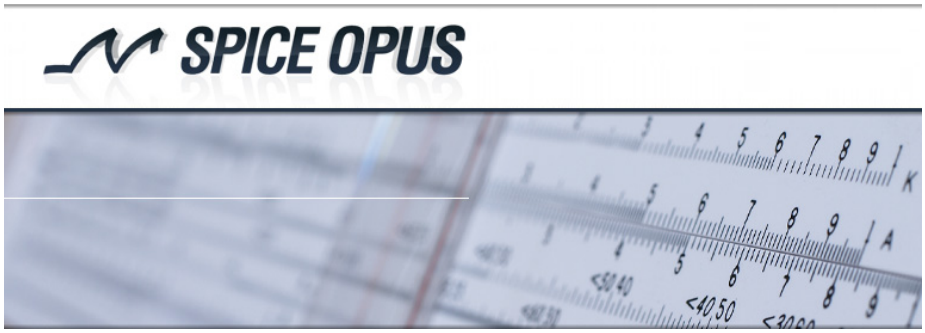

~~~

CIRCUIT ANALYSIS AND OPTIMISATION

Lecture Notes by Tadej Tuma, December 11, 2024
University of Ljubljana, Slovenia, Faculty of Electrical Engineering
~~~

Kataložni zapis o publikaciji (CIP) pripravili v Narodni in univerzitetni knjižnici v Ljubljani
COBISS.SI-ID 217853443
ISBN 978-961-243-474-8 (PDF)

URL: https://fides.fe.uni-lj.si/~tuma/Circuit_Analysis_and_Optimisation.pdf

Naslov: Circuit Analysis and Optimisation
Avtor: Tadej Tuma

Založnik: Založba FE, Ljubljana
Izdajatelj: Fakuleta za elektrotehniko, Ljubljana
Urednik: prof. dr. Sašo Tomažič
Kraj in leto izida: Ljubljana, 2024
1. elektronska izdaja

Contents

1 Introduction

This course attempts to explain the mechanisms of SPICE OPUS (`http://www.spiceopus.si/`) in formal terms. You can certainly use a circuit simulator without understanding its mathematical background, just as you can drive a car without knowing what is going on under the hood. This may sound like a good excuse to skip this course, in which case many phenomena will just have to go unexplained.

For instance, why does SPICE only compute nodal voltages and currents through independent voltage sources? Or why is it that sometimes a simple operating point analysis will last longer than a complex transient? What exactly is the meaning of all those cryptic warnings and error messages? Where is the source of convergence problems and how can one avoid them? How accurate and reliable are the obtained simulation results?

1.1 What is Electronic Design Automation (EDA)?

Electronic design automation, also referred to as electronic computer-aided design (ECAD), is a category of software tools for designing electronic systems such as integrated circuits and printed circuit boards. The tools work together in a design flow enabling engineers to device and analyze entire semiconductor chips. The central EDA component is always a circuit simulator like SPICE OPUS.

As a matter of fact, in the 1960s and 70s the IC industry was a rising star on the high-tech horizon. It quickly became obvious that designing ICs by trial and error was just too tedious and expensive. Prediction of circuit behavior using mathematical models of circuit components and digital computers seemed to be a far better approach. 1981 marks the beginning of EDA as an industry.

Today, semiconductor chips can have millions of components operating on a nanometer scale. One nanometer is 10^{-9} meters or a row of 10 hydrogen atoms. As a circuit designer, how are you going to test your circuit of millions of tiny transistors? No way of connecting a probe to some internal node and verify a voltage. All the testing must be done on EDA software level! Obviously the numerical results must be extremely reliable, which is difficult enough. Moreover, manufacturing processes can't ensure constant device parameters like capacitances, resistances, voltage gains, etc. On the contrary, all parameters deviate widely from their nominal values. EDA software must be capable of predicting the circuit behavior down to the last detail including manufacturing parameter deviations.

The modern chip fabrication process basically has three actors. Everything starts with the customer (investor) coming up with specifications for a new chip. Then a chip design team is contracted, which has qualified and highly experienced manpower as well as all the necessary EDA tools. The design house must also have a close working relationship with the foundry eventually producing the chip because the EDA software critically relies on accurate models of the fabrication process. The fabrication flow can be seen in figure 1.

Once the designers have all the input data they use EDA tools to come up with the chip's blueprint. In this step, the design team totally relies on software simulation. The final output is traditionally called tape-out, because it used to be delivered on magnetic tapes. At this point things become really

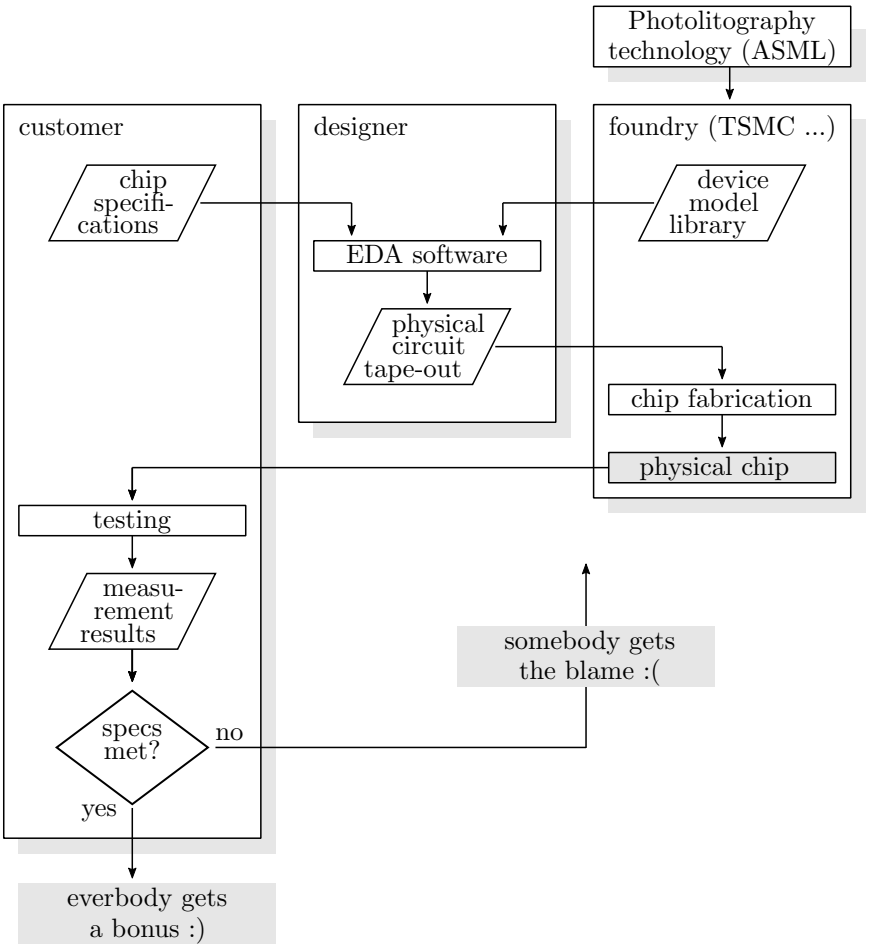

Figure 1: Chip fabrication flow.

expensive, since the tape-out data is given to the foundry which starts the physical implementation. The final chip is then cycled back to the customer who compares the performance against the initial specification. If everything adds up, mass production starts, otherwise the blaming game begins. This process is widely known as 'fabless manufacturing', because the physical production is outsourced to a highly specialized manufacturing plant or foundry. There are many chip design companies out there, but only a handful of fabs that can produce state of the art chips, the most famous being TSMC (Taiwan Semiconductor Manufacturing Company). Actually things become really weird when it comes to the production technology of these high tech fabs. Worldwide, there is only one company supplying cutting edge photolitography technology, namely the Dutch ASML holding!

Now let us take a closer look at the process of chip design. It basically consists of an electrical and a physical design cycle, as in figure 2. Obviously everything starts with the specifications of what the chip should actually do and the properties of the building blocks that a fab is offering. The designer's initial

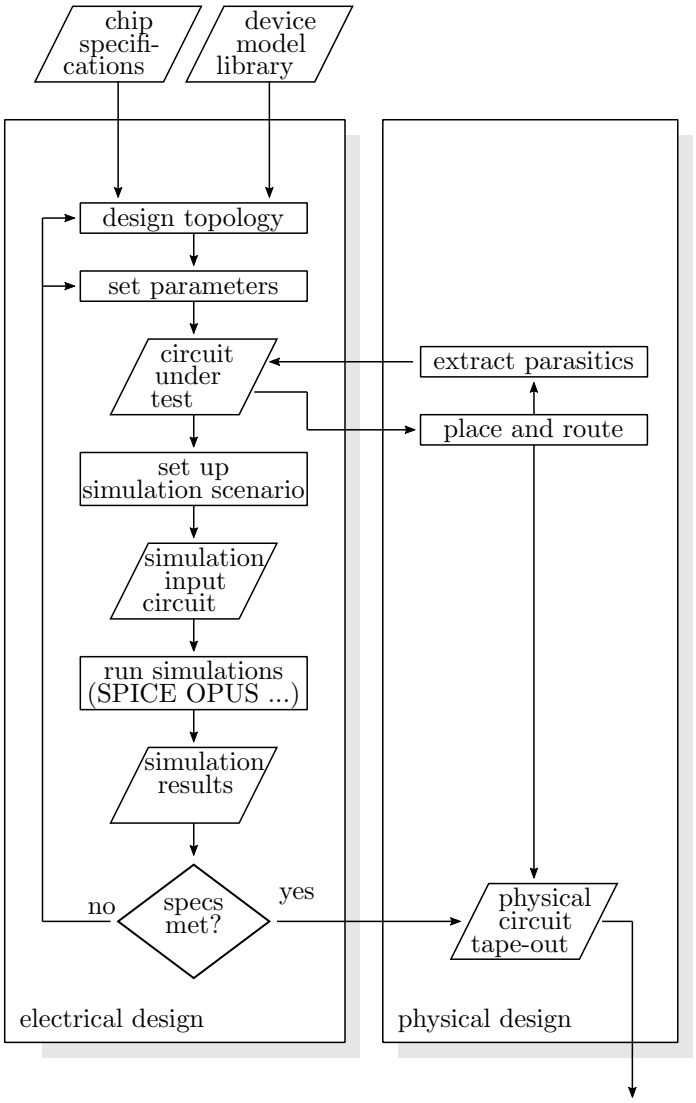

Figure 2: Chip design flow.

step is to come up with a suitable circuit topology and set respective device parameters. Next the circuit needs to be put into many different testbench configurations and a number of analyses have to be specified. Usually this step is called the simulation scenario and it includes the circuit as well as all testing specifications. Now the actual simulation is run on an analog circuit simulator (like SPICE OPUS). The results are compared against the given specifications and they most certainly won't match the first time around. Based on the type of mismatch the designer must now cycle back and adjust the circuit parameters and maybe even the topology. This loop is executed until the specs are roughly met.

Once the dimensions of the devices (transistors) are roughly known their physical placement on the wafer must be planned and they must be connected

according to the given circuit topology. This is done with highly specialized place and route tools. The result is the final manufacturing information (tape-out). Unfortunately the placement and the metallic connections introduce parasitic resistances, inductions and most notably capacitances to the circuit under test. These have to be added to the circuit and the electric design loop has to be run again. Obviously we have a bit of a chicken and egg situation. So both loops in figure 2 must be run alternately many times until the design finally meets the specs. Remember, so far all this is on a simulation level! Only now is the tape-out sent to the foundry for manufacturing and this is when expenses really start clocking.

After production the final product arrives at the customer (figure 1) for measurements. This is the moment of truth. Can the mass production begin or is another design cycle necessary? If the chip performance is insufficient something has gone wrong. Either the design team has inadequately set up the simulation scenario, or the device models from the foundry are not correct or the EDA simulation tools are off. In any case another manufacturing cycle is necessary, so the customer looses money and valuable time-to-market. Who is to blame?

1.2 The SPICE pedigree from UC Berkeley

The EDA tools are relatively easy to eliminate from the blaming game. There is a huge variety of circuit simulators out there, ranging from free academic tools to very expensive commercial ones. They all go back to a common ancestor, the SPICE project from UC Berkeley, which started in 1971 and ended 1993 with the publicly released source code SPICE 3F5. All contemporary circuit simulators – including SPICE OPUS – are based on this internal numerical apparatus. If used properly, most circuit simulators will give you surprisingly similar results. However close enough is not good enough if expensive redesign cycles in figure 1 are at stake.

Actually everything revolves around the extremely complex device model library at the top of figure 2. You see, the results are only as good as the process models supplied by the fab. So this is the holy grail. The fab must come up with a cutting edge manufacturing process and it must supply an accurate device library for a given EDA simulator. Up to a point the fab must guarantee that their devices will behave as predicted by the simulations. And there's the rub! The fab can't possibly check their model library on every analog circuit simulator out there. They usually just check two simulators, Spectre (Cadence Design Systems) and HSPICE (Synopsis), thereby making these two the golden standard! Should the design cycle in figure 1 be based on any simulator but Spectre or HSPICE the fab will promptly wash it's hands off any prototypes not matching the specs.

This puts a lot of technical and legal responsibility on Cadence and Synopsis resulting in very expensive simulators. A single annual license for Spectre is typically in the same price range as the annual salary of the senior designer using it. Many design companies keep their costs down by using free simulators like SPICE OPUS in early stages of the design, but no design company dares to tape-out anything that has not be thoroughly checked on a standard simulator.

Even if not the golden standard, SPICE OPUS is valuable in early stages of contemporary chip design as well as in eduction and research.

1.3 The Prerequisite Quiz

The lectures in this course are based on the assumption that you have a solid mathematical background and sufficient analog circuit knowledge. You should be able to answer the following questions.

1. What are the rules for matrix multiplications (additions)?
2. Can you explain the matrix determinant?
3. What are eigenvalues and eigenvectors?
4. How do you find the inverse of a matrix?
5. What is a matrix permutation?
6. How are systems of equation represented in matrix form?
7. Can you explain Gaussian elimination steps?
8. What is a singular matrix?
9. What does Kirchhoff's current (voltage) law state?
10. How is a circuit graph defined?
11. How are nodal equation formulated?
12. What are Bode's theorems?
13. How do you test circuits for stability?
14. What is a positive (negative) feedback?
15. What are the types of MOSFETs?
16. What are the basic MOSFET orientations?
17. What are the sources of noise?

1.4 A Short Refresher

This is a rapid rundown of basic matrix math. You might take a quick look, just to make sure you're up to speed. However, if your math needs brushing up ask Google for more details. So here it goes.

Matrix $\mathbf{A}_{n\times m}$ is a two dimensional array of arbitrary elements $a_{i,j}$ with n rows and m columns

$$\mathbf{A}_{n\times m} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}_{n\times m} .$$

Matrices are usually marked in bold font, while element positions are given as subscripts. For clarity reasons the comma in the subscript may be omitted. Where applicable, the subscripts as well as the matrix dimensions are dropped altogether.

Square matrices are special since $n = m$. For instance, the identity or unity matrix

$$\mathbf{I}_{4\times 4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{4\times 4} .$$

Then, there is the diagonal matrix with its strange sister, the anti-diagonal matrix

$$\mathbf{D} = \begin{bmatrix} d_{11} & 0 & 0 & 0 \\ 0 & d_{22} & 0 & 0 \\ 0 & 0 & d_{33} & 0 \\ 0 & 0 & 0 & d_{44} \end{bmatrix}, \qquad \mathbf{D}_a = \begin{bmatrix} 0 & 0 & 0 & d_{14} \\ 0 & 0 & d_{23} & 0 \\ 0 & d_{32} & 0 & 0 \\ d_{41} & 0 & 0 & 0 \end{bmatrix}.$$

Diagonal elements $a_{i,i}$ in general are also called pivots. Therefore, a diagonal matrix consists of nonzero pivots only. Next come the lower triangular and upper triangular matrices

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix}, \qquad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}.$$

As soon as matrix $\mathbf{A}_{n\times m}$ has only one row $n = 1$, it is called a row vector. Consequently, a matrix with $m = 1$ is called a column vector. Obviously, a matrix with only one row and one column is just another scalar $\mathbf{A}_{1\times 1} = [a_{11}] = a$. There are even some exotic matrices with at least one zero dimension $n = 0$ or $m = 0$. They are called empty matrices, not to be confused with zero matrices $\mathbf{A}_{n\times m} = \mathbf{0}$, having zero elements $a_{i,j} = 0$ rather than zero dimensions.

~~~
~~~

There is a lot of fun stuff you can do with matrices. For example, you can transpose a matrix by exchanging its rows and columns

$$\mathbf{A}^T_{n\times m} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix}^T_{n\times m} = \begin{bmatrix} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ a_{1,2} & a_{2,2} & \dots & a_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,m} & a_{2,m} & \dots & a_{n,m} \end{bmatrix}_{m\times n} .$$

As expected, by transposing the transposition, you get your original matrix back $(\mathbf{A}^T)^T = \mathbf{A}$. A symmetric matrix satisfies $\mathbf{A} = \mathbf{A}^T$.

How about adding two matrices? As long as they have the same dimensions, you just add individual elements $\mathbf{A}_{n\times m} + \mathbf{B}_{n\times m} = \mathbf{C}_{n\times m}$ where $c_{i,j} = a_{i,j} + b_{i,j}$. As usual, matrix additions are commutative, so $\mathbf{A}+\mathbf{B} = \mathbf{B}+\mathbf{A}$. By the way, the transposition of matrix additions is distributive, hence $\mathbf{A}^T + \mathbf{B}^T = (\mathbf{A} + \mathbf{B})^T$.

Matrix multiplication, on the other hand, is a bit more tricky. By definition, the number of the multiplier's columns must match the number of the multiplicand's rows. The product will inherit its row count from the multiplier and its column count from the multiplicand. Therefore, $\mathbf{A}_{n\times l} \cdot \mathbf{B}_{l\times m} = \mathbf{C}_{n\times m}$, where each element of $\mathbf{C}$ is the scalar product of a row vector from $\mathbf{A}$ and a corresponding column vector from $\mathbf{B}$, namely

$$c_{i,j} = \sum_{k=1}^{l} a_{i,k} b_{k,j}.$$

Matrix multiplication is obviously not a commutative operation $\mathbf{AB} \neq \mathbf{BA}$. In order to explicitly distinguish the multiplier from the multiplicand, you often hear that $\mathbf{B}$ is either pre-multiplied or post-multiplied by $\mathbf{A}$. A properly sized identity matrix is a neutral multiplier $\mathbf{IA} = \mathbf{A}$ as well as multiplicand $\mathbf{AI} = \mathbf{A}$. Matrix multiplication satisfies the rules of associativity $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ and distributivity $(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC}+\mathbf{BC}$. A square matrix $\mathbf{A}$ is called invertible or non-singular if there exists a matrix $\mathbf{B}$ such that $\mathbf{AB} = \mathbf{BA} = \mathbf{I}$, in which case $\mathbf{B} = \mathbf{A}^{-1}$. A matrix satisfying $\mathbf{A}^T = \mathbf{A}^{-1}$ is called orthogonal.

One more fun fact. Multiplying a matrix by a scalar $x \cdot \mathbf{A}$ means multiplying each element $x \cdot a_{i,j}$. On the other hand, adding a scalar to a matrix $x + \mathbf{A}$ is mathematically not defined.

~~~

Apart from additions and multiplications, there are three elementary matrix row and column operations. If you want to manipulate rows in $\mathbf{A}$, you pre-multiply it by a respective operator $\mathbf{F}$, mapping $\mathbf{F} \cdot \mathbf{A} = f_r(\mathbf{A})$. You can do the same to columns by post-multiplication $\mathbf{A} \cdot \mathbf{F} = f_c(\mathbf{A})$.

Multiplication, that is multiplying all elements of a row or column by a constant. In order to multiply row number 2 in matrix $\mathbf{A}_{4\times 2}$ by x, you pre-multiply it by a diagonal matrix $\mathbf{D}_{4\times 4}$ with $d_{2,2} = x$ and all other pivots equal to one

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & x & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ xa_{21} & xa_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} .$$
~~~

If you want to erase a row, you just set $x = 0$. This also works for columns. Suppose you want to multiply column 1 by y. Use a post-multiplication by a correctly sized diagonal matrix

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} y & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} ya_{11} & a_{12} \\ ya_{21} & a_{22} \\ ya_{31} & a_{32} \\ ya_{41} & a_{42} \end{bmatrix}.$$

Addition, that is adding one row or column to another. Let us say you want to add row number 4 to row number 2 in the same matrix as above. All you need is to pre-multiply it by an operator matrix $\mathbf{F}$, which is equal to a unit matrix with an additional 1 in column 4 of row 2, $f_{2,4} = 1$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} + a_{41} & a_{22} + a_{42} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix}.$$

The same goes for column additions, where you use $\mathbf{F}$ in post-multiplication.

Permutation, that is interchanging two rows or columns of a matrix. To exchange rows, the operator $\mathbf{F}$ must be a unit matrix with respectively interchanged rows. So in order to switch rows 3 and 4 in $\mathbf{A}_4$ you pre-multiply it by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{41} & a_{42} \\ a_{31} & a_{32} \end{bmatrix}.$$

Again, a post-multiplication will perform a respective column permutation.

~~~

If you think the above is just a boring theory, think again! By cleverly combining multiple column and row operators, a whole new world opens up.

Remember the Gaussian elimination algorithm? It goes somewhat like this: '... to eliminate one element below the pivot, multiply the pivot row by the element to be eliminated and divide it by the pivot and then subtract the pivot row from the one below, and then do this for all rows below the pivot row ...'. Just like a lawyer talking. With matrix operators all this becomes very elegant. Just one single pre-multiplication is needed to eliminate all elements below a_{11}

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{21}a_{11}^{-1} & 1 & 0 & 0 \\ -a_{31}a_{11}^{-1} & 0 & 1 & 0 \\ -a_{41}a_{11}^{-1} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ 0 & a_{22} - a_{21}a_{11}^{-1}a_{12} \\ 0 & a_{32} - a_{31}a_{11}^{-1}a_{12} \\ 0 & a_{42} - a_{41}a_{11}^{-1}a_{12} \end{bmatrix}.$$

There are countless examples of beautiful matrix transformations based operator pre- and post-multiplications, leading to efficient computer algorithms. However, this section is just a refresher, so we really have to stop here.
~~~

2 Linear Resistive Networks

We do have to start with some 101-course basics, but we will move very fast from there.

