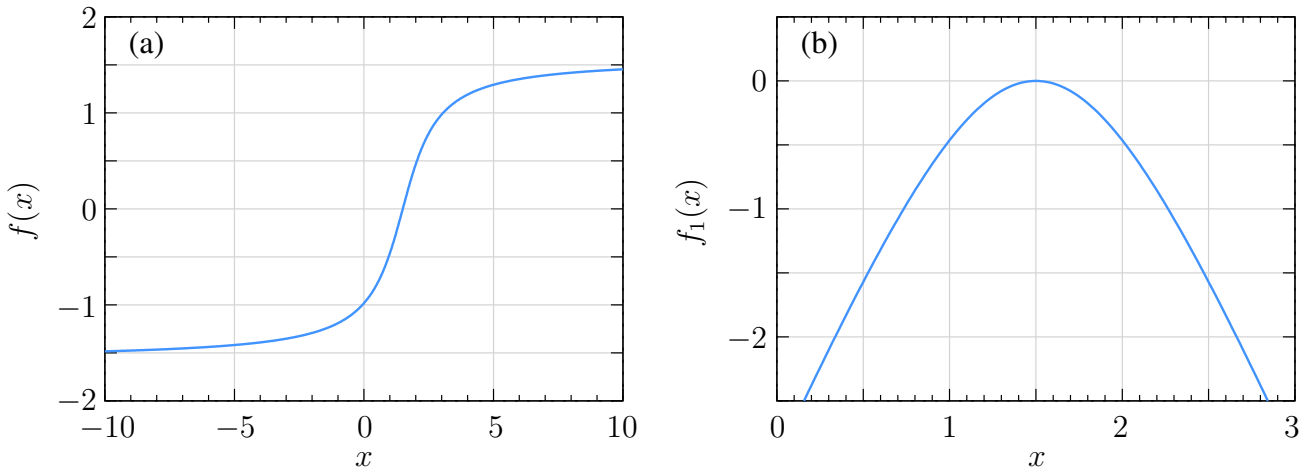


Figure 3.6: (a) $f(x)$ (Eq. 3.15) and (b) $f_1(x)$ (Eq. 3.16) versus x

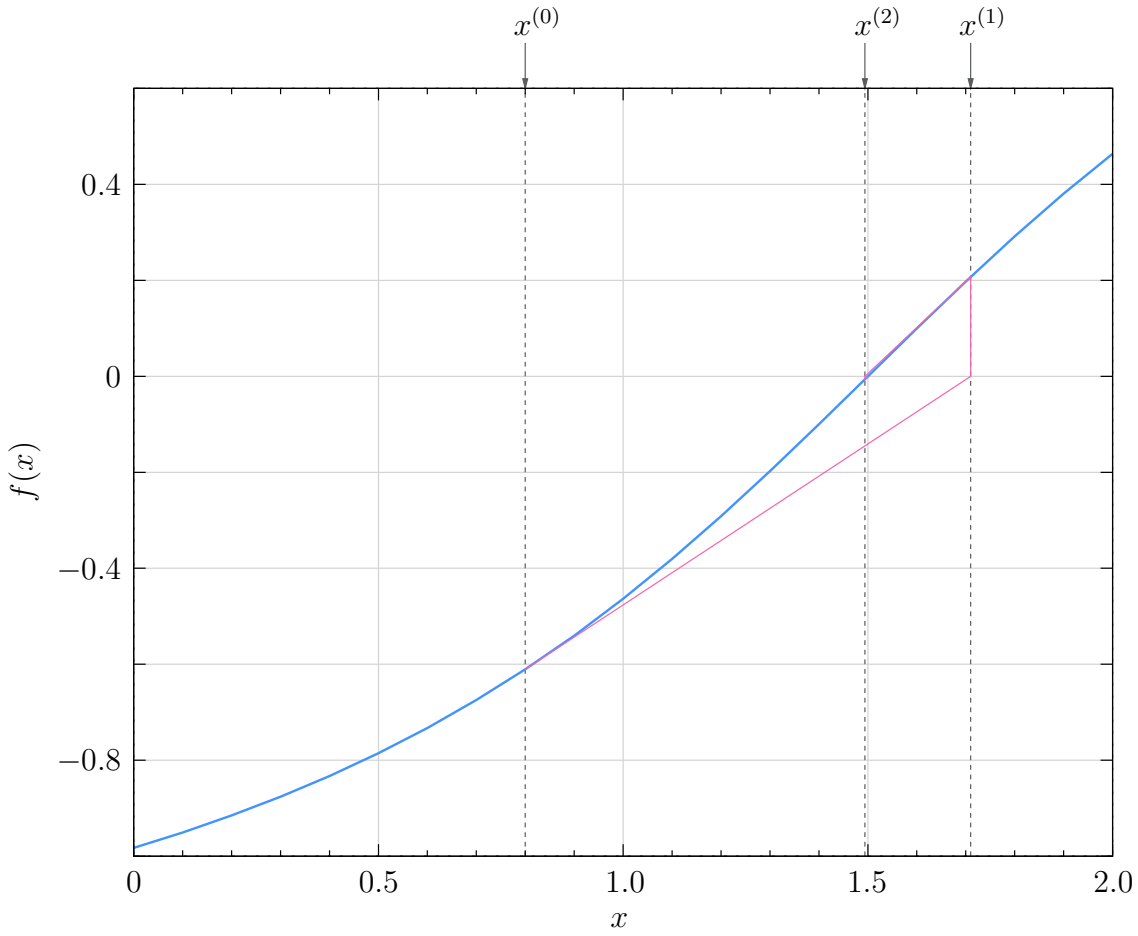


Figure 3.7: NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 0.8$.

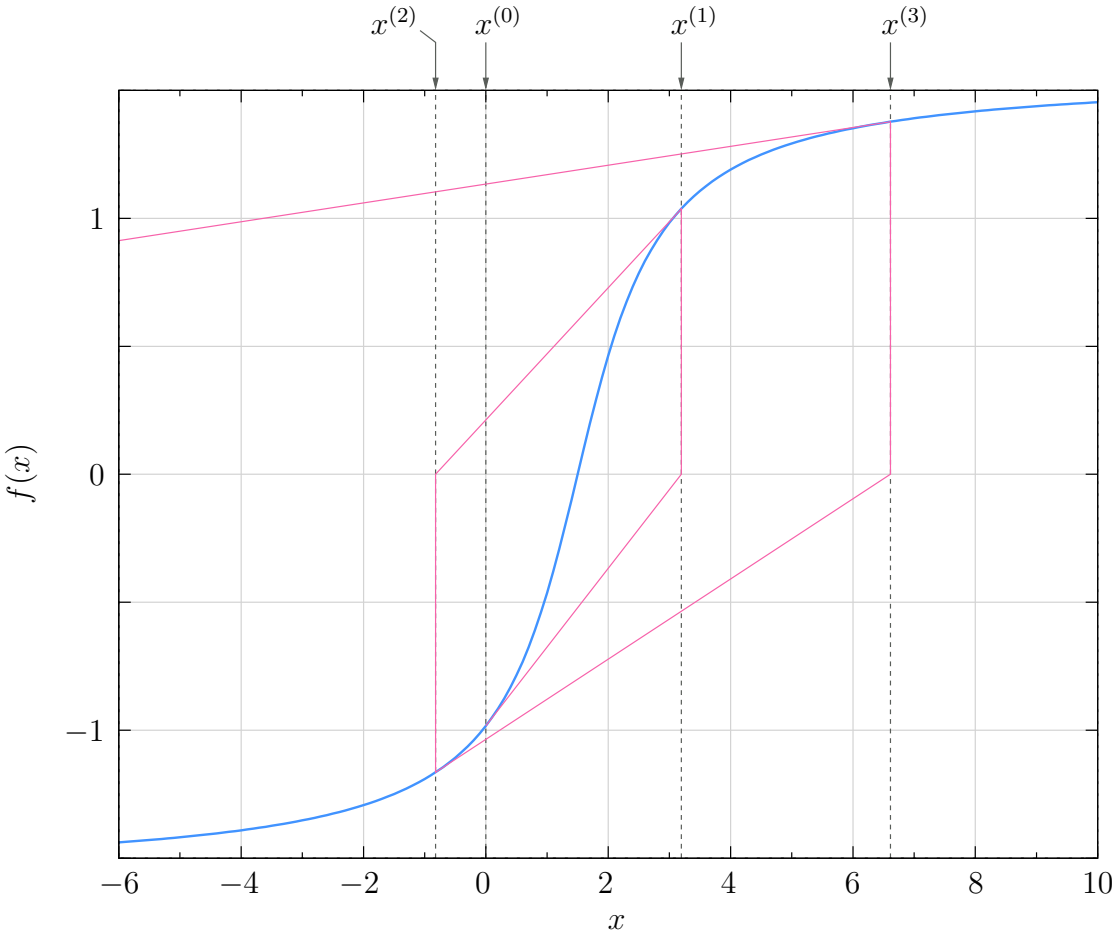


Figure 3.8: NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 0$.

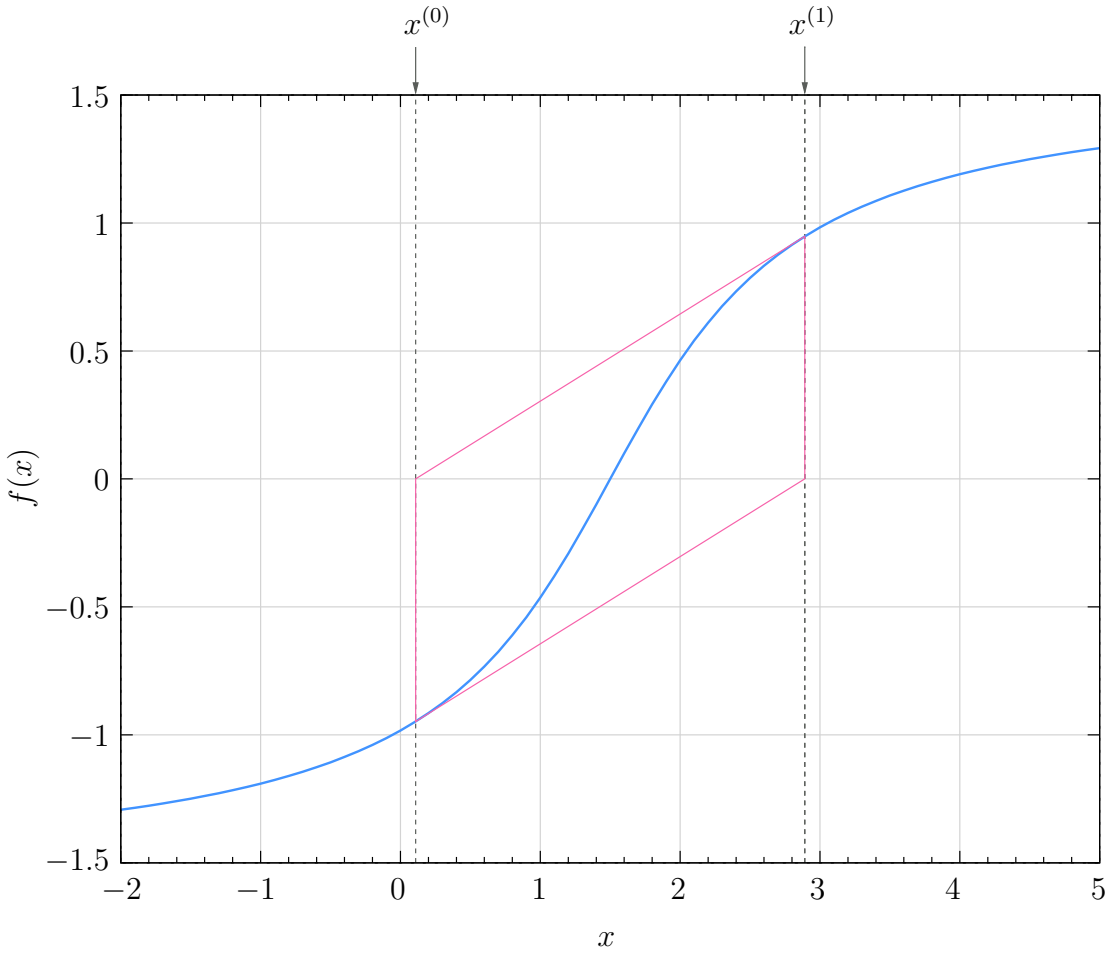


Figure 3.9: NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 2.89175$.

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*[4] (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 tne 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.

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

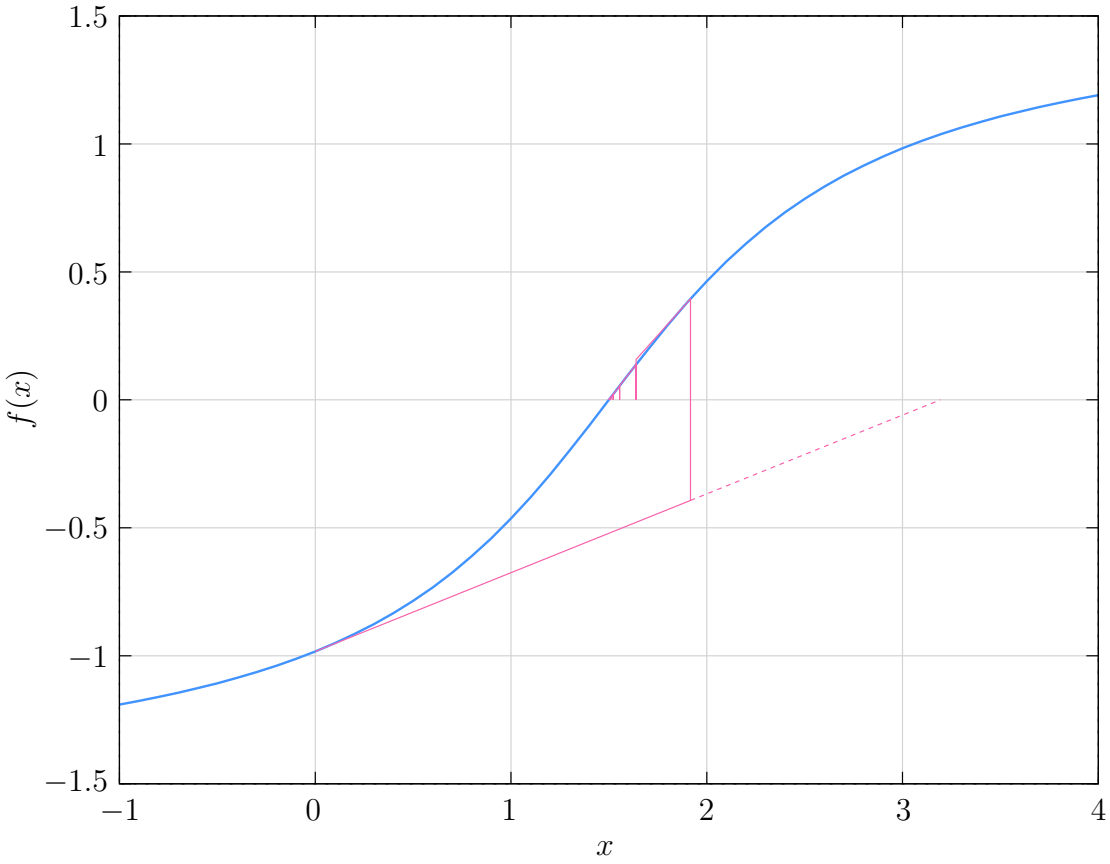


Figure 3.10: Damped NR process for $f(x) = \tan^{-1}(x - 1.5)$ with $x^{(0)} = 0$, and $k = 0.6$.

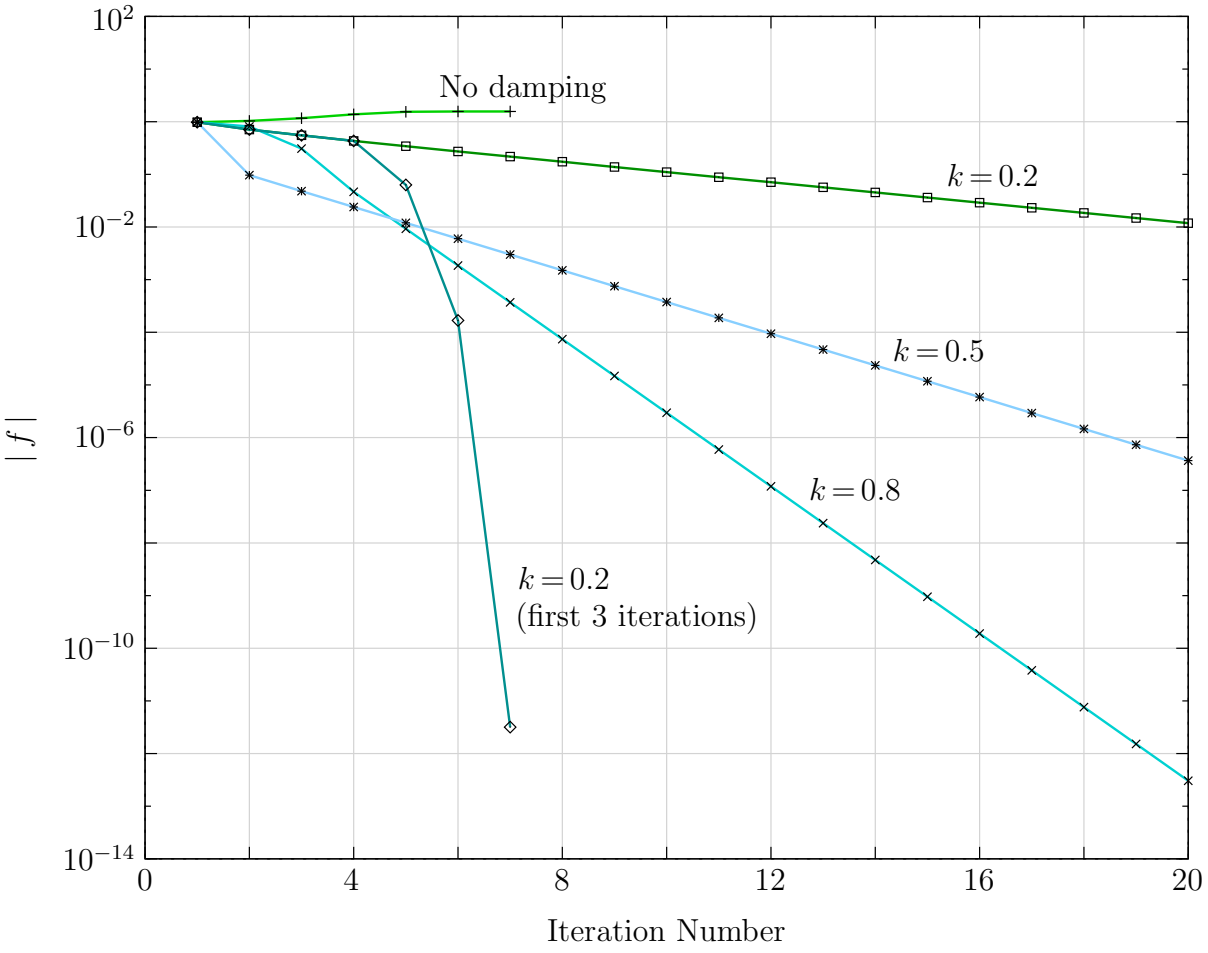


Figure 3.11: $|f|$ versus NR iteration number for different values of k for $f(x) = \tan^{-1}(x)$ with $x^{(0)} = 1.5$.

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.

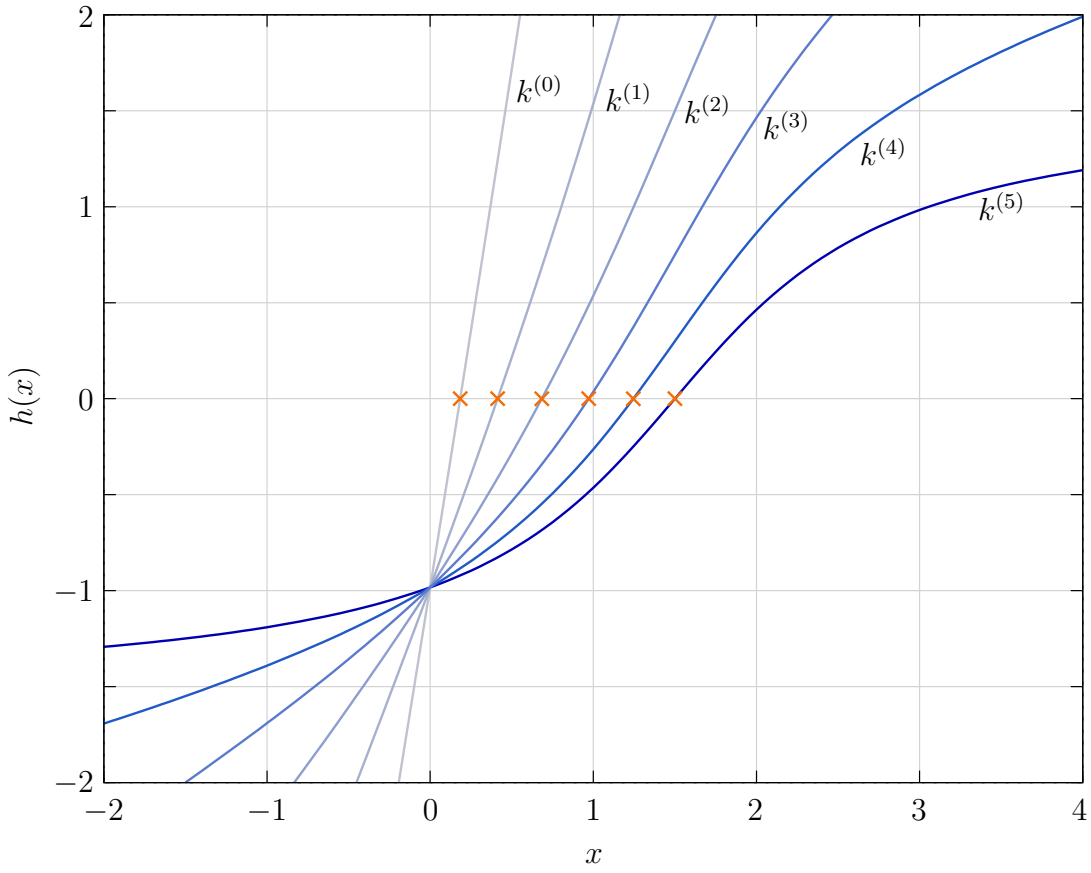


Figure 3.12: NR process for solving $f(x)\equiv\ \tan^{-1}(x-a)=0$ with $a=1.5$, and $x^{(0)}=0$ as the initial guess. An auxiliary function $g^{(i)}(x)=k^{(i)}\,x$ is used, and $h^{(i)}(x)=f(x)+g^{(i)}(x)$ is solved with the NR method. $x=0$ is used as the initial guess for solving $h^{(0)}=0$. Thereafter, 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[5], 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.

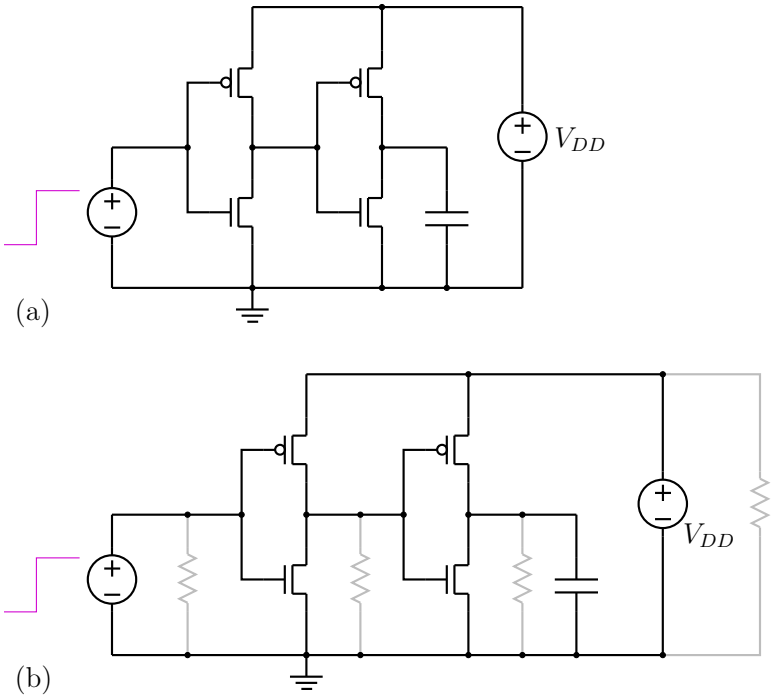


Figure 3.13: Illustration of $g_{\min}$ stepping: (a) original circuit, (b) circuit with a resistor added from each node to ground.

(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[6]. 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."

[5] or between each pair of nodes of the nonlinear devices

[6] There could be signal sources in the circuit (e.g., a BJT amplifier) which should also be made zero.

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.

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 $V = 2\,\text{V}$ and $V_T = 26\,\text{mV}$, 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.

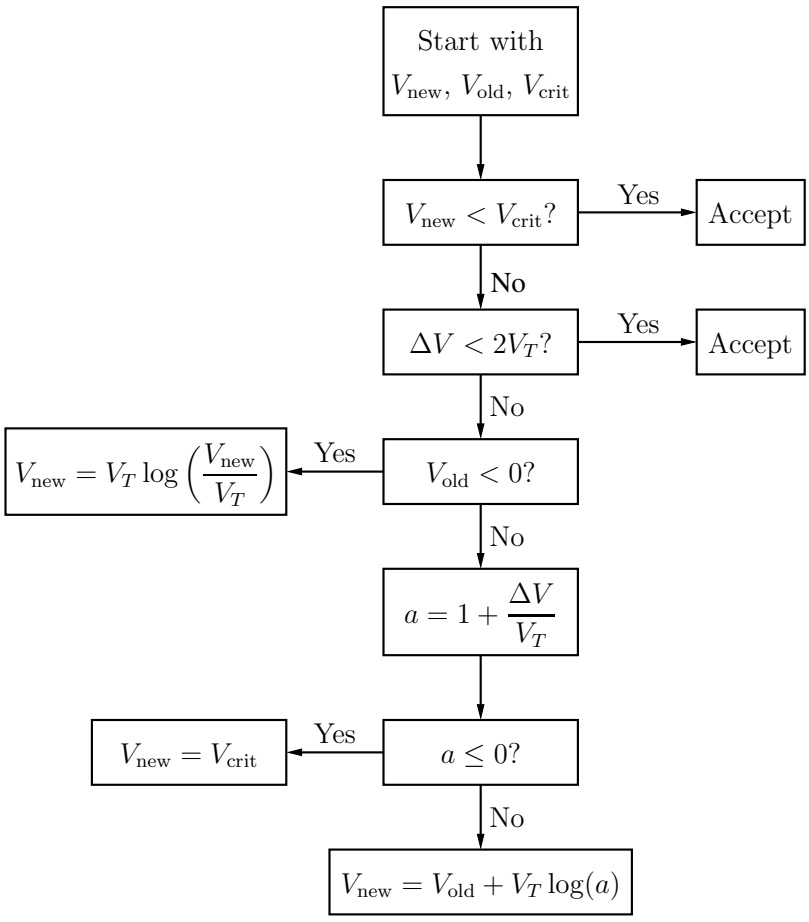


Figure 3.14: Flow chart for limiting junction voltages in SPICE. V_T is the thermal voltage, and $\Delta V = V_D^{(i+1)} - V_D(i)$.

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 6.

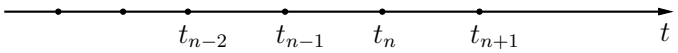


Figure 3.15: Discretisation of the time axis.

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.

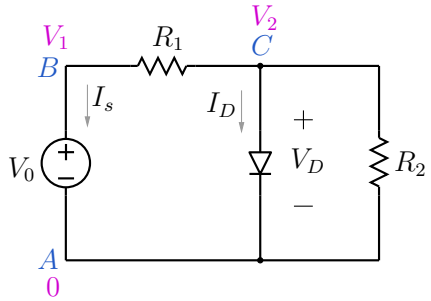


Figure 3.16: A nonlinear circuit example.

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

Numerical Solution of ODEs: Explicit Methods

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.

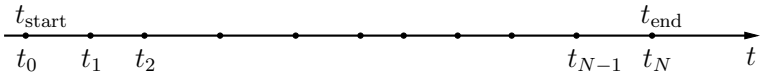


Figure 4.1: Discretisation of the time interval from t_{start} to t_{end}.

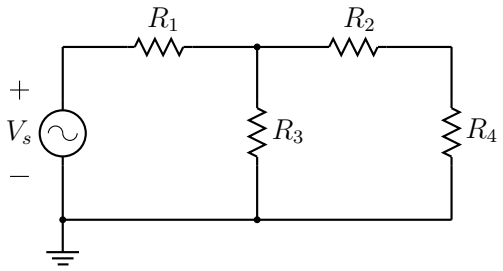


Figure 4.2: A circuit with a time-dependent voltage source.

The situation is different when time derivatives are involved, e.g., in the form of capacitors or inductors. Let us see how time derivatives can be handled.

4.1 Forward Euler method

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[1], i.e., the *discrete* values x_1, x_2, $\cdots$, corresponding to t_1, t_2, etc. 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 $\Delta t_n = t_{n+1} - t_n$.

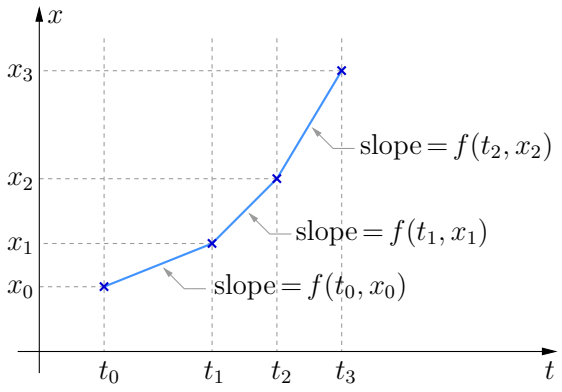


Figure 4.3: Illustration of the Forward Euler method.

Let us apply the FE method to the ODE,

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

which has the analytical (exact) solution,

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

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;
```

[1]We will denote the exact solution by $x(t)$; e.g., $x(t_1)$ means the exact solution at t_1.

```
  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[2] along with the analytical (exact) solution given by Eq. 4.4. We notice some difference between the two, but it can be made smaller[3] 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 [8]),

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

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.

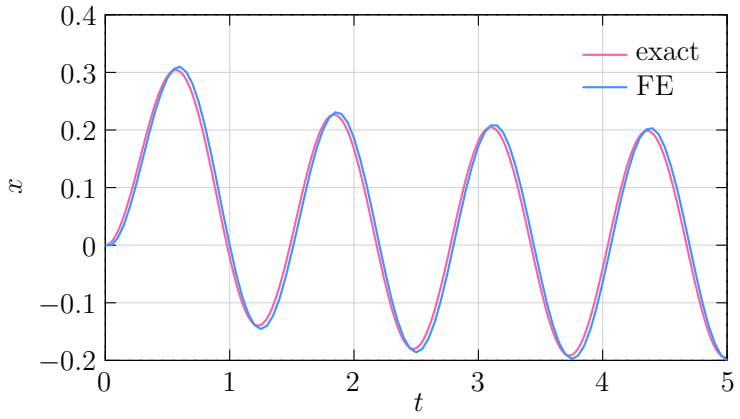


Figure 4.4: Analytical solution (red curve) and numerical solution obtained with the FE method (blue curve) for the ODE given by Eq. 4.3.

Let us look at the LTE of the FE method for the test equation[4],

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

[2]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."

[3]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.

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

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.

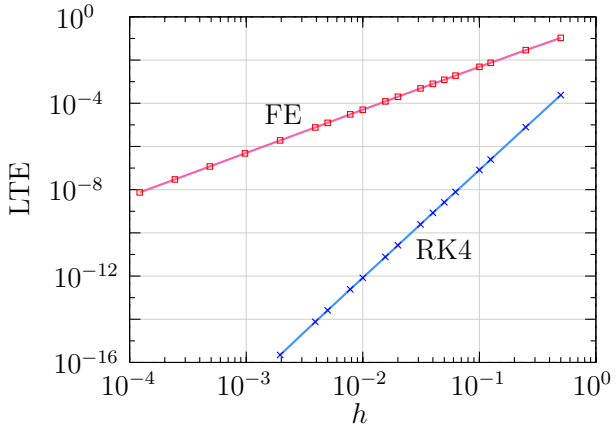


Figure 4.5: LTE versus step size h for the FE and RK4 methods in solving the ODE given by Eq. 4.6.

4.2 Runge-Kutta method of order 4

The Runge-Kutta method of order 4 is given by[5]

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

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

[5]There are other Runge-Kutta methods of order 4 (see [9]); the method described here is called the "classic" form and is commonly used.

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.

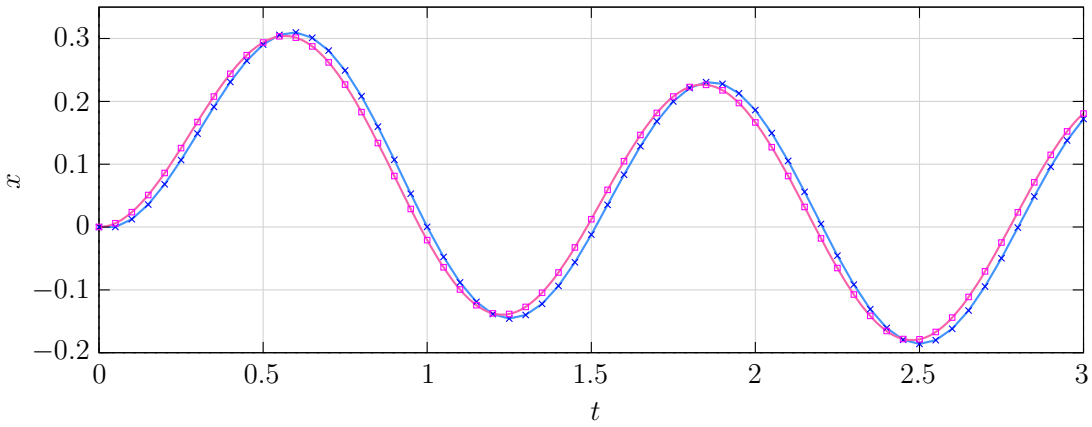


Figure 4.6: Numerical solution of Eq. 4.3 obtained with the FE and RK4 methods (crosses and squares, respectively) with a step size of $h = 0.05$. The exact solution is also shown (red curve).