Let us observe an arbitrary network consisting of branches numbered $1, 2, \ldots, b$ with two terminals each, interconnected through $n+1$ nodes numbered $0, 1, \ldots, n$. We know that voltages cannot be considered absolute, so we need to name exactly one reference node, also referred to as the ground node. Let the ground node be the one numbered 0. So we have n nodal voltages across each respective node to the ground node $v_{n_1}, v_{n_2}, \ldots, v_{n_n}$. Each of the b branches is conducting a branch current $i_{b_1}, i_{b_2}, \ldots, i_{b_b}$, which is flowing from the respective positive branch terminal t_+ to the negative one t_-. We also formulate voltages across each branch $v_{b_1}, v_{b_2}, \ldots, v_{b_b}$, that is, from the respective positive branch terminal t_+ to the negative t_-.

We have barely started and we already have a lot of indices in our expressions, so let us simplify the notation by introducing vectors. We will refer to column vectors $\mathbf{v}_n$, $\mathbf{v}_b$, and $\mathbf{i}_b$ rather than their individual components,

$$\mathbf{v}_n = \begin{bmatrix} v_{n_1} \\ v_{n_2} \\ \vdots \\ v_{n_n} \end{bmatrix}, \quad \mathbf{v}_b = \begin{bmatrix} v_{b_1} \\ v_{b_2} \\ \vdots \\ v_{b_b} \end{bmatrix}, \quad \mathbf{i}_b = \begin{bmatrix} i_{b_1} \\ i_{b_2} \\ \vdots \\ i_{b_b} \end{bmatrix}. \tag{1}$$

Next, we need to determine the node numbers of the positive and negative terminals for each branch. The configuration of the branches is also called the circuit topology and is usually given as a full incidence matrix $\mathbf{A}_f$ with $n+1$ rows corresponding to nodes and b columns corresponding to branches,

$$\mathbf{A}_f = \begin{bmatrix} a_{0,1} & a_{0,2} & \ldots & a_{0,b} \\ a_{1,1} & a_{1,2} & \ldots & a_{1,b} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,b} \end{bmatrix}. \tag{2}$$

The positive terminal of each branch is marked by $+1$ in the respective row, while the negative terminal is denoted by -1. All other elements in $\mathbf{A}_f$ have zero values. So branch number k in figure 3 with its positive terminal at node p and its negative terminal at node m would have zero elements in the k-th column of $\mathbf{A}_f$ except for $a_{p,k} = +1$ and $a_{m,k} = -1$. Each column in $\mathbf{A}_f$ thus has exactly two nonzero elements and each row has one nonzero element for each branch connected to it.

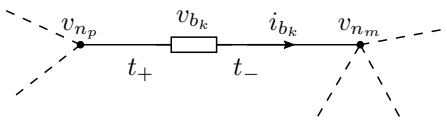

Figure 3: Arbitrary branch in the network.

Note that, by definition, a positive branch current i_{b_k} is flowing from t_+ to t_- through the branch regardless of the branch type. This seems a bit unnatural when the branch contains an independent voltage source – we expect supply

voltage sources to drive supply current through the network, rather than the other way around. This is why you normally see negative supply currents in SPICE OPUS output listings.

Notation	Dimension	Description
$n+1$	–	number of nodes + one reference node (ground)
b	–	number of branches
$\mathbf{v}_n$	$n \times 1$	column vector of nodal voltages
$\mathbf{v}_b$	$b \times 1$	column vector of branch voltages
$\mathbf{i}_b$	$b \times 1$	column vector of branch currents
$\mathbf{A}_f$	$n+1 \times b$	full incidence matrix of the network
$\mathbf{A}$	$n \times b$	reduced incidence matrix of the network

Table 1: The vectors and matrices used in the mathematical description of the circuit topology.

Besides the full incidence matrix $\mathbf{A}_f$, we also need the reduced incidence matrix $\mathbf{A}$, which actually is $\mathbf{A}_f$ minus the first row, describing the connections to the ground node. By reducing the matrix in this way, we are not losing any information because we know that each column in $\mathbf{A}$ with a single nonlinear element has to be connected to ground. In fact, by reducing $\mathbf{A}_f$ to $\mathbf{A}$ we are removing redundant information. We summarize the notation we have so far in table 2.

At this point we are ready to write down Kirchhoff's voltage law applied to each branch,

$$\mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{v}_b = \mathbf{0}, \tag{3}$$

and its twin, Kirchhoff's current law applied to each node,

$$\mathbf{A}\mathbf{i}_b = \mathbf{0}. \tag{4}$$

In a very formal but elegant way, equation (3) is actually saying that the sum of voltage drops along the loop from ground to positive branch terminal, negative branch terminal and back to ground is equal to zero. This holds for all branches. Equation (4) is stating that the sum of all branch currents entering or leaving each node equals zero.

Let us now focus on the devices in the branches, where the functional dependency between the branch voltages $\mathbf{v}_b$ and the respective branch currents $\mathbf{i}_b$ is determined. Because we are discussing only linear resistive networks in this section, we can state the branch equations as a linear function of the branch variables,

$$\mathbf{Y}\mathbf{v}_b + \mathbf{Z}\mathbf{i}_b - \mathbf{e} = \mathbf{0}. \tag{5}$$

$\mathbf{Y}$ and $\mathbf{Z}$ are b-dimensional square matrices containing admittance and impedance. In most cases these matrices will have nonzero elements only on their respective diagonal, as most branch relations involve only their own branch voltage and current. However, sometimes we need extra-diagonal nonzero elements to accommodate different kinds of controlled sources. We also need to consider independent voltage and current sources, as they will obviously cause zero rows in $\mathbf{Z}$ and $\mathbf{Y}$, respectively. The additions to the notation are listed in table 2

Notation	Dimension	Description
$\mathbf{e}$	$b \times 1$	column vector of excitation
$\mathbf{Y}$	$b \times b$	admittance coefficient matrix
$\mathbf{Z}$	$b \times b$	impedance coefficient matrix

Table 2: The vector and the matrices used in the mathematical description of the circuit's branches.

Before continuing we will demonstrate, that the expression (5) really can accommodate all eight basic linear elements. Let us start with a 2kΩ resistance in branch 1, a 20mS conductance in branch 2, a 3V independent voltage source in branch 3 and a 15mA independent current source in branch 4

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 20\text{mS} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_{b_1} \\ v_{b_2} \\ v_{b_3} \\ v_{b_4} \end{bmatrix} + \begin{bmatrix} 2\text{k}\Omega & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \\ i_{b_3} \\ i_{b_4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 3\text{V} \\ 15\text{mA} \end{bmatrix}. \tag{6}$$

If you do the multiplications and additions in (6) you will promptly end up with $v_{b_1} = 2\text{k}\Omega \cdot i_{b_1}$, $i_{b_2} = 20\text{mS} \cdot v_{b_2}$, $v_{b_3} = 3\text{V}$ and $i_{b_4} = 15\text{mA}$.

Likewise, all four types of controlled sources can be included, predictably causing extra diagonal nonzero elements in $\mathbf{Y}$ and $\mathbf{Z}$

$$\begin{bmatrix} -1 & 0 & 50 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_{b_1} \\ v_{b_2} \\ v_{b_3} \\ v_{b_4} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 400 \\ 60 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \\ i_{b_3} \\ i_{b_4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7}$$

Performing the operations in (7) is getting us a voltage-controlled voltage source $v_{b_1} = 50 \cdot v_{b_3}$ in branch 1, a current-controlled current source $i_{b_2} = 400 \cdot i_{b_4}$ in branch 2, a current-controlled voltage source $v_{b_3} = 60 \cdot i_{b_1}$ in branch 3 and a voltage-controlled current source $i_{b_4} = 2 \cdot v_{b_2}$ in branch 4.

Maybe you have noticed some inconsistency with regard to units of measurement in expression (5). In example (6) we have used the correct standard SI units, which doesn't fully agree with $\mathbf{Y}$ being a pure admittance matrix and $\mathbf{Z}$ being a pure impedance matrix as suggested in table 2. The same goes for vector $\mathbf{e}$, it contains a wild mix of units. Even worse, the units in example (7) are completely missing. The correct expression for the current-controlled voltage source would be really $v_{b_3} = 60\Omega \cdot i_{b_1}$, but no engineer would ever call the controlling coefficient a resistance.

So it really depends on who you are. All the above expressions are written from an electrical engineering point of view. A mathematician couldn't care less about units, he would only see variables with numerical values and so would a programmer. A physicist, on the other hand would be perfectly OK with a controlling coefficient in Ωs. He might even call it a transimpedance.

The important lesson here is, whatever results you get out of SPICE OPUS are numerically correct, but the corresponding units of measurement are your responsibility.

2.1 Circuit Tableau

Equation (5) can be viewed as an extension to the well-known Ohm's law, so we can set up a complete equation set for any linear resistive network. To this end we simply combine (3), (4), and (5) in matrix notation,

$$\begin{bmatrix} \mathbf{A}^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \mathbf{Y} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_b \\ \mathbf{i}_b \end{bmatrix} - \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{e} \end{bmatrix} = \mathbf{0}. \tag{8}$$

The equation set (8) is also known as the circuit tableau, where $\mathbf{A}^{\mathrm{T}}$ denotes the transposition of the reduced incidence matrix and $\mathbf{I}$ is a diagonal unity matrix. In principle, we could solve (8) directly and obtain all circuit variables. The equation set, however, is very large as it involves $2b + n$ equations with $2b + n$ unknowns. Due to the topological part, the equation set includes a large trivial portion with integer coefficients. A direct solution would thus be rather inefficient.

But before we continue, it is time we illustrated the circuit tableau with a simple example. Consider the resistive circuit in figure 4. Besides the ground node, it only has two more nodes and a total of four branches.

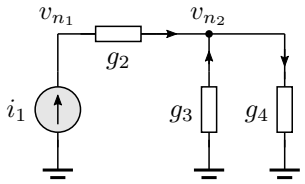

Figure 4: A simple resistive network.

First we need to capture the circuit topology by setting up the reduced incidence matrix. To this end we need the branch numbers, the node numbers, and the branch orientation, that is, we need to decide which is the positive and the negative terminal of each branch. With some devices, like the independent current source, we have no choice, because their functionality is terminal dependent, but with simple resistors, we can choose any orientation we want. So let us set up the incidence matrix according to branch current directions as indicated in figure 4,

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{bmatrix}. \tag{9}$$

With the incidence matrix in place, we can immediately formulate Kirchhoff's voltage law using equation (3),

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_{n_1} \\ v_{n_2} \end{bmatrix} - \begin{bmatrix} v_{b_1} \\ v_{b_2} \\ v_{b_3} \\ v_{b_4} \end{bmatrix} = \mathbf{0}, \tag{10}$$

as well as Kirchhoff's current law from (4),

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \\ i_{b_3} \\ i_{b_4} \end{bmatrix} = \mathbf{0}. \tag{11}$$

When setting up branch relations, we again have to make some choices. Simple resistors can be set up either as admittances in $\mathbf{Y}$ or as impedances in $\mathbf{Z}$. As we will see later, the admittance form is much more efficient. Therefore, we would like to express branch currents with branch voltages. In this case the impedance matrix is $\mathbf{Z} = -\mathbf{I}$ and equation (5) becomes

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & g_2 & 0 & 0 \\ 0 & 0 & g_3 & 0 \\ 0 & 0 & 0 & g_4 \end{bmatrix} \begin{bmatrix} v_{b_1} \\ v_{b_2} \\ v_{b_3} \\ v_{b_4} \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ i_{b_2} \\ i_{b_3} \\ i_{b_4} \end{bmatrix} - \begin{bmatrix} -i_1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}. \tag{12}$$

The complete circuit tableau (8) for our simple resistive circuit thus looks like this:

$$\begin{bmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & g_2 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_3 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & g_4 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} v_{n_1} \\ v_{n_2} \\ v_{b_1} \\ v_{b_2} \\ v_{b_3} \\ v_{b_4} \\ i_{b_1} \\ i_{b_2} \\ i_{b_3} \\ i_{b_4} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -i_1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}. \tag{13}$$

As expected, we are facing a 10 by 10 equation set, which can be solved directly. However, in order to improve computational efficiency, let us try to compact this equation set.

2.2 Basic Nodal Equations

With some simplifications we can considerably reduce the circuit tableau. For easier handling we simplify our notation (8) by appending the excitation vector to the coefficient matrix,

$$\begin{bmatrix} \mathbf{A}^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{Y} & \mathbf{Z} & \mathbf{e} \end{bmatrix} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_b \\ \mathbf{i}_b \\ -1 \end{bmatrix} = \mathbf{0}. \tag{14}$$

Then we rearrange the order of variables as well as the order of equations. In terms of matrices, we are permuting columns and rows. Specifically, we move column 1 in equation (14) between columns 3 and 4. At the same time, we

exchange rows 2 and 3, thus getting

$$\begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{A}^{\mathrm{T}} & \mathbf{0} \\ \mathbf{Y} & \mathbf{Z} & \mathbf{0} & \mathbf{e} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_b \\ \mathbf{i}_b \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \tag{15}$$

As the right-hand side of the equation is zero, we are allowed to multiply the left-hand side with any nonsingular matrix expression as long as the dimensions of the matrix match. We choose a unity matrix with the addition of $\mathbf{Y}$ in the first column of the second row. It is not difficult to see that the inverse of such a matrix always exists, thus guaranteeing nonsingularity. After the multiplication is carried out, we get

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{Y} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{A}^{\mathrm{T}} & \mathbf{0} \\ \mathbf{Y} & \mathbf{Z} & \mathbf{0} & \mathbf{e} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_b \\ \mathbf{i}_b \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{A}^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{Z} & \mathbf{Y}\mathbf{A}^{\mathrm{T}} & \mathbf{e} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_b \\ \mathbf{i}_b \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \tag{16}$$

By this transformation we have eliminated branch voltages $\mathbf{v}_b$ from the equation set, much as one step of a Gaussian elimination would eliminate one variable. The only difference is that we are working with entire submatrices instead of real numbers. Instead of the $2b + n$ dimensional equation set, we now have only $b + n$ independent remaining equations,

$$\begin{bmatrix} \mathbf{Z} & \mathbf{Y}\mathbf{A}^{\mathrm{T}} & \mathbf{e} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{i}_b \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}, \tag{17}$$

which can be solved directly. The solution ($\mathbf{i}_b$ and $\mathbf{v}_n$) would then be substituted back into (16) to obtain $\mathbf{v}_b$. The trick was to arrange for a zero column below the diagonal element $-\mathbf{I}$ in (16). If we repeat this strategy once more, we can eliminate $\mathbf{i}_b$ as well. So this time we multiply the system by a unity matrix with the addition of $-\mathbf{A}\mathbf{Z}^{-1}$ in the second column of the third row. Of course, we are assuming that the inverse of the impedance matrix $\mathbf{Z}^{-1}$ exists, which is not always the case, but we will return to this question later. Right now we get

$$\begin{aligned} &\begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & -\mathbf{A}\mathbf{Z}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{A}^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{Z} & \mathbf{Y}\mathbf{A}^{\mathrm{T}} & \mathbf{e} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_b \\ \mathbf{i}_b \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \\ &\begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{A}^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{Z} & \mathbf{Y}\mathbf{A}^{\mathrm{T}} & \mathbf{e} \\ \mathbf{0} & \mathbf{0} & -\mathbf{A}\mathbf{Z}^{-1}\mathbf{Y}\mathbf{A}^{\mathrm{T}} & -\mathbf{A}\mathbf{Z}^{-1}\mathbf{e} \end{bmatrix} \begin{bmatrix} \mathbf{v}_b \\ \mathbf{i}_b \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \end{aligned} \tag{18}$$

The final reduced equation set only has n independent variables and is the well-known circuit nodal equation set,

$$-\mathbf{A}\mathbf{Z}^{-1}\mathbf{Y}\mathbf{A}^{\mathrm{T}}\mathbf{v}_n + \mathbf{A}\mathbf{Z}^{-1}\mathbf{e} = \mathbf{0}. \tag{19}$$

We can directly solve (19) for the nodal voltage vector $\mathbf{v}_n$. All other circuit variables can be obtained by substitution back in (18). Note that we need not

solve the equation sets, just evaluate the matrix expressions. As expected, the branch voltages are trivial,

$$\mathbf{v}_b = \mathbf{A}^{\mathrm{T}}\mathbf{v}_n. \tag{20}$$

The branch currents, on the other hand, involve the inverse of the impedance coefficient matrix,

$$\mathbf{i}_b = -\mathbf{Z}^{-1}\mathbf{Y}\mathbf{A}^{\mathrm{T}}\mathbf{v}_n + \mathbf{Z}^{-1}\mathbf{e}. \tag{21}$$

We can postpone the discussion about the existence of $\mathbf{Z}^{-1}$ for just a little longer: fortunately, the example circuit in figure 4 does have a very convenient impedance matrix $\mathbf{Z} = -\mathbf{I}$. The inverse is trivial, so nodal equations (19) are greatly simplified, $\mathbf{A}\mathbf{Y}\mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{A}\mathbf{e} = \mathbf{0}$. By taking $\mathbf{A}$ from (9) and $\mathbf{Y}$ and $\mathbf{e}$ from (12), we can directly formulate the nodal equation set for the example circuit,

$$\begin{bmatrix} g_2 & -g_2 \\ -g_2 & g_2+g_3+g_4 \end{bmatrix} \begin{bmatrix} v_{n_1} \\ v_{n_2} \end{bmatrix} - \begin{bmatrix} i_1 \\ 0 \end{bmatrix} = \mathbf{0}. \tag{22}$$

~~~

However, in computer programs like SPICE OPUS mathematical expressions are not necessarily implemented directly into algorithms. In this case the computer algorithm will cut a few corners.

The coefficient matrix in nodal equations is $\mathbf{A}\mathbf{Y}\mathbf{A}^T$. Suppose the network is composed exclusively of admittances y_i, which makes $\mathbf{Y}$ a diagonal matrix. We know that the incidence matrix $\mathbf{A}$ only has 1 and -1 nonzero elements. So postmultiplying $\mathbf{A}$ with $\mathbf{Y}$ will multiply column i of $\mathbf{A}$ with the corresponding y_i from $\mathbf{Y}$. Let us assume that the admittance y in branch i is connected to node p and m. The multiplication product $(\mathbf{A}\mathbf{Y})$ will have the same dimensions as $\mathbf{A}$, moreover it will have the same nonzero-element pattern. Only now all nonzero values are multiplied by the respective admittance

$$\begin{bmatrix} & 0 & \\ \cdots & +1_{p,i} & \cdots \\ & 0 & \\ \cdots & -1_{m,i} & \cdots \\ & 0 & \end{bmatrix}_{n\times b} \begin{bmatrix} & & 0 & & \\ & \ddots & 0 & & \\ 0 & 0 & y_{i,i} & 0 & 0 \\ & & 0 & \ddots & \\ & & 0 & & \end{bmatrix}_{b\times b} = \begin{bmatrix} & 0 & \\ \cdots & +y_{p,i} & \cdots \\ & 0 & \\ \cdots & -y_{m,i} & \cdots \\ & 0 & \end{bmatrix}_{n\times b}.$$

Thus, in the final coefficient matrix $(\mathbf{A}\mathbf{Y})\mathbf{A}^T$ the entries are the inner product of a row of $(\mathbf{A}\mathbf{Y})$ with a column of $\mathbf{A}^T$. But the columns of $(\mathbf{A}\mathbf{Y})$ are the rows of $\mathbf{A}^T$, so the entry corresponds to the inner product of two rows of $\mathbf{A}$ resulting in a perfectly symmetric matrix

$$\begin{bmatrix} & 0 & \\ \cdots & +y_{p,i} & \cdots \\ & 0 & \\ \cdots & -y_{m,i} & \cdots \\ & 0 & \end{bmatrix}_{n\times b} \begin{bmatrix} & \vdots & & \vdots & \\ 0 & +1_{i,p} & 0 & -1_{i,m} & 0 \\ & \vdots & & \vdots & \end{bmatrix}_{b\times n} =$$
~~~

$$= \begin{bmatrix} & \vdots & & \vdots & \\ \cdots & +y_{p,p} & \cdots & -y_{p,m} & \cdots \\ & \vdots & & \vdots & \\ \cdots & -y_{m,p} & \cdots & +y_{m,m} & \cdots \\ & \vdots & & \vdots & \end{bmatrix}_{n\times n} . \tag{23}$$

We have arrived at a simple algorithm for building nodal equations directly from admittances. An admittance y in branch i connected to node p and m will add y to the p-th and m-th diagonal as well as $-y$ to the p-th and m-th anti-diagonal.

Sounds confusing? Let us go back to the circuit in figure 4 and its nodal equations (22). Sure enough, g_2 between nodes 1 and 2 is contributing to the diagonal and anti-diagonal 1 and 2. The other conductances g_3 and g_4 are connected from node 2 to ground and therefore only add to the diagonal 2.

The computer algorithm can be extended in a similar way to any other linear element, not just admittances.

Anyway, the nodal equation set (22) is much easier to solve than the entire circuit tableau (13), but the trick is to have a simple impedance coefficient matrix $\mathbf{Z}$ which is easily inverted. With resistive devices in branches we always have the choice of expressing the relation in admittance or impedance form. By choosing the admittance formulation, we automatically get -1 on the respective diagonal of $\mathbf{Z}$. Independent current sources will give us a -1 diagonal as well. Independent voltage sources, on the other hand, will necessarily cause a zero column and a zero row, thus rendering a singular $\mathbf{Z}$ without an inverse.

In order to solve this kind of problem, we need to modify the nodal equations.