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.

4.3 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

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

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! There is no need to rush to the weavers for every single problem. The numerical solution is shown in Fig. 4.7.

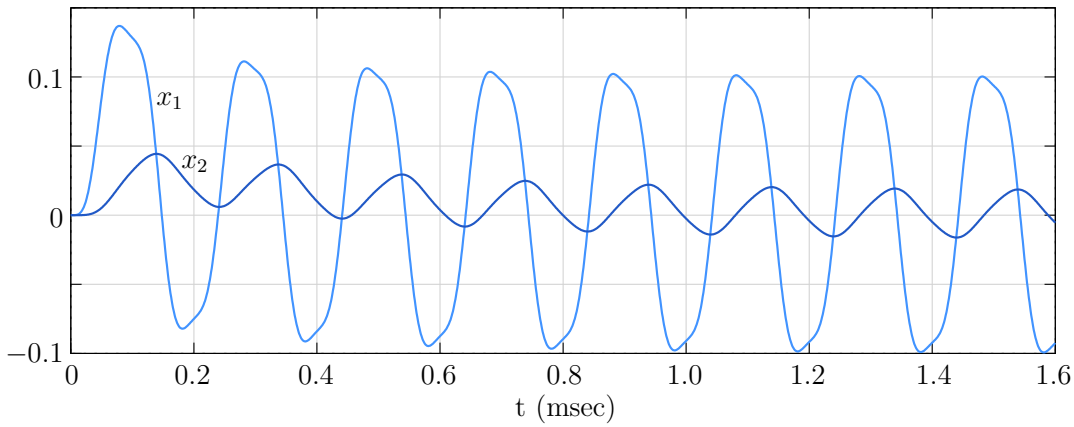


Figure 4.7: Numerical solution of Eq. 4.13 obtained with the FE method with a step size of $h = 1\,\mu$sec.

4.4 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[6] x_n to be equal to the exact solution $x(t_n)$, we have

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

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

Subtracting Eq. 4.16 from Eq. 4.15, we get

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

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

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

Having obtained an estimate for the LTE (denoted by LTE^{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 the LTE^{est} is larger than the specified tolerance, we reject the current step and try a smaller step. If LTE^{est}

[6] In reality, x_n and $x(t_n)$ would not be the same, but Eq. 4.17 remains valid (see [8]).

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 LTE^{est} using Eq. 4.18. Since LTE/h is $O(h^p)$, we can write

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

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

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

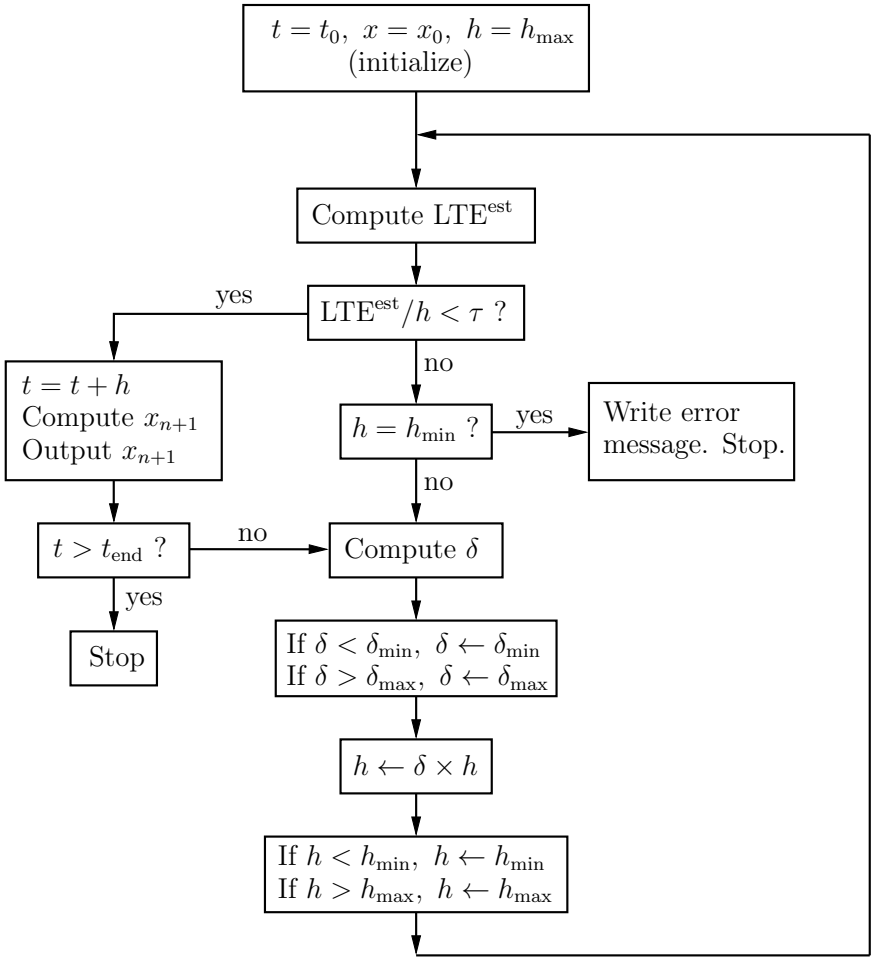


Figure 4.8: Flow chart for adaptive time step selection based on computation of local truncation error.

From Eqs. 4.19 and 4.20, we obtain δ as[7]

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

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 [8]), the value of δ is generally restricted to $\delta_{\min} \leq \delta \leq \delta_{\max}$, 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 [10]

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

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,

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

[7] Typically, δ is made a little smaller than that given by Eq. 4.21, see [10]. Also, note that controlling the LTE (rather than LTE/h) is also possible, and in that case Eq. 4.21 gets replaced by $\delta = \left(\frac{\tau}{|\tilde{x}_{n+1} - x_{n+1}|} \right)^{1/(p+1)}$.

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[8] (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.

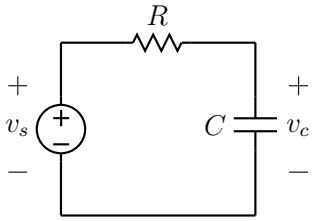


Figure 4.9: RC circuit example.

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, notwithstanding the aura of mystery created around them by some weavers. An induction motor control example and the results obtained with the RKF45 method are shown in Fig. 4.15.

[8] 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.

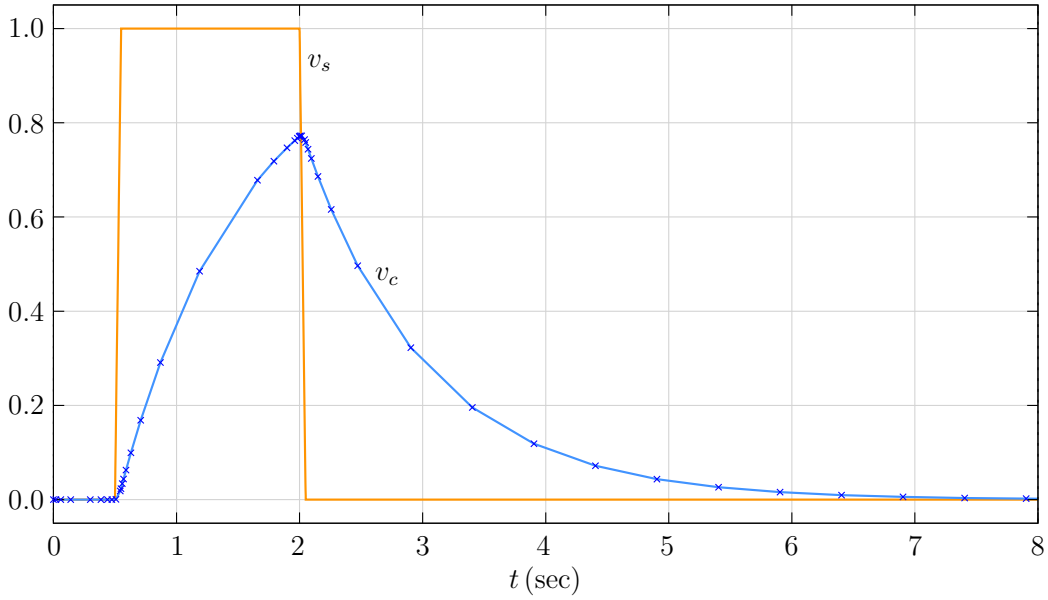


Figure 4.10: Numerical solution obtained for the RC circuit of Fig. 4.9 using the RKF45 method with $\tau = 10^{-4}$. The other parameters are $R = 1\,\Omega$, $C = 1\,\mathrm{F}$, $\delta_{\min} = 0.2$, $\delta_{\max} = 2$, $h_{\min} = 10^{-5}$, $h_{\max} = 0.5$.

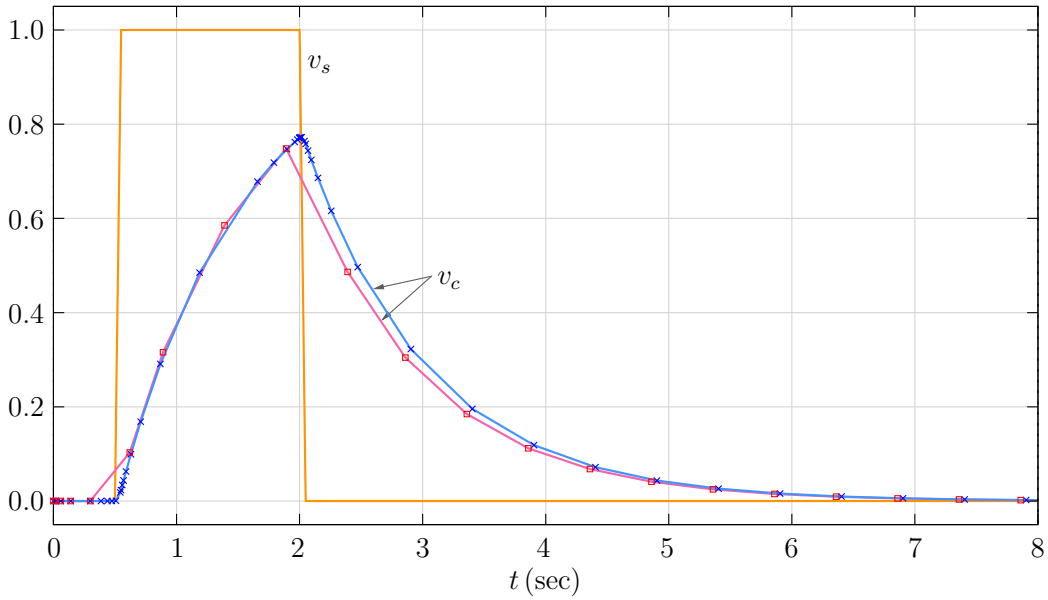


Figure 4.11: Numerical solutions obtained for the RC circuit of Fig. 4.9 using the RKF45 method with two values of τ: 10^{-2} (squares) and 10^{-4} (crosses). The other parameters are the same as in Fig. 4.10.

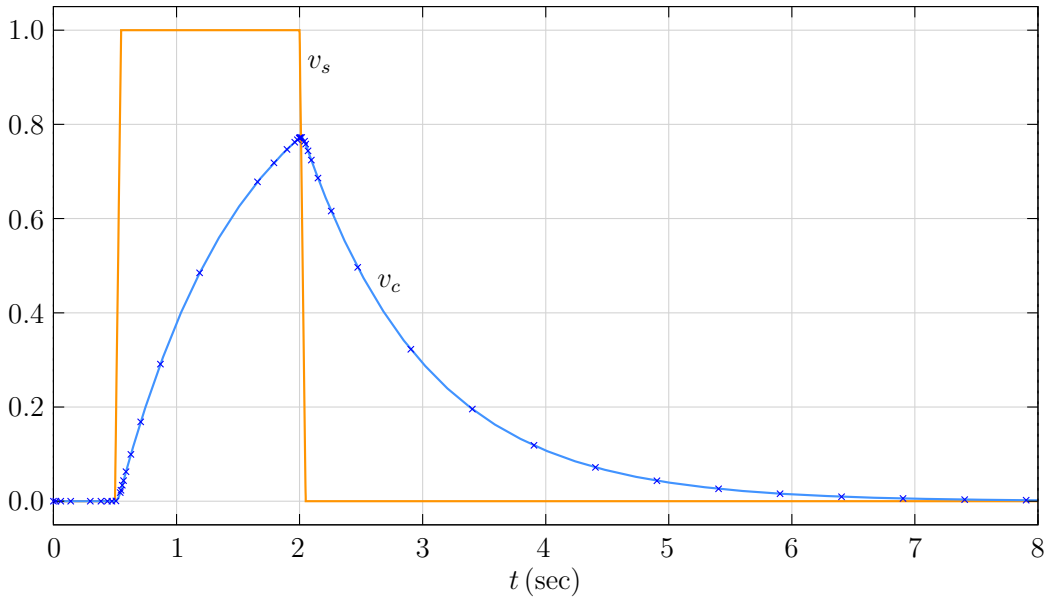


Figure 4.12: Numerical solutions obtained for the *RC* circuit of Fig. 4.9 using the RKF45 method with two values of τ: 10^{-4} (crosses) and 10^{-6} (blue graph). The other parameters are the same as in Fig. 4.10.

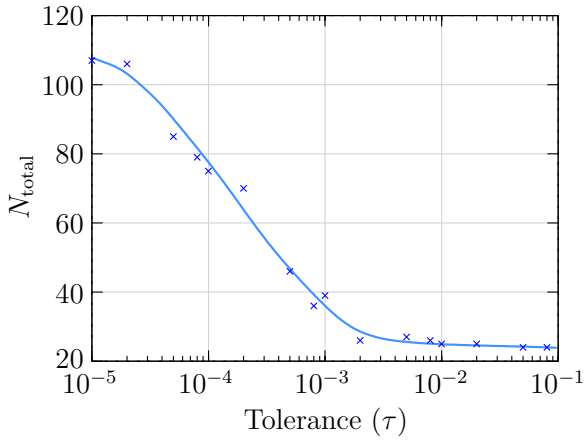


Figure 4.13: Total number of time points N_{total} used by the RKF45 method in computing the numerical solution for the *RC* circuit of Fig. 4.9 as a function of tolerance τ.

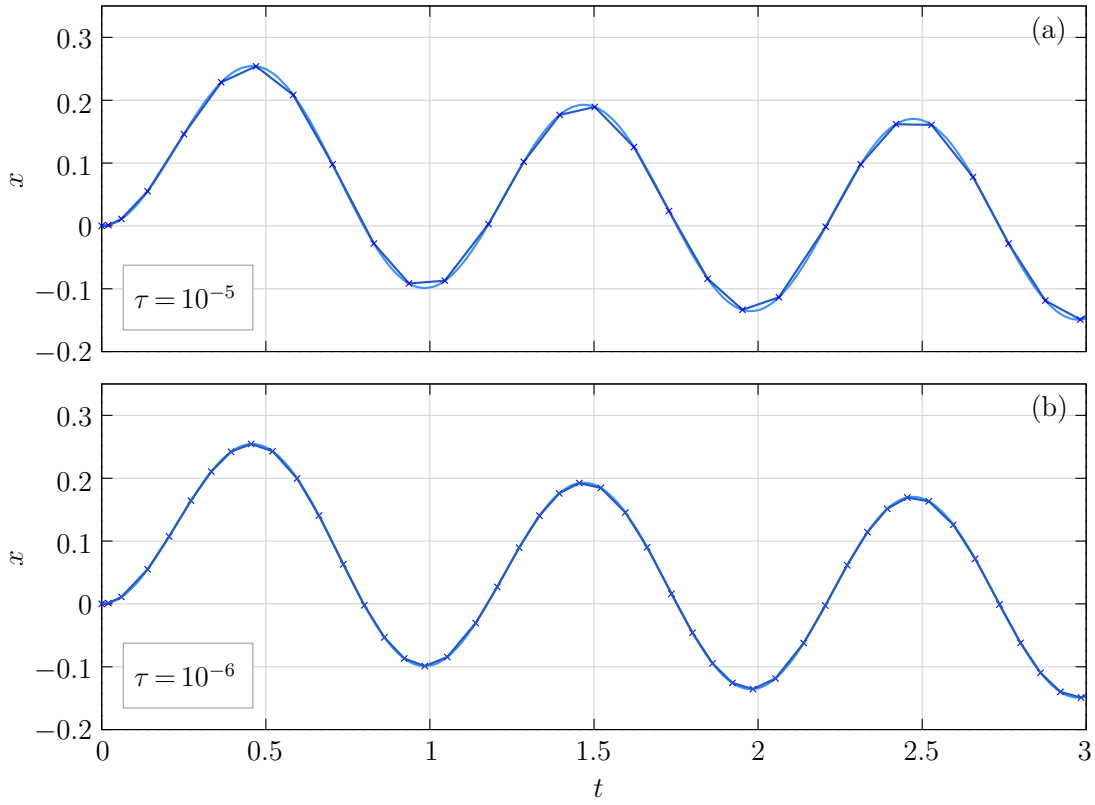


Figure 4.14: Numerical solutions obtained for the *RC* circuit of Fig. 4.9 with $v_s = \sin \omega t$ where $\omega = 2\pi$. Light blue curve: analytical solution, Crosses: numerical solution obtained with the RKF45 method. (a) $\tau = 10^{-5}$, (b) $\tau = 10^{-6}$. The other parameters are the same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

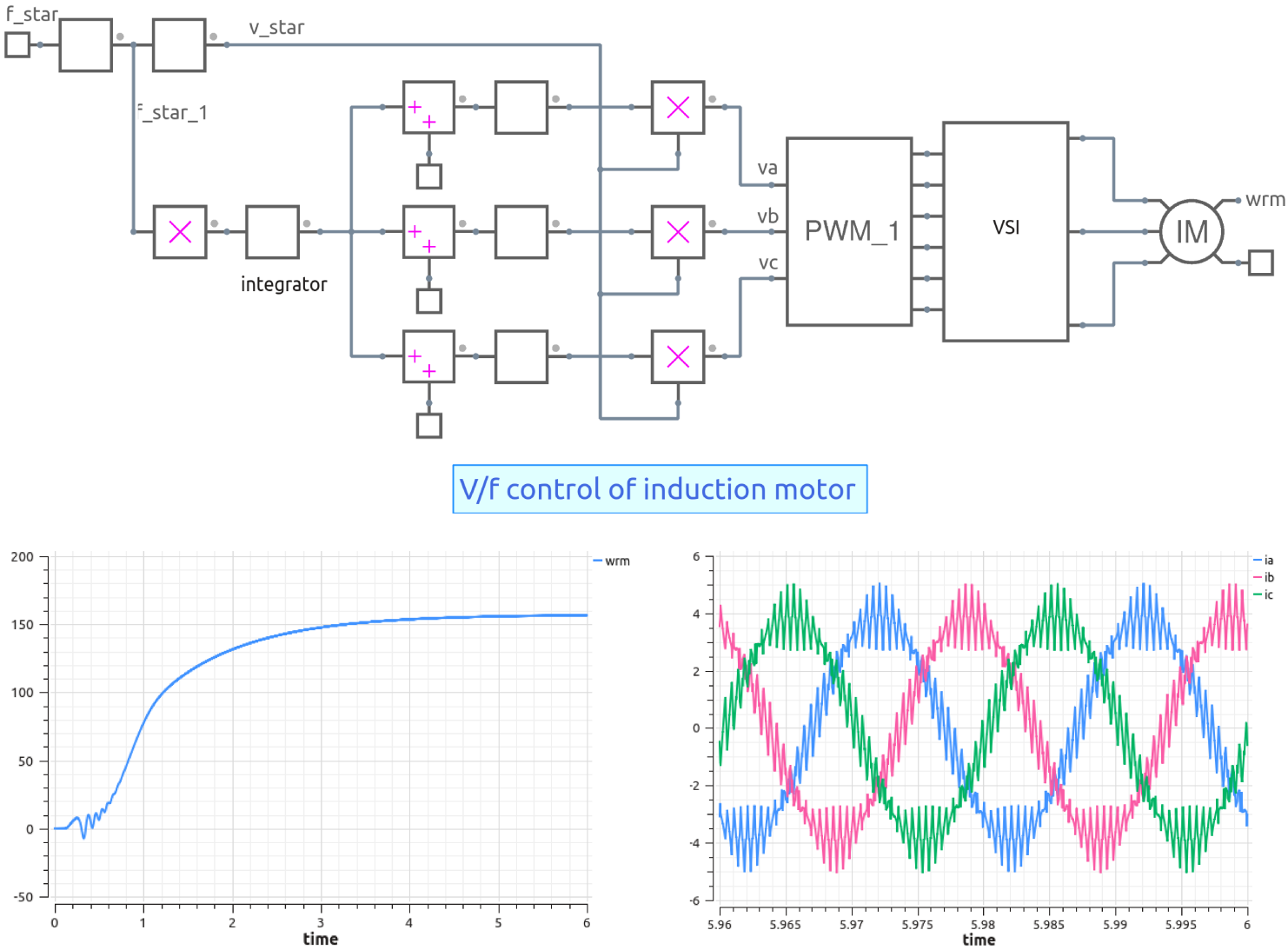


Figure 4.15: Simulation of V/f control of an induction motor using the RKF45 method. Circuit file: `x_vbyf1.sqproj`

4.5 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[9]. 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. 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 [6] 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. 4.16 (a) which can be described by the ODEs

$$\begin{aligned} \frac{dV_1}{dt} &= \frac{1}{R_1C_1}(V_s - V_1) - \frac{1}{R_2C_1}(V_1 - V_2), \\ \frac{dV_2}{dt} &= \frac{1}{R_2C_2}(V_1 - V_2). \end{aligned} \tag{4.25}$$

With a sinusoidal input, $V_s = V_m \sin \omega t$, we can use phasors (see Fig. 4.16 (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 $\mathbf{Z}_1 = -j11.8\,\text{k}\Omega$, $\mathbf{Z}_2 = -j6.4\,\Omega$. 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

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

The numerical solution obtained with the RK4 method with a time step of $h = 1\,\text{msec}$ is shown in Fig. 4.17 (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. 4.17, 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

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

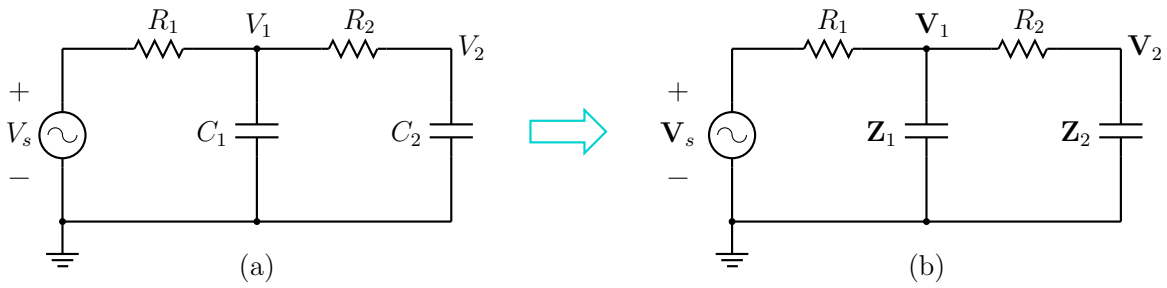


Figure 4.16: *RC* circuit with two time constants: (a) time-domain circuit, (b) frequency-domain circuit.

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. 4.18 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, 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. 4.4 – 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 it 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. 4.16 but with different component values, viz., $R_1 = 1\,\text{k}\Omega$, $R_2 = 1\,\text{k}\Omega$, $C_1 = 1\,\mu\text{F}$, $f = 1$ 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 with

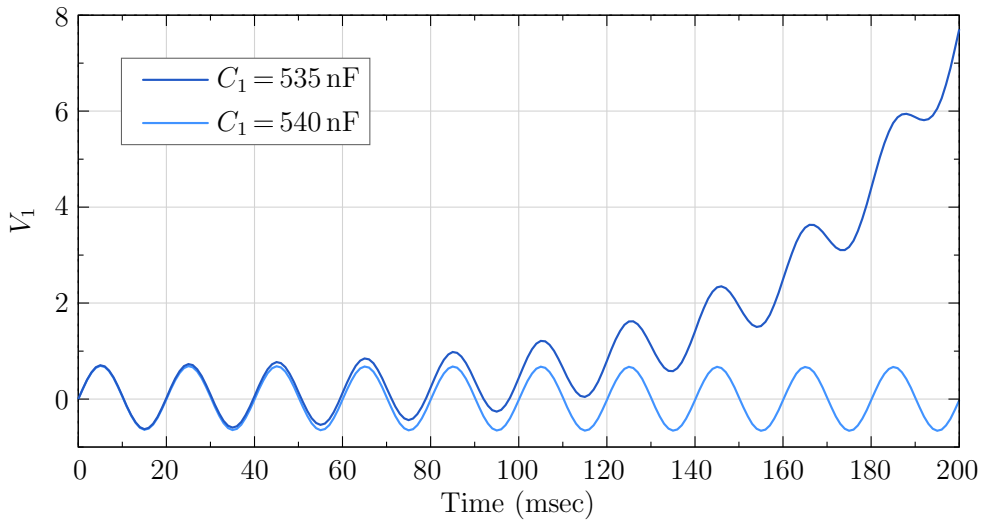


Figure 4.17: Numerical solution obtained with the RK4 method for the circuit of Fig. 4.16 for two values of C_1. The other parameters are $R_1 = 1\,\text{k}\Omega$, $R_2 = 2\,\text{k}\Omega$, $C_2 = 1\,\text{mF}$, $V_s = V_m \sin \omega t$, with $V_m = 1\,\text{V}$, $f = 50\,\text{Hz}$. The time step is $h = 1\,\text{msec}$.

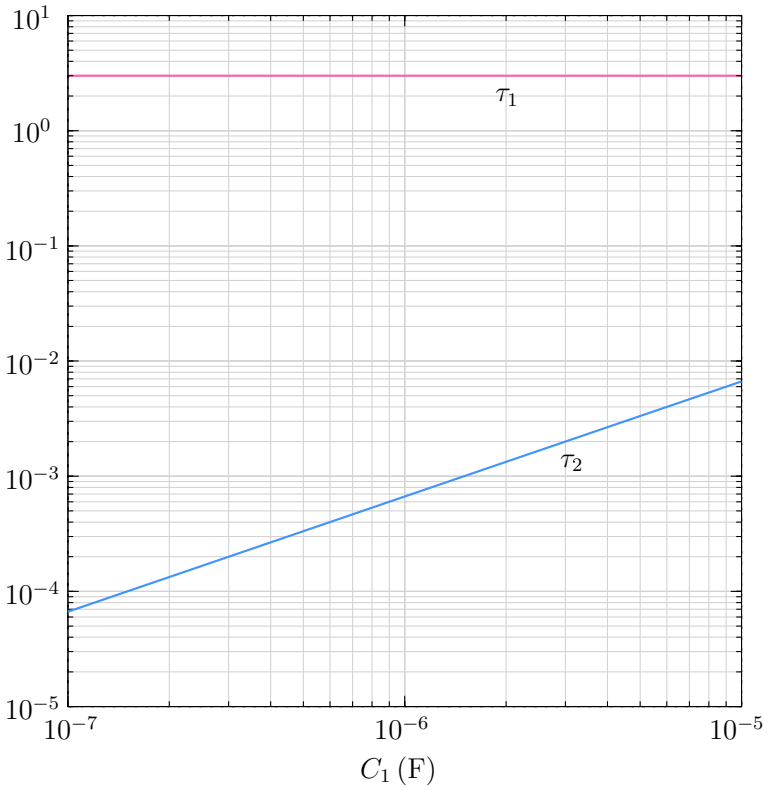


Figure 4.18: Time constants for the circuit of Fig. 4.16 (in seconds) as a function of C_1 with $R_1 = 1\,\text{k}\Omega$, $R_2 = 2\,\text{k}\Omega$, $C_2 = 1\,\text{mF}$.

time to meet the specified tolerance requirement, and a stable solution (not shown) is obtained. As C_2 is made smaller, the time constant (in particular, 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 $C_2 = 10^{-10}\,\mathrm{F}$. The impedance $\mathbf{Z}_2$ is then (with $f = 1\,\mathrm{kHz}$) $-j64\,\mathrm{M\Omega}$ 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 circuit 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.

Fig. 4.21 shows a Simulink implementation of the same equations (Eq. 4.25). The "model" (in Simulink lingo[10]) was simulated with the `ode45` algorithm. The time step variation for different values of C_2 is shown in Fig. 4.22, and its impact on the number of time points and the CPU time is shown in Table 4.2. Surprise, surprise, the rich cousin `ode45` – proprietary and what not – gets the same treatment as the poor cousin – our humble, free, open, publicly available RKF45! Sometimes, the world does seem to be flat.

C_2 (F)	T_{CPU}	N_{total}
10^{-6}	0.35	52
10^{-7}	0.36	92
10^{-8}	0.36	305
10^{-9}	0.39	3017
10^{-10}	0.65	30133
10^{-11}	3.03	301307
10^{-12}	26.23	3013054

Table 4.2: CPU time in seconds (T_{CPU}) and total number of time points (N_{total}) required to solve the model shown in Fig. 4.21 with Simulink using the `ode45` method for various values of C_2. Note that the CPU time should be treated as representative here; it would depend on the computer configuration being used.

[10] All over the world, the term "model" is commonly used to mean a representation of a component such as an electronic device, a machine, a heat sink, etc. Putting together several components makes up a circuit or a system. It is not clear why Simulink vendors have chosen to use "model" to mean a circuit or a system. Just to be different or is there some esoteric justification? We will never know since weavers do not share their secrets!

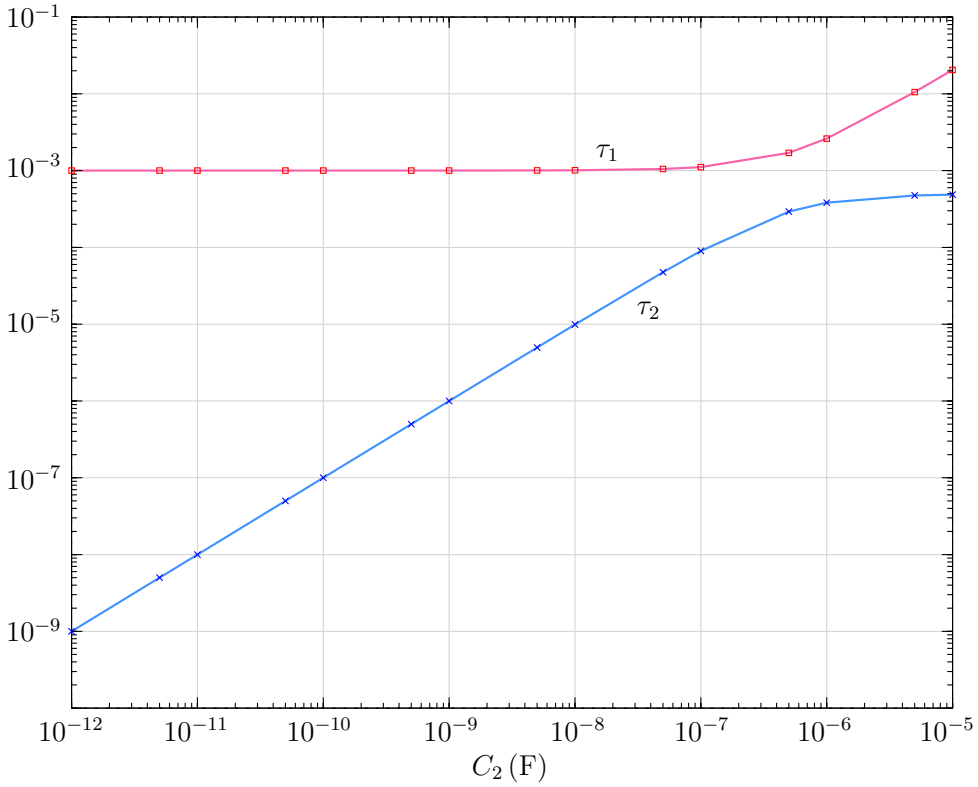


Figure 4.19: Time constants for the circuit of Fig. 4.16 (in seconds) as a function of C_2 with $R_1 = 1\,\mathrm{k}\Omega$, $R_2 = 1\,\mathrm{k}\Omega$, $C_1 = 1\,\mu\mathrm{F}$.

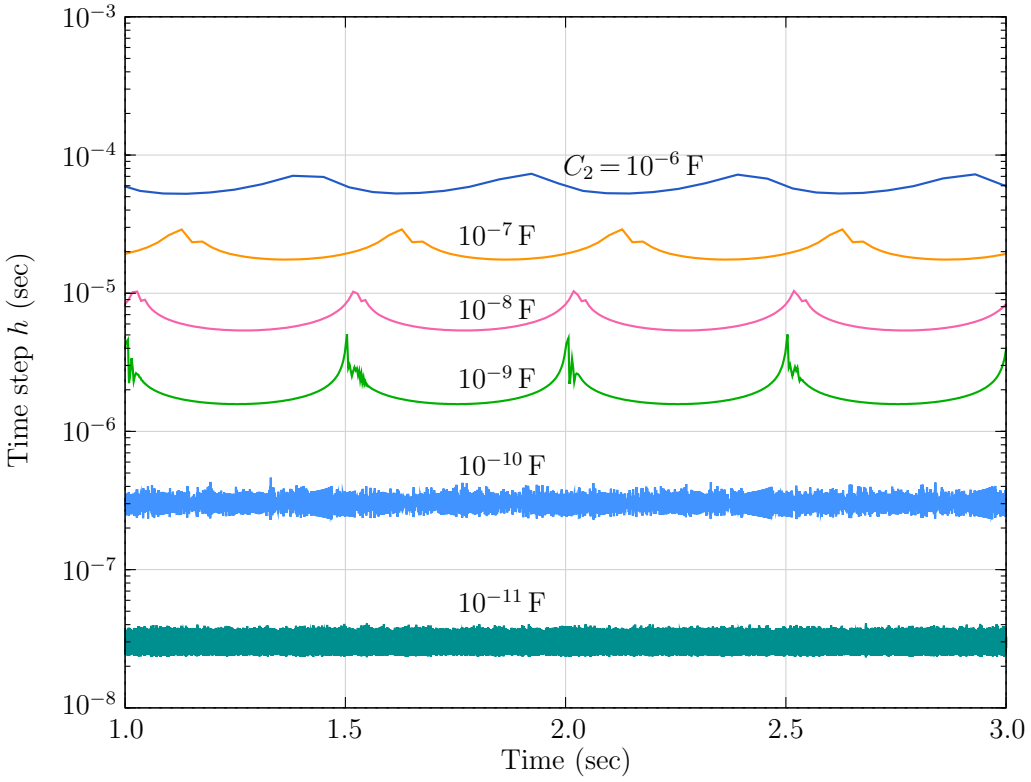


Figure 4.20: Time step (h) used by the RKF45 method versus elapsed time (t) in solving the set of ODEs for the circuit of Fig. 4.16 (Eq. 4.25) for different values of C_2. The other circuit parameters are $R_1 = 1\,\text{k}\Omega$, $R_2 = 1\,\text{k}\Omega$, $C_1 = 1\,\mu\text{F}$, $f = 1\,\text{kHz}$. The following algorithmic parameters were used: $\tau = 10^{-3}$, $\delta_{\min} = 0.2$, $\delta_{\max} = 2$, $h_{\min} = 10^{-5}$, $h_{\max} = 0.5$.

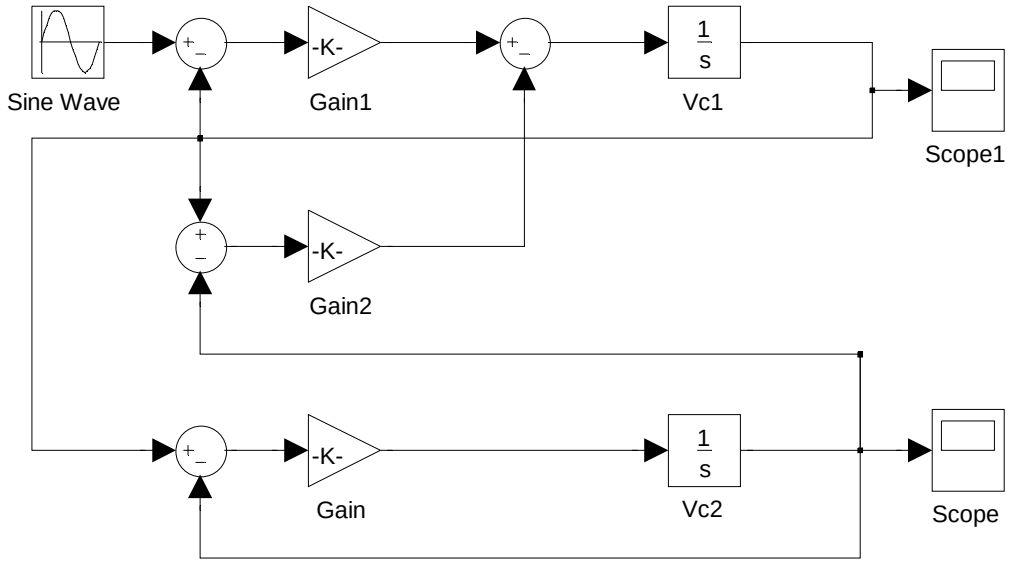


Figure 4.21: Simulink implementation of the circuit of Fig. 4.16. (`Gain1` $= 1/R_1C_1$, `Gain2` $= 1/R_2C_1$, `Gain` $= 1/R_2C_2$.)

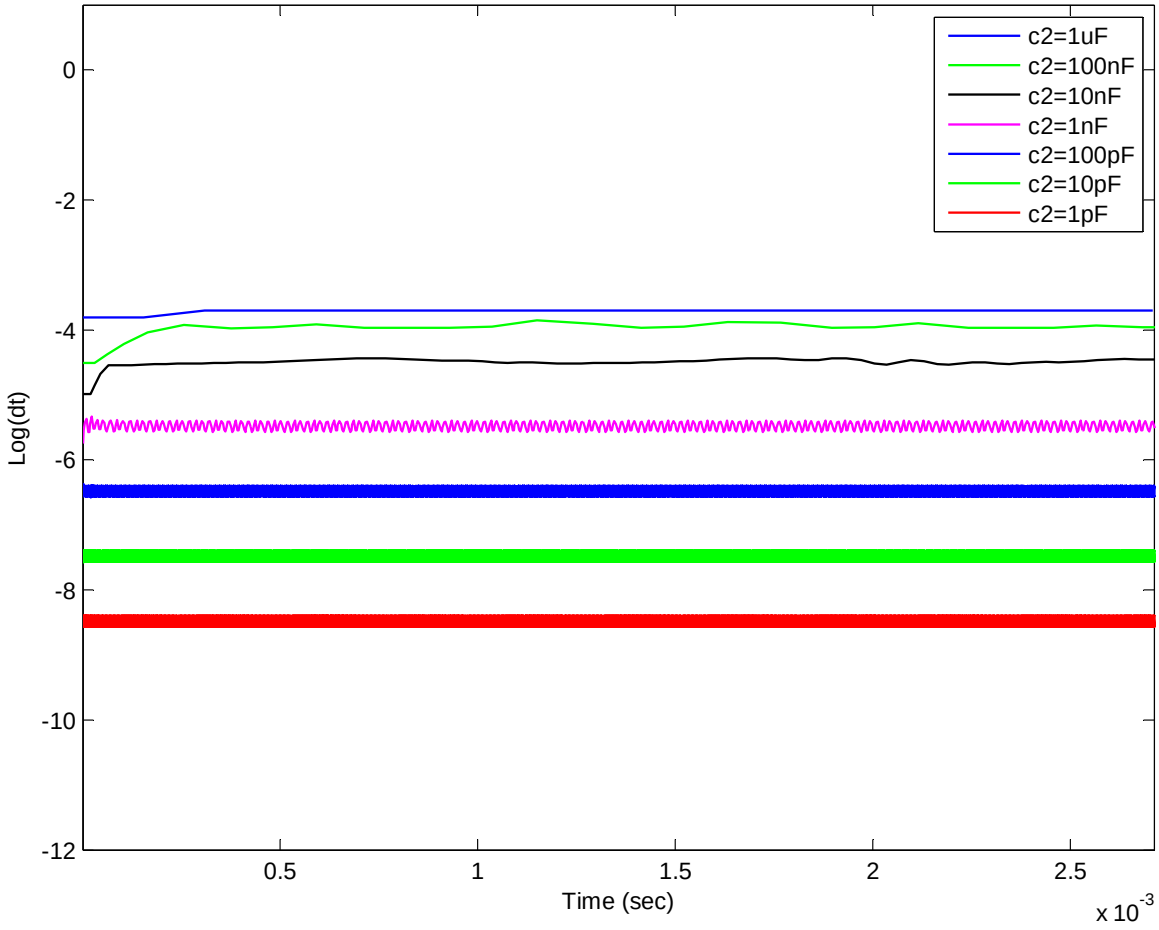


Figure 4.22: Time step Δt used by Simulink versus time with the `ode45` algorithm for the problem shown in Fig. 4.21 for various values of C_2. The other component values are $R_1 = 1\,\mathrm{k}\Omega$, $R_2 = 1\,\mathrm{k}\Omega$, $C_1 = 1\,\mu\mathrm{F}$ (same as those mentioned in Fig. 4.20).

4.6 What are those arrows in Simulink?

Consider the ODE,

$$\frac{dy}{dt} = k_1 x_1(t) + k_2 x_2(t), \tag{4.27}$$

where $x_1 = A_1 \sin \omega_1 t$ and $x_2 = A_2 \sin \omega_2 t$ are known functions of time. The Simulink implementation of Eq. 4.27 is shown in Fig. 4.23.

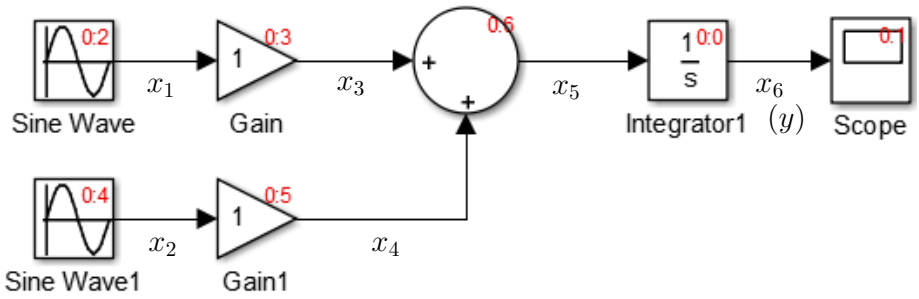


Figure 4.23: Implementation of Eq. 4.27 in Matlab/Simulink.

Simulink also gives the order in which the various blocks are executed[11], as marked in red in the figure. What does this order mean? Let us try to understand this point by writing the Forward Euler equation corresponding to Eq. 4.27:

$$y^{n+1} = y^n + \Delta t \ \left(k_1 x_1^n + k_2 x_2^n\right) \ , \tag{4.28}$$

where $\Delta t = t_{n+1} - t_n$ is the time step. A program like Simulink would internally execute the following steps[12] in solving the ODE with the FE method:

1. Start with $n = 0$ (i.e., $t = t_0$, the starting time for the simulation).
   Compute x_1^0 and x_2^0 (i.e., x_1 at $t = t_0$ and x_2 at $t = t_0$).
   Use $y^n = y^{(0)}$, the initial value for y.
2. Compute $y^{(1)}$ using Eq. 4.28.
3. Make $n \leftarrow n + 1$ and repeat until $t = t_{\text{final}}$, the ending time for the simulation.

It is now easy to understand why we require a certain order of execution (denoted by the red labels in Fig. 4.23). The variables for t_{n+1} are updated as follows.

1. $y^{n+1} = y^n + \Delta t \times x_5^n$ (element marked `0:0`)
2. $x_1^{n+1} = A_1 \sin \omega_1 t_{n+1}$ (element marked `0:2`)
3. $x_3^{n+1} = k_1 \, x_1^{n+1}$ (element marked `0:3`)

[11]Note that the block "1/s" is in integrator. If x and y are the input and output, respectively, of the integrator, we have $y = \int x \, dt$ which is the same as $\frac{dy}{dt} = x$.

[12]Because of the extreme secrecy observed by all weavers, we don't really know what exactly goes on internally in the program; what we have described here should therefore be treated as a likely scenario.

4. $x_2^{n+1} = A_2 \sin\omega_2 t_{n+1}$ (element marked `0:4`)
5. $x_4^{n+1} = k_2\, x_2^{n+1}$ (element marked `0:5`)
6. $x_5^{n+1} = x_3^{n+1} + x_4^{n+1}$ (element marked `0:6`)

When we complete this sequence of computations, we have performed *one* step of the simulation. We then proceed to the next step, and so on until $t_{\texttt{final}}$ is reached. Clearly, the order in which the above steps are executed is important[13]. For example, we could not have carried out step 3 before step 2 since step 3 depends on the *updated* value of x_1.

How can a program like Simulink figure out the correct order for processing the elements? The answer lies in the arrows in the block diagram. With arrows, the input and output nodes of the element can be distinguished. With that information, the program can order the elements such that the input values are updated before the output values for each element. It is for this reason that a Simulink block diagram has arrows connecting the different blocks.

Next, we consider a system of two ODEs given by

$$\frac{dx_8}{dt} = x_5 + x_2\, x_4\,, \tag{4.29}$$
$$\frac{dx_5}{dt} = x_1 + x_2\,, \tag{4.30}$$

where x_1, x_2, x_4 are known functions of time, viz., $x_1 = A_1 \sin\omega_1 t$, $x_2 = A_2 \sin\omega_2 t$, and $x_4 = k_0 = \text{constant}$. The Simulink implementation of the above equations is shown in Fig. 4.24. In this case, the Forward Euler integration formulas are given by

$$x_8^{n+1} = x_8^n + \Delta t \times (x_5^n + x_2^n\, x_4^n)\,, \tag{4.31}$$
$$x_5^{n+1} = x_5^n + \Delta t \times (x_1^n + x_2^n)\,. \tag{4.32}$$

The ordering shown in Fig. 4.24 (in red) corresponds to the following operations:

1. $x_8^{n+1} = x_8^n + \Delta t \times x_7^n$ (element marked `0:0`)
2. $x_4^{n+1} = k_0$ (element marked `0:2`)
3. $x_5^{n+1} = x_5^n + \Delta t \times x_3^n$ (element marked `0:3`)
4. $x_2^{n+1} = A_2 \sin\omega_2 t_{n+1}$ (element marked `0:4`)
5. $x_6^{n+1} = x_2^{n+1}\, x_4^{n+1}$ (element marked `0:5`)
6. $x_1^{n+1} = A_1 \sin\omega_1 t_{n+1}$ (element marked `0:6`)
7. $x_3^{n+1} = x_1^{n+1} + x_2^{n+1}$ (element marked `0:7`)
8. $x_7^{n+1} = x_5^{n+1} + x_6^{n+1}$ (element marked `0:8`)

[13] As we shall see later, the question of ordering the elements does not arise when an implicit method is used to solve the ODEs.

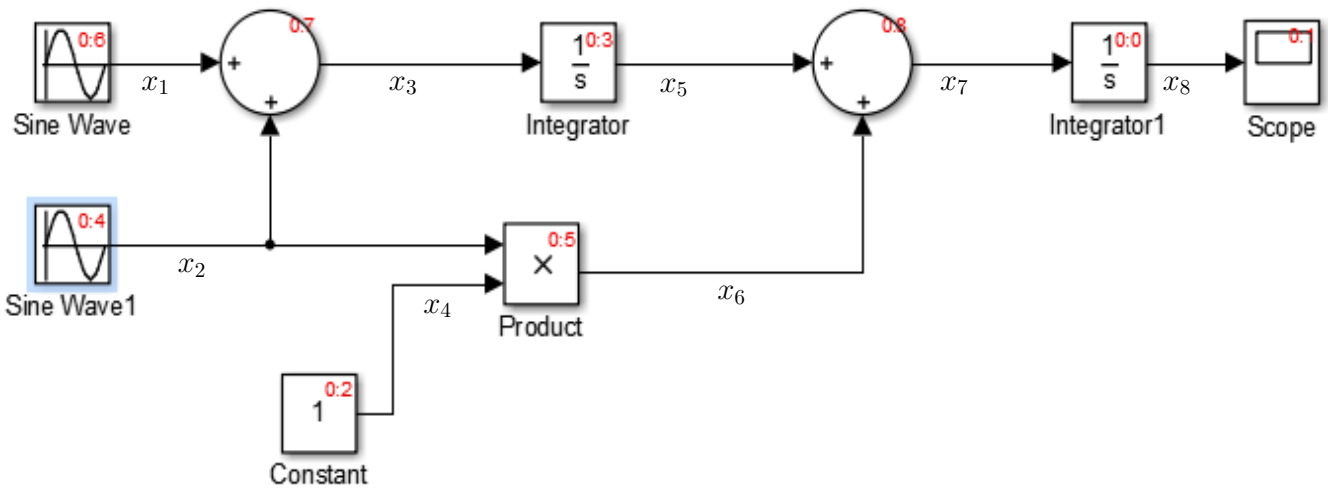


Figure 4.24: Implementation of Eqs. 4.29 and 4.30 in Matlab/Simulink.

It is easy to see that execution of the above sequence of steps is equivalent to updating x_8 and x_5 using the FE formulas[14] given by Eqs. 4.31 and 4.32. An important point to note is the *simplicity* of the process. Each of the above steps is straightforward – almost trivial – to execute. In fact, even a nonlinear operation (such as $x_6 = x_2 \times x_4$ in the block marked `0:5`) requires no special consideration; it is just another *evaluation*. In this respect, more complicated explicit methods such as the fourth-order Runge-Kutta method, the `ode45` method in Simulink, etc. are no different. They all involve *evaluation* of a left-hand side using *known* numbers on the right-hand side.

Algebraic Loops

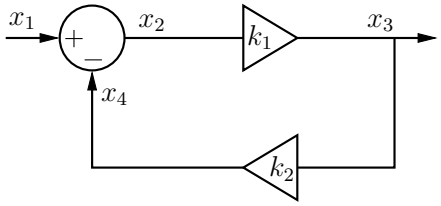


Figure 4.25: Block diagram showing an algebraic loop.

Consider the feedback loop shown in Fig. 4.25. We can write the following equation for this system:

$$x_3 = k_1\, x_2 = k_1\, (x_1 - x_4) = k_1\, (x_1 - k_2\, x_3)\,, \tag{4.33}$$

giving

$$x_3 = \frac{k_1}{1 + k_1 k_2}\, x_1\,. \tag{4.34}$$

[14]Note that the ordering is not unique; we can find other ways of ordering the blocks as well. All we want is *one* ordering that will work.

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 blocks, as we did earlier. We get the following equations.

$$x_2^{n+1} = x_1^{n+1} + x_4^{n+1}\,, \tag{4.35}$$
$$x_4^{n+1} = k_2\, x_3^{n+1}\,, \tag{4.36}$$
$$x_3^{n+1} = k_1\, x_2^{n+1}\,. \tag{4.37}$$

As before, we wish to evaluate these formulas *one at a time*, but this leads to a conflict. Eqs. 4.35 to 4.37 are supposed to valid *simultaneously*. For example, the value of x_3^{n+1} in Eqs. 4.36 and 4.37 is supposed to be the same. However, since Eq. 4.37 is evaluated after Eq. 4.36, 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[15], not involving time derivatives[16].

In Simulink, algebraic loops are allowed to some extent, and they are treated differently than the rest of the equations by iterating through the equations related to the algebraic loop(s). In the above example, if we evaluate the equations in the order $1 \to 2 \to 3 \to 1 \to 2 \to 3 \to \cdots$, the values of x_1^{n+1}, x_2^{n+1}, x_3^{n+1} may finally converge to the correct solution, i.e., values which simultaneously satisfy the three equations (Eqs. 4.35 to 4.37).

Will such an iterative procedure always converge? That will depend on the algebraic equation involved and of course on the method used for iterating through the equations. In several practical applications (e.g., electronic circuits), there are many algebraic loops in the system, and some of the equations could be highly non-linear. In such cases, Simulink may slow down considerably (since the algebraic loops require special treatment) or worse, it may fail to find a solution.

To circumvent the problems associated with algebraic loops, Simulink users are advised by the vendor to do the following.

(a) Avoid algebraic loops as far as possible. For example, instead of the system shown in Fig. 4.25, we could use a single block, $x_3 = k\, x_1$ with $k = \dfrac{k_1}{1 + k_1 k_2}$ pre-computed.

(b) Insert a delay element to "break" an algebraic loop. For example, we can replace the block diagram of Fig. 4.25 with that of Fig. 4.26 in which a "delay" element is added in the loop. How does this help?

The system equations can now be written as

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

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

[16]If there is an algebraic loop in the system, the elements (blocks) in the system cannot be ordered since there is a cyclic dependence. A simulator like Simulink would first try to order the elements. If the ordering algorithm fails, the simulator knows that there is an algebraic loop somewhere.

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.

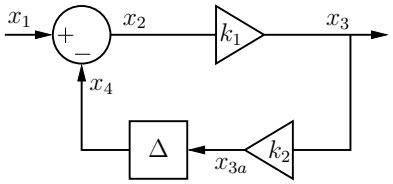


Figure 4.26: Block diagram showing breaking of an algebraic loop with a delay block.

The vendor acknowledges that the above two solutions cannot be pushed too far. There are cases when these *ad hoc*, quick-and-dirty fixes will not help. The vendor suggests a way out – buy another expensive package (from the *same* vendor) which will handle algebraic loops with the respect they deserve, i.e., by employing a robust iterative procedure such as the Newton-Raphson method for solving non-linear equations. Before taking this offer, the reader would do well to look around and check if there are alternative products which will do the same job for a lower price or in a better manner or both.

Acknowledgments

The author would like to acknowledge the contribution of Aneena Felix and Sai Suresh Veesam in generating the Simulink results reported in this chapter. This work was performed as part of the project "Simulation Centre for Power Electronics and Power Systems" under the National Mission on Power Electronics Technology (NaMPET), Phase 2, sponsored by the Department of Electronics and Information Technology, Govt. of India.

Chapter 5

Is $y = \dfrac{dx}{dt}$ same as $x = \displaystyle\int y\,dt$?

Mathematically, $y = \dfrac{dx}{dt}$ is the same as $x = \displaystyle\int y\,dt$ (assuming that the constant of integration is suitably assigned). However, the numerical solutions obtained from the two equations give different results in Matlab/Simulink. It is the purpose of this chapter to observe this difference, understand its origin, and look at the implications for simulation of filters[1].

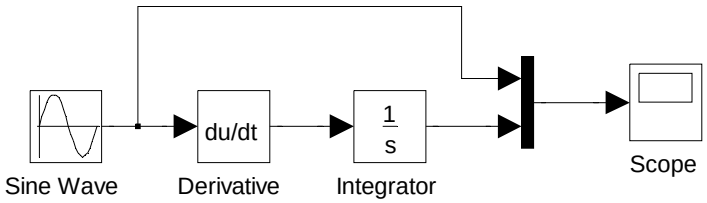


Figure 5.1: Simulink implementation of a differentiator followed by an integrator, with an input $x = \sin t$.

Consider the Simulink schematic shown in Fig. 5.1. We expect the output waveform $y(t)$ to be identical to the input waveform $x(t)$ (assuming that we have set the initial condition $y(0) = 0$). However, numerical results show some difference between the two, as shown in Figs. 5.2 and 5.3. The discrepancy is larger when a larger time step is used. Why does this happen?

The answer has to do with the fact that the integrator and differentiator elements are implemented in Simulink in drastically different ways. The integrator is implemented in a consistent manner, using well-known numerical integration (NI) methods such as the Runge-Kutta method and its variants. Let us consider the Forward Euler (FE) method, the simplest of NI schemes, to illustrate the implementation of an integrator. For the block diagram shown in Fig. 5.4 (a). the FE method gives

$$y = \int x\,dt \rightarrow \frac{dy}{dt} = x \rightarrow y_{i+1} = y_i + h\,x_i\,, \tag{5.1}$$

[1]This chapter is mainly about a subtle implementation issue; it can be skipped without loss of continuity.

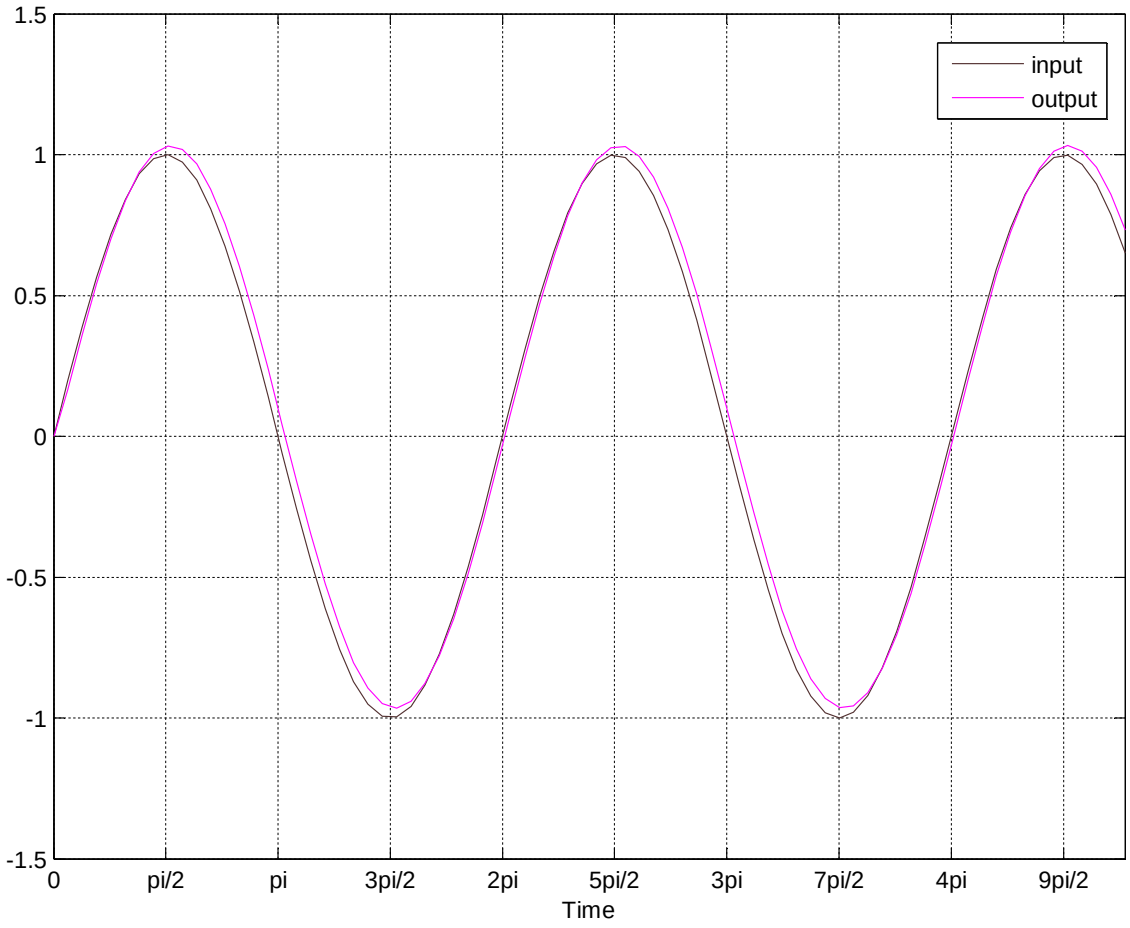


Figure 5.2: Simulink results for the block diagram of Fig. 5.1 with a time step of 0.2 seconds, using the `ode4` algorithm.

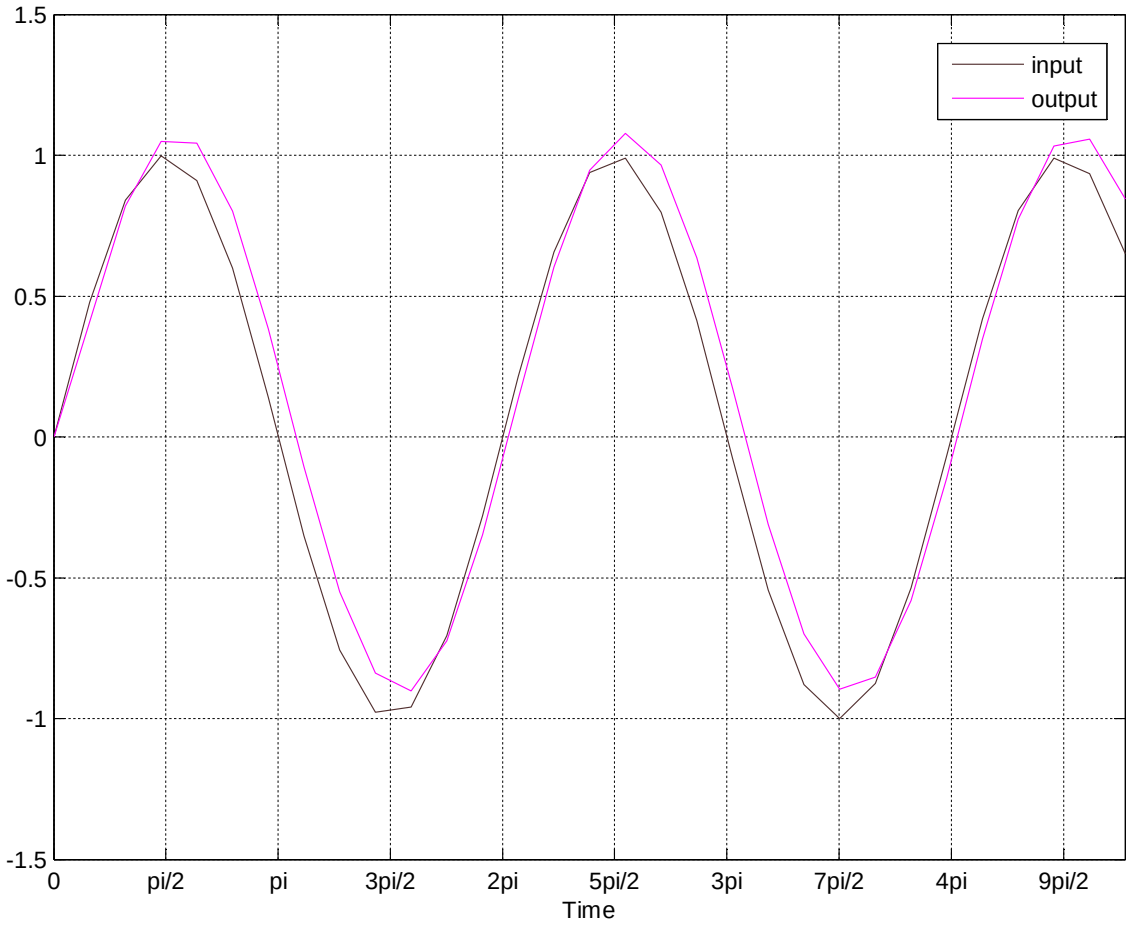


Figure 5.3: Simulink results for the block diagram of Fig. 5.1 with a time step of 0.5 seconds, using the `ode4` algorithm.

(a) (b)

Figure 5.4: (a) An integrator with a sinusoidal input, (b) A differentiator with a sinusoidal input.

where x_i and y_i represent the numerical solution at t_i, and $h = t_{i+1} - t_i$ is the time step. Eq. 5.1 has to be solved together with the equation for the sinusoidal source, viz.,

$$x_i = \sin \omega t_i \,. \tag{5.2}$$

That is straightforward – We evaluate x_i from Eq. 5.2, substitute it in Eq. 5.1 to obtain y_{i+1}, and repeat this procedure to obtain y_{i+2}, y_{i+3}, etc. For other methods used by Simulink (such as Runge-Kutta 4), the idea remains the same although the computation is more complex.

Now consider the block diagram shown in Fig. 5.4 (b). Using the FE method, we would obtain,

$$\frac{dx}{dt} = y \rightarrow x_{i+1} = x_i + h\, y_i \,. \tag{5.3}$$

Using the above equation, we can obtain x_{i+1}. However, x_{i+1} must also satisfy

$$x_{i+1} = \sin \omega t_{i+1} \,. \tag{5.4}$$

Clearly, there is a conflict between these two computations, and the FE approach (and other methods such as RK4) cannot be used. Simulink therefore uses an *estimate* of the derivative whenever the derivative block (denoted by $\dfrac{du}{dt}$ in the Simulink library) is present in the user's schematic. The derivative is estimated using past values x_i, x_{i-1}, etc. For example, a simple backward difference approximation is given by[2]

$$\left.\frac{dx}{dt}\right|_{t_i} = \frac{x_i - x_{i-1}}{t_i - t_{i-1}} \,. \tag{5.5}$$

This method introduces a lag between the numerically computed $\dfrac{dx}{dt}$ and the expected value of $\dfrac{dx}{dt}$, which becomes more pronounced when the time step is large. We have seen a manifestation of this phenomenon in Figs. 5.2 and 5.3.

To see this effect more clearly, let us generate $y = \cos t$ by employing an integrator and a differentiator, as shown in Figs. 5.5 and 5.6. Figs. 5.7, 5.8, 5.9, 5.10 show the results obtained with Simulink. The error introduced by the differentiator can be clearly seen by comparing the plots, in particular by looking at the zero-crossing time of the output waveform. When an integrator is used, the results are not sensitive to the time step; when a differentiator is used, the results show a significant dependence on the time step.

[2]Whether Simulink uses a first-order backward difference formula is not clear; something similar is probably being used.

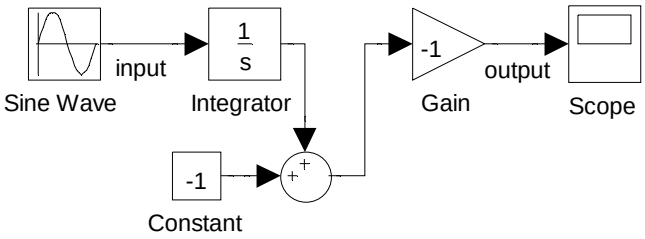


Figure 5.5: Simulink schematic diagram for obtaining $\cos t$ from $\sin t$ using an integrator.

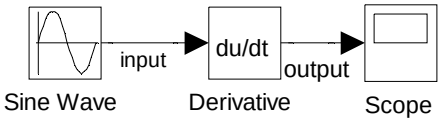


Figure 5.6: Simulink schematic diagram for obtaining $\cos t$ from $\sin t$ using a differentiator.

The difference between implementation of an integrator and a differentiator has other consequences. We will illustrate this point with respect to a first-order filter. For a low-pass filter (LPF), we have

$$y(s) = \frac{1}{1 + (s/\omega_0)} \rightarrow y(t) + \frac{1}{\omega_0}\frac{dy}{dt} = x(t)\,, \tag{5.6}$$

$$\text{i.e., } \frac{dy}{dt} = \omega_0 x(t) - \omega_0 y(t)\,, \tag{5.7}$$

which can be handled with standard explicit NI methods (e.g., `ode4`, `ode45`, `ode23` in Simulink).

For a high-pass filter (HPF), we have

$$y(s) = \frac{(s/\omega_0)}{1 + (s/\omega_0)} \rightarrow y(t) + \frac{1}{\omega_0}\frac{dy}{dt} = \frac{1}{\omega_0}\frac{dx}{dt}\,, \tag{5.8}$$

The term $\dfrac{dx}{dt}$ needs to be computed approximately (see Eq. 5.5), and that gives rise to numerical errors over and above (and generally much larger than) the errors due to the NI method itself (such as `ode4` and `ode45`).