2.3 Modified Nodal Equations

As we have seen in the previous section, any nodal approach is greatly simplified if we can provide a unity impedance coefficient matrix $\mathbf{Z} = -\mathbf{I}$. So let us separate the voltage sources that cannot be expressed using $\mathbf{Z} = -\mathbf{I}$ right from the beginning by splitting the incidence matrix into two parts $\mathbf{A} = [\mathbf{A_1}\mathbf{A_2}]$, where $\mathbf{A_1}$ represents all branches containing independent voltage sources and $\mathbf{A_2}$ all other branches. Consequently, relational equations (5) are also split two ways,

$$\begin{bmatrix} \mathbf{Y}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{Y}_2 \end{bmatrix} \begin{bmatrix} \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \end{bmatrix} + \begin{bmatrix} \mathbf{Z}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{Z}_2 \end{bmatrix} \begin{bmatrix} \mathbf{i}_{b1} \\ \mathbf{i}_{b2} \end{bmatrix} - \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \end{bmatrix} = \mathbf{0}. \tag{24}$$

Consider all independent voltage sources indexed 1 and everything else indexed 2. Branch equations for independent voltage sources are trivial $\mathbf{v}_{b1} = \mathbf{e}_1$ rendering $\mathbf{Z}_1 = \mathbf{0}$ and $\mathbf{Y}_1 = \mathbf{I}$. With these changes the complete circuit tableau from (14) becomes

$$\begin{bmatrix} \mathbf{A}_1^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_2^{\mathrm{T}} & \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{e}_1 \\ \mathbf{0} & \mathbf{0} & \mathbf{Y}_2 & \mathbf{0} & \mathbf{Z}_2 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \\ \mathbf{i}_{b1} \\ \mathbf{i}_{b2} \\ -1 \end{bmatrix} = \mathbf{0}. \tag{25}$$

Similar to the manipulations in the previous section, we permute column 1 between columns 5 and 6 and row 3 to the bottom. The equation set is then multiplied by a nonsingular matrix in order to eliminate branch voltages $\mathbf{v}_{b1}$ and $\mathbf{v}_{b2}$,

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Y}_2 & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{A}_2^{\mathrm{T}} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{e}_1 \\ \mathbf{0} & \mathbf{Y}_2 & \mathbf{0} & \mathbf{Z}_2 & \mathbf{0} & \mathbf{e}_2 \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \\ \mathbf{i}_{b1} \\ \mathbf{i}_{b2} \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \tag{26}$$

So far the result is not much different in comparison to the elimination step (16) in the previous section,

$$\begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{A}_2^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{e}_1 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{Z}_2 & \mathbf{Y}_2\mathbf{A}_2^{\mathrm{T}} & \mathbf{e}_2 \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \\ \mathbf{i}_{b1} \\ \mathbf{i}_{b2} \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \tag{27}$$

As $\mathbf{Z}_1 = \mathbf{0}$ we now have a missing pivot in row 3. To avoid singularity we exchange columns 3 and 4 and move row number 3 to the bottom to obtain

$$\begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{A}_2^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Z}_2 & \mathbf{0} & \mathbf{Y}_2\mathbf{A}_2^{\mathrm{T}} & \mathbf{e}_2 \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{e}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \\ \mathbf{i}_{b2} \\ \mathbf{i}_{b1} \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \tag{28}$$

We cannot eliminate the entire branch current vector anymore; hence, we settle for $\mathbf{i}_{b2}$ only,

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & -\mathbf{A}_2\mathbf{Z}_2^{-1} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{A}_2^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Z}_2 & \mathbf{0} & \mathbf{Y}_2\mathbf{A}_2^{\mathrm{T}} & \mathbf{e}_2 \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{e}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \\ \mathbf{i}_{b2} \\ \mathbf{i}_{b1} \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}, \tag{29}$$

to obtain

$$\begin{bmatrix} -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{A}_2^{\mathrm{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Z}_2 & \mathbf{0} & \mathbf{Y}_2\mathbf{A}_2^{\mathrm{T}} & \mathbf{e}_2 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1 & -\mathbf{A}_2\mathbf{Z}_2^{-1}\mathbf{Y}_2\mathbf{A}_2^{\mathrm{T}} & -\mathbf{A}_2\mathbf{Z}_2^{-1}\mathbf{e}_2 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} & \mathbf{e}_1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_{b1} \\ \mathbf{v}_{b2} \\ \mathbf{i}_{b2} \\ \mathbf{i}_{b1} \\ \mathbf{v}_n \\ -1 \end{bmatrix} = \mathbf{0}. \tag{30}$$

The two bottom lines actually represent an independent equation set, which is often referred to as the first modification of nodal equations. Besides the nodal

voltages $\mathbf{v}_n$, it also involves branch currents of independent voltage sources $\mathbf{i}_{b1}$. Assuming all devices, apart from independent voltage sources, may have a clean admittance form, we can set $\mathbf{Z}_2 = -\mathbf{I}$, yielding

$$\begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2\mathbf{Y}_2\mathbf{A}_2^{\mathrm{T}} \\ \mathbf{0} & \mathbf{A}_1^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \mathbf{i}_{b1} \\ \mathbf{v}_n \end{bmatrix} - \begin{bmatrix} \mathbf{A}_2\mathbf{e}_2 \\ \mathbf{e}_1 \end{bmatrix} = \mathbf{0}. \tag{31}$$

One might be tempted to interpret the zero below pivot $\mathbf{A}_1$ as an elimination of $\mathbf{i}_{b1}$, but note that $\mathbf{A}_1$ is not a square matrix. All unknowns are therefore independent and must be solved for simultaneously.

We have arrived at an equation set of the order $n + b_1$, where b_1 is the number of independent voltage sources in the circuit. The basic nodal equations are easily recognized on the top right-hand side of the matrix in equation (31). They are now framed by b_1 additional unknowns and supplemented with b_1 equations. Normally a circuit will only have a few voltage sources, so the order of the set of equations is only insignificantly larger than the basic nodal equations (19). Looking at any default output of SPICE OPUS, you will notice a listing of all nodal voltages plus currents through independent voltage sources.

So far we have covered both kinds of independent sources as well as arbitrary resistive branches. Next we need to consider controlled sources as well.

Voltage-controlled current sources will have -1 on the diagonal of $\mathbf{Z}$ and an extra-diagonal nonzero coefficient in $\mathbf{Y}$, so they would be included in $\mathbf{Y}_2$ in the modification of (31). Voltage-controlled voltage sources will cause an empty row in $\mathbf{Z}$ and an additional nonzero coefficient beside the diagonal 1 in the respective row of $\mathbf{Y}$. Similar to independent voltage sources, they belong in $\mathbf{Y}_1$. Current-controlled current sources will have -1 on the diagonal of $\mathbf{Z}$ in addition to a nonzero element in the respective row. This does not cause singularity, but the inverse of $\mathbf{Z}$ is not that trivial anymore. Basically, they can be included in $\mathbf{Y}_2$. But the problem of current-controlled voltage sources still exists. They have some nonzero coefficient in the respective row of $\mathbf{Z}$, but not on the diagonal. They may or may not cause singularities, depending on the circuit topology. To solve this problem, we would need to introduce yet another modification of the basic tableau. The procedure is not difficult, but it is rather complicated because the relational equations now need to be split four ways. In order to include all types of controlled sources, SPICE OPUS is actually based on a more complex modification than (31).

However, the order of the set of equations and all its properties are heavily dominated by the basic nodal equations, so we will focus on these equations in the following section. Remember that all the procedures described next will equally apply to any modification.

~~~

Before continuing, it is time for a practical example to illustrate all the theory so far. Consider the simple BJT amplifier in figure 5.

It has a 12V voltage supply, a 1μA current input with a respective impedance, a voltage divider to set the transistor bias, a 2KΩ pullup resistor and a 10KΩ load resistor. There is no feedback loop and the bipolar junction transistor is operating in the classic common emitter mode. The circuit really doesn't need more explanations.
~~~

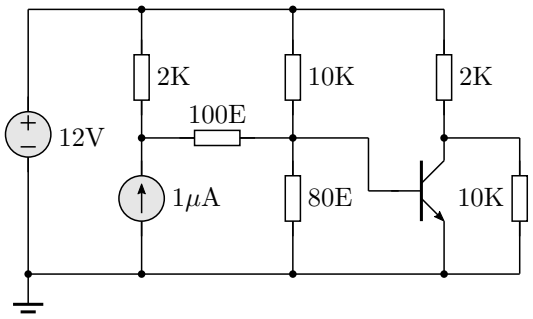

Figure 5: A simple transistor amplifier.

We want to see how the example will fit into (31). This chapter is about linear resistive networks, so we obviously need to replace the nonlinear transistor with a respective linear model prior to populating the equation set (31) with specific numbers. Suppose our BJT has a current gain of 100, an input resistance of 20KΩ and an output impedance of 50KΩ. A corresponding linear model depicted in figure 6.

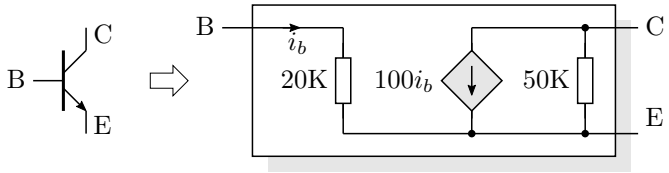

Figure 6: Replacing the transistor with a linear model.

There is a current-controlled current source in the model going against our assumption of $\mathbf{Z}_2$ being a unity. Luckily, we can transform it to a voltage-controlled current source, which will fit perfectly in our modified nodal equations. Since $i_b \cdot 20\text{K}\Omega = v_{be}$, we can use $\frac{v_{be}}{200}$ instead of $100i_b$ to control the current source. Next, we replace all resistances with conductances, add labels to the circuit nodes and arrive at the network in figure 7.

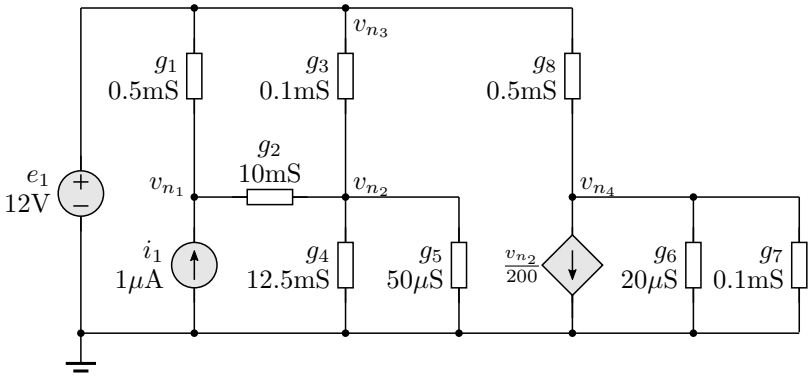

Figure 7: The nodal-equation-friendly network.

At this point we could simply set up $\mathbf{A}$, $\mathbf{Y}$, $\mathbf{e}$ and do the matrix multiplications required by (31). That would be the correct formal way. But we would rather demonstrate the building of nodal equations directly from branch elements is discussed in (23).

Armed with this knowledge we will illustrate the algorithm for composing modified nodal equations directly from the circuit network. In our case in figure 7, SPICE OPUS first parses the input netlist and stores all linear elements in an internal data structure. So it knows there are 4 nodes and 11 branches in the circuit. One branch holds a voltage source, therefore the final equation set will include 4 nodal voltages and one branch current as in

$$\left[\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right] \left[\begin{array}{c} i_{b_1} \\ \hline v_{n_1} \\ v_{n_2} \\ v_{n_3} \\ v_{n_4} \end{array}\right] = \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \end{array}\right]. \tag{32}$$

The lines in the equation set indicate the dominant basic nodal equations. As expected from (31), the modification only adds one equation and one variable.

Initially all values are zero, but this is the computer data space where equation (31) will gradually emerge. Next, the algorithm loops through all elements in the circuit and populates the matrix. Assuming the first element to go is g_1. According to (23), it will contribute symmetrically to four positions

$$\begin{bmatrix} 0 & 0.5\text{mS} & 0 & -0.5\text{mS} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -0.5\text{mS} & 0 & 0.5\text{mS} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_1} \\ v_{n_2} \\ v_{n_3} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{33}$$

Next comes g_2, adding 10mS to the diagonal 1 and 2 as well as subtracting 10mS from the anti-diagonal 1 and 2

$$\begin{bmatrix} 0 & 10.5\text{mS} & -10\text{mS} & -0.5\text{mS} & 0 \\ 0 & -10\text{mS} & 10\text{mS} & 0 & 0 \\ 0 & -0.5\text{mS} & 0 & 0.5\text{mS} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_1} \\ v_{n_2} \\ v_{n_3} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{34}$$

After all eight conductances in the circuit, we have an almost complete basic nodal equation set

$$\begin{bmatrix} 0 & 10.5\text{mS} & -10\text{mS} & -0.5\text{mS} & 0 \\ 0 & -10\text{mS} & 22.65\text{mS} & -0.1\text{mS} & 0 \\ 0 & -0.5\text{mS} & -0.1\text{mS} & 1.1\text{mS} & -0.5\text{mS} \\ 0 & 0 & 0 & -0.5\text{mS} & 0.62\text{mS} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_1} \\ v_{n_2} \\ v_{n_3} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{35}$$

Remains to add the voltage-controlled current source. It draws current from node 4 to ground, controlled by v_{n_2}. The controlling coefficient 0.005 therefore

pops up in row 4 of column 2 of the nodal equation set

$$\begin{bmatrix} 0 & 10.5\text{mS} & -10\text{mS} & -0.5\text{mS} & 0 \\ 0 & -10\text{mS} & 22.65\text{mS} & -0.1\text{mS} & 0 \\ 1 & -0.5\text{mS} & -0.1\text{mS} & 1.1\text{mS} & -0.5\text{mS} \\ 0 & 0 & 0.005 & -0.5\text{mS} & 0.62\text{mS} \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_1} \\ v_{n_2} \\ v_{n_3} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 1\mu\text{A} \\ 0 \\ 0 \\ 0 \\ 12\text{V} \end{bmatrix}. \tag{36}$$

As expected, the 1μA current source ends up on the respective right hand side, which completes the basic nodal equations.

Next, the modification part is added, that is everything with index 1 in (31). In our case this only includes the independent 12V voltage supply, which is represented with a single 1 in $\mathbf{A}_1$, adding the additional variable i_{b_1} to the equations set. One more variable means we need one more equation. You will find it right at the bottom of (36), namely $v_{n_3} = 12\text{V}$.

So far, the entire process of setting up modified nodal equations has been illustrated in the way a computer algorithm would do it. But, maybe a formal definition is even more explanatory

$$\begin{bmatrix} 0 & g_1+g_2 & -g_2 & -g_1 & 0 \\ 0 & -g_2 & g_2+g_3+g_4+g_5 & -g_3 & 0 \\ 1 & -g_1 & -g_3 & g_1+g_3+g_8 & -g_8 \\ 0 & 0 & 0.005 & -g_8 & g_6+g_7+g_8 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_1} \\ v_{n_2} \\ v_{n_3} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} i_1 \\ 0 \\ 0 \\ 0 \\ e_1 \end{bmatrix}.$$

3 Solving Linear Equations Sets

In this section we will look at ways to solve sets of linear equations. In general, a linear equation set may be formulated as n linear expressions with n unknowns. Assuming a square coefficient matrix $\mathbf{G}$, a constant vector $\mathbf{c}$, and a vector of unknowns $\mathbf{x}$, a linear equation set is formulated as $\mathbf{Gx} = \mathbf{c}$. The explicit notation of individual components thus is

$$\begin{aligned} g_{1,1}\, x_1 + g_{1,2}\, x_2 + ... + g_{1,n}\, x_n &= c_1 \\ g_{2,1}\, x_1 + g_{2,2}\, x_2 + ... + g_{2,n}\, x_n &= c_2 \\ &\vdots \\ g_{n,1}\, x_1 + g_{n,2}\, x_2 + ... + g_{n,n}\, x_n &= c_n. \end{aligned} \tag{37}$$

As we have established in the previous section, SPICE OPUS is actually based on a high order modification of the basic nodal equation set, but all equation set properties are heavily dominated by the basic nodal equations, so we will assume our general equation set (37) to be formulated according to (19). Assuming $\mathbf{Z} = -\mathbf{I}$, the coefficient matrix becomes $\mathbf{G} = \mathbf{AYA}^{\mathrm{T}}$, the constant vector $\mathbf{c} = \mathbf{Ae}$, and the unknowns $\mathbf{x} = \mathbf{v}_n$.

3.1 Sparse Matrices

A nonsingular circuit must necessarily have at least one branch more than there are nodes $b > n$; moreover, real-life circuits will have something like two branches for each node $b \approx 2n$. According to (2) each row of incidence matrix $\mathbf{A}_f$ will thus have an average of about four nonzero elements. Especially with large circuits, the reduced incidence matrix $\mathbf{A}$ will yield an insignificantly small lower row element average. The majority of elements in $\mathbf{A}$ will be zero. Matrices with mostly zero elements are referred to as sparse matrices.

Just looking at the definition (5) of the admittance matrix, we observe that $\mathbf{Y}$ is sparse as well. So both components of the coefficient matrix $\mathbf{G} = \mathbf{AYA}^{\mathrm{T}}$ are sparse. This should make $\mathbf{G}$ sparse too.

Consider the arbitrary admittance branch y_k in figure 3. It causes only two nonzero elements $a_{p,k} = +1$ and $a_{m,k} = -1$ in the k-th column of $\mathbf{A}$. It also places a single nonzero coefficient y_k on the diagonal of $\mathbf{Y}$. As we know from (23), the product $\mathbf{AY}$ will yield only two nonzero elements in the k-th column, namely $+y_k$ in row p and $-y_k$ in row m. Just as the multiplication by $\mathbf{A}$ has propagated the admittance y_k column-wise, the subsequent multiplication by its transposition $(\mathbf{AY})\mathbf{A}^{\mathrm{T}}$ will additionally propagate y_k row-wise. Therefore, an arbitrary admittance y_k between nodes p and m only contributes to four coefficients in G, namely to the diagonal $g_{p,p} = g_{m,m} = +y_k$, and to the anti-diagonal $g_{p,m} = g_{m,p} = -y_k$.

With approximately four branch connections to each node, we have the sum of all four admittances on the diagonal in addition to four nondiagonal nonzero elements in each row and column of $\mathbf{G}$. So we have typically something like five nonzero elements per row and column in $\mathbf{G}$ regardless of the circuit size. A circuit with 100 nodes will give us a 100 by 100 coefficient matrix $\mathbf{G}$, but of the 10,000 coefficients only about 500 will be nonzero. $\mathbf{G}$ definitely qualifies as a sparse matrix.

Consequently, a majority of coefficients in equation set (37) are equal to zero, and we can take advantage of this fact when attempting to solve (37).

3.2 LU Decomposition

One way of solving linear equation sets $\mathbf{Gx} = \mathbf{c}$ is by decomposing the square coefficient matrix $\mathbf{G}$ into the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$,

$$\mathbf{G} = \mathbf{LU}. \tag{38}$$

By definition, all elements above the diagonal of a lower triangular matrix are equal to zero, $l_{i,j} = 0 \ \forall i < j$. Similarly, all elements below the diagonal of an upper triangular matrix are zero, $u_{i,j} = 0 \ \forall i > j$. Consequently, the components of the product in (38) are

$$\begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,n} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{n,1} & g_{n,2} & \cdots & g_{n,n} \end{bmatrix} = \begin{bmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{bmatrix}. \tag{39}$$

After the multiplications we get n^2 individual equations with $n^2 + n$ unknowns,

$$g_{i,j} = \sum_{k=1}^{n} l_{i,k} u_{k,j} \quad \forall i, j = 1 \ldots n. \tag{40}$$

Obviously, the decomposition is not unique, because we have more unknowns than equations. We can freely choose the values for n variables. Usually the diagonal of $\mathbf{L}$ is set to 1,

$$l_{i,i} = 1 \quad \forall i = 1 \ldots n. \tag{41}$$

However, something is not right! We are trying to solve a system of n linear equations (37) by solving a system of n^2 linear equations (39). Did we make a bad deal?

Not at all. The trick is in the triangularity of $\mathbf{L}$ and $\mathbf{U}$. Because of all the zero elements, the sum in (40) doesn't really have to go all the way up to n. If you study (39) closely, you'll see that the sum only needs to go to $\min(i, j)$, because as soon as k is larger than i or j, either $l_{i,k}$ or $u_{k,j}$ is zero. Therefore (40) becomes

$$g_{i,j} = \sum_{k=1}^{\min(i,j)} l_{i,k} u_{k,j} \quad \forall i, j = 1 \ldots n. \tag{42}$$

Let us see where this is taking us. By resolving (42) with (41) in mind, we get the following n^2 equations.

$$\begin{array}{llll} g_{11} = u_{11} & g_{12} = u_{12} & g_{13} = u_{13} & \cdots \\ g_{21} = l_{21} u_{11} & g_{22} = l_{21} u_{12} + u_{22} & g_{23} = l_{21} u_{13} + u_{23} & \cdots \\ g_{31} = l_{31} u_{11} & g_{32} = l_{31} u_{12} + l_{32} u_{22} & g_{33} = l_{31} u_{13} + l_{32} u_{23} + u_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

Remember, all elements of $\mathbf{G}$ are given, while all components of $\mathbf{L}$ and $\mathbf{U}$ are unknowns. The first row of equations is easy. We know all $u_{1,i}$ since they are simply equal to the $g_{1,i}$. From the first equation of the second row we can calculate l_{21}, since we already know u_{11}. As luck would have it, the second equation of the second row will give us u_{22}, the third will yield u_{23} and so on. Each equation seem to hold only one unknown. But that has got nothing to do with luck. The secret lies in the right order of calculation. By proceeding from left to right and form top to bottom we can extract exactly one l or one u in each step

$$\begin{array}{llll} u_{11} = g_{11} & u_{12} = g_{12} & u_{13} = g_{13} & \dots \\ l_{21} = \frac{1}{u_{11}} g_{21} & u_{22} = g_{22} - l_{21} u_{12} & u_{23} = g_{23} - l_{21} u_{13} & \dots \\ l_{31} = \frac{1}{u_{11}} g_{31} & l_{32} = \frac{1}{u_{22}} (g_{32} - l_{31} u_{12}) & u_{33} = g_{33} - (l_{31} u_{13} + l_{32} u_{23}) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

But that is not all. If you look closely at the above equations, you will notice, that we can just as well proceed from top to bottom and then from left to right.