As an example, let us consider first-order low-pass and high-pass filters (Eqs. 5.6, 5.8) with $f_0 = 1$ Hz. If the signal frequency is equal to f_0 (i.e., $\omega = \omega_0$), the phase difference between the output $y(t)$ and the input $x(t)$ is expected to be $-45°$ and $+45°$ for the integrator and differentiator, respectively. This phase difference corresponds to $\pm T/8$ on the time axis, i.e., the zero-crossing points of the input and output are expected to differ by $\pm T/8$.

Figs. 5.11 and 5.12 show the block diagrams of the LPF (Eq. 5.6) and HPF (Eq. 5.8) as implemented in Simulink. Figs. 5.13, 5.14, 5.15 show the Simulink results for the LPF, as obtained with the `ode4` method with a time step of 0.02, 0.05, and 0.1 second, respectively. Note that the phase difference continues to be $-45°$ even when a relatively large time step is

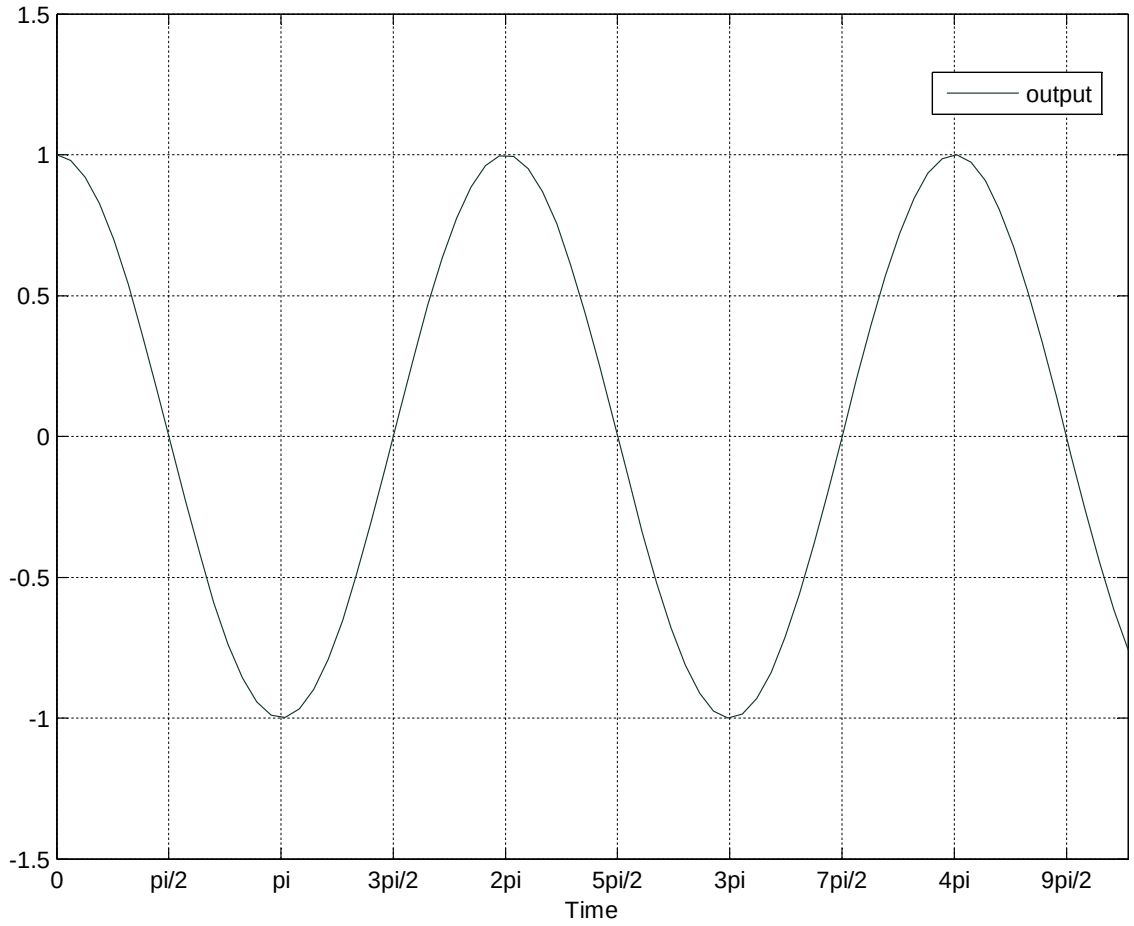


Figure 5.7: Simulink output for the schematic of Fig. 5.5, using the `ode4` method with a time step of 0.2 seconds.

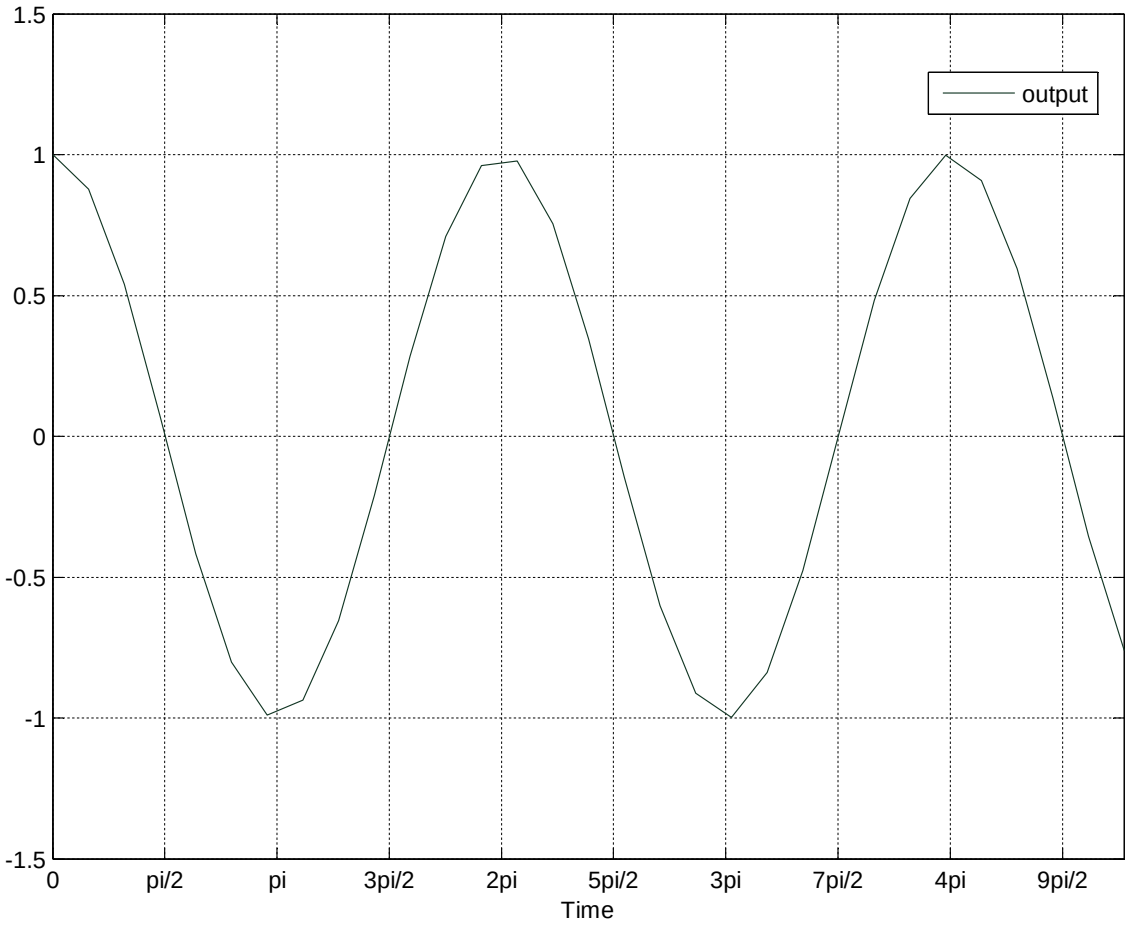


Figure 5.8: Simulink output for the schematic of Fig. 5.5, using the `ode4` method with a time step of 0.5 seconds.

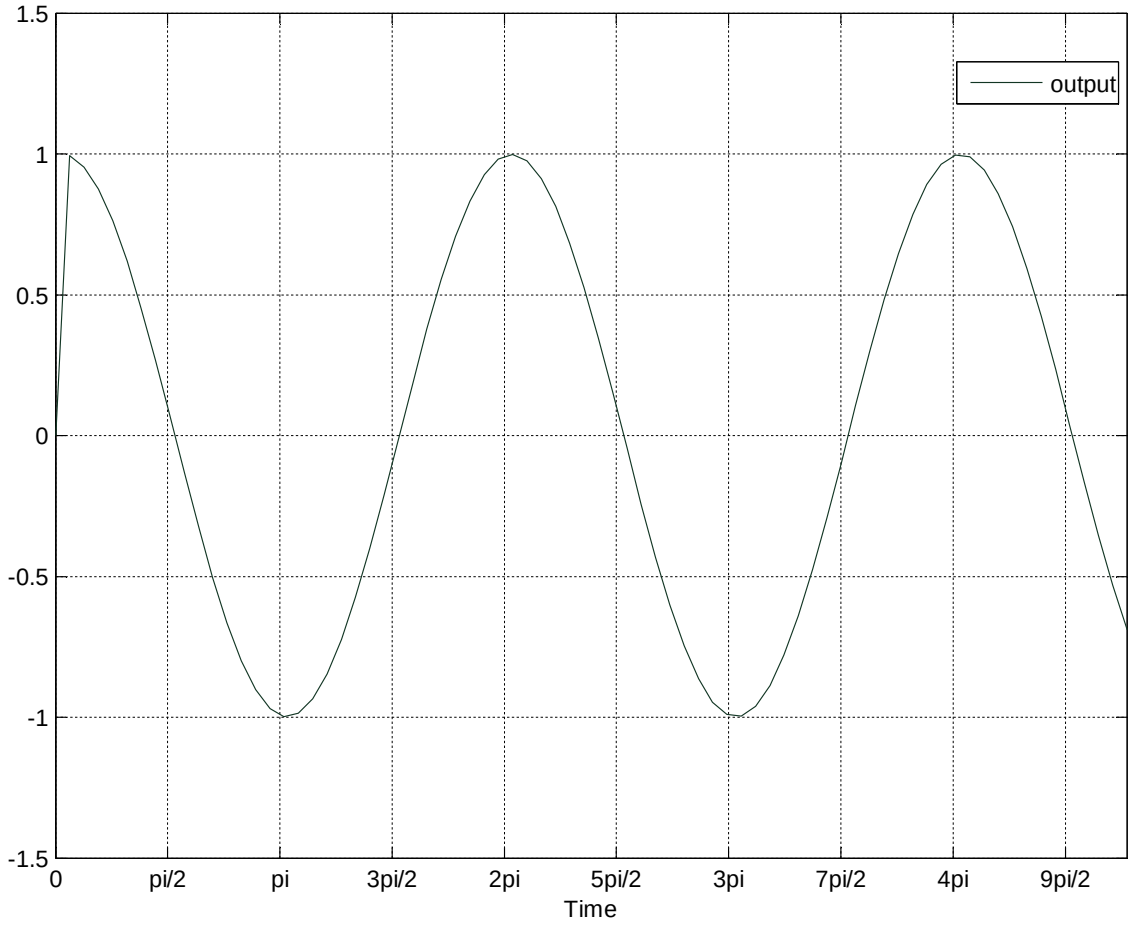


Figure 5.9: Simulink output for the schematic of Fig. 5.6, using the `ode4` method with a time step of 0.2 seconds.

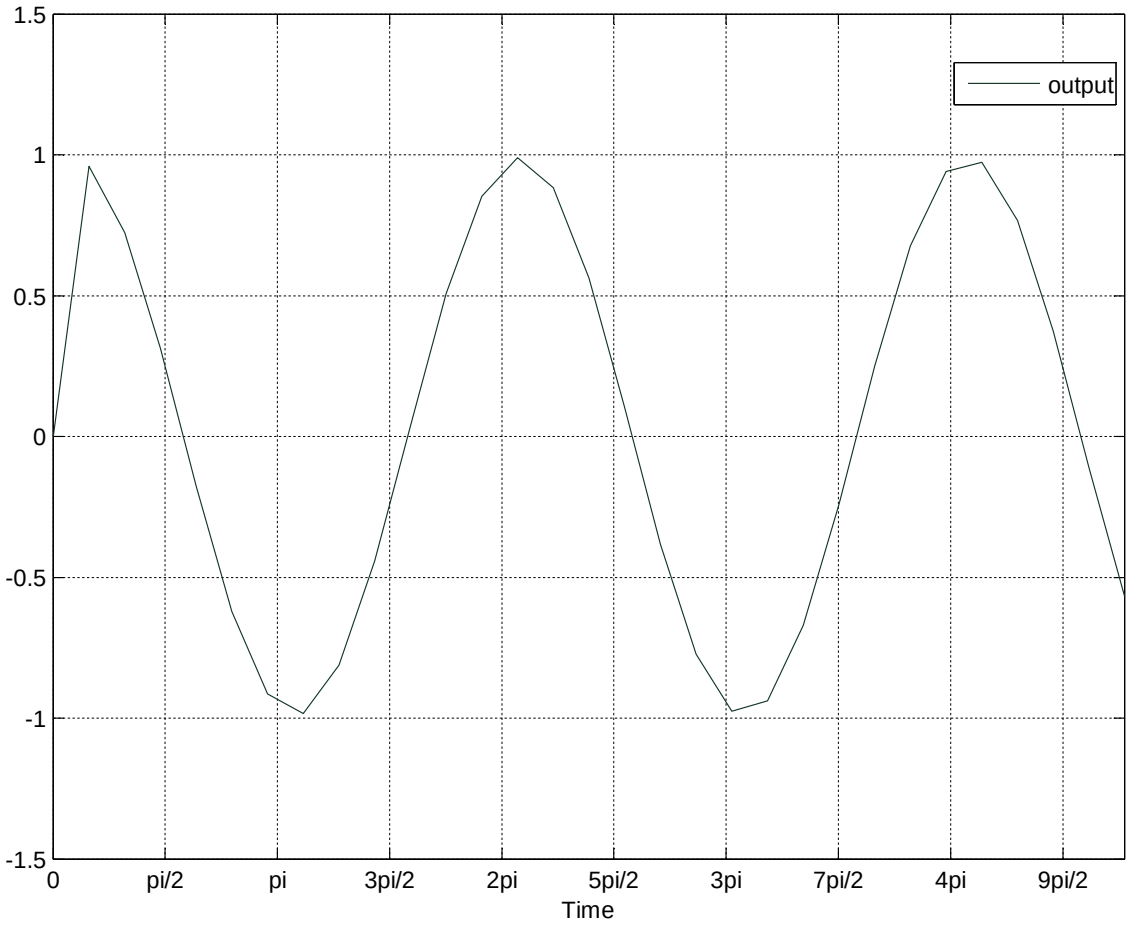


Figure 5.10: Simulink output for the schematic of Fig. 5.6, using the `ode4` method with a time step of 0.5 seconds.

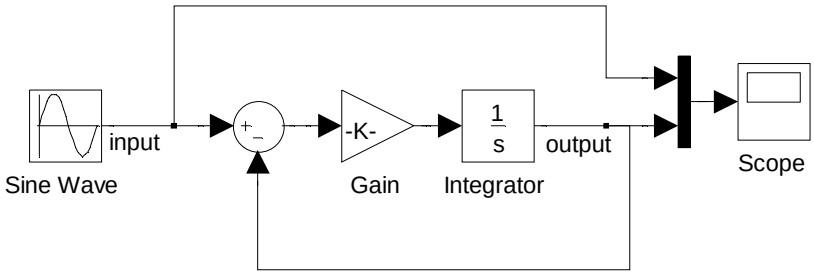


Figure 5.11: Simulink block diagram of LPF (see text).

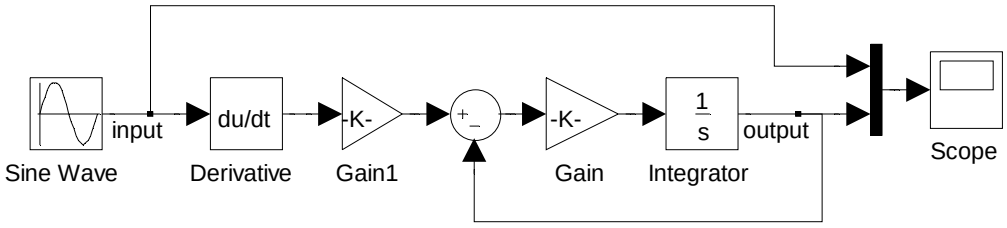


Figure 5.12: Simulink block diagram of HPF (see text).

used[3]. On the other hand, for the HPF case, the phase difference between the input and output starts deviating from 45° as the time step is made larger (see Figs. 5.16, 5.17, 5.18). Table 5.1 summarises the LPF and HPF results.

Time Step (sec)	0.02	0.05	0.1
Phase Diff in LPF (°)	-45	-45	-45
Phase Diff in HPF (°)	42.5	38.7	33.0

Table 5.1: Phase difference obtained with the Simulink `ode4` method for different time steps (see text).

Simulink also provides a "transfer function" block which can be used directly as a high-pass filter. Using this block, we can implement

$$H(s) = \frac{s}{1+s}\,, \tag{5.9}$$

which is a HPF with $\omega_0 = 1$ rad/s (see Eq. 5.8). When this block is used, the output waveform shows[4] the expected phase shift of +45° for all three time steps, viz., 0.02, 0.05, and 0.1 s, and the discrepancy we observed in Table 5.1 is removed. What is the secret?

The secret (likely) is that a transfer function with s, s^2, etc. in the numerator can also be written using long division and partial fractions. An expression of the form

$$H(s) = \frac{N(s)}{D(s)} = \frac{a_0 + a_1 s + \cdots + a_M s^M}{b_0 + b_1 s + \cdots + b_N s^N} \tag{5.10}$$

with $M \le N$ can be reduced to[5]

$$H(s) = A_0 + \frac{A_1}{s - z_1} + \frac{A_2}{s - z_2} + \cdots + \frac{A_N}{s - z_N}\,. \tag{5.11}$$

With this change, $y(s) = H(s)\,x(s)$ becomes

$$y(s) = y_0(s) + y_1(s) + \cdots + y_N(s)\,, \tag{5.12}$$

[3]The waveforms appear segmented when a large time step is used. As we have seen in Chapter 4, this is a matter of appearance rather than accuracy of the numerical solution.

[4]The simulation results are not given here.

[5]We will assume that the roots of $D(s)$ (i.e., z_1, z_2 $\cdots$) are real; however, the method can be extended to handle complex roots of $D(s)$ as well.

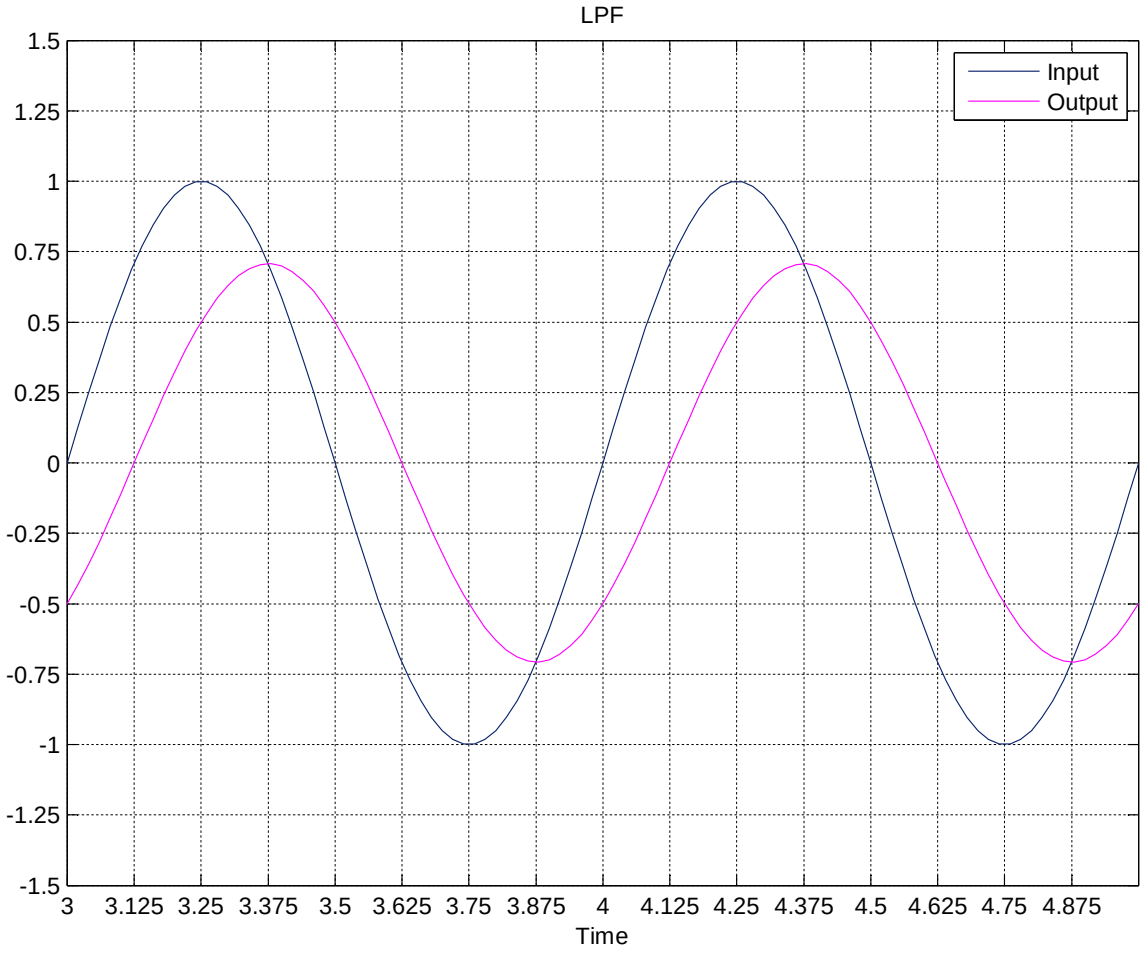


Figure 5.13: Simulink output for the low-pass filter (see text) obtained with the `ode4` method with a time step of 0.02 seconds.

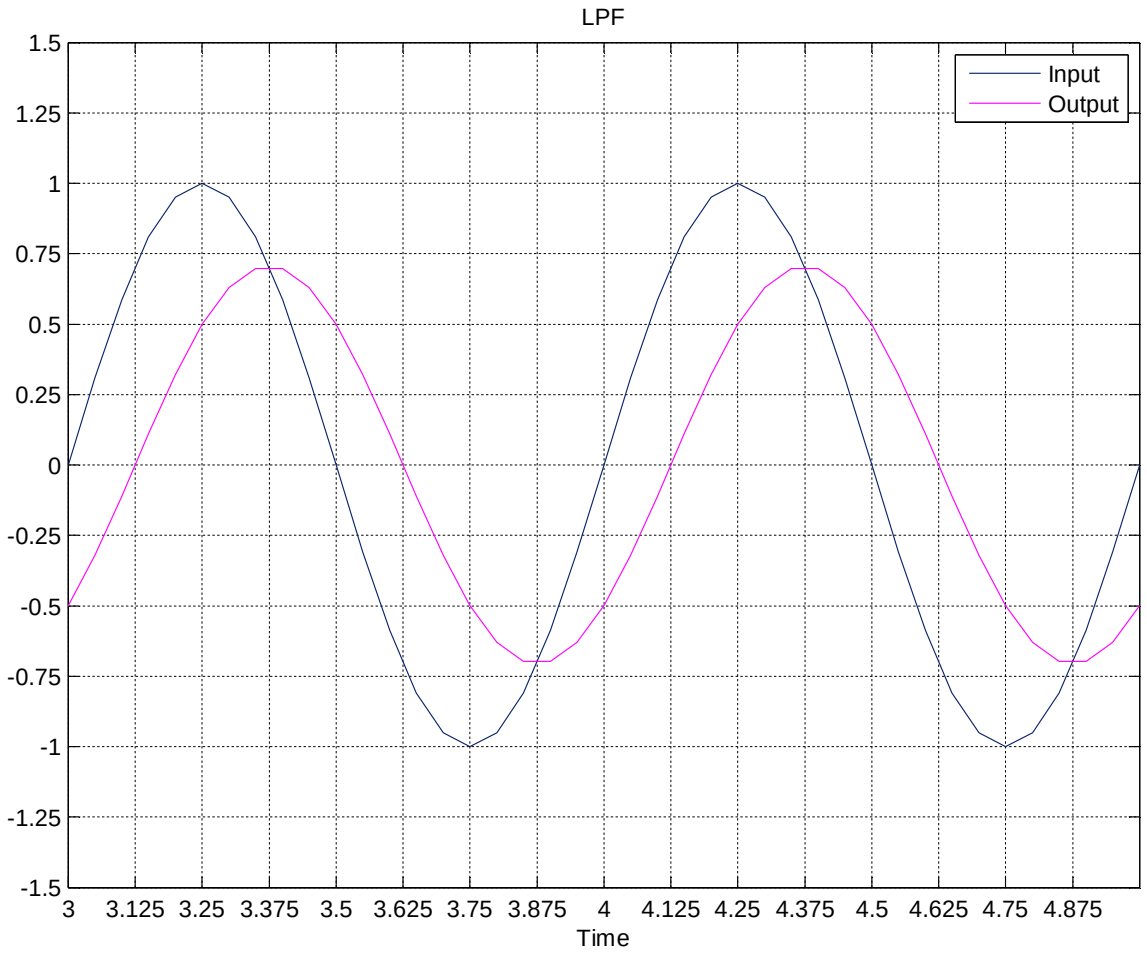


Figure 5.14: Simulink output for the low-pass filter (see text) obtained with the `ode4` method with a time step of 0.05 seconds.

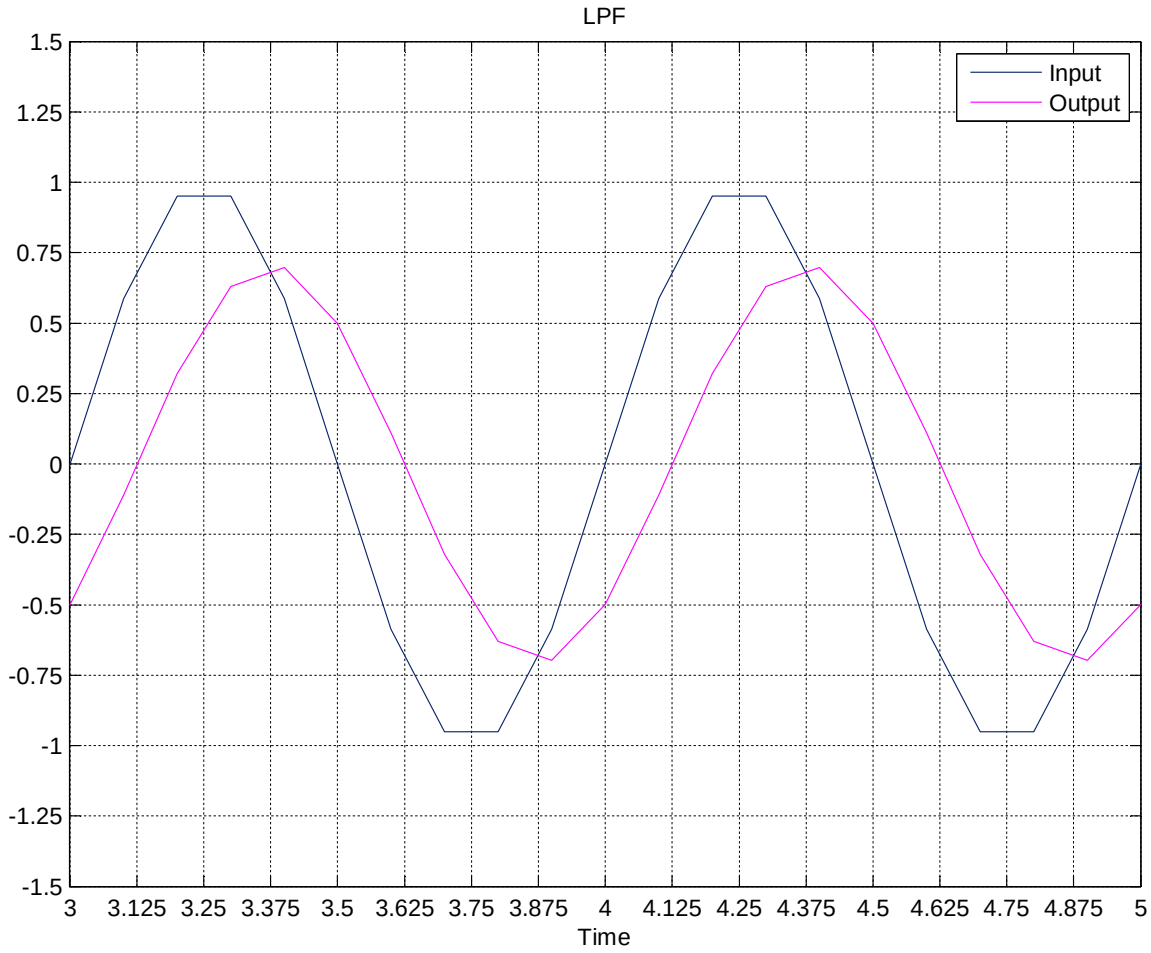


Figure 5.15: Simulink output for the low-pass filter (see text) obtained with the `ode4` method with a time step of 0.1 seconds.

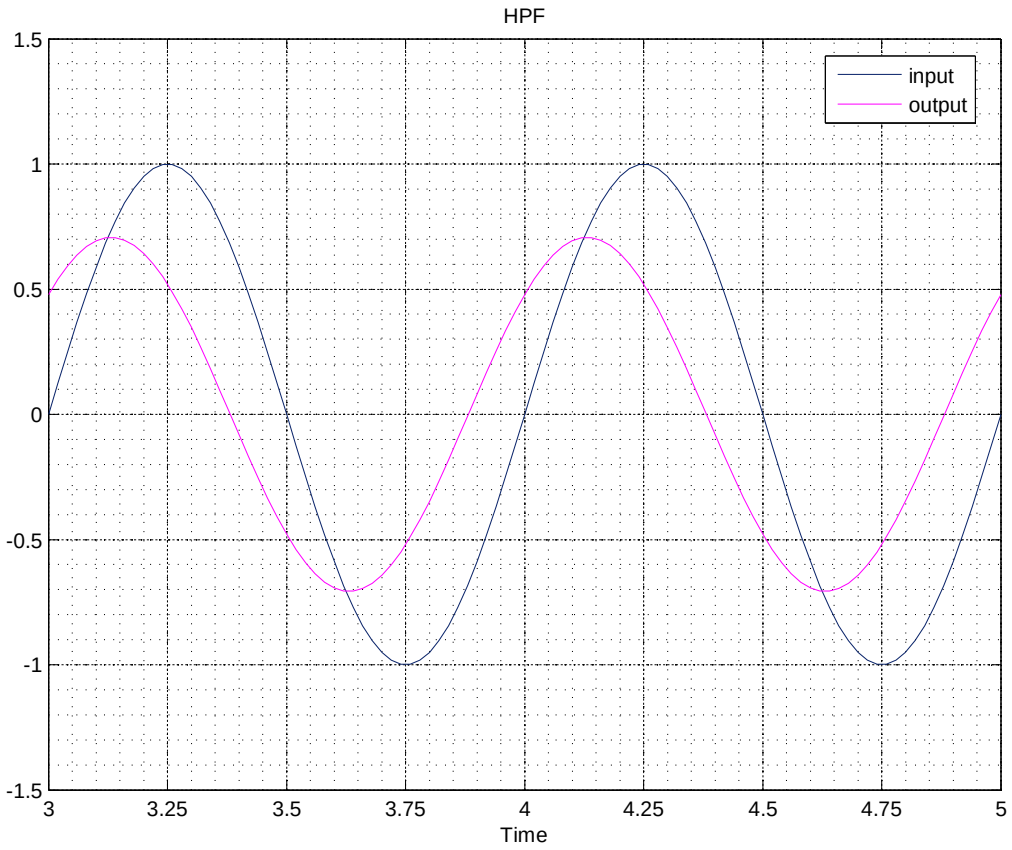


Figure 5.16: Simulink output for the high-pass filter (see text) obtained with the `ode4` method with a time step of 0.02 seconds.