The formal notation for resolving (42) for the individual components of $\mathbf{L}$ and $\mathbf{U}$ is

$$l_{i,j} = \frac{1}{u_{j,j}} \left(g_{i,j} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j} \right) \quad \forall i = 2 \dots n, \;\; j = 1 \dots i-1 \tag{43}$$

and

$$u_{i,j} = g_{i,j} - \sum_{k=1}^{i-1} l_{i,k} u_{k,j} \quad \forall i = 1 \dots n, \;\; j = i \dots n. \tag{44}$$

In fact, we could define the same thing with an elegant series of matrix operators as discussed in section 1.4, but let us not be too enthusiastic.

Anyway, the expressions look complex, but if we compute them in the right order, then each equation holds only one unknown. All $l_{i,j}$ and $u_{i,j}$ are computed as a difference between the corresponding element $g_{i,j}$ and a sum of products of previously computed $l_{i,k}$ and $u_{k,j}$, that is, elements with lower indices $k < i, j$.

This is very convenient, because we can store the components of $\mathbf{L}$ and $\mathbf{U}$ in the original matrix $\mathbf{G}$ as we proceed. Note that the unit diagonal of $\mathbf{L}$ does not need storing. The decomposition can proceed row-wise or column-wise, but we prefer an alternating pattern between rows and columns is in figure 8.

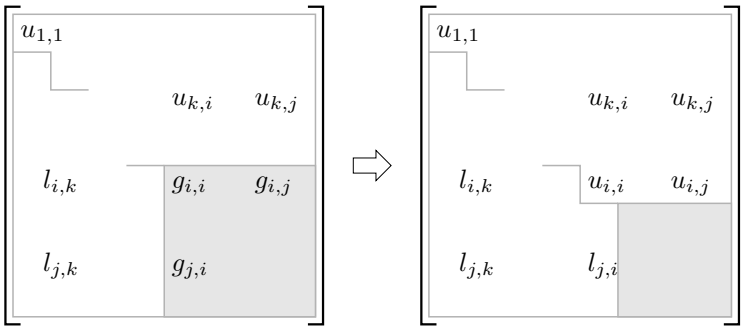

Figure 8: The LU decomposition algorithm, proceeding alternating row- and column-wise.

In each step we first finish the row, mapping $g_{i,i} \dots g_{i,n}$ to $u_{i,i} \dots u_{i,n}$ and then the respective column $g_{i+1,i} \dots g_{n,i}$ to $l_{i+1,i} \dots l_{n,i}$. This pattern is very important because $\mathbf{G}$ is a sparse matrix and needs special treatment, as we shall see in the next section.

The decomposition of $\mathbf{G}$ in itself does not solve our equation set directly. Two more steps are needed: a forward and a backward substitution.

Due to the rule of associativity, our equation set (38) can be computed either as $(\mathbf{LU})\mathbf{x} = \mathbf{c}$ or $\mathbf{L}(\mathbf{Ux}) = \mathbf{c}$. By introducing a new vector of unknowns $\mathbf{y} = \mathbf{Ux}$, we first solve $\mathbf{Ly} = \mathbf{c}$ for $\mathbf{y}$

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{2,1} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \dots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}. \tag{45}$$

As $\mathbf{L}$ is lower triangular, the top row only holds one unknown, namely y_1. With this in mind, we can extract y_2 from row two and so on. This process is called forward substitution

$$y_i = c_i - \sum_{j=1}^{i-1} l_{i,j} y_j \quad \forall i = 1 \dots n. \tag{46}$$

Knowing $\mathbf{y}$, we now solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$. This time the coefficient matrix is upper triangular, so we employ a so called back substitution

$$x_i = \frac{1}{u_{i,i}} \left(y_i - \sum_{j=i+1}^{n} u_{i,j} x_j \right) \quad \forall i = n \dots 1, \tag{47}$$

giving us the final solution $\mathbf{x}$.

Let us summarize. We can solve a linear equation set $\mathbf{Gx} = \mathbf{c}$, by decomposing the coefficient matrix $\mathbf{G} = \mathbf{LU}$, according to (43) and (44) using the algorithm in figure 8. So, the equation set becomes $\mathbf{LUx} = \mathbf{c}$. Next, we introduce a temporary new variable $\mathbf{y} = \mathbf{Ux}$ and by substitution (46) solve $\mathbf{Ly} = \mathbf{c}$ for $\mathbf{y}$. Finally, $\mathbf{y}$ is used in the back substitution (47) to solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$.

Thus, we have lined up all the math. But at what computational cost? With a little arithmetic we can establish that the LU decomposition in expressions (43) and (44) demands $\frac{n^3}{3} + \frac{n^2}{2} - \frac{n}{3}$ multiplications and just as many additions. The forward (46) and the backward substitution (47) each need another $\frac{n^2}{2} + \frac{n}{2}$ operations. So we have a total of $\frac{n^3}{3} + \frac{3n^2}{2} + \frac{2n}{3}$ multiplications and additions to solve the equation set $\mathbf{Gx} = \mathbf{c}$. With large circuits (e.g., $n > 100$) the n^3 term is predominant, so we can state that the computational effort is approximately $\frac{n^3}{3}$.

So far we have discussed the process of LU decomposition from a general point of view. However from the previous section we know that our equation set will only yield approximately five nonzero elements in each row in the coefficient matrix $\mathbf{G}$. This is potentially dangerous because we have divisions by $u_{i,i}$ in the decomposition (43) as well as the back substitution (47). Another consideration is the total number of arithmetic operations $\frac{n^3}{3}$ needed for the solution. This number should be much lower if we employ an efficient pivoting technique as explained in the next section.

3.3 Successful Pivoting Techniques

For several reasons, the key elements in the LU decomposition are the pivots, that is, the diagonal elements $u_{i,i}$ and $l_{i,i,}$. The latter are not really a problem because we have wisely chosen unit pivots in (41). However, the pivots in $\mathbf{U}$ are used for divisions in (43) and (47). Obviously, all pivots need to be nonzero; moreover, the pivot values must not be too small compared to all other coefficients since this can cause fatal numerical errors, as will be seen in the following section.

The way to influence the pivots is by matrix permutation. Consider the component notation of $\mathbf{Gx} = \mathbf{c}$ in (37). We have n individual equations with n unknowns. The order in which we write the equations surely cannot have any effect on the results and neither can the order of the unknowns. We are allowed to change the order of rows and columns in $\mathbf{G}$ as long as we reflect the changes in $\mathbf{x}$ and $\mathbf{c}$. Specifically, the order of elements in $\mathbf{x}$ must follow the order of columns in $\mathbf{G}$. Similarly, the order of $\mathbf{c}$ must accompany the order of rows in $\mathbf{G}$. Changing the order of rows and columns in a matrix or vector is termed permutation.

So let us consider the birth of pivots (44) in figure 8 from the permutation point of view. Each pivot is computed from the respective coefficient $g_{i,i}$ and the sum of products of all ls and gs above and to the left of it. If we ensured large, nonzero diagonals $g_{i,i}$ in advance, we should get nonzero pivots $u_{i,i}$ which are large enough not to cause fatal numerical errors. In some unlucky situations, however, the sum of products may be equal or very close to $g_{i,i}$, in which case we get pivots which are either zero or too small. These rare occasions cannot be foreseen in advance.

Nevertheless, the first step is a permutation of $\mathbf{G}$ to make sure the first pivot $u_{11} = g_{11}$ is not zero. This is not difficult as most diagonal elements are nonzero and relatively large already. Consider the fact explained in section 3.1 that all resistive branches contribute to the diagonal. Thus, only a few permutations will be required to prepare the diagonal in $\mathbf{G}$ for solid pivots during the LU decomposition.

Let us illustrate this procedure with the example circuit from figure 5. After having set up the respective modified nodal equation set, we have arrived at the coefficient matrix $\mathbf{G}$ in (36). At first, we will focus on the pattern of nonzero elements only. In figure 9, all nonzero elements are symbolically represented by dark gray squares and denoted by g.

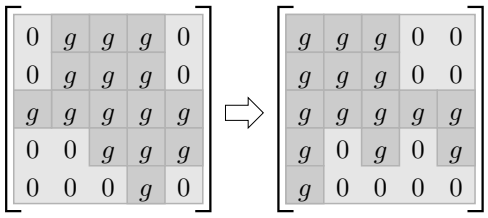

Figure 9: Fixing the first pivot in the coefficient matrix (36) by exchanging columns 1 and 4.

Unfortunately, the first pivot is zero, so we really need to make some permutations before starting the decomposition of $\mathbf{G}$. By exchanging columns 1 and 4, this problem is solved. As you can see, this leaves us with the last two pivots in $\mathbf{G}$ being zero, which can lead to zero pivots in $\mathbf{U}$. Regardless, let us start the decomposition by alternatively applying (44) row-wise and (43) column-wise as in figure 10.

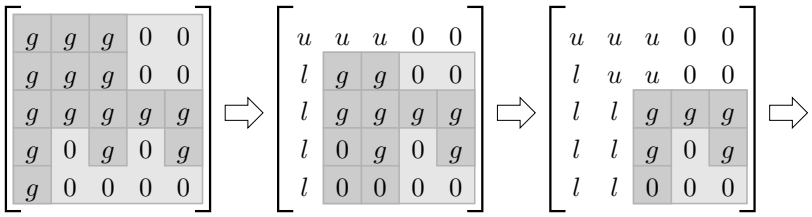

Figure 10: Mapping row 1 according to (44), column 1 according to (43) and marking three future fill-ins at g_{42}, g_{52}, g_{53}. Then mapping row 2 and column 2.

As expected, the fist row of $\mathbf{U}$ and the first column of $\mathbf{L}$ appear, replacing their original elements from $\mathbf{G}$ according to the algorithm in figure 8. Notice that the pattern of nonzero elements has not changed, however three zero elements in $\mathbf{G}$ are marked by dark gray squares, indicating future fill-ins.

A fill-in means that an initial zero element changes to nonzero, thus reducing the sparsity of the matrix. After the second mapping in figure 10, g_{42} and g_{52} promptly become nonzero. How come, we knew this in advance? According to (43), we know that $l_{42} = g_{42} - l_{41}u_{12}$. Since the product $l_{41}u_{12}$ is nonzero, the zero element g_{42} is mapped to a nonzero l_{42}. The same goes for g_{52}. The fate of future fill-ins is already sealed when rows and columns of $\mathbf{G}$ are mapped to respective rows of $\mathbf{U}$ and columns of $\mathbf{L}$. The products $l_{i,k}u_{k,j}$ in (44) and (43) are nonzero only if $l_{i,k} \neq 0$ and $u_{k,j} \neq 0$. So, with each new line of $\mathbf{U}$ and column of $\mathbf{L}$ we just check all remaining zero elements in $\mathbf{G}$ whether they just got a new nonzero l to the left as well as a nonzero u above. Wherever this is the case, we have a future fill-in.

With this in mind we can complete the decomposition in three more steps as in figure 11.

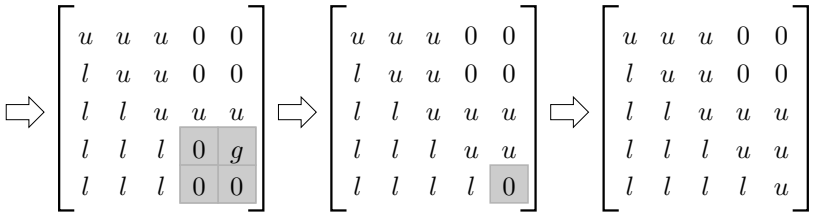

Figure 11: Three more mappings with respective fill-in predictions in dark gray squares.

Luckily, the two fill-ins at g_{44} and g_{55} have provided us with nonzero pivots u_{44} and u_{55}. However, as engineers we can't rely on luck. We need to find a way to control our fill-in! But what options do we still have?

In figure 9, we have initially done a column permutation in order to secure a nonzero first pivot. But who is to say that we can't permute rows and columns during the entire process of decomposition? After each step, we can pick any nonzero g as the next pivot. For instance, if the current pivot is $g_{k,k}$, but we would rather have $g_{i,j}$, we just exchange row k with row i and column k with column j. It goes without saying that by permuting rows in the remaining part of $\mathbf{G}$ must be accompanied by the same permutations in the emerging $\mathbf{L}$ as well as in vector $\mathbf{c}$. Likewise, the columns of $\mathbf{U}$ and the variable vector $\mathbf{x}$ must match the movements of the columns in $\mathbf{G}$.

This should give us the means to predict and control our fill-in. So it's back to square one. Again, we start with the modified nodal equation set (36), only this time around we think harder before selecting the first pivot. The algorithm is called prospecting for pivots. It's a relatively simple brute force algorithm. We know how to predict the fill-in of any given pivot, so we just check all pivot candidates.

Initially, we have 15 nonzero elements in the coefficient matrix (36). In principle, we have to permute each of them to the pivot position and check for future fill-in just like we did in figure 10. However, the permutation is not even necessary. We can just skip it and analyze the pattern of nonzero elements directly in $\mathbf{G}$ as demonstrated in figure 12.

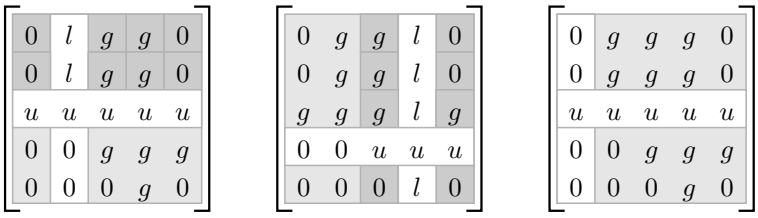

Figure 12: Checking the future fill-in for pivot candidates g_{32}, g_{44} and g_{31}.

In the example we are checking 3 of the 15 nonzero elements, namely g_{32}, g_{44} and g_{31}. In the left most case we are considering g_{32} for the pivot position. This would map row 3 to $\mathbf{U}$ and column 2 to $\mathbf{L}$. Fill-ins would potentially occur

at any position in $\mathbf{G}$ with a nonzero u in its column, as well as a nonzero l in its row. These position are marked by dark gray squares. This means, that g_{32} would add to 8 positions, from which 4 already are nonzero. Therefore g_{32} as pivot would cause 4 fill-ins at g_{11}, g_{15}, g_{21} and g_{25}.

Checking g_{44} in the same manner results in 4 fill-ins, while g_{31} is yielding no fill-ins at all. By looping through all 15 nonzero elements in $\mathbf{G}$ you can discover several candidates with no future fill-ins. For some reason, we prefer pivots with large values. So, we go for $g_{31} = 1.00000$ because it has the largest absolute value of all the 15 candidates. Figures 13, 14 and 15 visualize all the following steps of the pivoting algorithm.

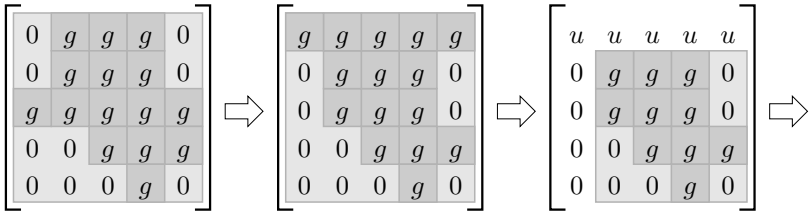

Figure 13: Permuting g_{31} to pivot position. Mapping row 1 and column 1 and prospecting for the next pivot. The largest element without future fill-in is $g_{54} = 1.00000$.

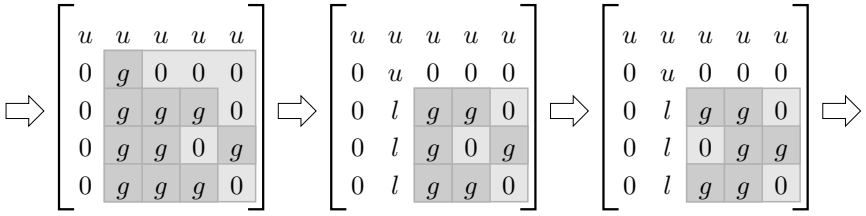

Figure 14: Permuting g_{54} to pivot position. Mapping row 2 and column 2. Finding and permuting the next pivot without fill-in, which is $g_{34} = 0.01050$.

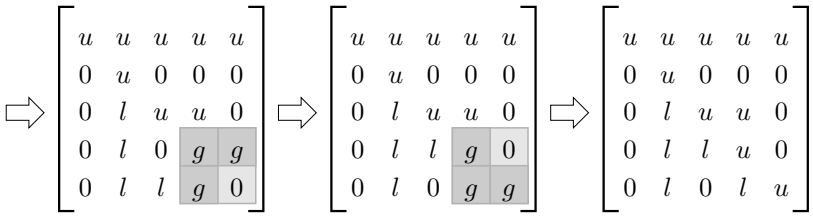

Figure 15: Mapping row 3 and column 3. Finding and setting the next pivot, which turns out to be $g_{54} = 0.02265$ and completing the decomposition.

Our pivoting technique has paid off well. Not only have we ensured nonzero pivots $u_{i,i}$, we have also greatly reduced the overall matrix fill-in. Consider the outcome of the decomposition with pivoting in figure 15 in comparison to the earlier outcome without pivoting in figure 11. Without pivoting we had 6 fill-ins

and no control over the pivot values. With pivoting we see no fill-in at all, plus we are getting large pivots.

Admittedly, the pivoting algorithm has been demonstrated on the extremely small circuit from figure 5, which is by no means representative. In fact the circuit is so small that the sparsity of its coefficient matrix is barely recognizable. Even so, the example is illustrative. It turns out that the presented pivoting algorithm is even more efficient when applied to realistic analog circuits with up to 50 nodes.

Notice, that so far we have not performed any actual calculations. All we have done in figures 13, 14 and 15 is simulating the nonzero element patterns during the decomposition. You could call it a decomposition dry-run. Its outcome is actually an optimized permutation of the original equation set $\mathbf{Gx} = \mathbf{c}$. The exact pivoting algorithm as it is implemented in SPICE OPUS is rather complex, so only the basic flowchart is presented in figure 16.

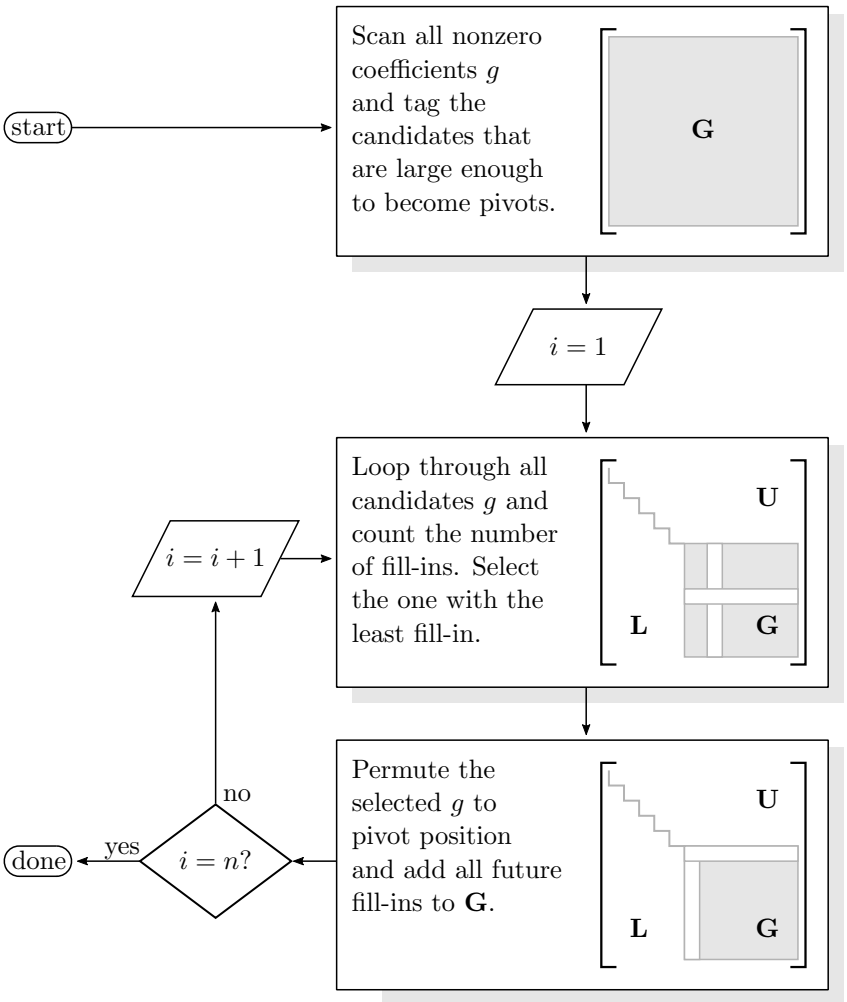

Figure 16: The pivoting algorithm simulating the nonzero pattern of the decomposition.

Keep in mind that the pivoting algorithm is run before the actual decomposition. It simulates the decomposition fill-in only. There is no information regarding the actual coefficient values, except for the initial $\mathbf{G}$ values. Consequently, the fill-ins are not pivot candidates. Nevertheless, the algorithm performs sufficiently in the majority of real cases.

The weak spot however might pop up during the subsequent decomposition if some pivot candidate is reduced too much or some fill-in becomes too large compared with its pivot. There are several advanced algorithms solving these kind of problems, but they would exceed the scope of this document.

3.4 Numerical Error Control

Finally everything is set to actually solve the modified equation of our example circuit in figure 5. So, we take the equation set (36) and permute it using the quick fix for the first pivot as in figure 9, or the fill-in optimized permutation from figure 15. Either way, we just alternatively run (44) and (43) to obtain $\mathbf{L}$ and $\mathbf{U}$. Then, we run a forward substitution (46) to get $\mathbf{y}$ followed by a back substitution (47) calculating our final result $\mathbf{x}$.

At this point, we are a bit worried about numerical errors during the entire process of LU decomposition. However, with such a small and simple circuit we surely get perfect results any way around. So we will not do the calculations just yet.

In the previous section we kept stressing the fact that pivots should not be too small compared to all other coefficients. Now we will explain why a small pivot may cause fatal numerical errors and discuss how small is too small. Numerical error analysis is a very difficult subject in mathematics, so we will simplify things as much as possible.

SPICE OPUS represents real numbers in the 64 bit double precision IEEE floating point format. This involves a 52 bit significand with an 11 bit exponent. Consequently, real numbers have a 10^{-16} relative round-off error. Our coefficients in $\mathbf{G}$ are based on model parameters with relative errors anywhere from 0.5 to, say, 0.001. Thus, the 10^{-16} round-off error is negligible.

We further know that each arithmetic operation will produce a round-off error in the 10^{-16} order of magnitude. So even if we had the worst case of error accumulation it would take something like 10^{14} operations to endanger the numerical quality of the final result. With $\frac{n^3}{3}$ operations per LU decomposition this would take a circuit of over 60,000 nodes with a nonsparse coefficient matrix. So the round-off errors remain at a safe distance at the far end of the significand.

However, there is a situation where the round-off errors can completely flood the entire significand. The root of this evil is the subtraction of two floating point numbers which are very close in value to each other. Suppose that we only had a 5 digit significand. Then the simple subtraction $5.2356? \cdot 10^3 - 5.2353? \cdot 10^3$ would yield something like $3.????? \cdot 10^{-1}$. A general rule says that if we see the exponent of a sum drop several magnitudes below the largest exponent of any summand, then the round-off error advances for just as many magnitudes in the significand.

To illustrate this, let us run the decomposition on (36) with just the basic pivot fix from figure 9. In order to study the round-off error propagation we need to scale the floating point format down to our circuit. Instead of calculating with a 16 digit significand we go with only 3. Our initial equation set $\mathbf{Gx} = \mathbf{c}$

thus becomes

$$\begin{bmatrix} -0.500 & 10.5 & -10.0 & 0 & 0 \\ -0.100 & -10.0 & 22.7 & 0 & 0 \\ 1.10 & -0.500 & -0.100 & 1000 & -0.500 \\ -0.500 & 0 & 5.00 & 0 & 0.620 \\ 1000 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_{n_3} \\ v_{n_1} \\ v_{n_2} \\ i_{b_1} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 0.001 \\ 0 \\ 0 \\ 0 \\ 12000 \end{bmatrix}. \tag{48}$$

To make the equation as illustrative as possible, we have multiplied the equation set by the factor of 1000, which doesn't change anything in terms of numerical error. Now, we run the decomposition rounding the significand to 3 digits after each addition and multiplication.

The result $\mathbf{Ly} = \mathbf{c}$ is ready for the forward substitution

$$\begin{bmatrix} 1000 & 0 & 0 & 0 & 0 \\ 200 & 1000 & 0 & 0 & 0 \\ 2200 & -1980 & 1000 & 0 & 0 \\ 1000 & 920 & -300 & 1000 & 0 \\ -2000000 & -1830000 & 954000 & -3180000 & 1000 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0.001 \\ 0 \\ 0 \\ 0 \\ 12000 \end{bmatrix}. \tag{49}$$

So far, everything is looking just fine. Notice that most elements have increased during the decomposition by several magnitudes. This is the direct consequence of our relatively small pivots in (48). During the decomposition all elements of $\mathbf{L}$ have been obtained by division with pivots, hence the large values.

The backward substitution $\mathbf{Ux} = \mathbf{y}$ is also facing large elements in the lower part of $\mathbf{U}$

$$\begin{bmatrix} -0.500 & 10.5 & -10.0 & 0 & 0 \\ 0 & -12.0 & 25.0 & 0 & 0 \\ 0 & 0 & 27.0 & 1000 & -0.500 \\ 0 & 0 & 0 & 300 & 0.470 \\ 0 & 0 & 0 & 0 & 1980 \end{bmatrix} \begin{bmatrix} v_{n_3} \\ v_{n_1} \\ v_{n_2} \\ i_{b_1} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}. \tag{50}$$

This comes as no surprise, since large values naturally propagate as $\mathbf{G}$ is gradually mapping to $\mathbf{L}$ and $\mathbf{U}$.

After performing the forward and back substitution, we finally arrive at the solution with a bit of a surprise

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1.00 \cdot 10^{-6} \\ 2.00 \cdot 10^{-7} \\ 1.82 \cdot 10^{-6} \\ -2.81 \cdot 10^{-7} \\ 1.20 \cdot 10^{1} \end{bmatrix}, \quad \begin{bmatrix} v_{n_3} \\ v_{n_1} \\ v_{n_2} \\ i_{b_1} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 11.9\text{V} \\ 0.961\text{V} \\ 0.463\text{V} \\ -9.55\text{mA} \\ 6.09\text{V} \end{bmatrix} \neq \begin{bmatrix} 12.000\text{V} \\ 1.0733\text{V} \\ 0.52683\text{V} \\ -9.8963\text{mA} \\ 5.4288\text{V} \end{bmatrix}. \tag{51}$$

As we can see, $\mathbf{x}$ turns out to be far off the correct solution on the right-hand side. Observing the computation of y_5 according to (46), we can see that the entire sum, after all round offs, equals zero. In other words y_5 is completely independent of all $y_1 \dots y_4$. This can't be good, y_5 is probably completely wrong. Consequently, the subsequent back substitution is doomed, but not only because it depends on a wrongly calculated y_5. It includes more exponent reductions by addition.

The fate of the substitutions has been decided right at the beginning of the decomposition. The first pivot $g_{11} = -0.500$ is actually one of the smallest of

all nonzero elements in $\mathbf{G}$. Many elements of $\mathbf{L}$ are divided by g_{11} in (43), which makes them large. These large values then propagate to $\mathbf{U}$ as well (44). This is why most elements have increased by several orders of magnitude during the decomposition. In itself this is not a problem. However, what goes up must come down, as the final results are independent of the pivot selections. And there are only two ways for coefficients to come down, either by multiplication (division) or by addition (subtraction). Multiplication would be numerically safe. Unfortunately, this is not the case for all coefficients.

~~~

What can we do to avoid all this? The numbers are defined by the circuit at hand. We don't want to change the circuit, so we can't change the numbers. But we can change the order of things!

Let us bake a loaf of bread. The recipe for the dough calls for 100g of starter, 250g of water, 450g of flour and 5g of salt. Since we want to keep dirty dishes to a minimum, we put a single bowl on a scale, reset the scale reading and start adding the ingredients. We pour sourdough stater into the bowl until the reading is 100g. Next we add water until the scale shows 350g and flour until we get 800g. Finally we pour salt into the bowl until the reading is 805g. Any engineers will wince at this! Why? The amount of salt we've actually just added is determined by subtracting the last two readings, namely 805g−800g=5g. The no go, a subtraction of two floating point numbers which are very close to each other in value! The standard digital kitchen scale is accurate down to 0.2%. So a reading of 805g could be anything between 803g and 807g, which is no problem so far. However, after the subtraction we end up with anything between 4g and 6g of salt in our bread, which is only 20% accurate.

A remedy is simple. By choosing a smart weighing order of ingredients, we can easily avoid the fatal subtraction. We start with the lightest ingredient (the salt) and work our way to the heavy ones. The target readings would be 5g, 105g, 355g and 805g. This time around, the accuracy of all ingredients is better then 0.5%.

So, from baking bread we have learned that order matters greatly. Back to our circuits, where things are not that obvious.

The idea is to avoid large coefficients during the decomposition by keeping the pivots as large as possible. Let us see what happens if we use the fill-in optimized permutation from figure 15. We get a nice string of large pivots to start with

$$\begin{bmatrix} 1000 & 1.10 & -0.500 & -0.100 & -0.500 \\ 0 & 1000 & 0 & 0 & 0 \\ 0 & -0.500 & 10.5 & -10.0 & 0 \\ 0 & -0.100 & -10.0 & 22.7 & 0 \\ 0 & -0.500 & 0 & 5.00 & 0.620 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_3} \\ v_{n_1} \\ v_{n_2} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} 0 \\ 12000 \\ 0.001 \\ 0 \\ 0 \end{bmatrix}. \tag{52}$$

Moreover, we know there won't be any fill-ins during the decomposition. Sure enough, the numerical values of $\mathbf{Ly} = \mathbf{c}$ remain in the same order of magnitude

$$\begin{bmatrix} 1000 & 0 & 0 & 0 & 0 \\ 0 & 1000 & 0 & 0 & 0 \\ 0 & -0.500 & 1000 & 0 & 0 \\ 0 & -0.100 & -95.2 & 1000 & 0 \\ 0 & -0.500 & 0 & 38.2 & 1000 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 12000 \\ 0.001 \\ 0 \\ 0 \end{bmatrix}. \tag{53}$$
~~~

The same goes for the elements of $\mathbf{Ux} = \mathbf{y}$

$$\begin{bmatrix} 1000 & 1.10 & -0.500 & -0.100 & -0.500 \\ 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 10.5 & 10.0 & 0 \\ 0 & 0 & 0 & 13.1 & 0 \\ 0 & 0 & 0 & 0 & 0.620 \end{bmatrix} \begin{bmatrix} i_{b_1} \\ v_{n_3} \\ v_{n_1} \\ v_{n_2} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}. \tag{54}$$

Next, both substitutions are run to obtain the final results

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 1.20 \cdot 10^{1} \\ 6.00 \cdot 10^{-3} \\ 6.91 \cdot 10^{-3} \\ 3.36 \cdot 10^{-3} \end{bmatrix}, \quad \begin{bmatrix} i_{b_1} \\ v_{n_3} \\ v_{n_1} \\ v_{n_2} \\ v_{n_4} \end{bmatrix} = \begin{bmatrix} -9.90\text{mA} \\ 12.0\text{V} \\ 1.07\text{V} \\ 0.527\text{V} \\ 5.42\text{V} \end{bmatrix} \approx \begin{bmatrix} -9.8963\text{mA} \\ 12.000\text{V} \\ 1.0733\text{V} \\ 0.52683\text{V} \\ 5.4288\text{V} \end{bmatrix} \tag{55}$$

This time the results are as accurate as they can possibly be considering that we have used a 3 digit significand only.

For simplicity, the involved example has been extremely downsized, which makes it unrealistic. However, the principle of round-off error propagation is realistic. The strategy of keeping pivots as large as possible by permutation actually works well in the majority of cases. However, it is not a magic wand–in some special cases large pivots can do more harm than good.

There are two simulator parameters in SPICE OPUS to control the pivot selection criteria: an absolute tolerance `pivtol` and a relative one `pivrel` with respective default values 10^{-12} and 10^{-3}. The pivot candidates in figure 8 must meet both conditions

$$\begin{aligned} |g_{i,i}| &> \texttt{pivtol} \\ |g_{i,i}| &> \texttt{pivrel} \cdot |g_{j,i}| \quad \forall j = i+1, i+2, \ldots, n. \end{aligned} \tag{56}$$

Note that the relative tolerance is checked only for elements below the pivot, because only $\mathbf{L}$ components are divided by the pivot (43). Although the default pivot simulator parameters can be changed, even advanced users are strongly discouraged to alter these settings, as unpredictable numeric behavior may occur.

4 Introducing Nonlinear Devices

So far we have seen how to set up a linear equation set from a circuit netlist and how to efficiently solve the equations. We have allowed only linear devices, which is a tough assumption in circuit design.

Now we are ready to take the next step by introducing nonlinear devices. In general, we cannot express branch currents and branch voltages explicitly anymore. We now have to use the general expression

$$\mathbf{R}(\mathbf{v}_b, \mathbf{i}_b) = 0 \tag{57}$$

instead of (5). $\mathbf{R}(\mathbf{v}_b, \mathbf{i}_b)$ is a column vector of functions of branch voltages ($\mathbf{v}_b$) and branch currents ($\mathbf{i}_b$) describing the nonlinearity contained in each branch,

$$\begin{aligned} R_1(v_{b_1}, v_{b_2} \dots v_{b_b}, i_{b_1}, i_{b_2} \dots i_{b_b}) &= 0 \\ R_2(v_{b_1}, v_{b_2} \dots v_{b_b}, i_{b_1}, i_{b_2} \dots i_{b_b}) &= 0 \\ &\vdots \\ R_b(v_{b_1}, v_{b_2} \dots v_{b_b}, i_{b_1}, i_{b_2} \dots i_{b_b}) &= 0. \end{aligned}$$

Of course, real circuits will not include devices with functional dependences of all circuit variables, so the above equations are the most general case. Also, under normal circumstances not all branches include nonlinear devices; hence, many of the above functions will actually be simple linear expressions as in (5).

Nevertheless, our complete general nonlinear equation set reads

$$\begin{bmatrix} \mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{v}_b \\ \mathbf{A}\mathbf{i}_b \\ \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b) \end{bmatrix} = \mathbf{0} \tag{58}$$

instead of the more explicit form in (8). Although we still have a linear part, which describes the circuit topology, we cannot attempt to solve this equation set analytically. In special cases one can find direct solutions, but in general we have to resort to iterative techniques.

4.1 Fixed Point Iteration

One of the most fundamental principles for solving nonlinear equations with computers is fixed point iteration. The algorithm is suitable for solving equations of the form

$$x = f(x), \tag{59}$$

not unlike our problem in (58). Equation (59) is solved by computing the sequence

$$x^{(k+1)} = f(x^{(k)}), \text{ where } k = 0, 1, 2, \dots, m. \tag{60}$$

Obviously, the iteration series needs a starting point $x^{(0)}$ and some convergence criterion to determine the necessary number of iterations m. We assume that a value for m exists such that $|x^{(m)} - x^{(m-1)}| < \epsilon$, for an arbitrary ϵ. The

last iteration $x^{(m)}$ is then accepted as the approximation of the solution $x^{(*)}$ to (59).

If we are to use fixed point iteration to solve our nonlinear equation set (58), which has the form $g(x) = 0$ instead of (59), we need to transform it slightly. The idea is to add x to both sides of the equation to obtain

$$x = x + g(x). \tag{61}$$

Let us look at an example of fixed point interation. Assume the simple function $g(x) = \frac{5}{3} - e^x$. In this case we know the solution of (61) as $x^{(*)} = \ln\left(\frac{5}{3}\right)$. Further assume that the starting point $x^{(0)} = 0.1$ and the convergence criterion is $\epsilon = 10^{-4}$.

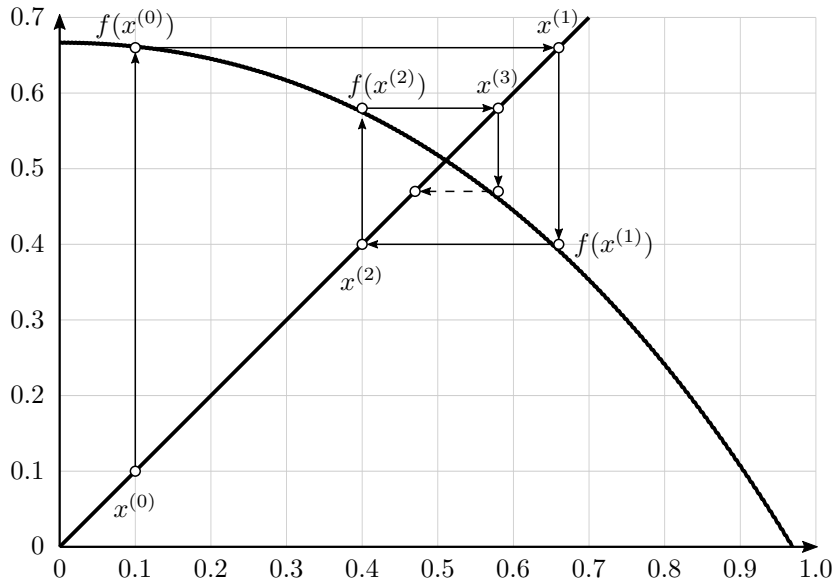

Figure 17: Fixed point iteration convergence.

In figure 17 you can see a straight line representing the left-hand side of (61) and the curve depicting the nonlinear right-hand side. The iteration sequence starting at 0.1 needs 21 steps for convergence,

$$\begin{aligned}
x^{(0)} &= 0.1000000 \\
x^{(1)} &= 0.6614957 \\
x^{(2)} &= 0.3904740 \\
x^{(3)} &= 0.5794596 \\
x^{(4)} &= 0.4610527 \\
&\vdots \\
x^{(20)} &= 0.5107529 \\
x^{(21)} &= 0.5108741.
\end{aligned}$$

The accepted approximation $x^{(21)} = 0.5108741$ is not far from the solution $x^{(*)} = \ln\left(\frac{5}{3}\right) = 0.5108256$. Although this looks promising, there are some problems we should consider.

First, there is the possibility that (61) does not have a solution. This means that there is no intersection between the two curves in figure 17. In this case any iteration is irrelevant. Second, there could be more than one solution, that is, more than one intersection of the curve and the straight line. However, knowing that our equation set is describing a correct electrical circuit with a stable solution, we can usually assume exactly one solution to the equation set. An exception would be a circuit with multiple operating points, like a Schmitt trigger or a flip-flop, where several stable and unstable solutions exist. In these cases fixed point iteration would find only one of the stable solutions, depending on the starting point $x^{(0)}$. So it is the responsibility of the engineer to ensure a correct circuit and manually supply a starting point if one is needed.

A more serious problem is the question of convergence itself. It is well known that fixed point iteration works only if the absolute derivative of the curve at the intersection is less than unity,

$$\left|f'(x^{(*)})\right| < 1 \text{ where } f'(x) = \frac{\partial f(x)}{\partial x}. \tag{62}$$

Moreover, the speed of convergence depends on the absolute derivative at the intersection. The closer it gets to zero, the faster the convergence. In our example we have the derivative $f'(x) = 1 + g'(x) = 1 - e^x$, which at the intersection is $f'(x^{(*)}) = -\frac{2}{3}$. This ensures convergence, as seen in figure 17, even if the iteration sequence is a little slow.

Let us now consider the function $g(x) = 5 - e^x$. The solution of (61) now is $x^{(*)} = \ln(5)$. We still have the same expression for the derivative, but the gradient at the solution now is $f'(x^{(*)}) = -4$, which is well outside the convergence interval $[-1 \ldots 1]$. Even if we start very close to the solution $x^{(0)} = 1.55$, our fixed point iteration now fails to find the solution, as seen from figure 18.

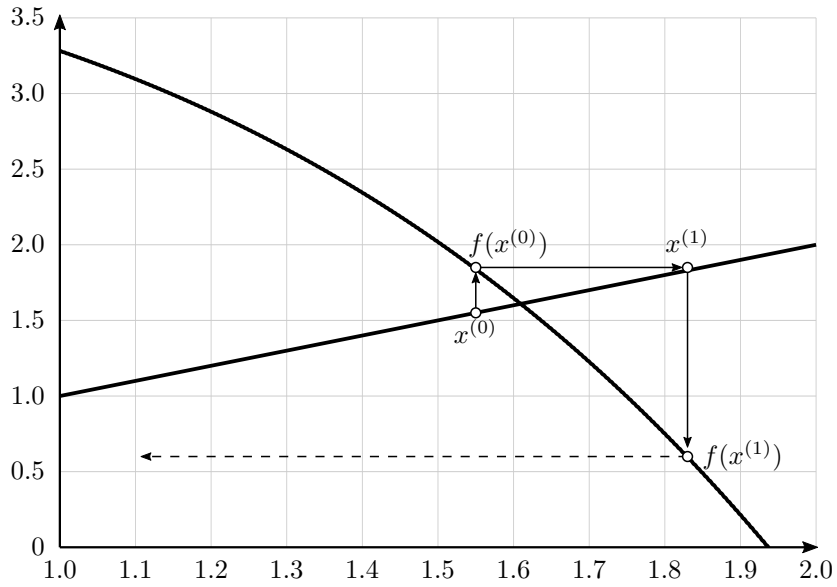

Figure 18: Fixed point iteration divergence.

We have a clear divergence as a consequence of the steep intersection angle.

In electronic circuits we expect a multitude of highly nonlinear devices, so we clearly cannot hope for the basic fixed point algorithm to converge. We need an improved algorithm.

4.2 Newton–Raphson Algorithm

Let us return to transformation (61) for a moment. We had to add x to both sides of the equation in order to get the right equation form for fixed point iteration. Keep in mind that the original equation set (58) reads $g(x) = 0$. So before adding x we can multiply both sides by any arbitrary function $k(x)$ without altering the solution. The only condition is that $k(x)$ be nonzero at the solution $x^{(*)}$. In general, our equation now is

$$x = x + k(x)g(x). \tag{63}$$

So we have some freedom of choice regarding $k(x)$. We can ensure convergence (62) and ideally even speed up the iteration sequence. The best thing would be to achieve a zero gradient at the point of intersection.

Take the right-hand side of (63), $f(x) = x + k(x)g(x)$, and formulate the first derivative $f'(x) = 1 + k'(x)g(x) + k(x)g'(x)$. We want $f'(x)$ to be zero at $x = x^{(*)}$. Knowing that $g(x^{(*)}) = 0$, we get $f'(x^{(*)}) = 1 + k(x^{(*)})g'(x^{(*)}) = 0$, from which it follows that $k(x) = -g'(x)^{-1}$. Equation (63) now becomes

$$x = x - \frac{g(x)}{g'(x)}. \tag{64}$$

With a properly defined circuit there is no danger that $g'(x)$ could be zero at $x = x^{(*)}$, as this would mean multiple adjacent solutions.

This modified equation ensures that fixed point iterations always converge; moreover, the convergence should be very fast. Applying fixed point iteration to this transformation is known as the Newton–Raphson algorithm.