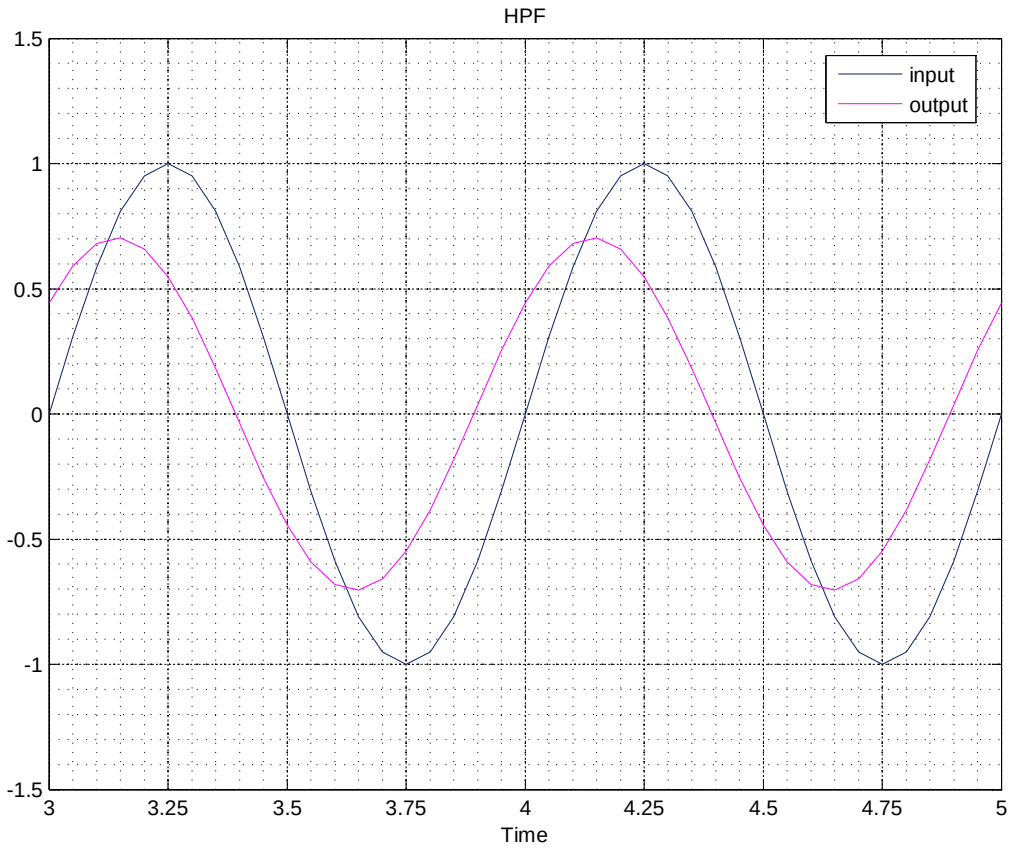


Figure 5.17: Simulink output for the high-pass filter (see text) obtained with the `ode4` method with a time step of 0.05 seconds.

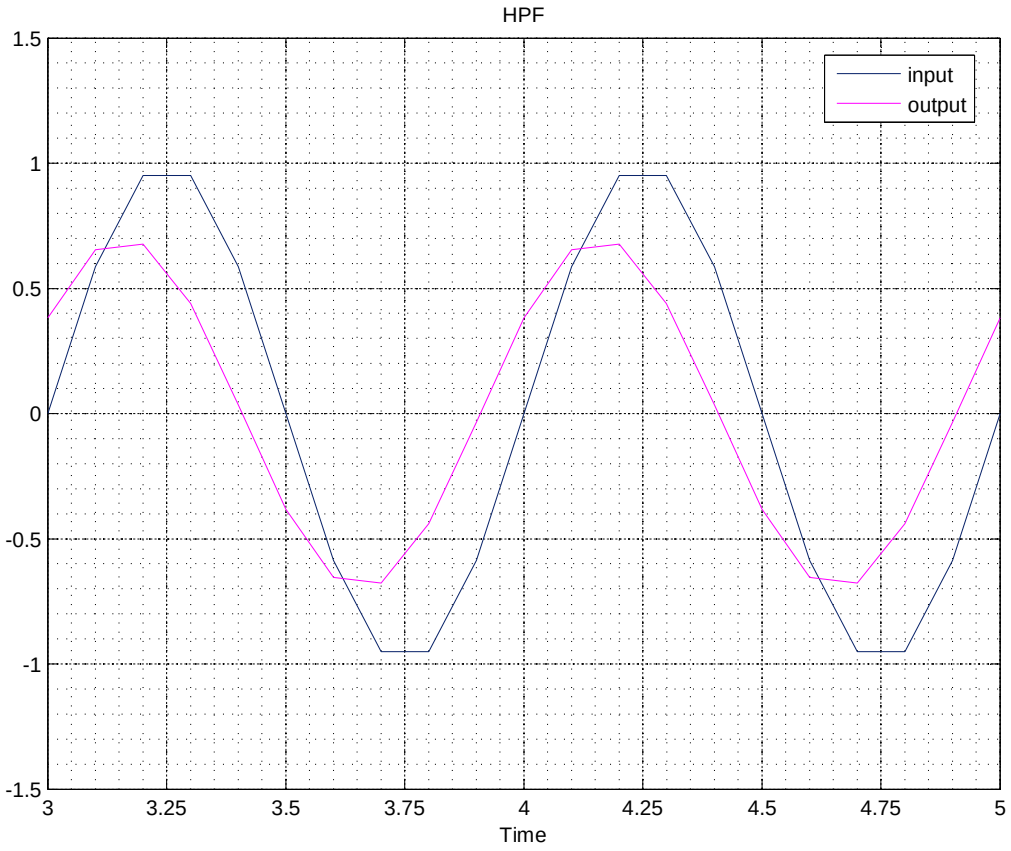


Figure 5.18: Simulink output for the high-pass filter (see text) obtained with the `ode4` method with a time step of 0.1 seconds.

where $y_0(s) = A_0\, x(s)$, $y_1(s) = \dfrac{A_1}{s - z_1}\, x(s)$, etc. In the time domain, we get

$$y_0(t) = A_0 x(t), \quad \frac{dy_1}{dt} = A_1 x + z_1 y_1, \cdots \quad , y(t) = \sum_0^N y_i(t). \tag{5.13}$$

Note that the second equation involving $\dfrac{dy_1}{dt}$ can be handled using standard explicit methods (`ode4` etc). In effect, the computation of $\dfrac{dx}{dt}$, $\dfrac{d^2x}{dt^2}$, etc. has been bypassed, and the derivative approximation (Eq. 5.5) is not required any more.

Acknowledgments

The author would like to acknowledge the contribution of Aneena Felix in generating the Simulink results reported in this chapter. This work was performed as part of the project "Simulation Centre for Power Electronics and Power Systems" under the National Mission on Power Electronics Technology (NaMPET), Phase 2, sponsored by the Department of Electronics and Information Technology, Govt. of India.

Chapter 6

Numerical Solution of ODEs: Implicit Methods

We have discussed a few explicit methods (in particular, the Forward Euler (FE), Runge-Kutta order 4 (RK4), and Runge-Kutta-Fehlberg (RKF45) methods) for solving the ODE

$$\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0. \tag{6.1}$$

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 [9], 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[1] methods provide that alternative.

6.1 Backward Euler, trapezoidal, and BDF2 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 [6]; here, we will present a simplistic intuitive picture. Suppose the solution $x(t)$ of Eq. 6.1 is given by the curve shown in Fig. 6.1. We are interested in obtaining a numerical solution at discrete time points $t_1, t_2, \cdots$.

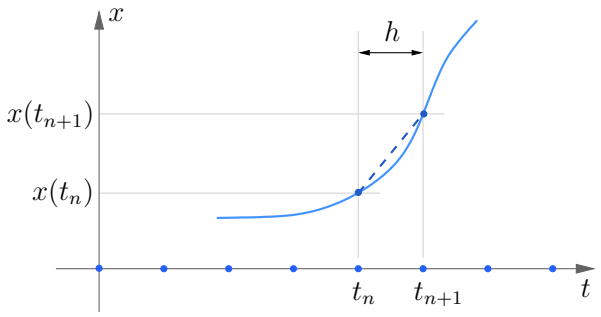


Figure 6.1: An intuitive view of the FE, BE, TRZ methods.

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

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \frac{x_{n+1} - x_n}{h} \approx \left.\frac{dx}{dt}\right|_{(t_n, x_n)} = f(t_n, x_n), \tag{6.2}$$

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

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \frac{x_{n+1} - x_n}{h} \approx \left.\frac{dx}{dt}\right|_{(t_{n+1}, x_{n+1})} = f(t_{n+1}, x_{n+1}), \tag{6.3}$$

leading to

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}). \tag{6.4}$$

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.,

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \frac{x_{n+1} - x_n}{h} \approx \frac{1}{2}\left(\left.\frac{dx}{dt}\right|_{(t_n, x_n)} + \left.\frac{dx}{dt}\right|_{(t_{n+1}, x_{n+1})}\right) = \frac{1}{2}\left(f(t_n, x_n) + f(t_{n+1}, x_{n+1})\right), \tag{6.5}$$

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

leading to

$$x_{n+1} = x_n + \frac{h}{2}\left(f(t_n, x_n) + f(t_{n+1}, x_{n+1})\right). \tag{6.6}$$

Note that we have shown the time steps to be uniform in Fig. 6.1, 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. 6.4 and 6.6. In other words, x_{n+1} cannot be simply *evaluated* in terms of known quantities[2]; instead, it needs to be obtained by *solving* Eq. 6.4 or 6.6. As an example, consider

$$\frac{dx}{dt} \equiv f(t, x) = \cos x, \quad x(0) = 0. \tag{6.7}$$

The FE and BE methods give the discretised equations,

$$\begin{aligned} \text{FE}: &\quad x_{n+1} = x_n + h\,\cos(x_n), \\ \text{BE}: &\quad x_{n+1} = x_n + h\,\cos(x_{n+1}). \end{aligned} \tag{6.8}$$

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. 6.7. The results are shown in Eq. 6.2.

```
#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);
```

[2]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., $f(t, x) = \cos \omega t$.

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

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

$$\begin{aligned} \text{BE:} \quad & \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + h\,\mathbf{f}(t_{n+1}, \mathbf{x}^{(n+1)}), \\ \text{TRZ:} \quad & \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \frac{h}{2}\left(\mathbf{f}(t_n, \mathbf{x}^{(n)}) + \mathbf{f}(t_{n+1}, \mathbf{x}^{(n+1)})\right), \end{aligned} \tag{6.9}$$

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:

$$\begin{aligned} \frac{dx_1}{dt} &= a_1\left(\sin\omega t - x_1\right)^3 - a_2\left(x_1 - x_2\right), \\ \frac{dx_2}{dt} &= a_3\left(x_1 - x_2\right), \end{aligned} \tag{6.10}$$

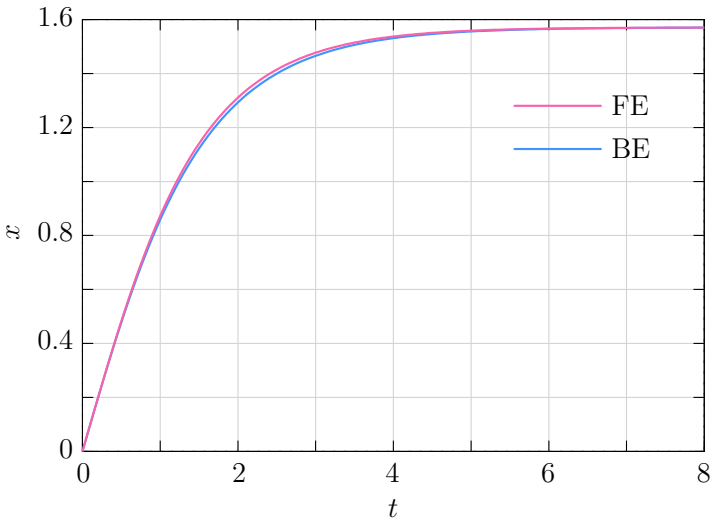


Figure 6.2: Numerical solutions of Eq. 6.7 obtained with the FE and BE using a step size of $h = 0.05$.

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

$$\begin{aligned} x_1^{(n+1)} &= x_1^{(n)} + h\left[a_1\left(\sin\omega t_{n+1} - x_1^{(n+1)}\right)^3 - a_2\left(x_1^{(n+1)} - x_2^{(n+1)}\right)\right], \\ x_2^{(n+1)} &= x_2^{(n)} + h\left[a_3\left(x_1^{(n+1)} - x_2^{(n+1)}\right)\right]. \end{aligned} \tag{6.11}$$

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[3]. 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.

Another implicit method commonly used in circuit simulation is the Backward Differentiation Formula of order 2 (BDF2), also known as Gear's method of order 2. It is given by

$$\frac{3}{2}\,x_{n+1} - 2\,x_n + \frac{1}{2}\,x_{n-1} = h\,f(t_{n+1}, x_{n+1}). \tag{6.12}$$

Note that the BDF2 method involves x_{n-1} (in addition to x_{n+1} and x_n), i.e., the numerical solution at t_{n-1}. In deriving Eq. 6.12, the time step is assumed to be uniform, i.e., $t_{n+1} - t_n = t_n - t_{n-1} = h$; however, it is possible to extend the BDF2 formula to the case of non-uniform time steps (see [5]). With the BDF2 method (methods involving multiple steps in general), starting is an issue since at the very beginning ($t = t_0$), the solution is available only at one time point, $x(t_0)$. In practice, a single-step method such as the BE method is used to first go from t_0 to t_1. The BDF2 method can now be used to compute x_2 from x_0 and x_1, x_3 from x_1 and x_2, and so on.

[3]It is $O(N^3)$ for a dense matrix.

6.2 Stability

As we have seen in Sec. 4.5, 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,

$$\frac{dx}{dt} = -\frac{t}{\tau}, \quad x(0) = 1. \tag{6.13}$$

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 [6], for example). The regions of stability of the AB methods with respect to Eq. 6.13 are shown in Fig. 6.3 (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. 6.13 are shown in Fig. 6.3 (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[4].

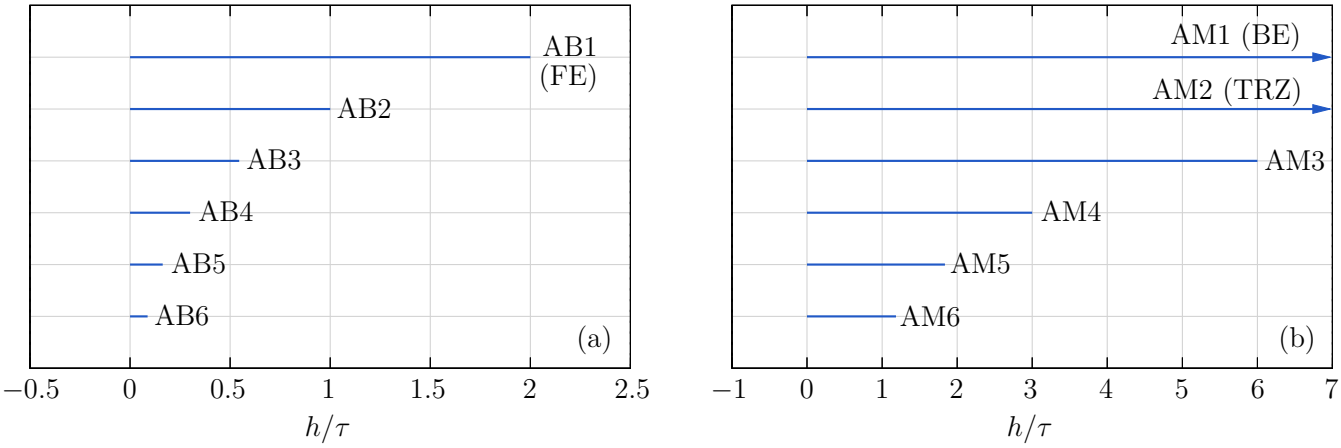
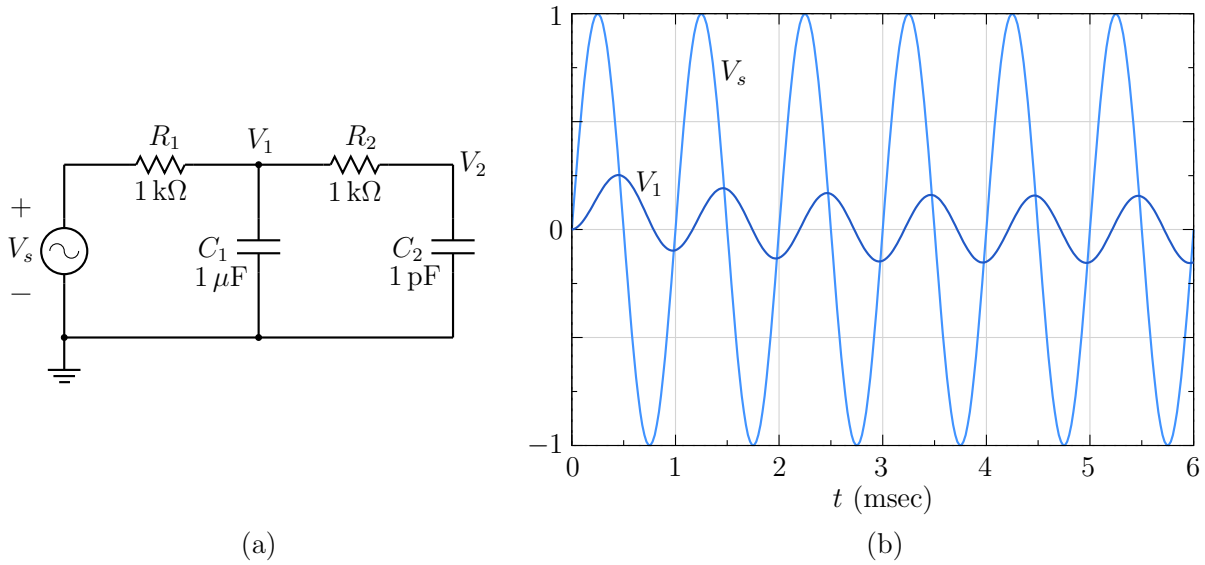


Figure 6.3: Regions of stability for Adams-Bashforth and Adams-Moulton methods of various orders with respect to the test equation $\frac{dx}{dt} = -t/\tau$.

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. 4.5). 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. 6.4 (a) which we have seen earlier. 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. 6.4 (b).

[4]The BDF2 formula given by Eq. 6.12 is also unconditionally stable, see [6].



(a) (b)

Figure 6.4: (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 $V_m = 1\,\text{V}$, $f = 1\,\text{kHz}$. The time step is $h = 0.02\,\text{msec}$.

As another example, consider the half-wave rectifier circuit of Fig. 6.5. The diode is represented with the $R_{\text{on}}/R_{\text{off}}$ model, with a turn-on voltage equal to 0 V and $R_{\text{off}} = 1\,\text{M}\Omega$. The following ODE describes the circuit behaviour.

$$\frac{dV_o}{dt} = \frac{1}{C}\left[\frac{V_s - V_o}{R_D} - \frac{V_o}{R}\right], \tag{6.14}$$

where R_D is the diode resistance (R_{on} if $V_s > V_o$; R_{off} otherwise). Fig. 6.5 shows the numerical solution of Eq. 6.14 obtained with the RKF45 method. The charging and discharging intervals can be clearly identified – charging takes place when the diode current is non-zero; otherwise, the capacitor discharges through the load resistor. When the diode conducts, the circuit time constant is $\tau_1 = (R_{\text{on}} \parallel R)\,C$ (approximately $R_{\text{on}}C$); 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}.

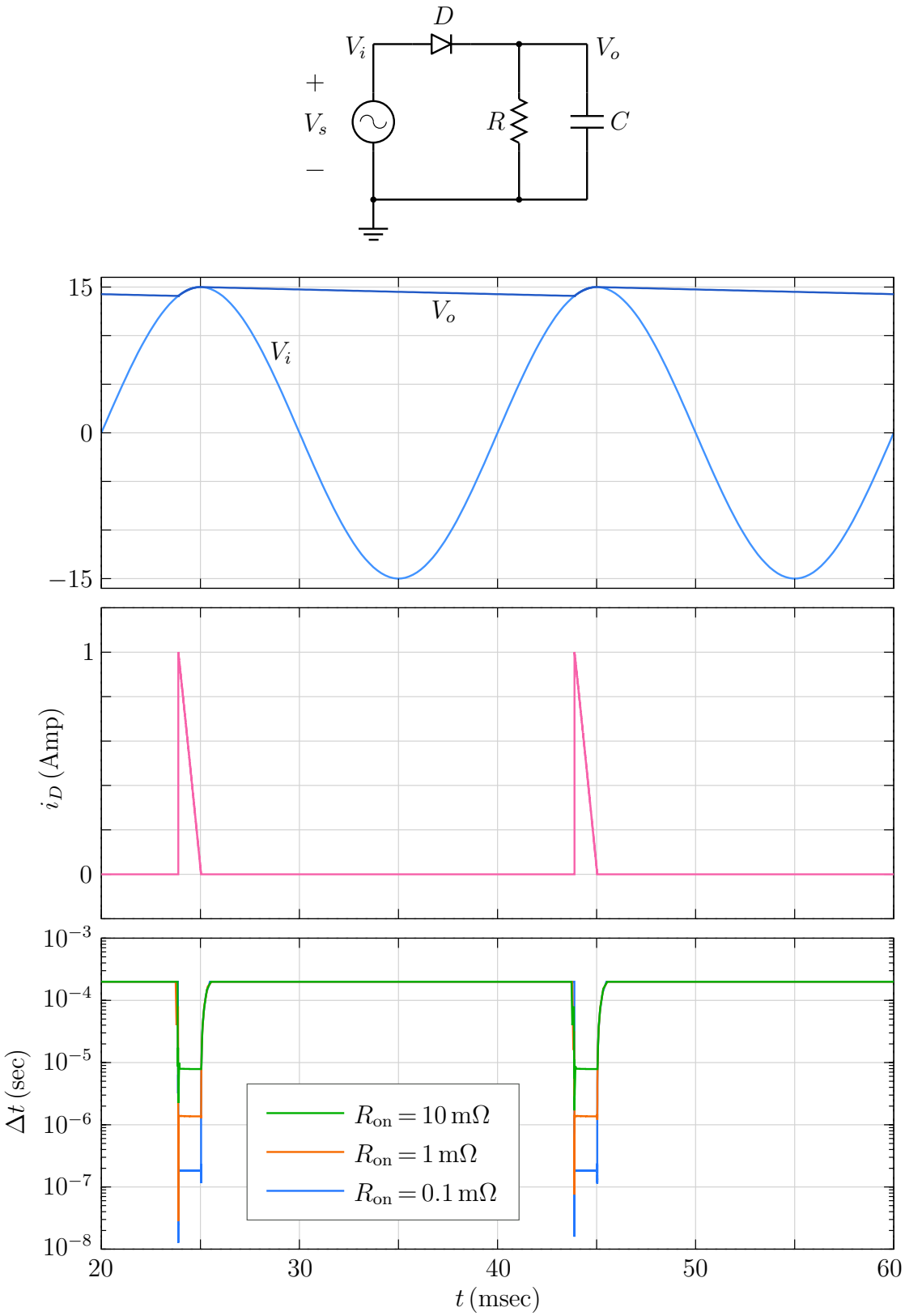


Figure 6.5: RKF45 results for a half-wave rectifier. (a) $V_i(t)$ and $V_o(t)$ in Volts, (b) diode current versus time, (c) time step used by the RKF45 method versus time. $V_s = V_m \sin \omega t$, with $V_m = 1\,\text{V}$, $f = 50\,\text{Hz}$, $R = 500\,\Omega$, $C = 600\,\mu\text{F}$. The R_{on}/R_{off} model is used for the diode, and the turn-on voltage is taken as 0 V. R_{off} is kept constant at $1\,\text{M}\Omega$.

6.3 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.

6.3.1 Oscillatory circuits

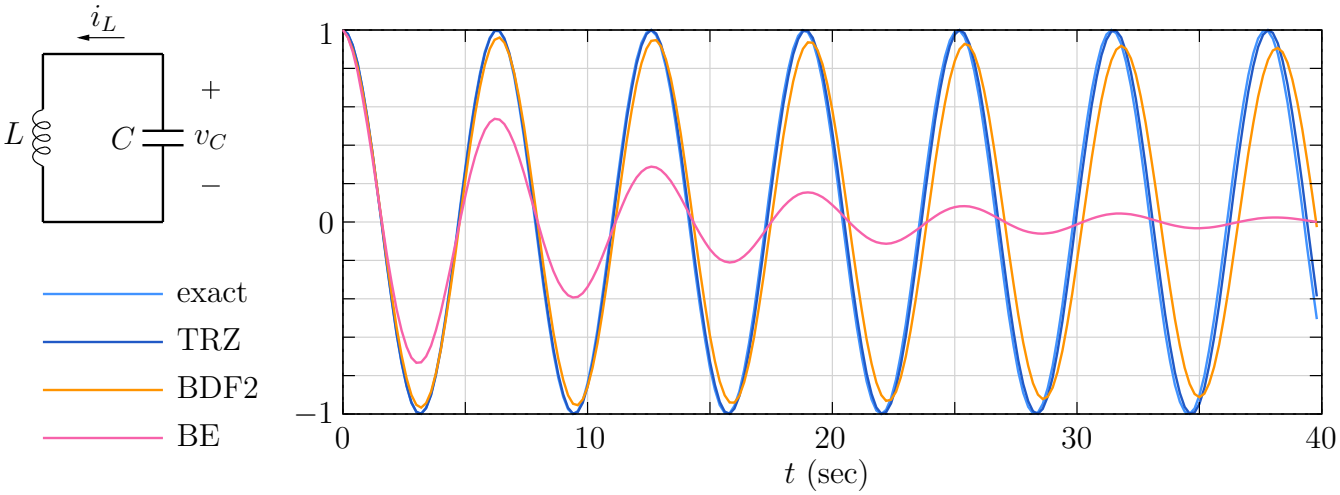


Figure 6.6: $v_C(t)$ obtained for an LC circuit with Backward Euler, Trapezoidal, and second-order Gear (BDF2) methods. $L = 1\,\text{H}$, $C = 1\,\text{F}$, and h (time step)=0.2 sec in all cases. The exact solution is also shown for comparison.

Consider an LC circuit without any resistance (see Fig. 6.6) and with the initial conditions, $v_C(0) = 1\,\text{V}$, $i_L(0) = 0\,\text{A}$. The circuit equations can be described by the following set of ODEs.

$$\begin{aligned} \frac{dv_C}{dt} &= -\frac{1}{C}\, i_L\,, \\ \frac{di_L}{dt} &= \frac{1}{L}\, v_C\,, \end{aligned} \tag{6.15}$$

with the analytic solution given by

$$\begin{aligned} v_C &= \cos(\omega t)\,, \\ i_L &= \sin(\omega t)\,, \end{aligned} \tag{6.16}$$

with $\omega = 1/\sqrt{LC}$. We will consider $L = 1\,\text{H}$, $C = 1\,\text{F}$ which gives the frequency of oscillation $f_0 = 1/2\pi\,\text{Hz}$, i.e., a period $T = 2\pi\,\text{sec}$. Fig. 6.6 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[5]. In such cases, the TRZ method should be used.

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[6]. The phase error can be reduced by selecting a smaller time step.

6.3.2 Ringing

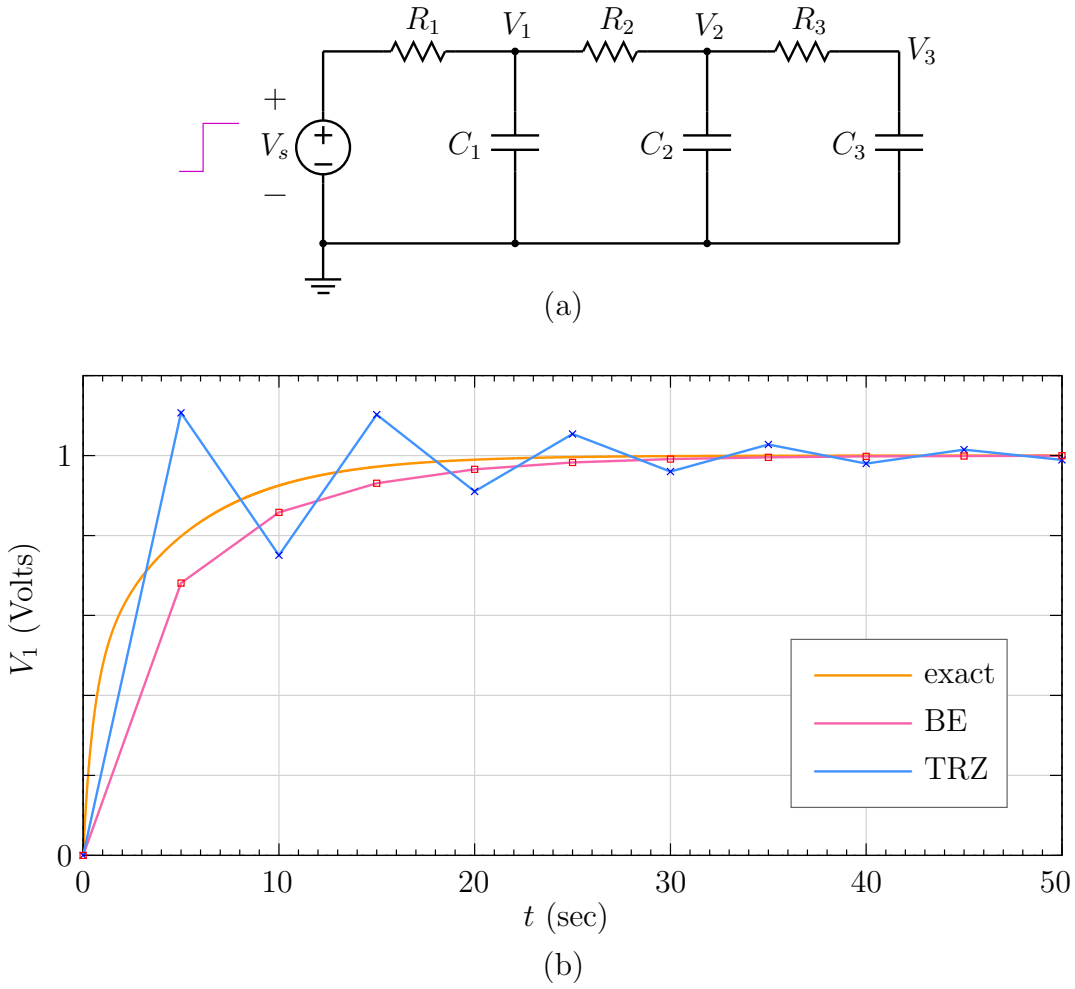


Figure 6.7: (a) *RC* circuit example, (b) Simulation results for V_1, with $R_1 = R_2 = R_3 = 1\,\Omega$, $C_1 = C_2 = C_3 = 1\,\text{F}$, $V_C(0) = 0\,\text{V}$ for all capacitors, and a step input going from 0 V to 1 V applied at $t = 0$.

Consider the *RC* circuit shown in Fig. 6.7. The time constants for this circuit are 0.31, 0.64, and 5.05 seconds, the smallest being $\tau_{\min} = 0.31\,\text{sec}$. Fig. 6.7 shows the BE and TRZ results with a relatively large time step. Sine the time step (5 sec) is much larger than $\tau_{\min}$, neither of the two methods can track closely the expected solution. However, there is a

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

[6]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.

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 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.

6.4 TR-BDF2 method

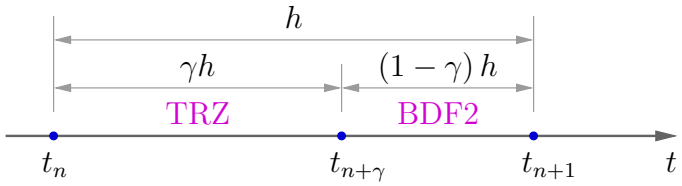


Figure 6.8: Division of the time interval h in the TR-BDF2 method.

The TR-BDF2 method is a combination of TRZ and BDF2 methods. In this method [11], the time interval h from t_n to t_{n+1} is divided into two intervals (see Fig. 6.8). The TRZ method is used to go from x_n to $x_{n+\gamma}$ (i.e., the numerical solution at $t_{n+\gamma} \equiv t_n + \gamma h$). In the second step, the BDF2 method is used (using the three points, t_n, $t_{n+\gamma}$, and t_{n+1}) to compute x_{n+1}. The constant γ is selected such that[7]

$$\frac{2}{\gamma} = \frac{2-\gamma}{1-\gamma}\,, \tag{6.17}$$

or $\gamma = 2 - \sqrt{2} \approx 0.59$.

The TR-BDF2 method has the following advantages.

(a) It does not exhibit ringing [11].

(b) Although BDF2 is a two-step method (see Eq. 6.12), TR-BDF2 is a single-step method since the solution at $t_{n+\gamma}$ is an intermediate result in going from t_n to t_{n+1}. Therefore, no special procedure for starting is required.

(c) The TR-BDF2 method can be used for auto (adaptive) time stepping by estimating the LTE and comparing it with a user-specified tolerance [11].

[7]For this choice, the TRZ and BDF2 Jacobian matrices are identical which means that the same *LU* factorisation can be used in the TRZ and BDF2 steps (see [11]).

6.5 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.

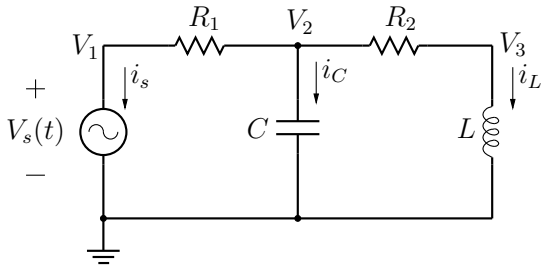


Figure 6.9: Circuit to illustrate equation assembly.

Consider the circuit in Fig. 6.9. The following equations describe the circuit behaviour, taking into account the relationship to be satisfied for each element in the circuit.

$$C\frac{dv_C}{dt} = G_1(V_s - V_2) - i_L\,, \tag{6.18}$$

$$L\frac{di_L}{dt} = V_3\,, \tag{6.19}$$

$$i_L = G_2(V_2 - V_3)\,, \tag{6.20}$$

where $G_1 = 1/R_1$. $G_2 = 1/R_2$. Note that the above system of equation is a "mixed" system – Eq. 6.20 is an algebraic equation whereas Eqs. 6.18 and 6.19 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. 6.18-6.20? 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

$$i_s + G_1(V_1 - V_2) = 0\,, \tag{6.21}$$

$$G_1(V_2 - V_1) + G_2(V_2 - V_3) + i_C = 0\,, \tag{6.22}$$

$$G_2(V_3 - V_2) + i_L = 0\,, \tag{6.23}$$

$$V_1 = V_s(t)\,, \tag{6.24}$$

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 $t = t_{n+1}$:

$$i_s^{n+1} + G_1(V_1^{n+1} - V_2^{n+1}) = 0\,, \tag{6.25}$$

$$G_1(V_2^{n+1} - V_1^{n+1}) + G_2(V_2^{n+1} - V_3^{n+1}) + i_C^{n+1} = 0\,, \tag{6.26}$$

$$G_2(V_3^{n+1} - V_2^{n+1}) + i_L^{n+1} = 0\,, \tag{6.27}$$

$$V_1^{n+1} = V_s(t_{n+1})\,. \tag{6.28}$$

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[8], 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

$$\frac{dV_2}{dt} = \frac{1}{C}\, i_C \;\rightarrow\; \frac{V_2^{n+1} - V_2^n}{h} = \frac{i_C^{n+1}}{C} \;\rightarrow\; i_C^{n+1} = \frac{C}{h}\,(V_2^{n+1} - V_2^n)\,, \tag{6.29}$$

$$\frac{di_L}{dt} = \frac{1}{L}\, V_3 \;\rightarrow\; \frac{i_L^{n+1} - i_L^n}{h} = \frac{V_3^{n+1}}{L} \;\rightarrow\; i_L^{n+1} = i_L^n + \frac{h}{L}\,(V_3^{n+1})\,, \tag{6.30}$$

where $h = t_{n+1} - t_n$ is the time step. Substituting for i_C^{n+1} and i_L^{n+1} in Eqs. 6.26 and 6.27, we get

$$G_1 V_1^{n+1} - G_1 V_2^{n+1} + i_s^{n+1} = 0\,, \tag{6.31}$$

$$-G_1 V_1^{n+1} + \left(G_1 + G_2 + \frac{C}{h}\right) V_2^{n+1} - G_2 V_3^{n+1} = \frac{C}{h}\, V_2^n\,, \tag{6.32}$$

$$-G_2 V_2^{n+1} + \left(G_2 + \frac{h}{L}\right) V_3^{n+1} = -i_L^n\,, \tag{6.33}$$

$$V_1^{n+1} = V_s(t_{n+1})\,, \tag{6.34}$$

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[9] 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. 6.10. The diode is described by

$$I_D = I_s\left(e^{V_D/V_T} - 1\right). \tag{6.35}$$

We can write the MNA equations for this circuit as

$$i_s + i_C = 0, \tag{6.36}$$

$$-i_C - i_D + GV_2 = 0 \;\rightarrow\; -i_C - I_s\left(e^{-V_2/V_T} - 1\right) + GV_2 = 0, \tag{6.37}$$

$$V_1 = V_s(t), \tag{6.38}$$

[8]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.

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

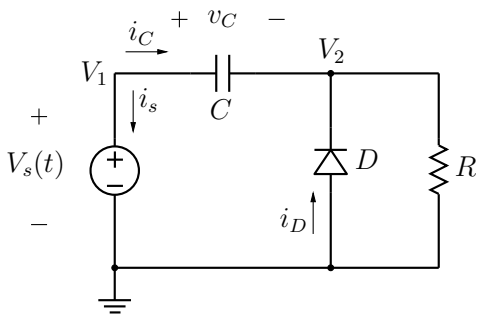


Figure 6.10: A nonlinear circuit.

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

$$i_s^{n+1} + i_C^{n+1} = 0, \tag{6.39}$$

$$-i_C^{n+1} - I_s\left(e^{-V_2^{n+1}/V_T} - 1\right) + GV_2^{n+1} = 0, \tag{6.40}$$

$$V_1^{n+1} = V_s(t_{n+1})\,. \tag{6.41}$$

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

$$\frac{dv_C}{dt} = \frac{1}{C}\,i_C \;\rightarrow\; \frac{v_C^{n+1} - v_C^n}{h} = \frac{i_C^{n+1}}{C} \;\rightarrow\; i_C^{n+1} = \frac{C}{h}\left[(V_1^{n+1} - V_2^{n+1}) - (V_1^n - V_2^n)\right], \tag{6.42}$$

Finally, substituting for i_C^{n+1} in Eqs. 6.39 and 6.40, we get

$$i_s^{n+1} + \frac{C}{h}V_1^{n+1} - \frac{C}{h}V_2^{n+1} - \frac{C}{h}(V_1^n - V_2^n) = 0, \tag{6.43}$$

$$-\frac{C}{h}V_1^{n+1} + \frac{C}{h}V_2^{n+1} - I_s\left(e^{-V_2^{n+1}/V_T} - 1\right) + GV_2^{n+1} + \frac{C}{h}(V_1^n - V_2^n) = 0, \tag{6.44}$$

$$V_1^{n+1} - V_s(t_{n+1}) = 0. \tag{6.45}$$

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[10], but it simply cannot be helped. 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.

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.

[10]Some weavers get around the nonlinearities by approximating nonlinear functions with piecewise linear functions, thus avoiding an NR loop in each time step. Such simulators can give a significant speed advantage, but they can handle only a small subset of circuits of general interest.

(b) "Algebraic loops" (see Chapter 4) 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, no matter what weavers of other packages tell us about their products.

6.6 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. 6.11. The basic idea is to allow only a certain maximum number $N_{\text{NR}}^{\max}$ 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 $N_{\text{NR}}^{\max}$ 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. 6.12 [6] 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 $h_{\max}$.

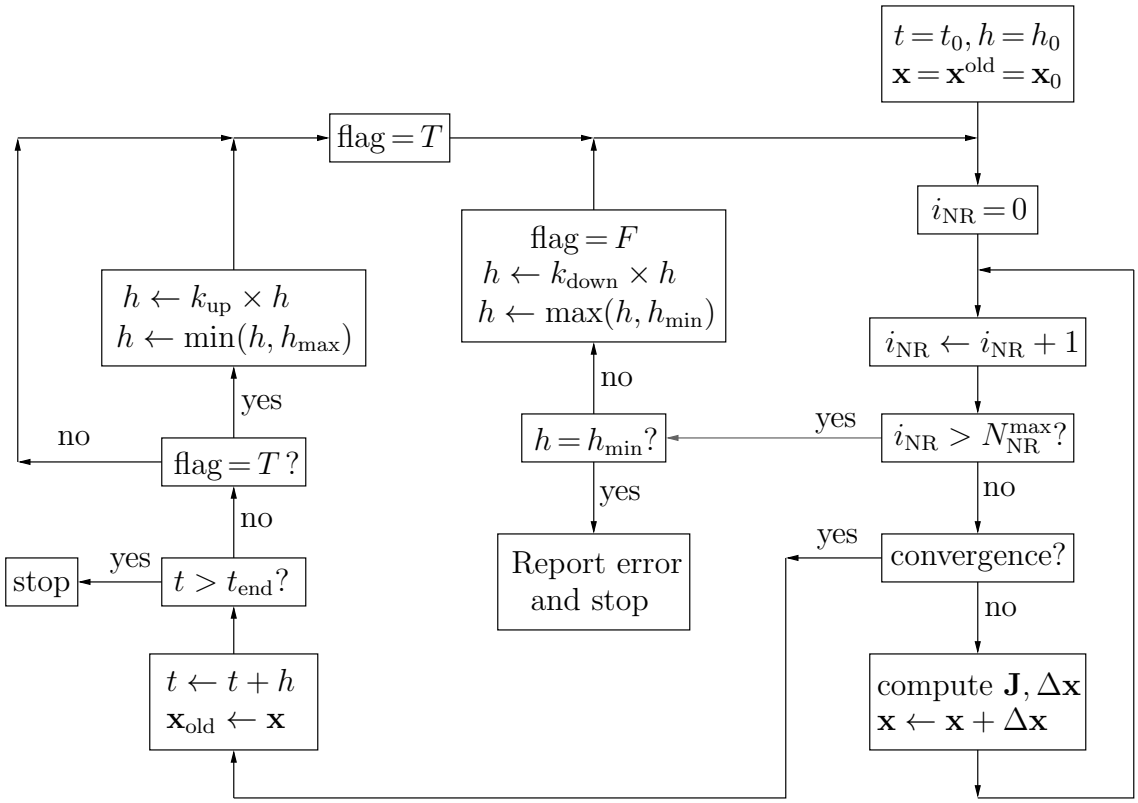


Figure 6.11: Flow chart for automatic time step adjustment using convergence of the NR process.

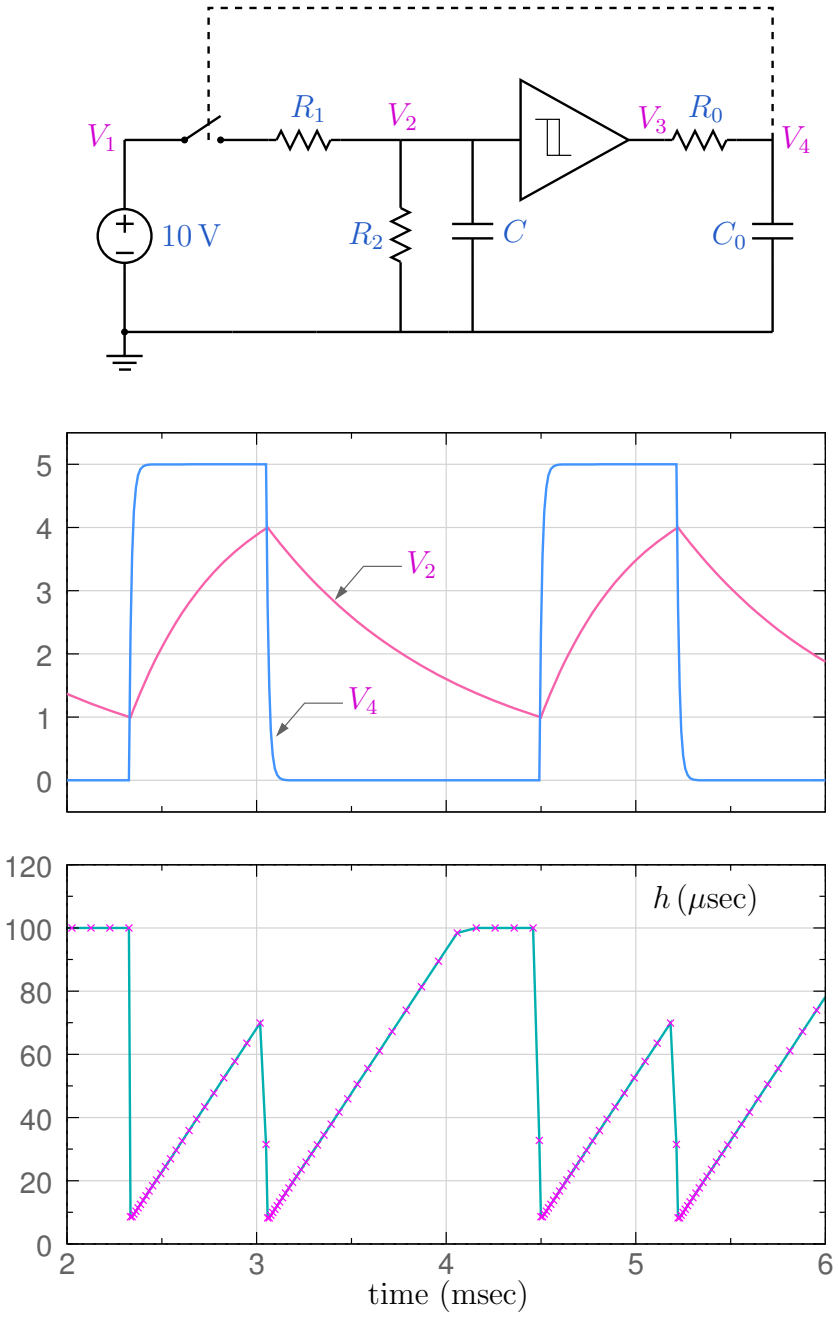


Figure 6.12: Example of automatic time step adjustment using convergence of the NR process. Parameter values are $R_1 = R_2 = 1\,\text{k}\Omega$, $R_0 = 100\,\Omega$, $C = 1\,\mu\text{F}$, $C_0 = 0.1\,\mu\text{F}$, $V_{IL} = 1\,\text{V}$, $V_{IH} = 4\,\text{V}$, $V_{OL} = 0\,\text{V}$, $V_{OH} = 5\,\text{V}$, $h_{\min} = 10^{-9}\,\text{sec}$, $h_{\max} = 10^{-4}\,\text{sec}$, $k_{\text{up}} = 1.1$, $k_{\text{down}} = 0.8$, $N_{\text{NR}}^{\max} = 10$.

Chapter 7

Steady-State Waveform (SSW) Computation

Consider the boost converter circuit shown in Fig. 7.1 (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.

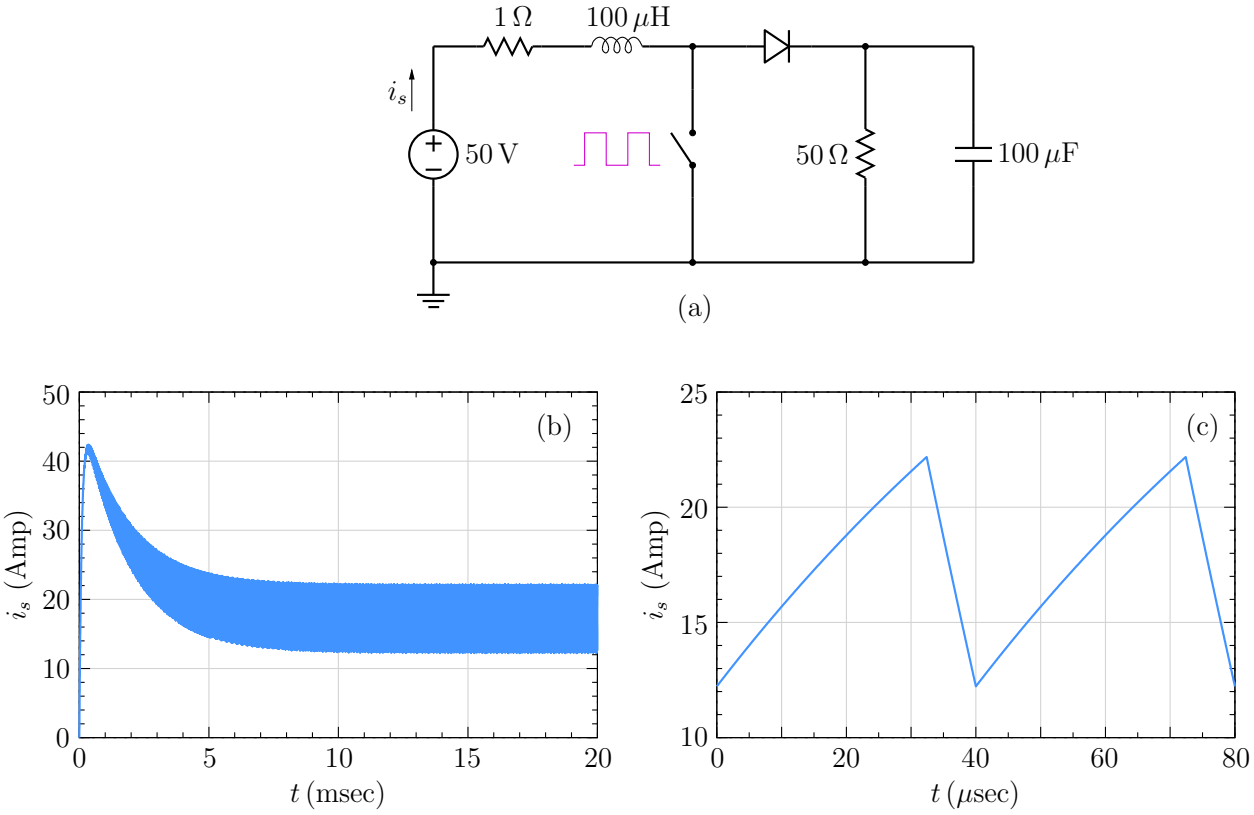


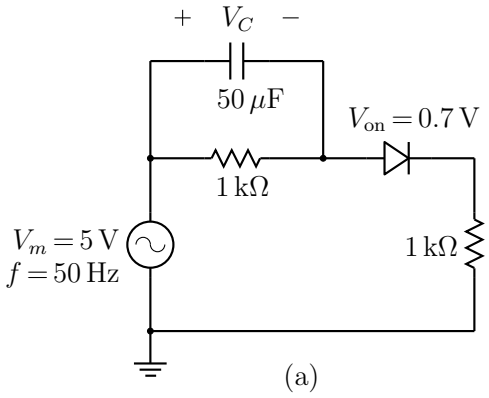
Figure 7.1: (a) Boost converter, (b) Source current obtained with transient simulation, (c) Steady-state source current.

Fig. 7.1 (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 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

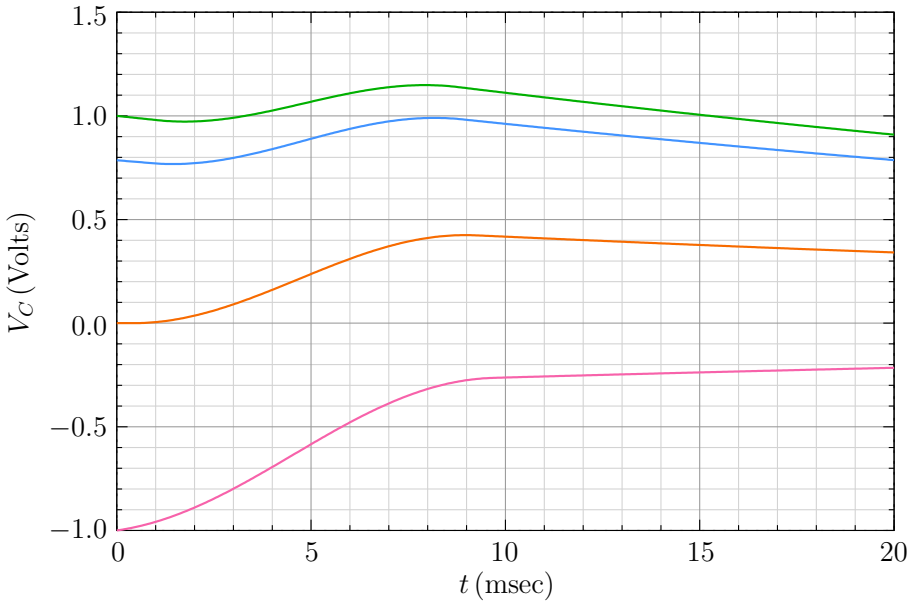
[1]This is also true about several other converter circuits.

comprising a time interval of $T = 40\,\mu$sec (see Fig. 7.1 (c)). To get to that one cycle in the steady state, we have ended up 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 7.2: (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 [12],[13])[2]. The basic idea behind SSW computation is illustrated in Fig. 7.2 with an example. In this circuit, the capacitor voltage V_C is the only state variable. If $V_C(t_0)$ is

[2]It is also possible to use a frequency-domain approach to compute the SSW solution (see [14], for example); we will describe only the time-domain approach here.

known, then the behaviour of the circuit for $t > t_0$ can be uniquely determined[3]. Fig. 7.2 (b) shows the results obtained with 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 [12] 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. 7.3 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.

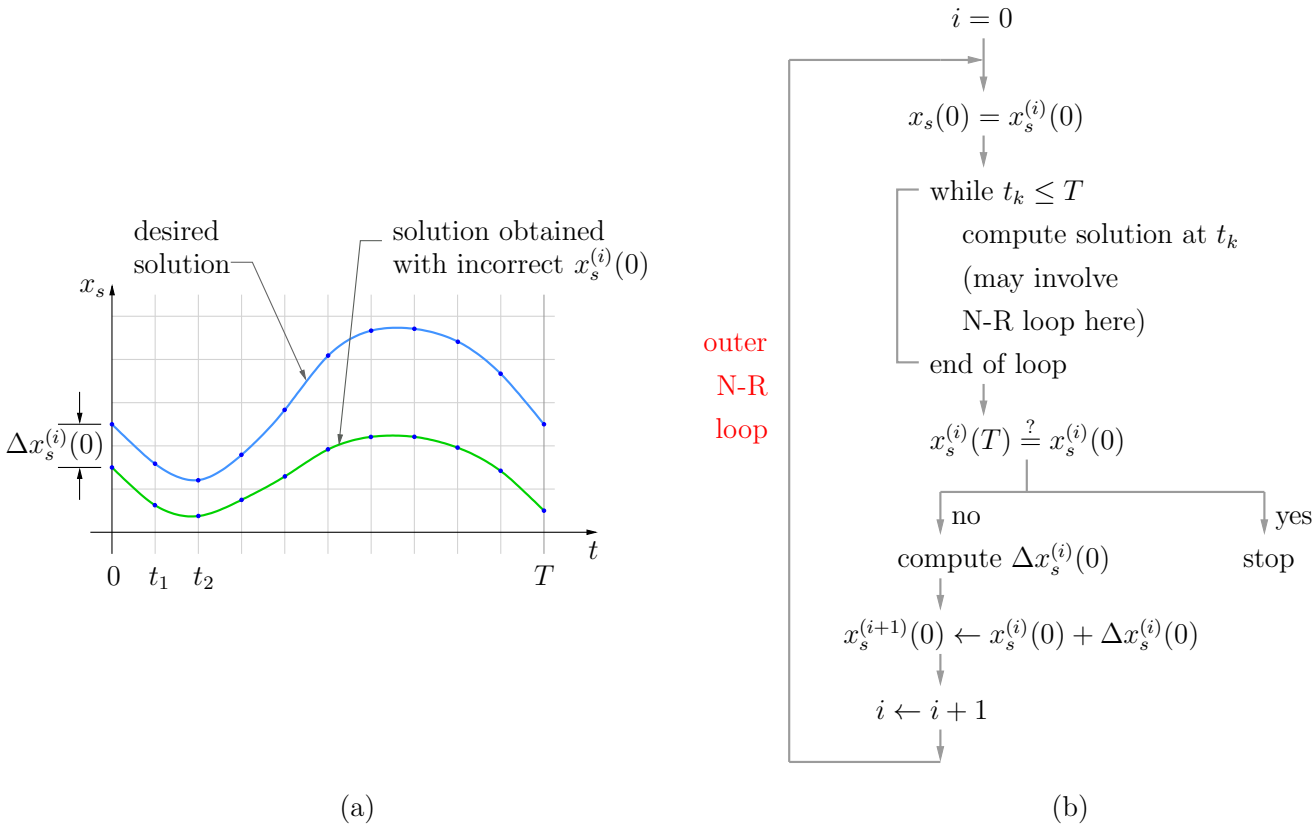


Figure 7.3: (a) Illustration of Newton-Raphson approach for SSW computation, (b) flow chart.

The SSW "outer loop" (see the flow chart in Fig. 7.3 (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

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

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

is computed for one period. This computation involves several time points, as shown in Fig. 7.3 (a). Furthermore, at each time point, there may be an *inner* NR loop if the system is nonlinear.

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 periodic steady-state 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 [12].

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. 7.3) 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 8

Start-up Simulation

One question we have not yet answered is: With what initial condition do we start when we perform transient simulation of a circuit? There are three options.

(a) Start with zero voltages and currents.

(b) Use a previously saved solution (DC or transient) as the starting point.

(c) Use the "start-up" solution obtained with some specified values of the state variables (capacitor voltages, inductor currents, etc.) as the starting point.

The first option is frequently used although it is somewhat artificial in the sense that it does not correspond to a valid solution for the circuit being simulated. The second option corresponds to a valid solution since it has been obtained by simulating the same circuit previously. It is the third option we want to discuss in this chapter, viz., the start-up solution.

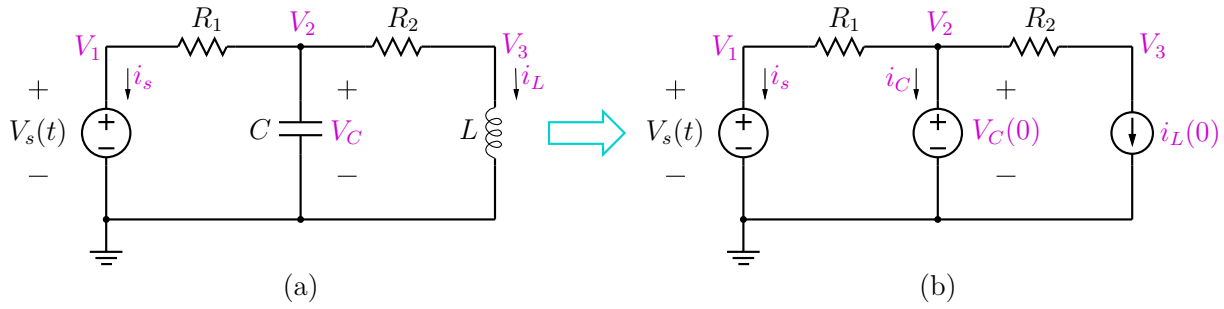


Figure 8.1: (a) An *RLC* circuit, (b) Same circuit under the start-up condition.

Let us illustrate the idea of a start-up solution with an example, the circuit shown in Fig. 8.1 (a). Suppose we are interested in performing transient simulation of the circuit from $t = 0$ to some known final time t_{end}. Suppose the initial values of the state variables, $V_C(0)$ and $i_L(0)$, are known. Corresponding to these initial values, there is a "start-up" solution (consisting of the node voltages and currents) which satisfies the circuit equations. It is this start-up solution that we want to obtain first, and then use it as a starting point for transient simulation.

With the conditions, $V_C = V_C(0)$ and $i_L = i_L(0)$, the circuit can be replaced by that shown in Fig. 8.1 (b). Since the voltage across the capacitor is known, we replace it with a DC

voltage source. Similarly, since the current through the inductor is known, we replace it with a DC current source. We can now use the MNA approach to assemble the circuit equations as

$$G_1(V_1 - V_2) + i_s = 0, \tag{8.1}$$

$$G_1(V_2 - V_1) + G_2(V_2 - V_3) + i_C = 0, \tag{8.2}$$

$$G_2(V_3 - V_2) = -i_L(0), \tag{8.3}$$

$$V_1 = V_s(0), \tag{8.4}$$

$$V_2 = V_C(0), \tag{8.5}$$

where $G_1 = 1/R_1$, $G_2 = 1/R_2$. Note that we have introduced the capacitor current as an additional system variable as required by the MNA formulation (since the capacitor has been replaced with a voltage source). By solving the above equations, we obtain the start-up solution[1]. If there are nonlinear elements in the circuit, the procedure remains the same except that the equations would need to be solved iteratively, using the Newton-Raphson method, for example.

It is not always possible to know in advance the initial values of the state variables except in simple cases. The situation may arise in closed-loop control of an induction motor, for example. We may be interested in how the speed varies in response to a step input in the reference speed from ω_1 to ω_2. However, we may not have the steady-state solution for ω_1 to begin with. What can we do in that case?

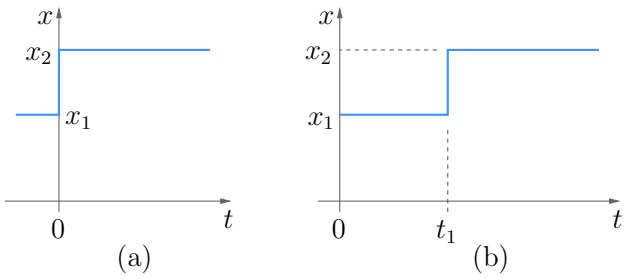


Figure 8.2: Two ways to simulate step response: (a) using start-up option, (b) using transient simulation with zero initial conditions.

Consider a circuit (or a general system) with a step input going from x_1 to x_2 at $t = 0$, as shown in Fig. 8.2 (a). If we know the values of the state variables at $t = 0$, we can use start-up simulation, obtain the complete solution at $t = 0$, and use it as a starting point for transient simulation. If not, we have no option but to first generate the solution for $x = x_1$ by performing transient simulation from $t = 0$ to $t = t_1$ (see Fig. 8.2 (b)) and then use it as the starting point for the step change. The time t_1 should be chosen to ensure that the circuit settles down to its steady-state solution before the step appears.

[1]Note that the start-up solution is different from the DC solution which is obtained by replacing capacitors with open circuits and inductors with short circuits.

Chapter 9

AC Simulation

We have seen so far how a circuit simulator handles DC and transient (time-domain) simulation. In this Chapter, we discuss how the response of a circuit in the sinusoidal steady state can be computed. Examples include[1] filters, power systems, amplifiers, etc.

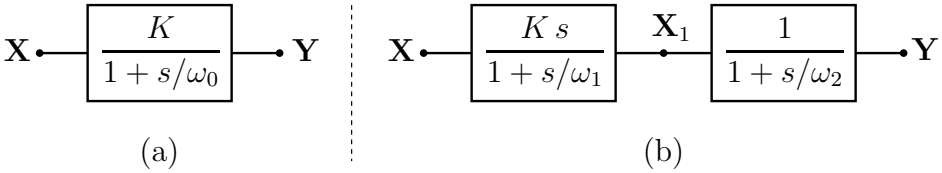


Figure 9.1: Filter examples.

Consider the low-pass filter shown in Fig. 9.1 (a). The variables $\mathbf{X}$ and $\mathbf{Y}$ are phasors[2] corresponding to the time-domain variables $x(t)$ and $y(t)$. Take the case where $x(t) = A\sin(\omega t + \phi)$, which corresponds to the phasor $\mathbf{X} = A\angle\phi = A\cos\phi + j\sin\phi$. The following equations can be written for this situation:

$$\mathbf{X} = A\cos\phi + j\sin\phi, \tag{9.1}$$

$$\frac{K}{1 + j\omega/\omega_0}\mathbf{X} - \mathbf{Y} = \mathbf{0}. \tag{9.2}$$

For a given frequency ω and known parameters (A, ϕ, K, ω_0), the above equations can be written as

$$\mathbf{A}_{11}\mathbf{X}_1 + \mathbf{A}_{12}\mathbf{X}_2 = \mathbf{B}_1, \tag{9.3}$$

$$\mathbf{A}_{21}\mathbf{X}_1 + \mathbf{A}_{22}\mathbf{X}_2 = \mathbf{B}_2, \tag{9.4}$$

a linear system with two equations in two variables $\mathbf{X}_1 \equiv \mathbf{X}$ and $\mathbf{X}_2 \equiv \mathbf{Y}$. This $\mathbf{Ax} = \mathbf{b}$ problem is no different than the one we have seen in DC circuits with linear elements (see Eq. 2.3, for example) except that the numbers are complex. Techniques such as Gaussian elimination and LU decomposition can be used to solve the above linear equations if suitable modifications are made to handle complex numbers.

[1] We will only consider circuits which can be represented with linear elements such as resistors, capacitors, inductors, independent sources, linear dependent sources, scalar multiplier, summer, etc.

[2] We will follow the commonly used convention of representing phasors (complex numbers) by bold letters.

As another example, consider the filter shown in Fig. 9.1 (b), with $\mathbf{X} = A\angle\phi$. The equations can now be written as

$$\mathbf{X} = A\cos\phi + j\sin\phi, \tag{9.5}$$

$$\frac{j\omega K}{1 + j\omega/\omega_1}\,\mathbf{X} - \mathbf{X}_1 = \mathbf{0}, \tag{9.6}$$

$$\frac{1}{1 + j\omega/\omega_2}\,\mathbf{X}_1 - \mathbf{Y} = \mathbf{0}, \tag{9.7}$$

once again a linear system. The above approach can be extended to any AC simulation problem which can be described by a "block diagram" containing several elements (transfer functions, sum, difference, scalar multiplication, etc.) interconnected in some manner. For each element, we write an equation(s) in terms of the variables associated with that element. When all elements are treated, we get a linear system whose solution yields the desired phasors.

The "frequency response" of a system refers to how a phasor(s) of interest changes with frequency. It can be computed by solving the system equations for several frequency values, one frequency at a time. Fig. 9.2 shows an example[3]. The gain in the figure refers to $|\mathbf{Y}(j\omega)/\mathbf{X}(j\omega)|$ and the phase to $\angle\mathbf{Y} - \angle\mathbf{X}$. If $\mathbf{X} = 1\angle 0$, the gain is the same as $\mathbf{Y}$, and the phase is the same as $\angle\mathbf{Y}$.

What about circuits? Let us consider the *RLC* circuit shown in Fig. 9.3 in the sinusoidal steady state. In the frequency domain, each component is represented by a phasor equation, e.g., $\mathbf{V} = R\,\mathbf{I}$ for a resistor, $\mathbf{V} = j\omega L\,\mathbf{I}$ for an inductor, $\mathbf{I} = j\omega C\,\mathbf{V}$ for a capacitor, $\mathbf{V} = V_m\angle\phi$ for an independent AC voltage source. Note that these are all linear equations. In addition, the voltage and current phasors in the circuit must satisfy the KCL and KVL constraints. It is easy to see that the AC circuit problem is similar to the DC circuit problem with linear components, and we can use the same systematic approach that we used in the DC case, viz., the MNA approach. For the circuit of Fig. 9.3, the MNA equations can be written as

$$G(\mathbf{V}_1 - \mathbf{V}_2) + \mathbf{I}_s = \mathbf{0}, \tag{9.8}$$

$$G(\mathbf{V}_2 - \mathbf{V}_1) + j\omega C\mathbf{V}_2 + \frac{\mathbf{V}_2}{j\omega L} = \mathbf{0}, \tag{9.9}$$

$$\mathbf{V}_1 = \mathbf{V}_s, \tag{9.10}$$

where $G = 1/R$. Solving the above linear system yields $\mathbf{V}_1$, $\mathbf{V}_2$, and $\mathbf{I}_s$. Subsequently, other variables of interest such as current through an element, voltage across an element, or average power absorbed by an element can be computed by post-processing.

When there are nonlinear elements in the circuit – as in the common-emitter (CE) amplifier shown in Fig. 9.4 (a) – we need to first *linearise* the semiconductor device behaviour using Taylor series, and represent it with an equivalent small-signal model. The meaning of "small" depends on the device; for a BJT, it means that the AC base-emitter voltage is small compared to the thermal voltage $V_T = kT/q$ which is about 26 mV at room temperature.

[3]Note that, although the gain and phase appear to be continuous functions of frequency in Fig. 9.2, the graphs actually contain a finite number of points joined with straight line segments.

Under this condition, the amplifier can be represented by the small-signal equivalent circuit[4] shown in Fig. 9.4 (b). Finally, the small-signal equivalent circuit is converted to the frequency domain (see Fig. 9.4 (c)), and the corresponding MNA equations are assembled and solved.