Before continuing, let us see how Newton–Raphson iterations will solve our problem from the previous section, $g(x) = 5 - e^x$. After the transformation we have to solve $x = x - 1 + 5e^{-x}$. Assuming the starting point $x^{(0)} = 1.1$ and the usual convergence criterion $\epsilon = 10^{-4}$, we only need a sequence of four steps,

$$\begin{aligned} x^{(0)} &= 1.100000 \\ x^{(1)} &= 1.764355 \\ x^{(2)} &= 1.620841 \\ x^{(3)} &= 1.609503 \\ x^{(4)} &= 1.609438. \end{aligned}$$

As expected, we can see the zero gradient intersection in figure 19 guaranteeing convergence and speeding up things nicely.

But there is always a price to pay. Instead of solving the simple equation $g(x) = 0$ we need to compute the complex expression (64) in each iteration. The expression involves the inverse of the first derivative of $g(x)$. So we are looking at a considerable additional computational effort, but we gain shorter iteration loops compensating the extra effort.

Most importantly, we now have a guarantee for convergence. Or do we? We have ensured a zero gradient, but this holds only in the close neighborhood of

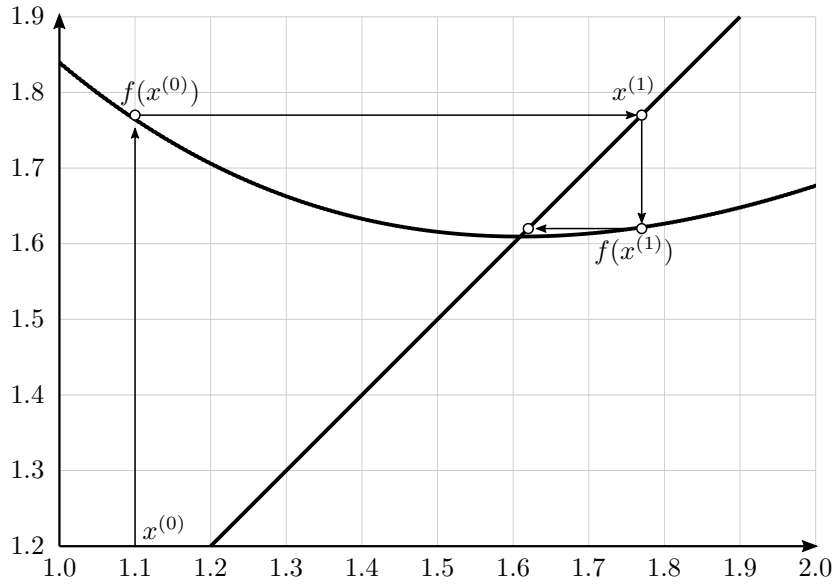

Figure 19: Newton–Raphson iteration convergence.

the solution. In other words, as long as the starting point $x^{(0)}$ is close enough to the solution $x^{(*)}$, the convergence really is guaranteed.

However, the right-hand side of (64) might not be well behaved outside the immediate neighborhood of the solution. In these cases Newton–Raphson iterations still can experience convergence problems. In real circuit simulations this is not a very likely scenario but is by no means unheard of. Circuits with strong feedback loops, for instance, tend to diverge, so it is very important to have methods which bring the iterations close enough to the region where the Newton–Raphson algorithm is convergent.

Finally, let us discuss a special situation where the multiplier $k(x)$ introduces additional solutions to $k(x)g(x) = 0$. We have selected $k(x) = -g'(x)^{-1}$. So we are concerned by the fact that at some x the derivative becomes infinite. This would introduce a new and unrealistic solution. However, an infinite gradient could only be caused by a nonlinearity in (58) having zero conductance at some voltage. This is something that we can control on the circuit level, so we do not really have to worry about it here.

Now we are finally ready to apply Newton–Raphson iteration (64) to our equation set (58). So far we have discussed iterative algorithms for one-dimensional problems. We need to extend them to spaces of arbitrary dimensionality. Instead of a single real number, we now consider x an n-dimensional column vector of variables $x_1, x_2 \dots x_n$ and $g(x)$ a respective column vector of n nonlinear functions $g_1(x_1, x_2 \dots x_n), g_2(x_1, x_2 \dots x_n) \dots g_n(x_1, x_2 \dots x_n)$. Our first derivative of $g'(x)$ becomes a square matrix of all partial derivatives, also known as the Jacobi matrix $\mathbf{J}(\mathbf{x})$,

$$\mathbf{g}'(\mathbf{x}) = \mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial g_1(\mathbf{x})}{\partial x_1} & \frac{\partial g_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial g_2(\mathbf{x})}{\partial x_1} & \frac{\partial g_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial g_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial g_n(\mathbf{x})}{\partial x_1} & \frac{\partial g_n(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial g_n(\mathbf{x})}{\partial x_n} \end{bmatrix}. \tag{65}$$

Let us substitute the general expressions for the Jacobi matrix with our circuit tableau notation, where

$$\mathbf{x} = \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_b \\ \mathbf{i}_b \end{bmatrix} \quad \text{and} \quad \mathbf{g}(\mathbf{x}) = \begin{bmatrix} \mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{v}_b \\ \mathbf{A}\mathbf{i}_b \\ \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b) \end{bmatrix}. \tag{66}$$

In general, the Jacobi matrix appears in the following form:

$$\mathbf{g}'(\mathbf{x}) = \begin{bmatrix} \frac{\partial(\mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{v}_b)}{\partial \mathbf{v}_n} & \frac{\partial(\mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{v}_b)}{\partial \mathbf{v}_b} & \frac{\partial(\mathbf{A}^{\mathrm{T}}\mathbf{v}_n - \mathbf{v}_b)}{\partial \mathbf{i}_b} \\ \frac{\partial(\mathbf{A}\mathbf{i}_b)}{\partial \mathbf{v}_n} & \frac{\partial(\mathbf{A}\mathbf{i}_b)}{\partial \mathbf{v}_b} & \frac{\partial(\mathbf{A}\mathbf{i}_b)}{\partial \mathbf{i}_b} \\ \frac{\partial \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b)}{\partial \mathbf{v}_n} & \frac{\partial \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b)}{\partial \mathbf{v}_b} & \frac{\partial \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b)}{\partial \mathbf{i}_b} \end{bmatrix}. \tag{67}$$

We can partially simplify this expression, because the topological part of the circuit tableau is linear. We get

$$\mathbf{g}'(\mathbf{x}) = \begin{bmatrix} \mathbf{A}^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \frac{\partial \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b)}{\partial \mathbf{v}_b} & \frac{\partial \mathbf{R}(\mathbf{v}_b, \mathbf{i}_b)}{\partial \mathbf{i}_b} \end{bmatrix}. \tag{68}$$

This expression is very similar to a linear equation set as in (8). Actually, we are looking at the linear equivalent of our nonlinear circuit, where all nonlinearities are replaced by gradients at a specific operating point $\mathbf{v}_b$, $\mathbf{i}_b$.

Unfortunately, we will have to compute the inverse of the Jacobi matrix if we want to use the Newton–Raphson iteration form (64). The only alternative is to solve a linear equation set instead. Let us first apply the iterative algorithm to (64),

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - [\mathbf{g}'(\mathbf{x}^{(k)})]^{-1}\mathbf{g}(\mathbf{x}^{(k)}). \tag{69}$$

Let us multiply both sides of this equation by $\mathbf{g}'(\mathbf{x})$. We have to be careful because $\mathbf{g}'(\mathbf{x})$ is a square matrix, so we must premultiply by $\mathbf{g}'(\mathbf{x})$,

$$\mathbf{g}'(\mathbf{x}^{(k)})\mathbf{x}^{(k+1)} = \mathbf{g}'(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{g}(\mathbf{x}^{(k)}). \tag{70}$$

The result looks a bit confusing, because the left-hand side now includes the variable vector from the current iteration $\mathbf{x}^{(k)}$ as well as the new one $\mathbf{x}^{(k+1)}$. Keep in mind that the latter is to be computed from (70). We are actually looking at a linear equation set of the type $\mathbf{G}\mathbf{x} = \mathbf{c}$. The entire right-hand side is known from iteration k and so is the Jacobi matrix on the left-hand side. From (70), (66), and (68) we have

$$\begin{bmatrix} \mathbf{A}^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \frac{\partial \mathbf{R}}{\partial \mathbf{v}_b} & \frac{\partial \mathbf{R}}{\partial \mathbf{i}_b} \end{bmatrix}^{(k)} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_b \\ \mathbf{i}_b \end{bmatrix}^{(k+1)} = \begin{bmatrix} \mathbf{A}^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \frac{\partial \mathbf{R}}{\partial \mathbf{v}_b} & \frac{\partial \mathbf{R}}{\partial \mathbf{i}_b} \end{bmatrix}^{(k)} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_b \\ \mathbf{i}_b \end{bmatrix}^{(k)} - \begin{bmatrix} \mathbf{A}^{\mathrm{T}} v_n - \mathbf{v}_b \\ \mathbf{A}\mathbf{i}_b \\ \mathbf{R} \end{bmatrix}^{(k)}. \tag{71}$$

For simplicity, we have dropped the explicit functional dependence of $\mathbf{R}$ on $\mathbf{v}_b, \mathbf{i}_b$. For the same reason, the iteration superscripts are symbolically placed outside the matrices rather than at each variable instance. We can further resolve part of the right-hand side, so that equation (71) simplifies to

$$\begin{bmatrix} \mathbf{A}^{\mathrm{T}} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \frac{\partial \mathbf{R}}{\partial \mathbf{v}_b} & \frac{\partial \mathbf{R}}{\partial \mathbf{i}_b} \end{bmatrix}^{(k)} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{v}_b \\ \mathbf{i}_b \end{bmatrix}^{(k+1)} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \frac{\partial \mathbf{R}}{\partial \mathbf{v}_b}\mathbf{v}_b + \frac{\partial \mathbf{R}}{\partial \mathbf{i}_b}\mathbf{i}_b - \mathbf{R} \end{bmatrix}^{(k)}. \tag{72}$$

The algorithm of (72) resembles our well-known linear equation set in (8). The topological part is identical, but the branch equations are obviously a linearization at the point of the previous iteration. This can be illustrated separately as

$$\frac{\partial \mathbf{R}}{\partial \mathbf{v}_b}\mathbf{v}_b^{(k+1)} + \frac{\partial \mathbf{R}}{\partial \mathbf{i}_b}\mathbf{i}_b^{(k+1)} = \frac{\partial \mathbf{R}}{\partial \mathbf{v}_b}\mathbf{v}_b + \frac{\partial \mathbf{R}}{\partial \mathbf{i}_b}\mathbf{i}_b - \mathbf{R}. \tag{73}$$

All expressions above originate from the k-th iteration except the variable vectors $\mathbf{v}_b$ and $\mathbf{i}_b$ marked with $(k+1)$. The expression clearly describes a $2b$ dimensional plane touching the branch curves at the point where $\mathbf{v}_b^{(k+1)} = \mathbf{v}_b^{(k)}$ and $\mathbf{i}_b^{(k+1)} = \mathbf{i}_b^{(k)}$. The plane is tilted in the same direction as the curve at the point of inflection. In other words, we have a tangential plane at $\mathbf{R}(\mathbf{v}_b^{(k)}, \mathbf{i}_b^{(k)})$.

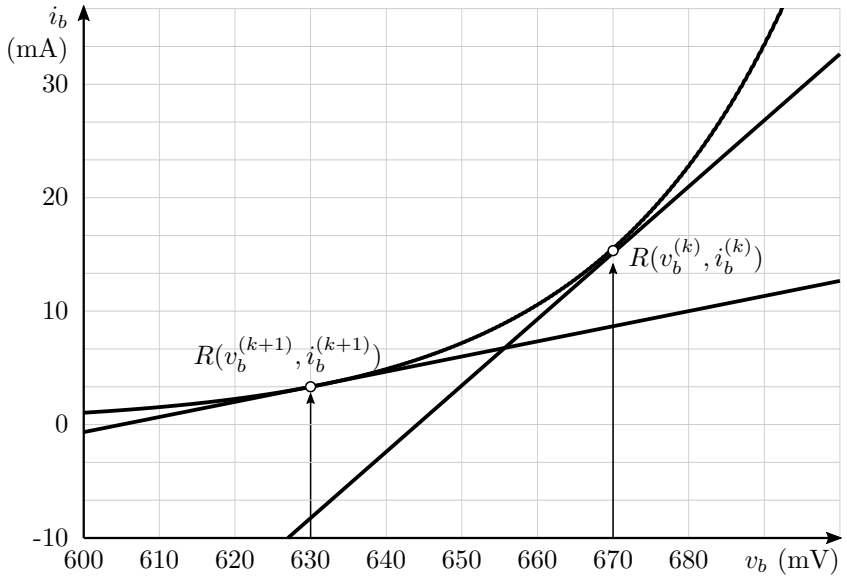

Figure 20: Newton–Raphson equivalent device.

To illustrate this point, consider a simple diode with the branch relation $R(v_b, i_b) = I_s(e^{\frac{v_b}{v_t}} - 1) - i_b = 0$ where $I_s = 10^{-14}$A and $v_t = 26$mV. In figure 20 this nonlinear relation is depicted by a curve. Substituting R in equation (73) with the diode expression, we get a straight line $\frac{I_s}{v_t}e^{\frac{v_b}{v_t}} \cdot v_b^{(k+1)} - i_b^{(k+1)} = I_s e^{\frac{v_b}{v_t}}(\frac{v_b}{v_t} - 1) + I_s$ touching $R(v_b, i_b) = 0$ at $v_b^{(k+1)} = v_b$.

Suppose the branch voltage in iteration k is $v_b = 0.67\text{V}$. We get a line representing the diode as a linear element for the duration of iteration $k+1$. After solving the linear equation set (72), we get a solution somewhere on that line. Suppose we have obtained $v_b = 0.63\text{V}$. The next iteration is going to be based on a linearization at that point.

The iterative process will continue in this manner until at some point the iteration falls very close to the previous one for all nonlinear branches. In this case the approximation of the nonlinearity by a straight line is justified, and the result of the linear equation set is considered to be sufficiently close to the solution of the underlying nonlinear equation set.

In this section we have seen that the Newton–Raphson algorithm very efficiently solves nonlinear circuit equations. Moreover, the algoritm can be applied at the device level, as illustrated in figure 21.

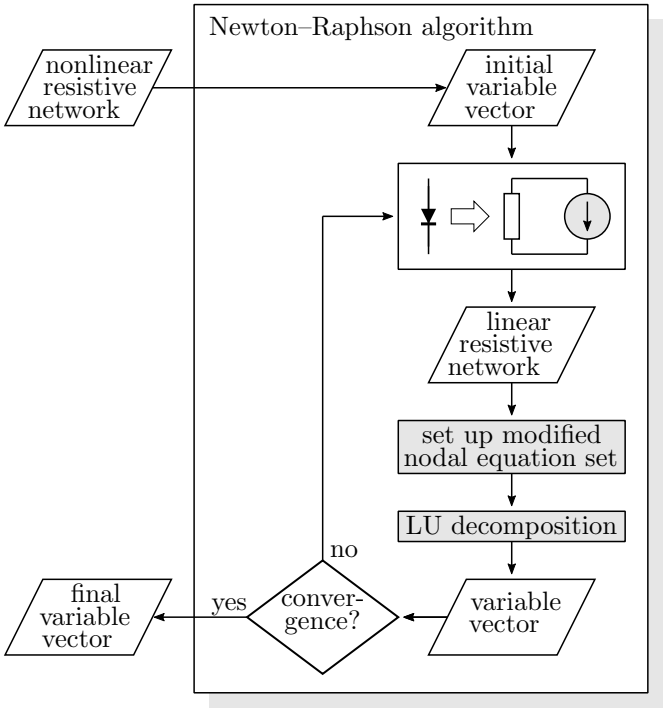

Figure 21: Newton–Raphson algorithm.

Before any circuit equations are formulated, all branches containing nonlinear devices are transformed according to (73); that is, the nonlinear relations are replaced with respective tangents. In terms of devices, the tangent is realized by the parallel combination of an independent current source and a conductance. After replacing all nonlinearities, we obtain a linear resistive network, for which an equation set is formulated, as described in Section 2, and solved exactly, as explained in Section 3.

Although we have not yet discussed dynamic devices, we can now consider the operating point and the DC sweep analysis in SPICE OPUS. In both cases we are interested in steady state circuit solutions. By definition, after all transients die out, the voltages across inductors and the currents through capacitors

become equal to zero. Consequently, branches containing capacitances are replaced with independent zero current sources while inductances are replaced with independent zero voltage sources. In this way, we get a nonlinear resistive network which can be solved by the Newton–Raphson algorithm, as depicted in figure 22.

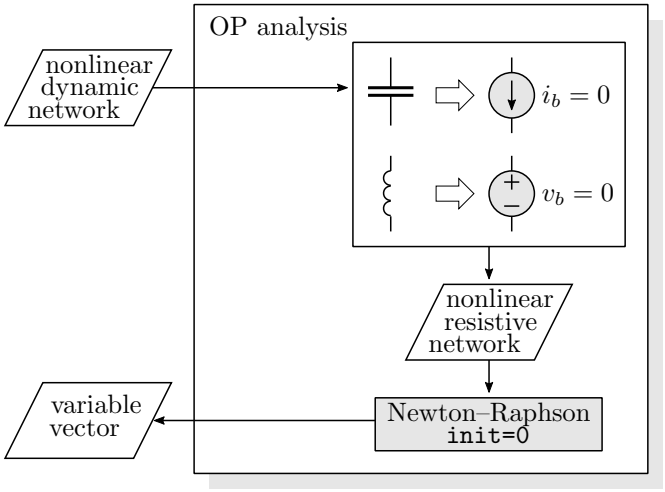

Figure 22: Operating point analysis.

This is the program flow on SPICE OPUS when running the operating point analysis. The DC sweep is also very similar. It consists of a series of operating point calculations while one or two sources are varied. The individual steps in the sweep are theoretically independent operating point anlyses, but SPICE OPUS takes advantage of the fact that adjacent steps in the sweep should produce adjacent operating point solutions. So the solution of each sweep step is wisely used as the initial variable vector in figure 22 for the next Newton–Raphson iteration. In other words, the initial iteration $k = 0$ in (73) is the result of the previous run. In this way the number of Newton–Raphson iterations is greatly reduced. Formally, the program flow of a DC sweep analysis is shown in figure 23.

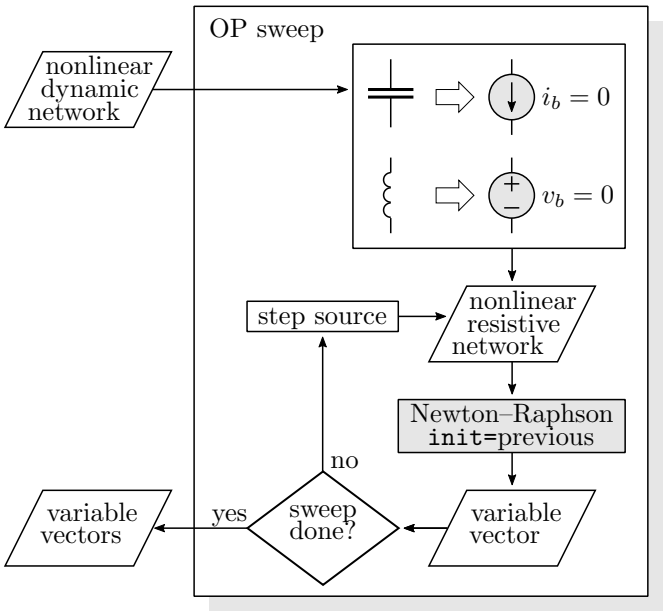

Figure 23: DC sweep analysis.

So far we have succeeded in solving nonlinear circuits by iteratively solving a linearized transformation until the variable vectors of two consecutive iterations differ by less than a predefined value.

A number of open questions still remain, like how to determine the initial Newton–Raphson iteration, how to detect convergence or the lack thereof, and what to do in cases of divergence.

4.3 Convergence Detection

We have already mentioned that the Newton–Raphson algorithm is stopped if two consecutive iterations in (72) are close enough. In fact, SPICE OPUS employs a more advanced convergence detection mechanism.

There are three simulator parameters involved in the convergence detection cirteria with the default values $\texttt{abstol} = 10^{-12}$, $\texttt{reltol} = 10^{-3}$, and $\texttt{vntol} = 10^{-6}$. SPICE OPUS actually checks not only the difference between iteration k and $k+1$ but also the one before, $k-1$. The latest iteration pair must satisfy

$$\begin{aligned} |v_{n_i}^{(k+1)} - v_{n_i}^{(k)}| &\leq \texttt{reltol} \cdot \max(|v_{n_i}^{(k+1)}|, |v_{n_i}^{(k)}|) + \texttt{vntol} \\ |i_{b1_i}^{(k+1)} - i_{b1_i}^{(k)}| &\leq \texttt{reltol} \cdot \max(|i_{b1_i}^{(k+1)}|, |i_{b1_i}^{(k)}|) + \texttt{abstol}. \end{aligned} \tag{74}$$

All nodal voltage differences must be small enough in a relative as well as an absolute sense. Moreover, all current defined branches (31) must fulfill the same criteria. The same applies for the previous iteration pair,

$$\begin{aligned} |v_{n_i}^{(k)} - v_{n_i}^{(k-1)}| &\leq \texttt{reltol} \cdot \max(|v_{n_i}^{(k)}|, |v_{n_i}^{(k-1)}|) + \texttt{vntol} \\ |i_{b1_i}^{(k)} - i_{b1_i}^{(k-1)}| &\leq \texttt{reltol} \cdot \max(|i_{b1_i}^{(k)}|, |i_{b1_i}^{(k-1)}|) + \texttt{abstol}. \end{aligned} \tag{75}$$

After all the above criteria are met, there still is one more check left. The three points form a triangle in a multidimensional space. We require the angle at the second point to be less or equal to 90°. This assures that the iteration series is in a sort of rotation around the solution, as illustrated in figure 24.