We see that the AC simulation problem and solution method are substantially different when nonlinear elements are present in the circuit:

(a) In a linear circuit such as the *RLC* circuit of Fig. 9.3, the time-domain voltages and currents are purely sinusoidal, and each of them has the form $K\sin(\omega t+\phi)$.

In a nonlinear circuit such as the amplifier of Fig. 9.4, a DC bias is required for the circuit to work satisfactorily. Each voltage and current would generally have a DC component and an AC component. The total voltage or current would be of the form $X^T = X^{\rm DC} + X_m \sin(\omega t+\phi)$.

(b) For a linear circuit, AC analysis can be performed directly, i.e., without any other preprocessing.

For a nonlinear circuit, the AC circuit parameters depend on the DC solution of the circuit. For the CE amplifier, for example, the parameter g_m depends on the DC collector current ($g_m = I_C/V_T$). DC analysis must be performed first, followed by computation of the AC circuit parameters, and then by assembly and solution of the MNA equations.

(c) In a linear circuit, there is no particular restriction on the amplitude of the AC input voltage.

In a nonlinear circuit such as the CE amplifier, the AC equivalent circuit is derived by assuming the small-signal approximation to be valid. If the AC input voltage is large, the small-signal approximation may get violated, and the AC simulation results would not make sense.

That said, we recall that the AC equivalent circuit is linear, and therefore an increase in the AC input simply causes the voltage and current phasors to increase proportionately. As a result, as far as *relative* numbers or *ratios* are concerned (such as the magnitude of a current or voltage phasor with respect to the input magnitude), the results remain valid even if a large input voltage is applied.

[4]Circuit simulators may not use the small-signal model of Fig. 9.4 (b) in that precise form. Instead, the current and charge derivatives (such as dI_C/dV_{BE}, dQ_C/dV_{BE}) – the first term in the corresponding Taylor expansion – may be computed and used in AC analysis.

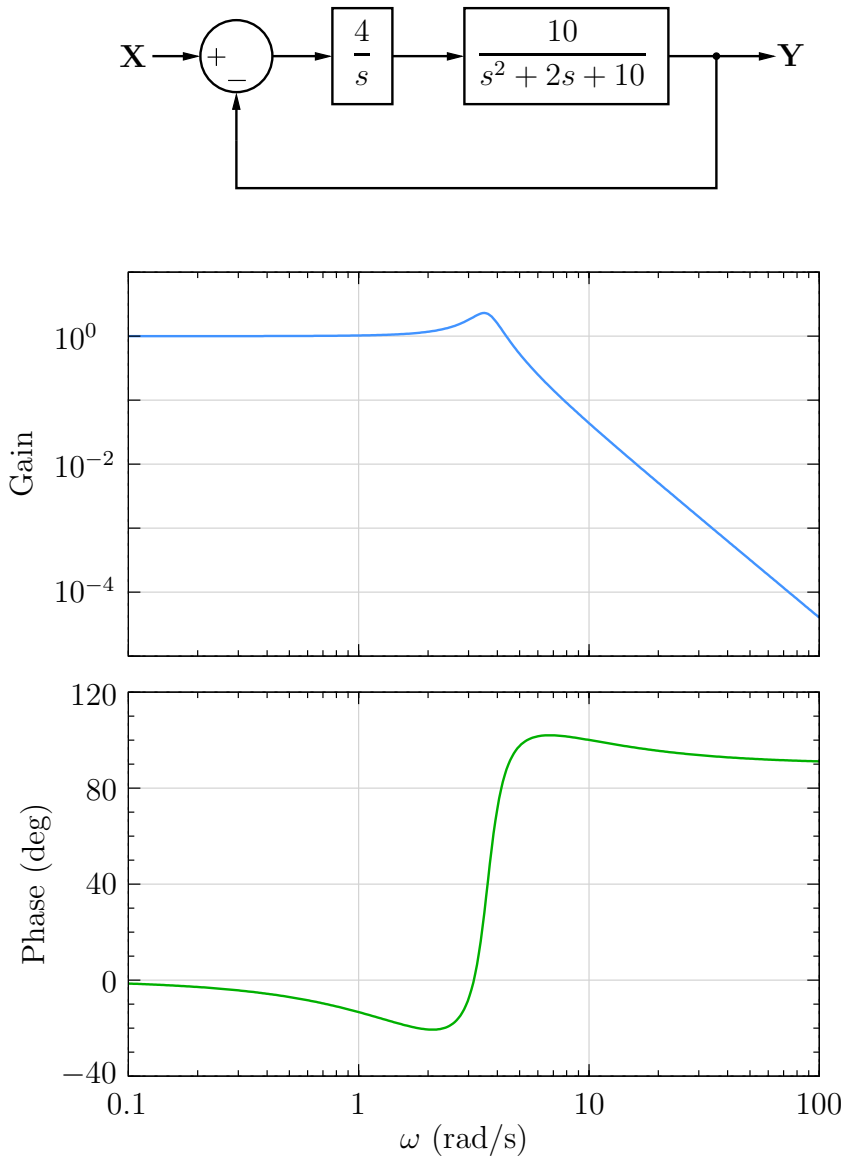


Figure 9.2: A closed-loop system and its frequency response.

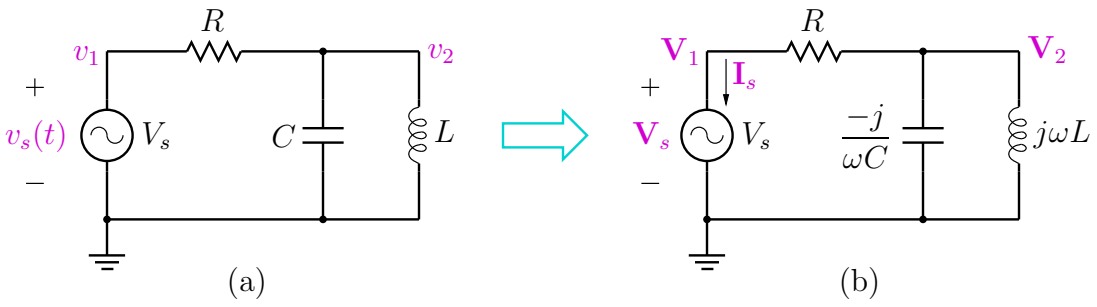


Figure 9.3: *RLC* circuit: (a) time domain, (b) frequency domain.

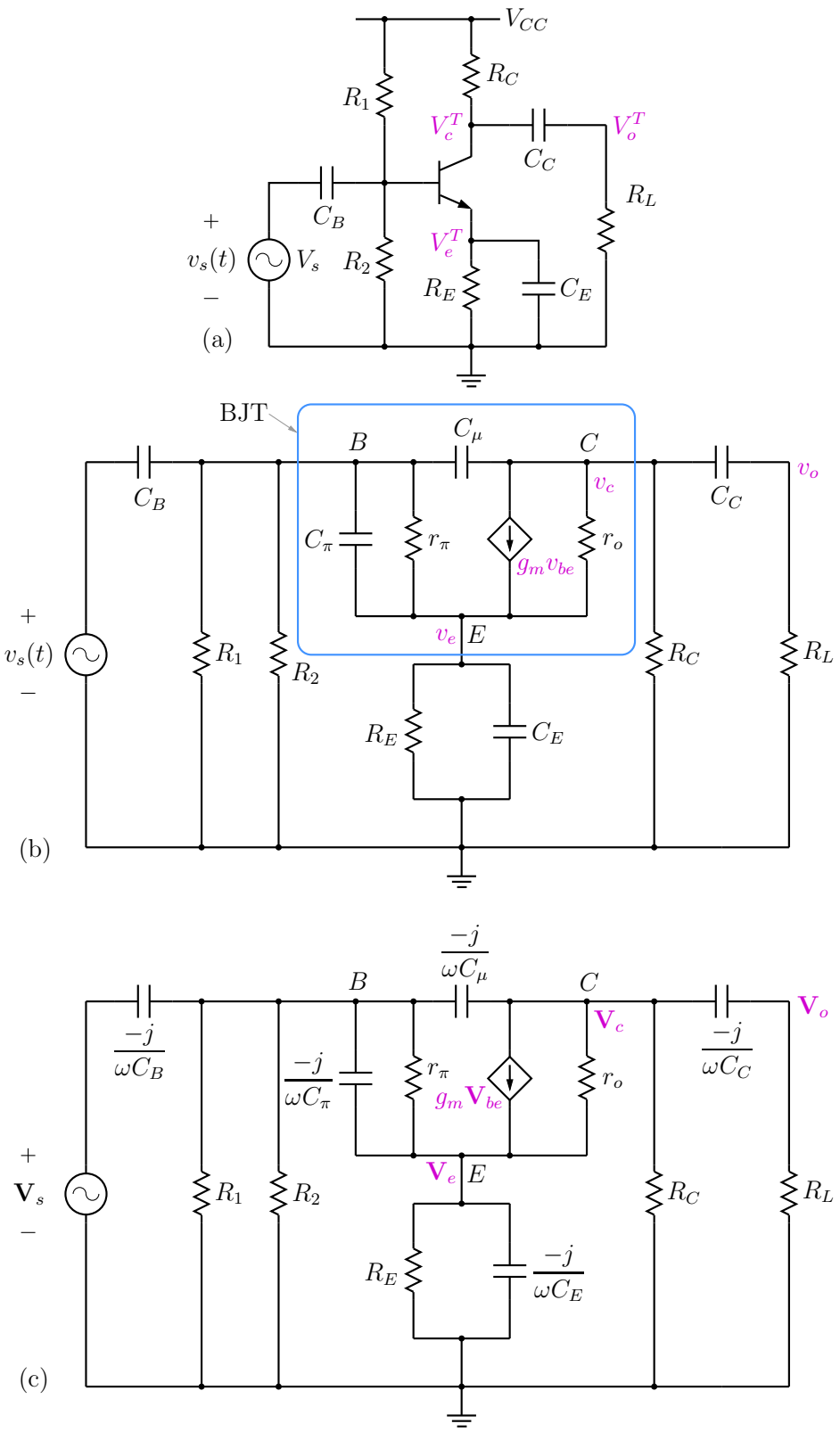


Figure 9.4: Common-emitter amplifier: (a) large-signal circuit, (b) small-signal equivalent circuit in time domain, (c) small-signal equivalent circuit in frequency domain.

Chapter 10

Digital Circuits

A digital circuit consists of digital (logical) elements such as gates, flip-flops, counters, adders, etc. Simulation techniques used for analog and digital circuits are fundamentally different due to the following factors.

(a) In an analog circuit, the variables are real-valued. In a digital circuit, the variables take only two values: `0` or `1`, `0` referring to a low voltage, and `1` to a high voltage.

(b) An analog circuit element (e.g., resistor, capacitor, BJT, voltage source, transformer) is described by an equation involving real numbers and possibly time derivatives. A digital circuit element (e.g., logic gates, flip-flops) is described by how the output(s) should change (to `0` or `1`) when certain conditions are met with respect to the current and past values of its input and output variables.

To simulate an analog circuit, we assemble the circuit equations using the MNA approach and solve them at each time point[1] (see Chapter 6). The solution process at a given time point involves a single matrix inversion (or equivalent) step for linear circuits, and several matrix inversion steps (one for each Newton-Raphson iteration) for nonlinear circuits. The outcome is a vector of real numbers representing the node voltages and voltage source currents.

On the other hand, for a digital circuit, we are not interested in the real (analog) value of a voltage; we are only interested in knowing whether it is high (`1`) or low (`0`) at a given time. The questions that arise are: (i) Which variables are expected to change? (ii) What is the change (from `0` to `1` or from `1` to `0`)? (iii) At what time will the change take place?

Let us see how these questions are tackled in "event-driven" simulation [15] which is the most effective technique for transient analysis of digital circuits. To illustrate the event-driven simulation approach, let us consider an inverter which has a delay of $\delta = 0.5$ nsec. Suppose its input has changed from `0` to `1` at $t = t_0$. This change ("event") triggers ("drives") a transition from `1` to `0` at the output. However, the output transition is delayed by δ and is therefore to be scheduled at $t_0 + \delta$. At that time, we "process" the scheduled change, i.e., we change the output from `1` to `0`. The change in the inverter output is a new event which may drive yet another change in the circuit, and so on. We keep repeating this process of scheduling and processing changes (events) until the end of the simulation interval t_{end} is reached.

[1] We will only consider transient simulation of analog circuits here, not DC or AC.

Fig. 10.1 shown a digital circuit with some given inputs X_1, X_2, X_3. Let us see how it can be handled using the event-driven simulation approach. We begin at $t = t_0$, assuming that the variables have already been suitably initialised. Subsequently, we proceed as follows.

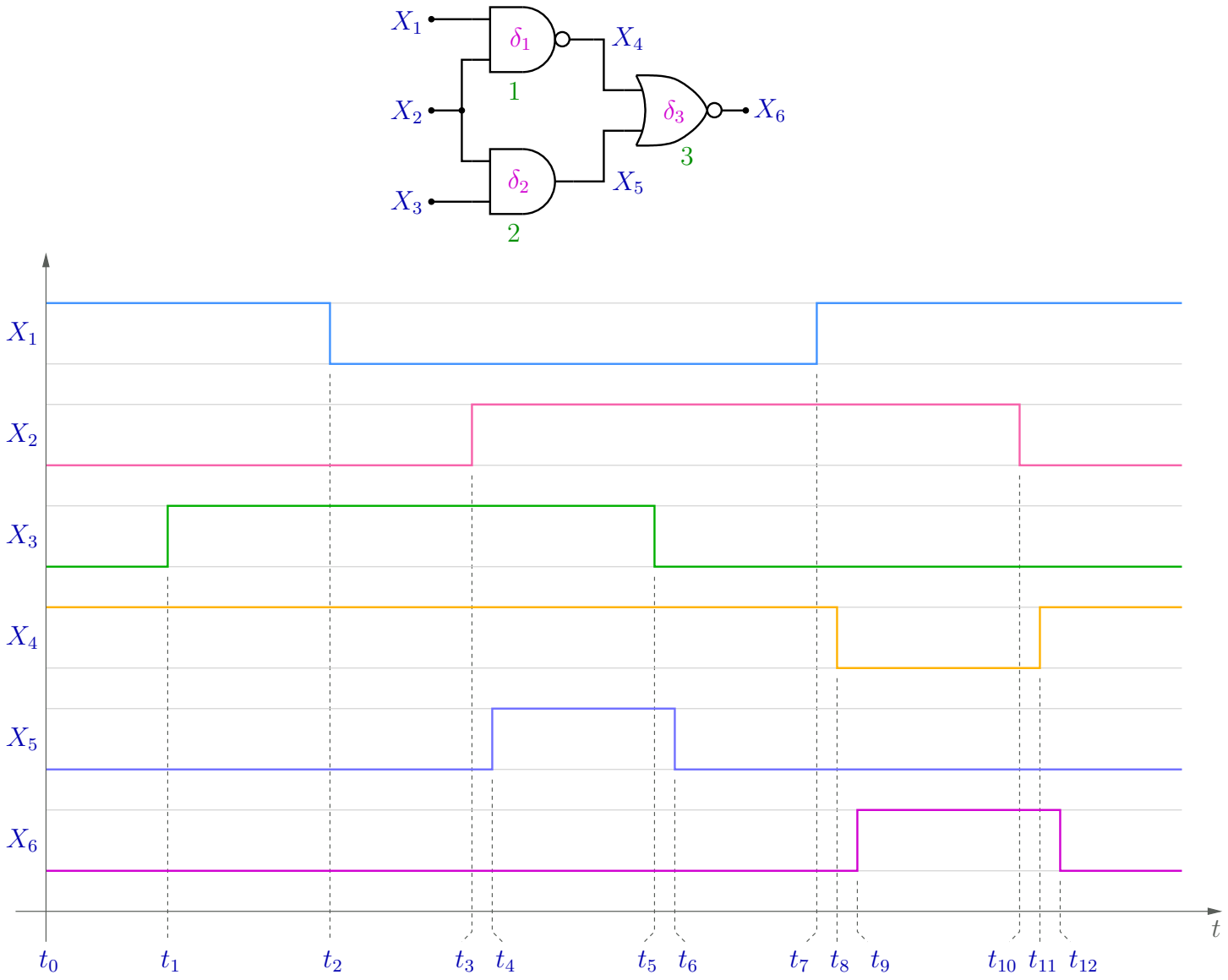


Figure 10.1: A digital circuit and associated waveforms.

(1) We scan the inputs and find that the first event is X_3 going from `0` to `1` at t_1. We ask: Will this event cause any change? To answer this question, we identify all elements in the circuit for which X_3 serves as an input – in this case, gate 2. We visit gate 2, look at its inputs, and find that its other input (X_2) is still `0`. We conclude that its output will continue to be `1`, and therefore no change is required to be scheduled.

(2) The next event is X_1 going from `1` to `0` at t_2. Following the process described above, we conclude that this event does not require any change to be scheduled.

(3) The next event is at t_3 when X_2 goes from `0` to `1`. We visit the gates for which X_2 serves as an input. For gate 1, we find that its other input is `0`, so its output will continue to be `1`, and no change is required. For gate 2, we find that both its inputs are `1`, and its output X_5 will therefore change from `0` to `1`. We must take into account the delay of gate 2 (δ_2) in scheduling this change, so the change is scheduled at $t_4 = t_3 + \delta_2$.

(4) At t_4, we *process* or *execute* the above change, viz., set X_5 to `1`.

(5) When we complete the above processing operation, we check if this event (i.e., X_5 going from `0` to `1`) will drive any other change, by visiting all elements for which X_5 is an input. There is only one such gate (gate 3), and since the other input of this gate (X_4) is already `1`, we figure that no change is required to be scheduled.

(6) The next event is at t_5, when X_3 goes from `1` to `0`. This event will cause X_5 to change from `1` to `0` at $t_6 = t_5 + \delta_2$, so we schedule that change.

(7) At t_6, we set X_5 to `0` as scheduled.

(8) By visiting gate 3 for which X_5 is an input, we conclude that the above change in X_5 does not drive any change at X_6.

(9) The next event is at t_7, when X_1 goes from `0` to `1`. This event will cause X_4 to change from `1` to `0` at $t_8 = t_7 + \delta_1$, so we schedule that change.

(10) At t_8, we set X_4 to `0` as scheduled.

(11) We find that the above event will drive a change in X_6 from `0` to `1`, so we schedule it at $t_9 = t_8 + \delta_3$.

(12) At t_9, we process the above change. Since X_6 is not an input for any gate, we do not need to check if a change in X_6 will cause any other change.

(13) The next event is at t_{10}, when X_2 goes from `1` to `0`. This event will not affect X_5, but it will cause X_4 to change from `0` to `1` at $t_{11} = t_{10} + \delta_1$, so we schedule that change.

(14) At t_{11}, we set X_4 to `1` as scheduled.

(15) We find that the above event will drive a change in X_6 from `1` to `0`, so we schedule it at $t_{12} = t_{11} + \delta_3$.

(16) At t_{12}, we process the above change. Since X_6 is not an input for any gate, no change needs to be scheduled because of the change at t_{12}.

When sequential elements (flip-flop, shift register, counter) are present in the circuit, the process of scheduling an event becomes more complex since it would involve the present input values as well as the past states of the sequential elements. However, the overall simulation scheme remains the same.

It is clear that simulation of digital circuits is completely different from that of analog circuits, and it is computationally much simpler, almost trivial. There are no equations to be *solved*, only values to be *assigned* (`0` or `1`). Thus, for the same complexity (say, the same

number of nodes), the CPU time required per time step would be significantly smaller for digital circuits.

There are circuits with both analog and digital elements, e.g., the 555-based oscillator circuit shown in Fig. 10.2. In these "mixed-signal" circuits, we can divide the elements into different categories:

(a) Purely analog elements (resistor, capacitor, DC voltage source in the figure).

(b) Purely digital elements (RS flip-flop in the figure)

(c) "ADC-type" elements in which the inputs are analog and outputs digital (comparators in the figure)

(d) "DAC-type" elements which behave like analog elements but also have digital inputs (controlled switch in the figure)

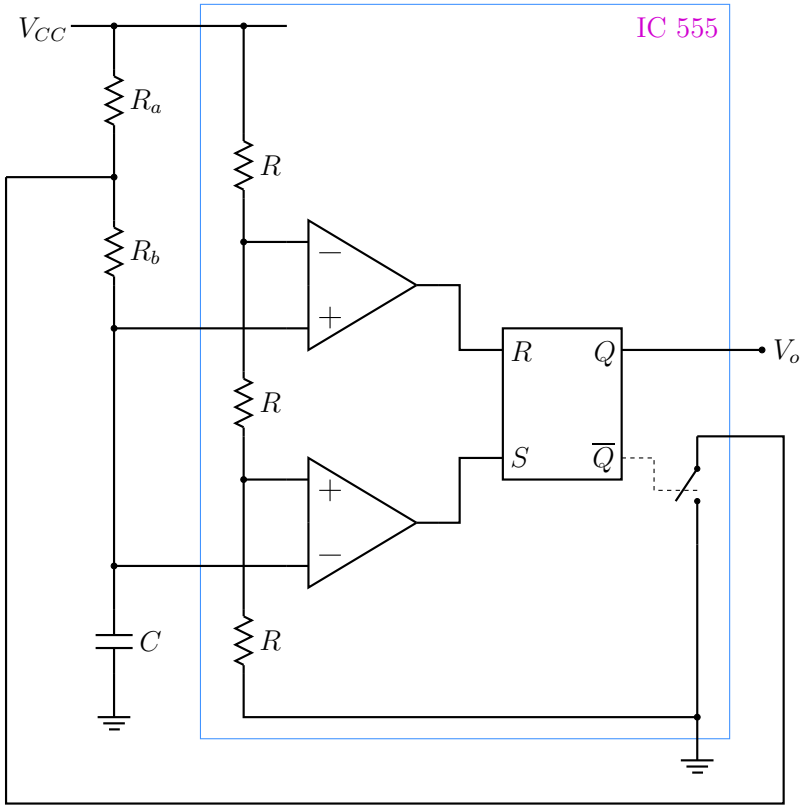


Figure 10.2: Oscillator based on the 555 timer.

In mixed-signal simulation, there are "analog time points" at which the analog variables are computed and "digital time points" at which the digital variables are treated (by scheduling and processing events). At the end of an analog time point, the analog variables would have changed; and if there are ADC-type elements in the circuit, some events may have to be scheduled as a result of the change in the analog variables. Similarly, at the end of a digital time point, the digital variables would have changed; and if there are DAC-type elements, the system of equations involving the analog variables must be solved to update the analog variables accordingly. Thus, the analog and digital variables are continuously upgraded, and the analog and digital time steps are automatically synchronised.

Chapter 11

SEQUEL library

The usefulness of a circuit simulator or ODE solver depends on the applications it can handle. Simulators are available for various applications, e.g., electronic circuits, power electronic circuits, automotive systems, power systems, "multi-physics" circuits (involving different domains simultaneously, such as electrical, thermal, mechanical), MEMS, digital circuits, etc. For a specific application, the developer must incorporate suitable elements inside the program. For example, a circuit simulator meant for electronic circuits must have diodes and transistors, apart from linear elements such as R, L, C, and sources.

SEQUEL is meant to be somewhat broad in scope, so it allows different types of elements. It is organised so that the main program and the element library are decoupled, making it easier to add new library elements[1]. The element help files can be accessed from the GUI, and they provide information about the nodes of an element, its parameters, and its behaviour. Here, we mainly want to look at how the element library is organised. Fig. 11.1 shows the library organisation. At the lowest level, we have the "basic" elements. Four types of basic elements are allowed, viz., electrical, general, digital, and explicit. Let us take a brief look at them.

(a) Electrical Basic Elements (EBE): These are elements with which voltages and currents are associated. When the circuit file contains EBEs, the program uses the MNA approach to assemble the circuit equations (see Chapter 2). Examples: R, L, C, sources, diode, BJT, FET, MOSFET, transformers, switches, induction motor, DC motor, thyristor, etc.

(b) General Basic Elements (GBE): These elements are used to implement relationships or equations between "general variables" (`gvar`). There is no KCL or KVL associated with these elements, only the relationships provided by the concerned GBE. Examples: summer, multiplier, integrator, transfer function, look-up tables, trigonometric functions, sources, clocks, etc.

(c) Digital Basic Elements (DBE): These elements are used to build a digital circuit. The event-driven simulation approach (see Chapter 10) is used to describe their behaviour. Their inputs and outputs are generally digital variables (except the "ADC"-type

[1]SEQUEL stands for "Solver of circuit EQuations with User-defined ELements." The original idea was to allow the user to add new elements, but for practical reasons, it was abandoned. Perhaps, it is time to change the name of the program.

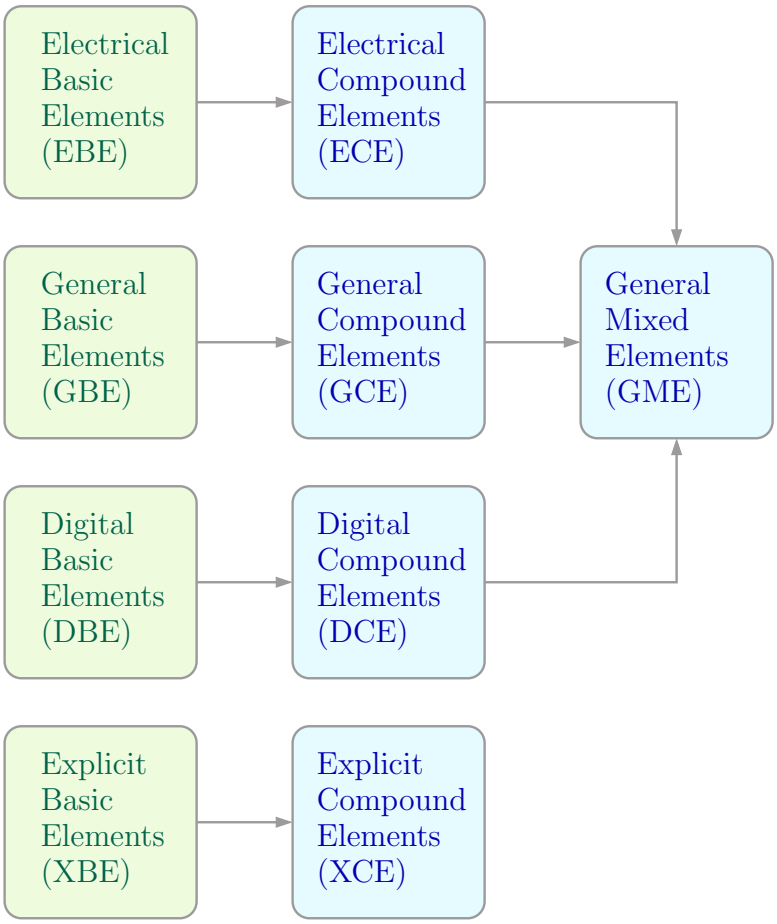


Figure 11.1: SEQUEL library organisation. Elements in the blue boxes can be used in the circuit file.

elements which have analog input nodes as well). Examples: gates, flip-flops, counters, multiplexer, demultiplexer, etc.

(d) Explicit Basic Elements (XBE): As explained in Chapter 4, explicit methods such as Forward Euler and Runge-Kutta-4 are advantageous for non-stiff problems. The XBEs provide the building blocks when an explicit method is used in transient simulation. The nodes of an XBE are of type `xvar` (explicit variable). Examples: summer, multiplier, integrator, transfer function, look-up tables, trigonometric functions, sources, induction motor, BLDC motor, clocks, etc. Note that several elements have been implemented as GBEs as well as XBEs. GBEs are used with implicit methods whereas XBEs are used with explicit methods.

At the next higher level, we have the "compound" elements. There are four types of compound elements: electrical, general, digital, and explicit. A compound element is made up of basic element of the same type. It may also include another compound element, again of the same type. Here are some examples.

(a) Electrical Compound Elements (ECE): An Op Amp macromodel can be constructed as an ECE and will possibly include BJTs, resistors, capacitors, and dependent sources as EBEs.

(b) General Compound Elements (GCE): A PI controller can be constructed as a GCE using a scalar multiplier, an integrator, and a summer as GBEs.

(c) Digital Compound Elements (DCE): A 4-bit counter can be constructed as a DCE using JK flip-flops as DBEs. An 8-bit counter can be constructed as a DCE using two 4-bit counters as DCEs.

(d) Explicit Compound Elements (XCE): A PI controller can be constructed as an XCE using a scalar multiplier, an integrator, and a summer as XBEs.

At the next higher level, we have the General Mixed Elements (GME) which allow compound elements of different types to be combined into a single element. The 555 timer (see Fig. 10.2) is a good example where we have ECEs (such as resistor, switch) and DCEs (RS flip-flop) combined into a single GME. Note that a GME cannot directly use the basic elements.

The user's circuit file –which is created by the SEQUEL GUI from the user's circuit schematic – can use elements of type ECE, GCE, DCE, XCE, and GME (the blue boxes in Fig. 11.1). At the circuit file level, we do not have direct access to the basic elements.

Bibliography

[1] http://www.ee.iitb.ac.in/~sequel.

[2] http://www.pitt.edu/~dash/type1620.html#andersen.

[3] http://jeanporter.cmswiki.wikispaces.net/The+Emperor%27s+New+Clothes+%28clipart+%26+images%29.

[4] D.O. Pederson, "A historical review of circuit simulation," *IEEE Trans. Circuits Syst.*, vol. 31, pp. 103–111, 1984.

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

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

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

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

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

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

[11] R.E. Bank, W.M. Coughran, W. Fichtner, E.H. Grosse, D.J. Rose, and R.K. Smith, "Transient simulation of silicon devices and circuits," *IEEE Trans. Electron Devices*, vol. 32, no. 10, pp. 1992–2006, 1992.

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

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

[14] 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.

[15] R. Raghuram, *Computer Simulation of Electronic Circuits.* New Delhi: Wiley Eastern, 1989.

SEQUEL Users' Manual: Part 2

Mahesh B. Patil

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

About Part 2 of the Manual

Simulating a circuit involves various steps: (i) creating the circuit schematic, (ii) assigning component values, (iii) choosing analysis type and assigning related parameters, (iv) running the simulator and viewing the results. SEQUEL provides a GUI for all of these steps. Video tutorials are available at the SEQUEL site to help a new user with the entire process in a step-by-step fashion.

The third task in the above list is achieved by defining "solve blocks." The purpose of Part 2 of the SEQUEL manual is to describe the syntax of the solve block statements. Before proceeding to the syntax rules, however, a few remarks are in order.

SEQUEL has been developed to simulate different kinds of circuits or systems: analog circuits, digital circuits, mixed-signal circuits, power electronic circuits including drives. If a simulator has to cater to a wide variety of circuits, it is difficult to set the algorithm/method parameters in a one-size-fits-all manner. For this reason, SEQUEL sets the method parameters to reasonable default values but allows the user to set them to suit his/her simulation need. Unfortunately, it calls for more work for the users since they need to know what each parameter means. The list of method parameters may seem intimidating at first, but it gets better after reading Part 1 of the SEQUEL manual. Several circuit files are provided with the SEQUEL distribution, and it is likely that the user will find an example similar to the circuit s/he wants to simulate. In such a case, the method parameters can be simply copied from that example, and chances are high that they will work without any changes.

There is a positive side to knowing a simulator in depth. Yes, it takes some effort, but it is a rewarding and enriching exercise. Apart from that, it would help the user in making better, more effective use of circuit simulators in general.

As Prof. V. Ramanarayanan of the Indian Institute of Science would put it, using a ready-made program is like eating canned food whereas writing your own program is like cooking your own meal, the way *you* would like it. Canned food has its advantages – it is quick and probably good enough for the average taste buds. Cooking yourself is more challenging – you need to start with chopping the ingredients and wait around when things get cooked or fried. But then you have complete control over the final product, and if all goes well, you have the satisfaction of doing a good job!

In the present context, understanding the functioning of a simulator and then using it is somewhere between eating canned food and preparing your own meal. With that brief introduction, let us get started.

In the following, we will group together solve block statements related to the same phenomenon or technique. That will make it easier to get an overall picture. Whenever applicable, we will cite equations or sections from Part 1 of the SEQUEL manual (as Sec. `xx` of Part-1, etc.) so that the reader can quickly make a connection between the solve block statements and the underlying method.

1 Newton-Raphson Method Parameters

The Newton-Raphson (NR) method, described in Chapter 3 of Part-1, is used in a variety of situations: DC, transient, SSW, and start-up (see Chapters 3, 6, 7, 8 in Part-1). The NR process is the same in all of these situations, and the NR method parameters are therefore common[1]. In the following, we describe these parameters.

* `itmax_newton`: (integer) maximum number of NR iterations (default: 500). Typically, the NR method converges in less than ten iterations, but the default value of `itmax_newton` has been made large to 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 `back_euler_auto` or `trapezoidal_auto` option is used, `itmax_newton` should be set to a much smaller number, say, 5.

* `dmp`: (`yes`/`no`) decides whether damping (see Eq. 3.19 in Part-1) should be used (default: `no`)

* `dmp_k`: (real number) damping factor k where $0 < k < 1$ (see Eq. 3.19 in Part-1, default: 0.2). Not relevant when `dmp` is set to `no`.

* `dmp_newt_max`: (integer) number of NR iterations for which damping is applied (default: 50). Not relevant when `dmp` is set to `no`.

* `chk_rhs2`: (`yes`/`no`) decides whether the 2-norm (see Eq. 3.9 in Part-1) should be used to check for convergence. If there are electrical elements in the system, `chk_rhs2` is set to `no` by default; otherwise, it is set to `yes`.

* `chk_only_rhs2`: (`yes`/`no`) Setting this flag to `yes` is equivalent to setting `chk_rhs2` to `yes` and all other covergence flags to `no`.

* `norm_2`: (real number) tolerance value for the 2-norm (default: 10^{-10}). Not relevant when `chk_rhs2` is `no`.

* `write_rhs2`: (`yes`/`no`) decides whether the 2-norm should be written to the console (for each NR iteration). default: `no`.

* `chk_delx_volt`: (`yes`/`no`) decides whether ΔV, the node voltage difference between successive NR iterations, should be used to check for convergence (see Sec. 3.3 in Part-1). Default: `no`.