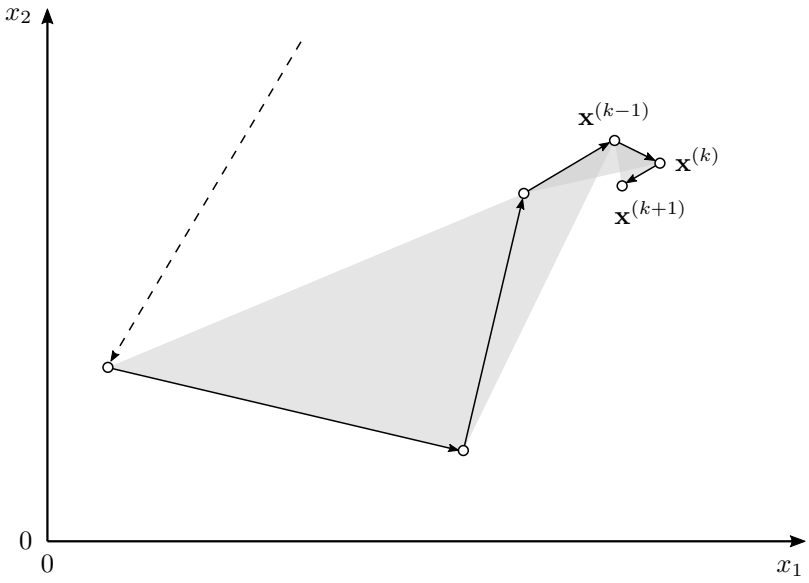

Figure 24: Newton–Raphson convergence detection.

Formally, Pythagoras theorem can be used to ensure an acute angle at $\mathbf{x}^{(k)}$

$$\begin{aligned} |\mathbf{v}_n^{(k+1)} - \mathbf{v}_n^{(k-1)}| &\leq \sqrt{|\mathbf{v}_n^{(k)} - \mathbf{v}_n^{(k-1)}|^2 + |\mathbf{v}_n^{(k+1)} - \mathbf{v}_n^{(k)}|^2} \\ |\mathbf{i}_{b1}^{(k+1)} - \mathbf{i}_{b1}^{(k-1)}| &\leq \sqrt{|\mathbf{i}_{b1}^{(k)} - \mathbf{i}_{b1}^{(k-1)}|^2 + |\mathbf{i}_{b1}^{(k+1)} - \mathbf{i}_{b1}^{(k)}|^2}. \end{aligned} \tag{76}$$

This complex convergence detection scheme is necessary, because sometimes two adjacent iteration points can be very close to each other only to drift apart again. By defining the simulator parameter `noconviter` one can switch off the double convergence check, but without a very special reason the user should not change any of the simulator parameters that control convergence.

Basically, there are two things that can go wrong in connection with convergence. Although we have chosen the very stable and robust Newton–Raphson iteration algorithm in Section 4.2, there still is no guarantee of convergence. Large numbers of transistors with their exponential characteristics, interconnected in a certain way, can cause havoc in any iteration algorithm. The iterations either oscillate instead of zooming in on the solution or they even diverge. In both cases the maximum iteration limit is exceeded `itl1` = 100 with still no convergence detected.

The other problem arises when at some iteration in Figure 21 the LU decomposition fails. This can happen for a number of reasons, e.g., diverging iterations can reach absurd numerical values. Whatever the reason, the alarm always goes off in the pivoting algorithm when at some step it cannot find a single coefficient that would satisfy (56).

But this is not the end of the Newton–Raphson iteration algorithm in SPICE OPUS; it is rather the beginning of a whole series of tricks. The convergence helpers described in the following section are among the most important quality marks of any circuit simulator.

4.4 Automatic Convergence Helpers

SPICE OPUS does not give up easily. There are six tricks built into SPICE OPUS, and all of them are activated automatically. Normally the user does not have to intervene, and the average user does not even have to know about them.

All six automatic convergence helpers rely on two assumptions. First, at least one solution of the circuit must exist. Obviously, no convergence helper can find a nonexisting solution. A circuit with no solution is not unheard of. Very often an erroneous netlist causes singularities; for instance, two parallel voltage sources, or an ideal transformer generating a floating subcircuit. In these cases all convergence helpers will fail.

The second assumption is that the proximity of the solution is convergent for the Newton–Raphson algorithm. In other words, if we can find an initial vector $k = 0$ for (72) which is close enough to the final solution, the iteration series will converge.

So, it's really all about finding that crucial proximity. We know that (64) is guaranteeing a zero gradient $f'(x^{(*)}) = 0$ at the solution, but that does not mean that $f(x)$ can't be very steep further away from the solution as illustrated in figure 25.

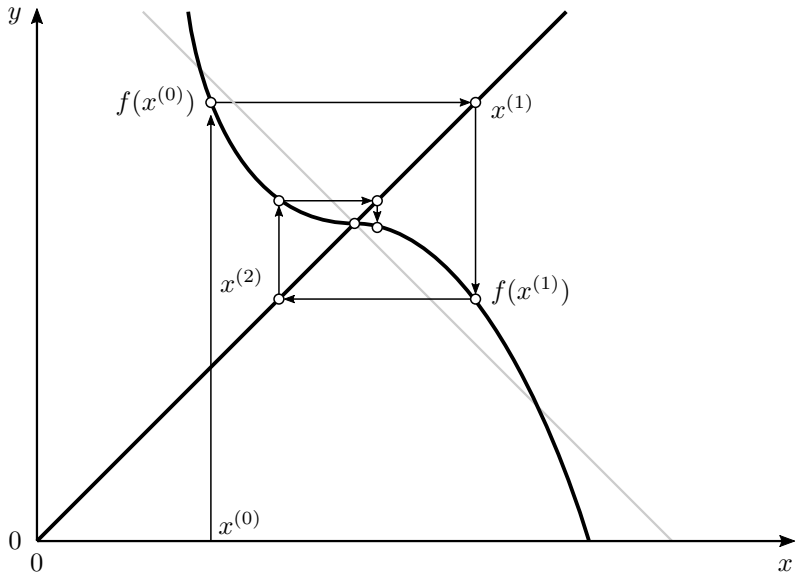

Figure 25: Newton–Raphson convergence.

The perpendicular gray line marks the Fixed Point Iteration convergence limit as defined by (62). Any iterations venturing outside the intersection of $f(x)$ and the gray line risk spiraling into divergence. The inital $x^{(0)}$ is just

barely squeaking by. However, just a little more to the left and the goose is cooked as in figure 26.

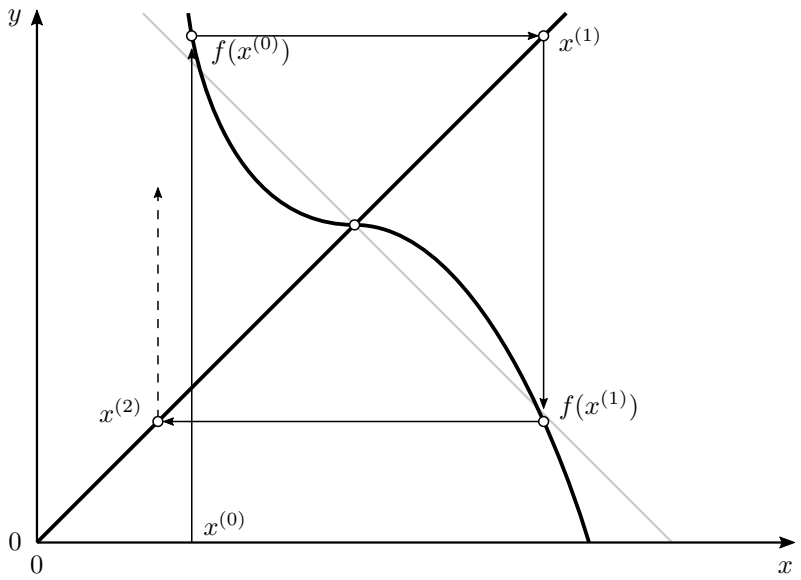

Figure 26: Newton–Raphson divergence.

Normally, Newton–Raphson iterations start from a zero vector. Whenever divergent iterations are detected, convergence helper algorithms are used to find a suitable starting point.

Junction Voltage Limitation

This is a very crude measure that prevents Newton–Raphson iterations from running amok. Especially in the beginning of an iteration series, the exponential nature of p-n junctions can cause extreme numerical values going right to the limit of floating point numbers. Sometimes the limit is even violated, causing exceptions in the mathematical package.

In order to prevent these rare occasions, SPICE OPUS sets an upper limit on voltages across a p-n junction regardless of their polarity. By changing the simulator parameter `voltagelimit`, the default limit of 10^{30}V can even be overridden.

So after each new iteration in (72), the branch voltages $\mathbf{v}_b^{(k+1)}$ containing p-n junctions are checked. Any value above or below `voltagelimit` is reset to the respective limit,

$$v_{b_i}^{(k+1)} = \begin{cases} v_{b_i}^{(k+1)}; & \forall\, |v_{b_i}^{(k+1)}| \leq \texttt{voltagelimit} \\ -\texttt{voltagelimit}; & \forall\, v_{b_i}^{(k+1)} < -\texttt{voltagelimit} \\ \texttt{voltagelimit}; & \forall\, v_{n_i}^{(k+1)} > \texttt{voltagelimit}. \end{cases} \tag{77}$$

Junction voltage limitation not only prevents numerical exceptions—in most cases it also speeds up convergence by limiting the iteration space. Normally the default value of `voltagelimit` does not need to be changed.

Damping Newton–Raphson Steps

As we have mentioned, one of the things that can prevent convergence is an oscillating iteration sequence as illustrated in figure 26. Instead of quickly zooming in on the solution, the iterations start oscillating around it. A major problem is the detection of oscillations as they are not necessarily periodic. Also note that we are looking for oscillating iterations in (72), where each vector has $2b + n$ components. In the worst scenario, we need to detect a stochastic limit cycle in a $2b + n$ dimensional space. This is not a trivial problem, and we do not want to spend more computational effort on the detection of oscillations than on solving the problem itself.

Let us put aside the question of detection and look at the solution first. An oscillation can also be seen as an infinite series of iterates in the proximity of the solution. Intuitively, one would limit the iteration steps. This approach is termed the damped Newton–Raphson algorithm, whatever limiting strategy is employed.

In SPICE OPUS the simulator parameter `sollim` (default value is 10) regulates the damping of iteration steps. The damping algorithm is based on the same voltage and current tolerance thresholds as the convergence detection. We first define the thresholds for iteration $k + 1$ (note that i is the index of a component in the solution vector $\mathbf{v}_n, \mathbf{i}_{b1}$),

$$\begin{aligned} v_{s_i}^{(k+1)} &= \texttt{reltol} \cdot \max(|v_{n_i}^{(k+1)}|, |v_{n_i}^{(k)}|) + \texttt{vntol} \\ i_{s_i}^{(k+1)} &= \texttt{reltol} \cdot \max(|i_{b1_i}^{(k+1)}|, |i_{b1_i}^{(k)}|) + \texttt{abstol}. \end{aligned} \tag{78}$$

With damping turned on, the Newton–Raphson algorithm still runs exactly as defined in (72). However, before feeding the variable vector back to the netlist, all variables which have stepped over their respective threshold (78) are limited. The Newton–Raphson flow now has one more transformation in the loop, as can be seen in figure 27.

Actually, the steps are limited to a fraction determined by `sollim` of the respective threshold. The damping of voltages is expressed in the following equation:

$$v_{n_i}^{(k+1)} = \begin{cases} v_{n_i}^{(k+1)}; & \forall\ |v_{n_i}^{(k+1)} - v_{n_i}^{(k)}| \le v_{s_i}^{(k+1)} \\ v_{n_i}^{(k)} - \frac{v_{s_i}^{(k+1)}}{\texttt{sollim}}; & \forall\ v_{n_i}^{(k+1)} - v_{n_i}^{(k)} < -v_{s_i}^{(k+1)} \\ v_{n_i}^{(k)} + \frac{v_{s_i}^{(k+1)}}{\texttt{sollim}}; & \forall\ v_{n_i}^{(k+1)} - v_{n_i}^{(k)} > v_{s_i}^{(k+1)}. \end{cases} \tag{79}$$

There are three possibilities. If the variable step was already within its convergence tolerance, nothing happens. If the new nodal voltage $v_{n_i}^{(k+1)}$ is lower than its predecessor $v_{n_i}^{(k)}$ by more than the threshold, then this step is reduced to a fraction of the threshold. An excessively long positive voltage step is limited accordingly.

Similarly, all branch currents that occur in the set of modified nodal equations are damped,

$$i_{b1_i}^{(k+1)} = \begin{cases} i_{b1_i}^{(k+1)}; & \forall\ |i_{b1_i}^{(k+1)} - i_{b1_i}^{(k)}| \le i_{s_i}^{(k+1)} \\ i_{b1_i}^{(k)} - \frac{i_{s_i}^{(k+1)}}{\texttt{sollim}}; & \forall\ i_{b1_i}^{(k+1)} - i_{b1_i}^{(k)} < -i_{s_i}^{(k+1)} \\ i_{b1_i}^{(k)} + \frac{i_{s_i}^{(k+1)}}{\texttt{sollim}}; & \forall\ i_{b1_i}^{(k+1)} - i_{b1_i}^{(k)} > i_{s_i}^{(k+1)}. \end{cases} \tag{80}$$

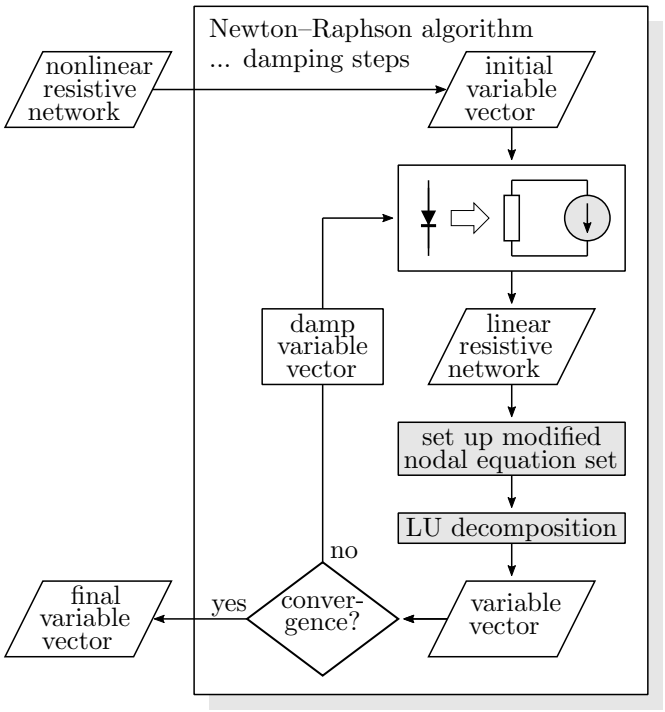

Figure 27: Newton–Raphson iterations with damped steps.

With a default damping fraction of 10, it is clear that the damped steps will be extremely tiny. This ensures robust convergence at the expense of many more iterations. The normal iteration limit `itl1` with its default value of 100 would thus be exceeded immediately, so SPICE OPUS has a multiplier simulator parameter `sollimiter` with the default value 10. Therefore, whenever damping is turned on, the iteration limit actually becomes `itl1` · `sollimiter`.

The damping factor `sollim`, which is actually defining a fraction of the threshold, would normally be greater than one. However, SPICE OPUS also accepts values smaller than one, in which case the steps are limited to more than the tolerance threshold. In this case the damped step is additionally limited by the original step. The idea is to reduce the original Newton–Raphson steps rather than enlarging them.

As we shall see in the following sections, the damped Newton–Raphson algorithm is used when the GMIN stepping and source value stepping algorithms fail to produce a solution. This removes the need for a sophisticated oscillation detection algorithm and works well in practice.

GMIN Stepping

The idea behind this convergence helper is strikingly simple and efficient. SPICE OPUS adds a conductance between each node and ground. In this way convergence is greatly improved, at the expense of not solving the original circuit, but rather a transformed version. Intuitively, it is clear that any circuit which causes problems for the Newton–Raphson algorithm can be tamed by adding

large conductances between the nodes.

Formally this can be verified. As discussed in Section 3.1, an arbitrary admittance y_k between nodes p and m contributes to four coefficients in the sparse equation set (37), namely to the diagonal $g_{p,p} = +y_k$, $g_{m,m} = +y_k$ and the anti-diagonal $g_{p,m} = -y_k$, $g_{m,p} = -y_k$. In our case one terminal is always the ground node, $m = 0$. Because the reference node is reduced from the incidence matrix, only one contribution is left on the diagonal, $g_{p,p} = +y_k$.

By adding a huge conductance y_k to each node, we get a diagonal with large terms in the equation set. However, a large and linear diagonal guarantees a solution. We have sure convergence because the linearity outweighs all nonlinear problems and the strong diagonal ensures perfect pivots for (56). The situation is illustrated in figure 28.

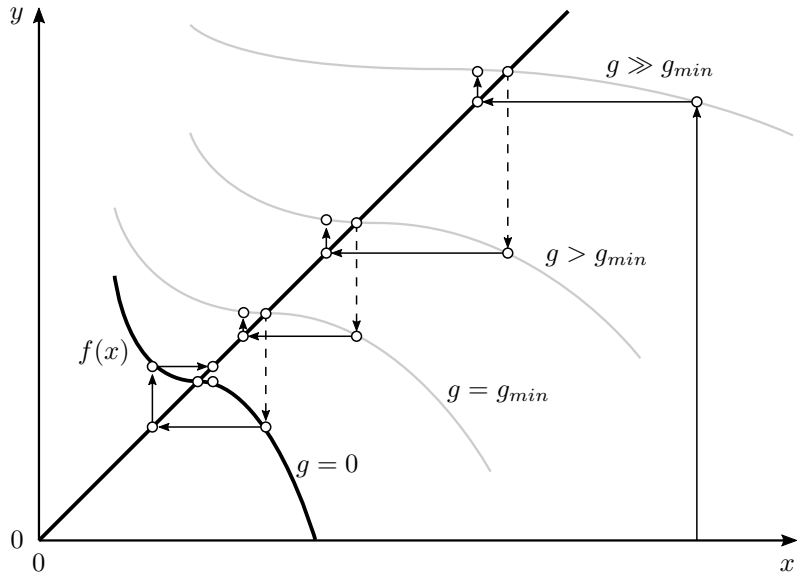

Figure 28: Newton–Raphson GMIN stepping.

With large enough conductances we always get a solution. However, this is the solution of the transformed circuit, whereas we want the solution of the original circuit. The trick is to gradually fade out the additional conductances (and that is where the name GMIN stepping comes from). In each step the conductances are reduced and the Newton–Raphson algorithm is run from the previous solution. Eventually, the conductances are stepped down to zero, rendering the original circuit. This procedure, called GMIN stepping, is very effective although also very computationally intensive. Note that each step involves one complete Newton–Raphson run.

So far we have conveyed just the general idea. Now let us be more specific. There are three simulator parameters with respective default values controlling this process in SPICE OPUS: the minimal conductance for the AC and TRAN analysis, `gmin` $= 10^{-12}$, the minimal conductance for DC analysis, `gmindc` $= 10^{-12}$, and the maximal number of GMIN steps before giving up, `gminsteps` $= 10$.

The simplest way would be to start with an extremely large conductance g

and then step it down gradually. However, it is uneconomical to start down-stepping from a needlessly high value. Therefore, SPICE OPUS starts with the default minimal conductance g_{min} and steps it up in huge three decade increments until convergence is reached. In the second part the conductance is gradually stepped down, always starting the Newton–Raphson iteration from the previous solution. If everything works out, the conductance g eventually falls below g_{min} and is finally completely removed. The three steps are illustrated in figure 29.

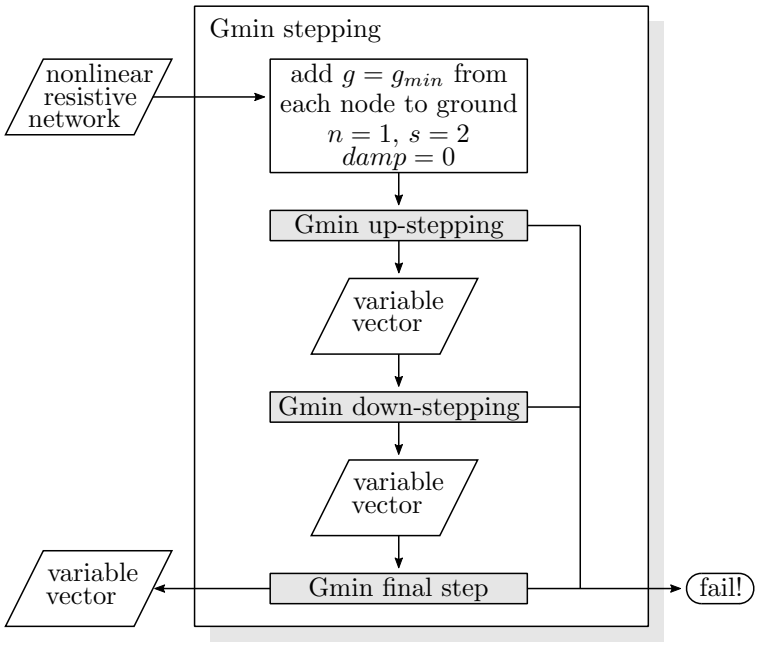

Figure 29: GMIN stepping algorithm.

The up-stepping part is simple. SPICE OPUS starts with $g = g_{min}$ and multiples it by 1000 for each unsuccessful Newton–Raphson run, as seen in figure 30.

The overall step counter n is initialized at the beginning and is limiting the total number of Newton–Raphson runs to the predefined value $n_{max} =$ `gminsteps`. Whenever this number is reached, the entire GMIN stepping procedure is considered to have failed.

The up-stepping is fast and should take only a few runs. However, singular circuits are bound to fail already during the up-stepping part.

Given a successful up-stepping, we have our first solution. From this point on we need to step-down the conductance. This process is fragile because the individual steps always need to be close enough not to break convergence. The algorithm is depicted in figure 31.