* `delxmax_volt`: (real number) tolerance value for ΔV (default (in Volts): 10^{-4}). Not relevant when `chk_delx_volt` is `no`.

* `write_delx_volt`: (`yes`/`no`) decides whether $\Delta V^{\max}$ should be written to the console (for each NR iteration). default: `no`.

[1]There are some exceptions such as NR parameters for the outer loop in SSW analysis (see Fig. 7.3 in Part-1) and NR parameters used during $g_{\min}$ stepping (see Sec. 3.5.2 in Part-1); these parameters are described separately.

* `chk_only_delx_volt`: (`yes`/`no`) Setting this flag to `yes` is equivalent to setting `chk_delx_volt` to `yes` and all other covergence flags to `no`.

* `chk_spice`: (`yes`/`no`) decides whether the SPICE convergence criteria (see Sec. 3.3 in Part-1) should be used to check for convergence. If there are electrical elements in the circuit/system, `chk_spice` is set to `yes` by default; otherwise, it is set to `no`.

* `chk_only_spice`: (`yes`/`no`) Setting this flag to `yes` is equivalent to setting `chk_spice` to `yes` and all other covergence flags to `no`.

* `norm_spice_rel`: (real number) k_{rel} in Eq. 3.10 of Part-1 (default: 10^{-3}).

* `norm_spice_nodev`: (real number) τ_{abs} for node voltages (see Eq. 3.10 of Part-1, default in Volts: 10^{-6}).

* `norm_spice_cur`: (real number) τ_{abs} for currents (default in Amps: 10^{-12}).

* `norm_spice_eaux`: (real number) τ_{abs} for EBE auxiliary variables (default: 10^{-4}).

* `norm_spice_locvar`: (real number) τ_{abs} for EBE local variables (default: 10^{-4}).

* `norm_spice_gvar`: (real number) τ_{abs} for GBE main variables (default: 10^{-4}).

* `norm_spice_gaux`: (real number) τ_{abs} for GBE auxiliary variables (default: 10^{-4}).

  In general, it is difficult to set `norm_spice_eaux`, `norm_spice_locvar`, `norm_spice_gvar`, and `norm_spice_gaux` in a meaningful manner because they correspond to variables of different kinds. For example, an auxiliary variable in an EBE may be a voltage or a current or a charge. They are made available to the user mainly for the sake of completeness.

* `write_spice`: (`yes`/`no`) decides whether information about SPICE convergence parameters should be written to the console (for each NR iteration). Default: `no`.

As we have seen in Chapter 3 of Part-1, 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.

* `itmax_newton_first`: (integer) maximum number of NR iterations for the first solution (default: 500).

* `dmp_first`: (`yes`/`no`) decides whether damping should be used for the first solution (default: `no`)

* `dmp_k_first`: (real number) damping factor k ($0 < k < 1$) for the first solution (default: 0.1). Not relevant when `dmp_first` is set to `no`.

* `dmp_newt_max_first`: (integer) number of NR iterations for which damping is applied for the first solution (default: 50). Not relevant when `dmp_first` is set to `no`.

2 Parameters related to $g_{\min}$ stepping

When the Newton-Raphson (NR) method fails to converge, SEQUEL uses $g_{\min}$ stepping (see Sec. 3.5.2 in Part-1). $g_{\min}$ stepping is allowed only for DC and transient simulation types. The following parameters are related to $g_{\min}$ stepping.

* `gmin_step`: (`yes`/`no`) decides whether $g_{\min}$ stepping should be used (when normal NR convergence fails). By default, this parameter is set to `yes` if there are highly nonlinear elements (semiconductor devices such as diodes and transistors) in the circuit; otherwise, it is set to `no`.

* `gmin_init`: (`yes`/`no`) decides whether $g_{\min}$ stepping should be used to obtain the initial solution (default: `yes`). Not relevant if `gmin_step` is `no`.

* `gmin_start`: The starting value of $g_{\min}$ in $\mho$ (default: 0.1)

* `gmin_end`: The final value of $g_{\min}$ in $\mho$ (default: 10^{-12})

* `gmin_npoints`: (integer) number of gmin points (default: 50).

  During $g_{\min}$ stepping, successive values of $g_{\min}$ are related by $g_{\min}^{(n+1)} = k' \times g_{\min}^{(n)}$. `gmin_start`, `gmin_end`, and `gmin_npoints` are used to compute the ratio k'.

* `gmin_itmax_newton`: (integer) maximum number of NR iterations allowed during $g_{\min}$ stepping (default: 500).

* `gmin_dmp`: (`yes`/`no`) decides whether damping (see Eq. 3.19 in Part-1) should be used during $g_{\min}$ stepping (default: `no`)

* `gmin_dmp_k`: (real number) damping factor k ($0 < k < 1$) to be used during $g_{\min}$ stepping (see Eq. 3.19 in Part-1). Not relevant when `gmin_dmp` is set to `no`.

* `gmin_dmp_newt_max`: (integer) number of NR iterations for which damping is applied during $g_{\min}$ stepping (default: 50). Not relevant when `gmin_dmp` is set to `no`.

3 Parameters related to SSW analysis

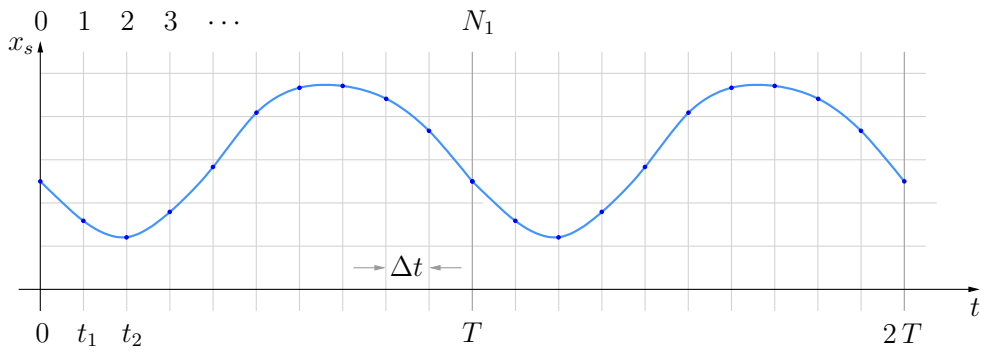


Figure 1: Illustration of parameters related to SSW analysis. A schematic plot of $x_s(t)$ is shown where x_s is a state variable.

SSW analysis is meant for computing the periodic steady-state solution (see Chapter 7 of Part-1). There are two sets of parameters related to SSW analysis: (a) waveform parameters, (b) NR parameters.

Waveform parameters:

* `ssw_period`: (real number) waveform period (T in the figure).
* `ssw_frequency`: (real number) waveform frequency ($1/T$).

  Either `ssw_period` or `ssw_frequency` must be specified.
* `ssw_ndiv`: (integer) number of divisions in one period (N_1 in the figure, default: 100). If `delt_const` (Δt in the figure) is specified, `ssw_ndiv` is ignored.
* `ssw_period_mult`: (integer) number of SSW periods to be simulated (default: 1). Only one period is really required to be simulated. However, the user may want to view a graph showing a few periods, and therefore this option is made available.

NR parameters: As shown in Fig. 7.3 in Part-1, the SSW state variable computation involves a Newton-Raphson (NR) loop (the *outer* NR loop in the figure). Parameters related to this loop are specific to SSW analysis and are described below[2].

* `ssw_itmax_newton`: (integer) maximum number of outer NR iterations allowed (default: 20).
* `ssw_dmp`: (`yes`/`no`) decides whether damping (see Eq. 3.19 in Part-1) should be used for the outer NR loop (default: `no`)
* `ssw_dmp_k`: (real number) damping factor k ($0 < k < 1$) for the outer NR loop (see Eq. 3.19 in Part-1, default: 0.8). Not relevant when `ssw_dmp` is set to `no`.

[2]If the circuit is nonlinear, there is *another* NR loop – the *inner* NR loop. Parameters for that loop (e.g., `itmax_newton`, `dmp`, etc.) are identical to those described elsewhere and are not repeated here.

* `ssw_dmp_newt_max`: (integer) number of NR iterations for which damping is to be applied in the outer NR loop (default: 10). Not relevant when `ssw_dmp` is set to `no`.

* `ssw_chk_rhs2`: (`yes`/`no`) specifies whether the 2-norm should be used to check for convergence of the outer NR iterations (default: `no` if there are electrical elements in the system being simulated; else, `yes`).

* `ssw_norm`: (real number) specifies the tolerance to check convergence of the outer NR iterations. It applies if `ssw_chk_rhs2` is `yes`, and in that case, the 2-norm, $\left[\frac{1}{N'}\sum_{i=1}^{N'} f_i\right]^{1/2}$, is compared with `ssw_norm` to test for convergence, where N' is the number of state variables.

* `ssw_chk_spice`: (`yes`/`no`) specifies whether the SPICE convergence criteria should be used to check for convergence of the outer NR iterations (default: `yes` if there are electrical elements in the system being simulated; else, `no`).

4 Statements related to frequency specification

An AC (frequency-domain) solution requires the frequency (frequencies) to be specified (see Chapter 9 of Part-1). SEQUEL allows the following options in that context.

* `set_frequency R`
  (`R`: real number)
  corresponds to the GUI statement `set frequency`. This statement is used to instruct the simulator to perform AC simulation at a single frequency `R`.

* `vary_freq from R1 to R2 type=linear/log n_points=I`
  (`R1,R2`: real numbers, `I`: integer)
  corresponds to the GUI statement `vary frequency`. This statement is used to instruct the simulator to perform AC simulation for a range of frequencies from `R1` to `R2`. The number of frequency points is specified by `I`.

  If the type is specified as `linear`, the frequency values are assumed to be distributed linearly between `R1` and `R2`.

  If the type is specified as `log`, the frequency values are assumed to be distributed logarithmically between `R1` and `R2`.

  In typical applications, `R1` and `R2` are orders of magnitude apart, and the `log` option is more appropriate.

* `vary_freq type=table R1 R2 R3 ...`
  (`R1,R2,R3,...`: real numbers)
  corresponds to the GUI statement `vary frequency`. This statement is used to instruct the simulator to perform AC simulation at frequencies given by `R1`, `R2`, `R3`,....

In an AC simulation solve block, either `set_freq` or `vary_freq` must be included.

5 Statements related to setting parameters

The SEQUEL GUI allows element parameters to be set with the "property editor." In addition, it is sometimes desirable to set a parameter of a given element inside a solve block, set it to another value in another solve block, and so on. The following statements may be used for that purpose.

* `set_parm S1_of_S2=I/R`
  (`S1,S2`: strings, `I`: integer, `R`: real number)
  corresponds to the GUI statement `set parameter`. This statement is used to set an integer or real parameter of an element. The parameter name is given by `S1`, and the element name by `S2`. The parameter value is specified by `I` or `R`.

* `set_parm S1_of_glbl=I/R`
  (`S1`: string, `I`: integer, `R`: real number)
  corresponds to the GUI statement `set global parameter`. This statement is used to set an integer or real global parameter `S1`. The parameter value is specified by `I` or `R`.

* `set_stparm S1_of_S2=R`
  (`S1,S2`: strings, `R`: real number)
  corresponds to the GUI statement `set startup parameter`. This statement is used to set a start-up parameter of an element (such as voltage across a capacitor or current through an inductor, see Chapter 8 of Part-1). The parameter name is given by `S1`, and the element name by `S2`. The parameter value is specified by `R`.

6 Statements related to varying parameters

The SEQUEL GUI allows element parameters to be set with the "property editor." In addition, it is sometimes desirable to vary a parameter of a given element from a starting value to an ending value inside a solve block. For example, in generating the *I-V* curve of a device, we would like to vary the voltage across the device and record the current. The following statements may be used to vary parameters.

* `vary_parm S1_of_S2 from R1 to R2 type=linear n_points=I1`
  (`S1,S2`: strings, `I1`: integer, `R1,R2`: real numbers)
  corresponds to the GUI statement `vary parameter (NPoints)`. This statement is used to vary a real parameter of an element from `R1` to `R2` in a linear fashion. The parameter name is given by `S1`, and the element name by `S2`. The number of parameter values is specified by `I1`.

  For example, `vary_parm vdc_of_Vs from 0 to 5 type=linear n_points=51`
  can be used to vary the parameter `vdc` of the element `Vs` from 0 to 5. Since `n_points` is specified as 51, the interval $(5-0)$ is divided into $(51-1)$ intervals (i.e., each interval equal to 0.1), and the parameter is varied as 0, 0.1, 0.2, $\cdots$, 4.9, 5.0.

* `vary_parm S1_of_S2 from R1 to R2 type=log n_points=I1`
  (`S1,S2`: strings, `I1`: integer, `R1,R2`: real numbers)
  corresponds to the GUI statement `vary parameter (NPoints)`. This statement is used to vary a real parameter of an element from `R1` to `R2` in a logarithmic fashion. The parameter name is given by `S1`, and the element name by `S2`. The number of parameter values is specified by `I1`.

* `vary_parm S1_of_S2 type=table R1 R2 R3 ...`
  (`S1,S2`: strings, `R1,R2,R3,...`: real numbers)
  corresponds to the GUI statement `vary parameter (NPoints)`. This statement is used to vary a real parameter of an element. The parameter values to be assigned are given by `R1`, `R2`, `R3`,.... The parameter name is given by `S1`, and the element name by `S2`. Note that `n_points` is not relevant in this case.

* `vary_parm S1_of_S2 from R1 to R2 type=linear div=R3`
  (`S1,S2`: strings, `R1,R2,R3`: real numbers)
  corresponds to the GUI statement `vary parameter (Div)`. This statement is used to vary a real parameter of an element from `R1` to `R2` in a linear fashion. The parameter name is given by `S1`, and the element name by `S2`. The interval between successive parameter values is specified by `R3`.

  For example, `vary_parm vdc_of_Vs from 0 to 5 type=linear div=0.2`
  can be used to vary the parameter `vdc` of the element `Vs` from 0 to 5. Since `div` is specified as 0.2, the parameter is varied as 0, 0.2, 0.4, $\cdots$, 4.8, 5.0.

Note: The `vary parameter` statement is not allowed in transient and SSW simulation.

7 Statements related to varying global parameters

The SEQUEL GUI allows global parameter values to be assigned with the "property editor." In addition, it is sometimes desirable to vary a global parameter from a starting value to an ending value inside a solve block. The following statements may be used for that purpose.

* `vary_parm S1_of_glbl from R1 to R2 type=linear n_points=I1`
  (`S1`: string, `I1`: integer, `R1,R2`: real numbers)
  corresponds to the GUI statement `vary parameter global`. This statement is used to vary a global real parameter from `R1` to `R2` in a linear fashion. The parameter name is given by `S1`. The number of parameter values is specified by `I1`.

* `vary_parm S1_of_glbl from R1 to R2 type=log n_points=I1`
  (`S1`: string, `I1`: integer, `R1,R2`: real numbers)
  corresponds to the GUI statement `vary parameter global`. This statement is used to vary a global real parameter from `R1` to `R2` in a logarithmic fashion. The parameter name is given by `S1`. The number of parameter values is specified by `I1`.

* `vary_parm S1_of_glbl type=table R1 R2 R3 ...`
  (`S1`: string, `R1,R2,R3,...`: real numbers)
  corresponds to the GUI statement `vary parameter global`. This statement is used to vary a global real parameter. The parameter values to be assigned are given by `R1`, `R2`, `R3`,.... The parameter name is given by `S1`. Note that `n_points` is not relevant in this case.

Note: The `vary parameter global` statement is not allowed in transient and SSW simulation.

8 Statements related to initial solution

SEQUEL allows the following options for generating the initial solution, which serves as the starting point for the Newton-Raphson process (in a nonlinear problem).

* `initial_sol initialize`
  When this option is selected, all variables are initialised to zero to generate the initial solution. This is the default option.

* `initial_sol previous`
  When this option is selected, the solution computed in the previous solve block is used as the initial solution.

* `initial_sol file filename=S1`
  (`S1`: string)
  When this option is selected, the initial solution is read from a file. The string `filename` specifies the name of the file, e.g., `x1.dat` (without quotation marks).

9 Output block

The output block is used to convey to the simulator several details such as which output files to generate, which variables to include in each file, etc. When the program executes successfully, the user-specified output files get created. To view the information contained in the output files (as a plot or a table), the SEQUEL GUI or any other plotting package can be used.

General attributes

* `FileName`: name of the output file (default: `output.dat`)
* `Output Variables`: output variables to be stored in the file.
* `LimitLines`: (integer) maximum number of lines to be stored. This is a "safety feature" to ensure that the user does not unknowingly generate very large files. If the number of lines generated by the program exceeds `LimitLines`, SEQUEL will produce an error message. If there is a genuine requirement of a large amount of data points, the user should increase `LimitLines` suitably. Default: 100,000.
* `Append`: (`yes`/`no`) decides whether the output data for the present output file should be appended to previously existing data (default: `no`).

  As an example, suppose that we have split a transient simulation from t_1 to t_3 to two intervals (i.e., two solve blocks): (a) t_1 to t_2 and (b) t_2 to t_3. We want a variable to be recorded for both these intervals in the *same* output file. In this case, we would make the output file names identical in the two solve blocks and set `Append` to `yes` in the second solve block.

Attributes related to transient simulation

* `FixedInterval`: (real number) if specified, the output variables are recorded at uniform intervals.

  For example, if `50u` is specified, the output variables are recorded every 50 μsec.
* `OutTStart`: (real number) if specified, the output of the output variables are recorded only for $t >$ `OutTStart`.
* `OutTEnd`: (real number) if specified, the output of the output variables are recorded only for $t <$ `OutTEnd`.

  if `OutTStart` and `OutTEnd` are not specified, the output variables are recorded for the entire simulation interval (from `t_start` to `t_end`).
* `Fourier`: (`yes`/`no`) decides whether Fourier components of the output variables should be computed and stored (default: `no`). If `Fourier` is specified as `yes`, the waveforms are assumed to be periodic with the period T computed as the difference `OutTEnd` − `OutTStart`.

When `Fourier` is specified as `yes`, the total harmonic distortion (THD) is also made available (in the `Solver Output` tab) for the variables listed in the output block. The following definition of THD is used.

$$\text{THD} = \frac{\sqrt{X_2^2 + X_3^2 + \cdots}}{X_1}, \tag{1}$$

where X_1, X_2, X_3 (etc.) are the f, $2f$, $3f$ components, respectively, of the concerned variable.

* `NFourier`: (integer) number of Fourier components to be computed.

Attributes related to AC simulation

* `MinPhase`: (real number) If `MinPhase` is specified, the phase data is written to the output file such that the phase angle is always larger than `MinPhase` (by adding suitable multiples of 360°).

* `MaxPhase`: (real number) If `MaxPhase` is specified, the phase data is written to the output file such that the phase angle is always smaller than `MaxPhase` (by adding suitable multiples of 360°).

  Note that either `MinPhase` or `MaxPhase` can be specified, but not both.

* `FreqHz`: (`yes`/`no`). If `yes`, the frequency values are written to the output file in Hz; else, in rad/s (default: `yes`).

10 `offs_dgtl`

Digital variables take on only two values: `0` and `1`. If several digital variables are plotted versus time in the same plot, it is difficult to view them (see Fig. 2, top plot). The parameter `offs_dgtl` (real number) is used to introduce an offset between successive variables (see Fig. 2, bottom plot).

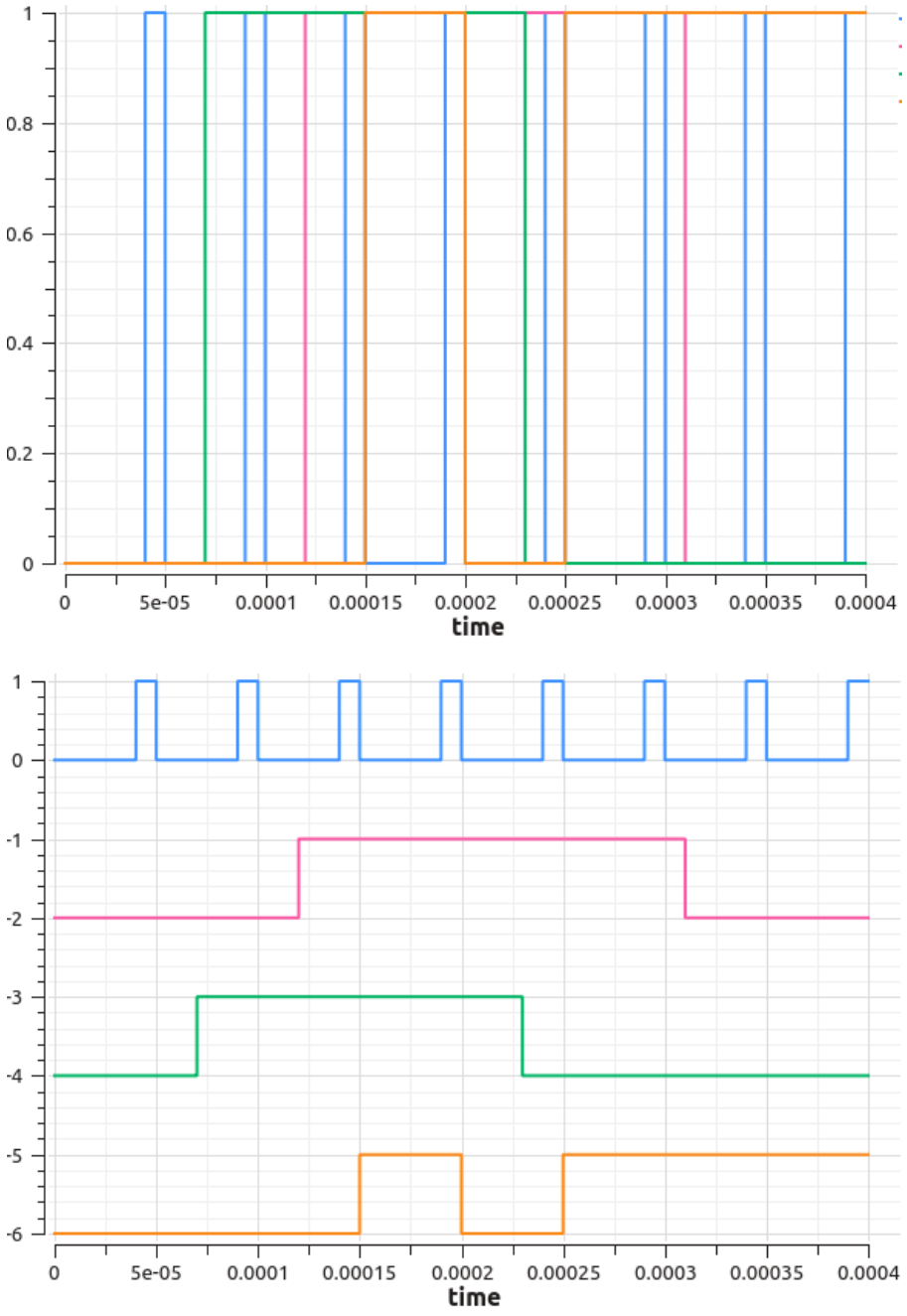


Figure 2: Example of digital output waveforms: `offs_dgtl` $= 0$ (top plot), `offs_dgtl` $= -2$ (bottom plot).

11 Parameters related to transient simulation

In the following, we describe parameters related to transient simulation (see Chapter 6 in Part-1). Some of the parameters would also apply to SSW computation since SSW involves transient simulation as well (see Fig. 7.3 in Part-1).

* `back_euler`: (`yes`/`no`) for Backward Euler method with constant time step (given by `delt_const`). In this case, a few smaller time steps may be taken to account for corners in input waveforms, for example.

* `back_euler_auto`: (`yes`/`no`) for Backward Euler method with NR-based adaptive time steps (see Sec. 6.6 in Part-1)

* `trapezoidal`: (`yes`/`no`) for trapezoidal method with constant time step (given by `delt_const`). In this case, a few smaller time steps may be taken to account for corners in input waveforms, for example.

* `trapezoidal_auto`: (`yes`/`no`) for trapezoidal method with NR-based adaptive time steps (see Sec. 6.6 in Part-1)

* `gear2`: (`yes`/`no`) for second-order Gear (BDF) method with constant time step (not available for SSW)

* `trbdf2`: (`yes`/`no`) for TR-BDF2 method (not available for SSW)

* `constant_step`: (`yes`/`no`) for forcing constant time steps. In this case, the step size is always equal to `delt_const`. If `constant_step` is selected, `back_euler` or `trapezoidal` must also be selected. (not available for SSW)

* `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: 100,000). This is a "safety feature." If the user by mistake creates conditions which calls for a large number of time points, SEQUEL will produce an error message. If there is a genuine requirement, the user should increase `itmax_trns` suitably.

* `delt_const`: (real number) serves as the constant time step for BE and TRZ methods, and as the first time step for methods with adaptive time steps.

* `delt_min`: (real number) smallest time step allowed (default: $0.0002 \times$ `delt_const`)

* `delt_max`: (real number) largest time step allowed (default: $10 \times$ `delt_const`)

* `fctr_stepred`: (real number) factor for reducing time step in adaptive time step methods (k_{down} in Sec. 6.6 in Part-1, default: 0.6)

* `fctr_stepinc`: (real number) factor for increasing time step in adaptive time step methods (k_{up} in Sec. 6.6 in Part-1, default: 1.5)

* `itmax_stepred`: (integer) maximum number of successive reductions in the time step at a given time point (default: 20)

* `trbdf2_tolr`: (real number) tolerance for TR-BDF2 method (default: 10^{-5})

* `itmax_trbdf2`: (integer) maximum number of successive reductions in the time step at a given time point when TR-BDF2 method is used (default: 20)

12 Parameters related to explicit methods

For transient simulation of a circuit with explicit compound elements (XCEs), SEQUEL employs explicit methods[3] (see Chapter 4 of Part-1). The following parameters apply in that case.

* `forward_euler`: (`yes`/`no`) for Forward Euler method with constant time step.
* `RK4`: (`yes`/`no`) for Runge-Kutta order-4 method with constant time step.
* `modified_euler`: (`yes`/`no`) for Modified (Improved) Euler method (see [1], for example) with constant time step.
* `Heun`: (`yes`/`no`) for Heun method (see [1], for example) with constant time step.

  When a constant step method (Forward Euler, RK4, Modified Euler, Heun) is used, a few smaller time steps may be taken to account for corners in input waveforms, for example.
* `RKF45`: (`yes`/`no`) for Runge-Kutta-Fehlberg 4/5 method (auto time step)
* `BS23`: (`yes`/`no`) for Bogacki-Shampine 2/3 method [2] (auto time step)
* `delt_const_x`: (real number) serves as the constant time step for Forward Euler, RK4, Modified Euler, and Heun methods, and as the first time step for methods with auto time steps.
* `delt_min_x`: (real number) smallest time step allowed (default: $0.0002 \times$ `delt_const_x`)
* `delt_max_x`: (real number) largest time step allowed (default: $10 \times$ `delt_const_x`)

Next, we list parameters related to the RKF45 and BS23 methods. These methods use auto time stepping. At each time point, a multiplier k is computed from the tolerance and an estimate of the local truncation error (see Sec. 4.4 of Part-1) to obtain the next time step as

$$\Delta t^{\text{new}} = k \times \Delta t^{\text{old}}. \tag{2}$$

The parameters are

* `rkf45_tolr`: (real number) tolerance value for the RKF45 method (default: 10^{-8})
* `rkf45_fctr_min`: (real number) lower limit on multiplier k (Eq. 2) in the RKF45 method. (default: 0.8)
* `rkf45_fctr_max`: (real number) upper limit on multiplier k (Eq. 2) in the RKF45 method. (default: 1.1)
* `bs23_tolr`: (real number) tolerance value for the BS23 method (default: 10^{-8})
* `bs23_fctr_min`: (real number) lower limit on multiplier k (Eq. 2) in the BS23 method. (default: 0.8)

[3]With explicit elements, only transient simulation is allowed; DC, start-up, AC, SSW are not allowed.

* `bs23_fctr_max`: (real number) upper limit on multiplier k (Eq. 2) in the BS23 method. (default: 1.1)

As we have seen in Sec. 4.6 of Part-1, algebraic loops create a difficulty for explicit methods. If the user's system has algebraic loops, the algebraic equations must be solved separately. As an example, consider the system shown in Fig. 3 which has two algebraic loops (the loops involving multipliers k_1 and k_2).

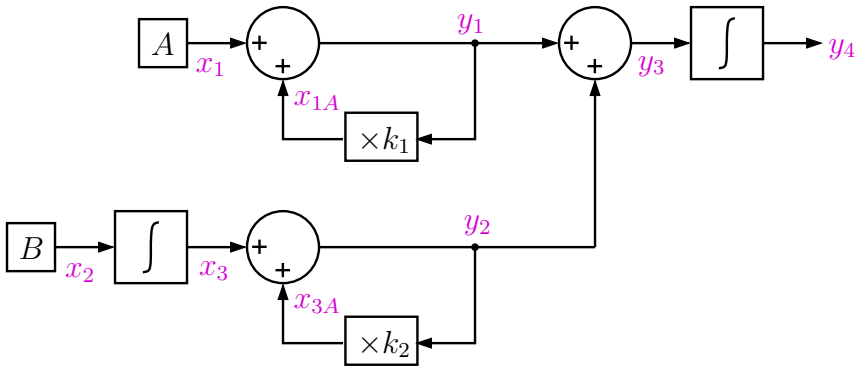


Figure 3: A system with algebraic loops. A and B are constants.

In such a case, SEQUEL would first update the variables associated with a time derivative, in this case, x_3 and y_4. For example, with the Forward Euler method, we have

$$y_4^{n+1} = y_4^n + hy_3^n, \tag{3}$$

$$x_3^{n+1} = x_3^n + hx_2^n, \tag{4}$$

where $h = t_{n+1} - t_n$ is the time step.

Having obtained y_4^{n+1} and x_3^{n+1}, the other variables are updated by solving the algebraic equations governing those variables. In the above example, all elements are linear, so it is a simple matter of solving a linear system of equations. If there are nonlinear elements in the system, the Newton-Raphson (NR) method (see Chapter 3 of Part-1) is used to solve the resulting equations.

Apart from algebraic loops, there is another situation in which a linear or nonlinear system of equations needs to be solved, and that is the presence of electrical-type elements which are implemented as XCEs. In that case, SEQUEL internally adds the required KCL and KVL equations.

The overall set of equations is divided into two sub-sets: (a) ODEs, (b) algebraic equations. An algebraic equation may arise from an algebraic loop, it may be a KCL/KVL equation, or it may be an element behaviour equation. The two sub-sets are treated separately – the ODEs with an explicit method (specified by the user) and the algebraic equations with a linear or nonlinear solver.

The following parameters are relevant in the above context.

* `x_eval_serial`: (`yes`/`no`) is used to indicate whether the elements should be evaluated in a serial fashion (see Sec. 4.6 in Part-1). It should be set to `no` when there are algebraic loops or electrical-type XCEs in the system.

Default: If there are electrical-type XCEs, `x_eval_serial` is set to `no` by default; else, it is set to `yes`.

* `x_itmax_newton`: (integer) maximum number of NR iterations (default: 500).

* `x_dmp`: (`yes`/`no`) decides whether damping (see Eq. 3.19 in Part-1) should be used (default: `no`)

* `x_dmp_k`: (real number) damping factor k where $0 < k < 1$ (see Eq. 3.19 in Part-1, default: 0.2). Not relevant when `x_dmp` is set to `no`.

* `x_dmp_newt_max`: (integer) number of NR iterations for which damping is applied (default: 50). Not relevant when `x_dmp` is set to `no`.

* `x_chk_rhs2`: (`yes`/`no`) decides whether 2-norm (see Eq. 3.9 in Part-1) should be used to check for convergence of NR iterations. (default: `yes`)

* `x_norm_2`: (real number) tolerance value for the 2-norm (default: 10^{-10}). Not relevant when `x_chk_rhs2` is `no`.

* `x_write_rhs2`: (`yes`/`no`) decides whether the 2-norm should be written to the console (for each NR iteration). Default: `no`.

* `x_chk_spice`: (`yes`/`no`) decides whether the SPICE convergence criteria (see Sec. 3.3 in Part-1) should be used to check for convergence of NR iterations. (default: `no`)

* `x_norm_spice_rel`: (real number) k_{rel} in Eq. 3.10 of Part-1 (default: 10^{-3}).

* `x_norm_spice_nodev`: (real number) τ_{abs} for node voltages (see Eq. 3.10 of Part-1, default in Volts: 10^{-6}).

* `x_norm_spice_cur`: (real number) τ_{abs} for currents (default in Amps: 10^{-12}).

* `x_norm_spice_xaux`: (real number) τ_{abs} for XBE auxiliary variables (default: 10^{-4}).

  In general, it is difficult to set `x_norm_spice_xaux` in a meaningful manner because it corresponds to variables of different kinds. For example, an auxiliary variable in an XBE may be a speed or force or current. It is made available to the user mainly for the sake of completeness.

* `x_write_spice`: (`yes`/`no`) decides whether information about SPICE convergence parameters should be written to the console (for each NR iteration). Default: `no`.

As we have seen in Chapter 3 of Part-1, 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.

* `x_itmax_newton_first`: (integer) maximum number of NR iterations for the first solution (default: 500).

* `x_dmp_first`: (`yes`/`no`) decides whether damping should be used for the first solution (default: `no`)

* `x_dmp_k_first`: (real number) damping factor k ($0 < k < 1$) for the first solution (default: 0.1). Not relevant when `x_dmp_first` is set to `no`.

* `x_dmp_newt_max_first`: (integer) number of NR iterations for which damping is applied for the first solution (default: 50). Not relevant when `x_dmp_first` is set to `no`.

References

1. M. B. Patil, V. Ramanarayanan, and V. T. Ranganathan, *Simulation of Power Electronic Circuits*, Narosa, New Delhi, 2009.

2. `https://en.wikipedia.org/wiki/Bogacki%E2%80%93Shampine_method`