Each successful step increases the stepping fraction s, speeding up the down-stepping. However, a too large step is likely to break the convergence chain, so the algorithm has to revert to the previous conductance $g = g_p$ and accordingly reduce s. This adaptive step sizing is computationally very efficient, but can lead to extremely small step sizes, in which case Newton–Raphson damping is

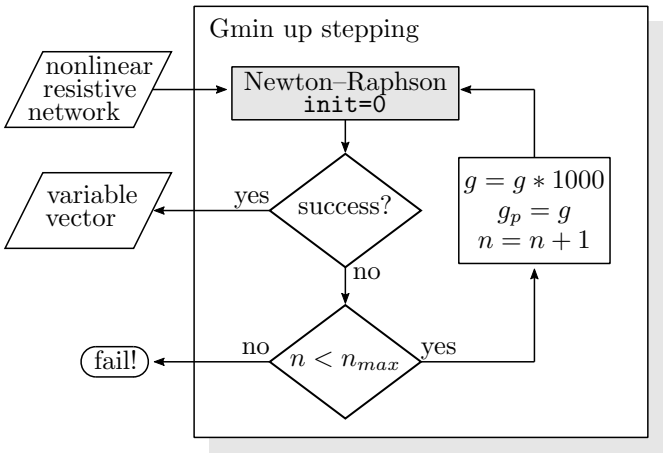

Figure 30: Up-stepping part of GMIN stepping.

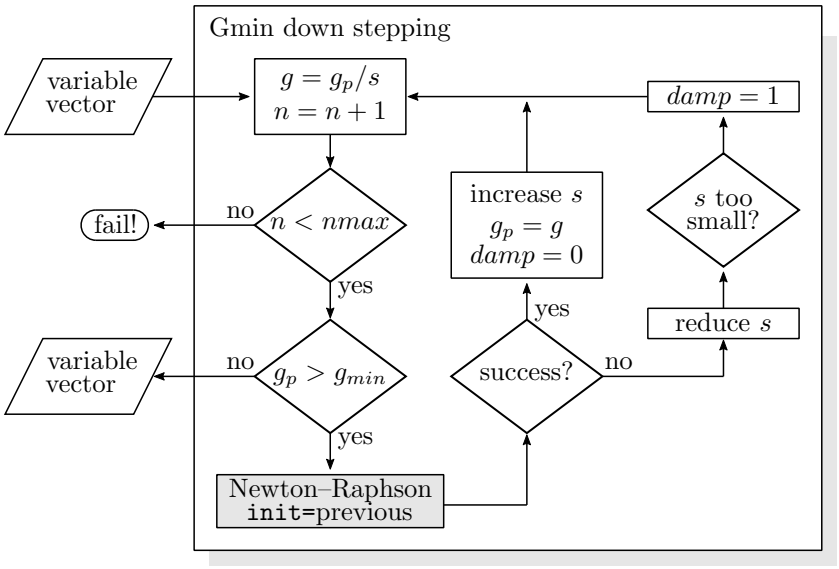

Figure 31: Down-stepping part of GMIN stepping.

turned on automatically.

The fact that at some point a very small step in g is causing a loss of convergence does not necessarily mean that the Newton–Raphson algorithm is experiencing oscillating iterations! Putting the Newton–Raphson algorithm in damping mode is based on pure heuristic reasoning. It just turns out that this measure very often works.

In the case of a successful down-stepping without exceeding the maximal number of Newton–Raphson iterations, we have convergence for some $g < g_{min}$. With the default value of $g_{min} = 10^{-12}$ this is almost our original circuit. Only a very small resistive network is still superimposed on each node. So the final part of the algorithm is to completely remove g from the network and do one

final Newton–Raphson series starting from the last known solution, as shown in figure 32.

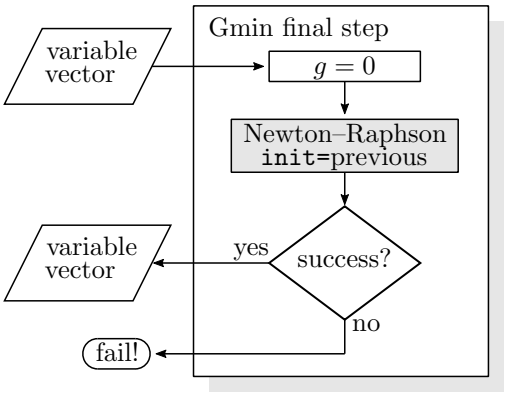

Figure 32: Final part of GMIN stepping.

One could argue that this part is not necessary as the difference between having $g = 10^{-12}$ and $g = 0$ is irrelevant. With normal circuits this is the case, but some singularities in the circuits can prevent convergence in this very last part. For instance, suppose that there is no DC path from some node to ground. By removing the smallest of g, we are facing a dangling node with a theoretically undeterminable voltage.

Looking back at the entire GMIN stepping procedure in figure 29, it is obvious that the default maximal Newton–Raphson count of `gminsteps` = 10 is rather low. In fact, the default is only meant to do a quick trial of GMIN stepping, producing successful results only for relatively simple circuits. In difficult cases GMIN stepping will run out of steps and will report this to the user. In this situation one should first review the circuit for systematic singularities before increasing `gminsteps` because the resulting GMIN stepping runs may take a long time to complete.

5 Frequency-Domain Analysis

There is one more step we need to make towards simulating real circuits. We have to find a way to include dynamic devices. We will not worry about nonlinear energy storing devices as they can be represented by a combination of a nonlinear controlled source and either a linear capacitor or inductor. So we are really talking about introducing simple linear capacitance and inductances into branch relations (5).

There are two ways to do this, either in the frequency or the time domain, each having its respective advantages and disadvantages.

5.1 Small Signal (AC) Analysis

Small signal response analysis (also referred to as frequency-domain analysis, AC analysis, or AC sweep), is based on the Fourier transformation of the circuit's equations. Unfortunately, we have to assume either linear circuits or small signals. As we are primarily interested in nonlinear circuits, we will first discuss the notion of small signals.

Observe any nonlinear analog circuit in the steady state. From previous sections we know how to obtain an operating point solution. Now imagine that some independent source between two nodes starts oscillating with an arbitrary frequency but with an extremely small magnitude. Unless the circuit has some discontinuity in the immediate proximity of the operating point, this tiny oscillation will propagate linearly through the entire circuit. In other words, the circuit can be considered linear in the neighborhood of the operating point, provided that the neighborhood is small enough. With this assumption small signal response analysis needs to calculate only the ratios of magnitudes and the phase delays to completely describe any sinusoidal signal.

This is done with complex arithmetic, where all resistive devices are represented by admittances equal to the gradients in the operating point. These gradients are already computed as part of each Newton–Raphson iteration, as shown in figure 20. Capacitances and inductances become imaginary functions of the frequency $y_c = j\omega C$ and $y_l = \frac{1}{j\omega L}$, respectively. Consequently, independent voltage sources become short circuits, and independent current sources are replaced by open circuits. With these replacements we are facing a simple linear resistive network, but not like that described by (37). The only differences are the parametric frequency dependence and the complex arithmetic. The latter is no problem; the procedure is exactly equivalent to that described in Section 3. The frequency dependence is usually solved by sweeping ω over a relevant interval.

The program flow of the small signal analysis in SPICE OPUS is summarized in figure 33.

New users often have misunderstandings regarding the AC analysis. We clarify some of them here.

- The linearized resistive network in figure 33 needs at least one AC excitation. So unless we have defined at least one AC source, the result will be zero, because the right-hand side of (37) is all zero.
- The AC excitation magnitude and phase basically are arbitrary, because we have a linear system where the superposition theorem applies. There-

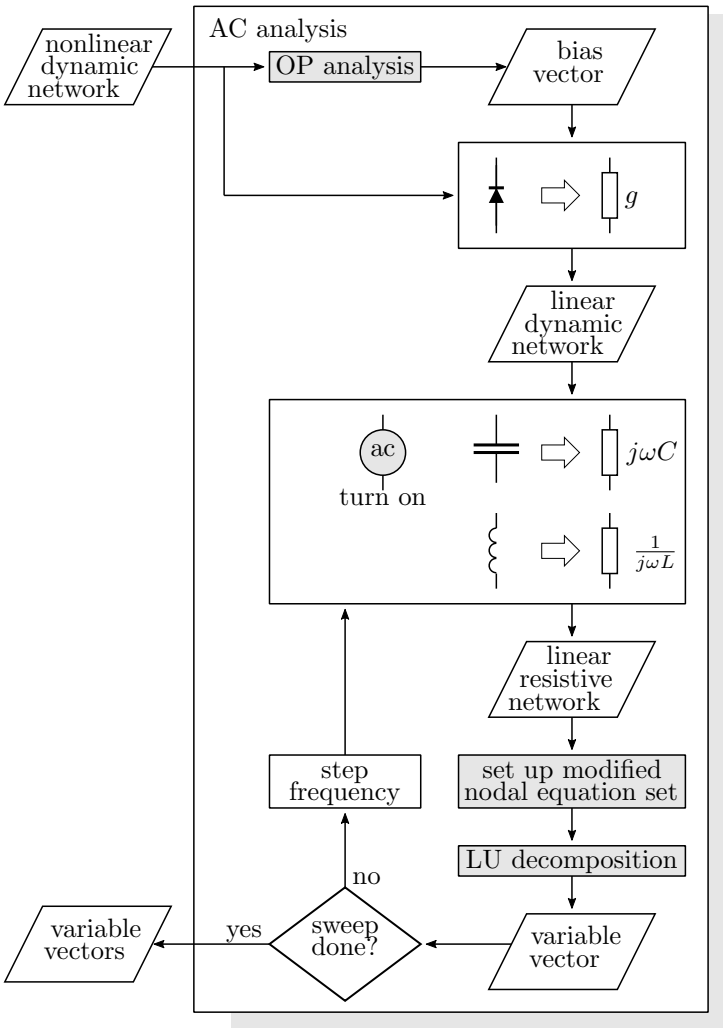

Figure 33: Small signal response analysis.

fore, it makes sense to chose unit magnitude and zero phase. In this way the results can be directly interpreted as ratios. Do not be disturbed by SPICE OPUS reporting 1MV at the output of an operational amplifier. Remember to interpret this as the amplification gain of a unit excitation.

- You can specify more than just one AC excitation source, in which case all excitation sources are swept with the same frequency. Thus, any output will be the product of the combined excitation. This questions the purpose of the entire analysis, but in some special cases one might make use of this possibility.

- Unknown x's in (37) should be considered in polar form where the absolute values signify the magnitude and the angle represents the relative phase shift. This puts some natural limits on the phase response calculation because the complex number space only allows angles up to 2π

radians. On the other hand, especially with modern high gain operational amplifiers, we experience phase shifts well outside the $\pm 180°$ range. The small signal analysis will still deliver correct results; however, the phase will be wrapped. For instance, $\varphi = 14°$ should actually be interpreted as $\varphi = 14° + k \cdot 360°$, where k can be any integer including zero. As circuit designers we can guess the value of k for each specific case. Nevertheless, there is the special NUTMEG command `unwrap()` which attempts to make an intelligent guess of k.

- Small signal analysis results depend heavily on the operating point of the circuit. Always double-check the resulting operating point before focusing on the AC analysis data.

One could say that small signal response analysis reveals the differential properties of a circuit in the neighborhood of a specific operating point. With a different bias (resulting in a different operating point), the gradients of the characteristics of nonlinear resistive devices change, and the AC analysis might yield completely different results. Despite the fact that the small signal analysis results in much information on the differential behavior of the circuit, it does not provide us with a picture of the circuit's nonlinear behavior. Nevertheless, the analysis is fast and almost never results in convergence problems. Convergence problems are usually the result of the initial operating point analysis that precedes AC analysis.

5.2 Small Signal (NOISE) Analysis

Small signal noise analysis is another small signal analysis derivative (also in the frequency domain), which is very important and is being heavily used in modern chip design. The goal is to find the total noise spectrum produced by a network of individual electronic devices. The noise level in electronic circuits puts a major constraint on the dynamic range, so small signal noise analysis is crucial to modern chip design. It is based on the knowledge of each individual noise source. In theory there are three noise sources: thermal noise from every resistance as well as shot noise and flicker noise from every semiconductor device. Noise sources can depend on the frequency (e.g., flicker noise) and on the operating point (shot noise, flicker noise).

A noise signal $v(t)$ can be represented by its power spectrum density $S^+_{vv}(f)$. If the signal is a node voltage, the corresponding unit for the signal is V and the unit for the power spectrum density is V^2/Hz. If the signal is filtered by an ideal bandpass filter with frequency range $f_1 \leq f \leq f_2$, a new noise signal $v'(t)$ is obtained. By integrating the power spectrum density $S^+_{vv}(f)$ from f_1 to f_2, the mean square of $v'(t)$ is obtained,

$$\lim_{\tau \to \infty} \frac{1}{2\tau} \int_{-\tau}^{\tau} |v'(t)|^2 dt = \int_{f_1}^{f_2} S^+_{vv}(f) df. \tag{81}$$

Every resistor and every semiconductor device is a source of noise. Because the noise is a small signal, it is assumed that the circuit is linear within the noise magnitude around the circuit's operating point. Therefore, the circuit can be linearized and an ordinary small signal response analysis applied to obtain the noise characteristics of the linearized circuit as in figure 34.

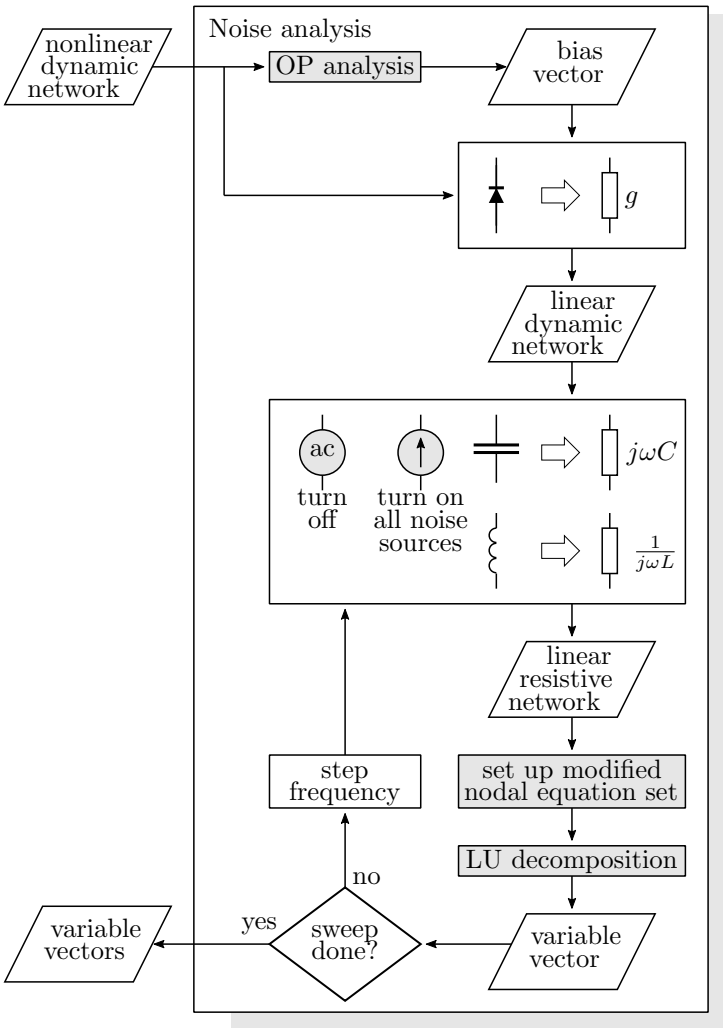

Figure 34: The block diagram of the noise analysis.

The noise introduced by a device is represented by one or more noise current sources in the AC small signal model of the device. The phase of such noise current sources is 0, whereas their magnitude generally depends on the frequency and the circuit's operating point.

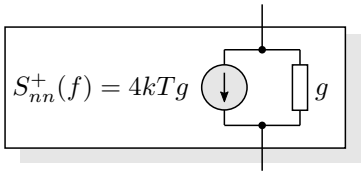

Figure 35: Resistor noise model.

Figure 35 depicts the noise model of a resistor, where k is the Boltzmann con-

stant, T is the absolute temperature, $g = 1/R$ is the conductance of the resistor, and $S^+_{nn}(f)$ is the current noise power spectrum density (A^2/Hz) representing the thermal noise.

Generally, there are n_{noise} noise current sources in the circuit. Let $S^+_{\mathrm{nn},i}(f)$ denote the power spectrum density of i-th noise source. Because individual noise sources are uncorrelated, the power spectrum density of the output noise $S^+_{\mathrm{out,out}}(f)$ is the sum of power spectrum density contributions from individual noise sources,

$$S_{\mathrm{out,out}}(f) = \sum_{i=1}^{n_{noise}} |A_i(f)|^2 S^+_{\mathrm{nn},i}(f), \tag{82}$$

where $A_i(f)$ is the transfer function from the i-th noise source to the output at frequency f.

The equivalent input noise is defined via a noiseless circuit (one in which the circuit elements generate no noise). It is the noise signal that must enter a noiseless circuit at its input in order to produce the same output noise power spectrum density as one would get with a noisy circuit and the input signal equal to zero.

The equivalent input noise power spectrum density is obtained by dividing the ouput noise power spectrum density by the squared magnitude of the circuit's transfer function from the input to the output. This transfer function can be obtained with an ordinary small signal response analysis.

6 Time-Domain (TRAN) Analysis

Time-domain analysis is the most computationally intensive approach, but also the most realistic in the sense that real circuits actually perform in the time domain. Besides the dynamic excitation from independent sources it is the reactances that influence the transient behavior of any circuit. In other words, we need to find a way to supplement branch equations (5) with equations for capacitors and inductors.

6.1 Backward Euler Integration

The time-domain model of a capacitor (inductor) is given by the following equation.

$$i_c(t) = C\frac{\mathrm{d}v_c(t)}{\mathrm{d}t}, \quad v_l(t) = L\frac{\mathrm{d}i_l(t)}{\mathrm{d}t}. \tag{83}$$

A direct inclusion would turn coefficient matrices Y and Z into operators (e.g., terms of the form $\frac{\mathrm{d}}{\mathrm{d}t}$ would appear as matrix elements) and the linear equation set (8) into a set of ordinary differential equations. In order to avoid this, we use the trick of partitioning time into small steps. The basic idea can be considered a type of finite element approach to time-domain analysis.

The simplest way is to assume constant currents through capacitors and constant voltages across inductors. First, let us examine the consequences on capacitors in figure 36.

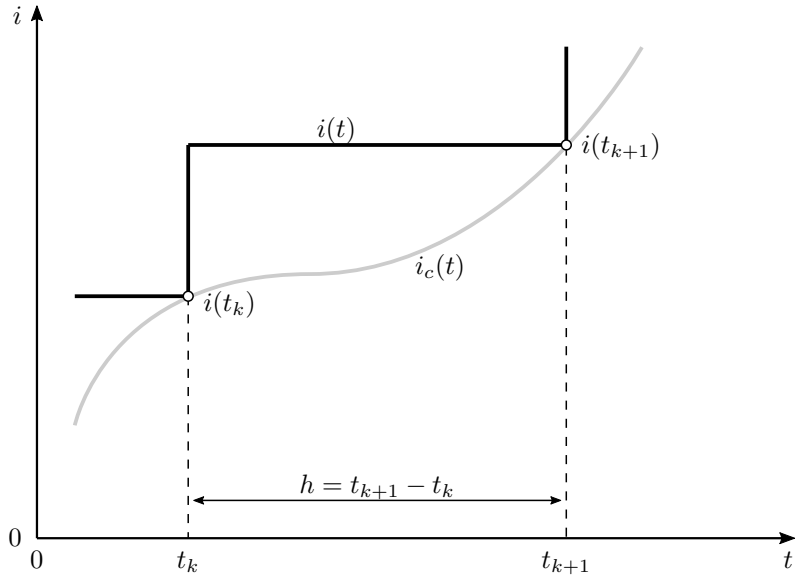

Figure 36: Backward Euler integration.

We are observing the time interval $[t_k, t_{k+1}]$. Inside this time interval we assume a constant capacitor current,

$$i(t) = i(t_{k+1}). \tag{84}$$

By substituting this expression for the current in the integral form of (83) and solving the simple integral, we get

$$v(t) = \frac{1}{C}\int i(t)\mathrm{d}t = \frac{1}{C}\int i(t_{k+1})\mathrm{d}t = \frac{i(t_{k+1})}{C}t. \tag{85}$$

As expected, the constant current through the capacitance causes a linear voltage ramp across it. However, we are not as interested in the transient inside the time slice as in the variable increments of the time step, that is the difference between variable values at times t_k and t_{k+1},

$$v(t_{k+1}) - v(t_k) = \frac{i(t_{k+1})}{C}(t_{k+1} - t_k). \tag{86}$$

The time step is usually referred to as $h = t_{k+1} - t_k$. With this substitution we get the following admittance form:

$$i(t_{k+1}) = \frac{C}{h}v(t_{k+1}) - \frac{C}{h}v(t_k). \tag{87}$$

This actually represents an admittance $g_c = \frac{C}{h}$ in parallel with a constant current source $i_c = -\frac{C}{h}v(t_k)$. So by knowing the circuit variables at t_k, we can compute the variables in the next time step t_{k+1} by solving a resistive network where each capacitor has been replaced by a respective g_c and i_c (figure 37).

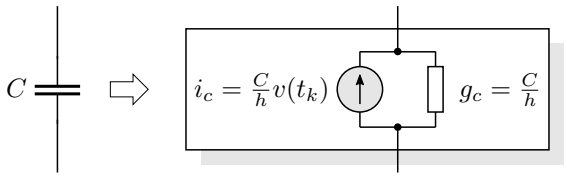

Figure 37: The model of a capacitor in time-domain analysis.

Naturally, the procedure for impedances is exactly dual. We assume a constand voltage across inductances,

$$v(t) = v(t_{k+1}). \tag{88}$$

Next we solve the basic integral

$$i(t) = \frac{1}{L}\int v(t)\mathrm{d}t = \frac{1}{L}\int v(t_{k+1})\mathrm{d}t = \frac{v(t_{k+1})}{L}t \tag{89}$$

and formulate the current step

$$i(t_{k+1}) - i(t_k) = \frac{v(t_{k+1})}{L}(t_{k+1} - t_k). \tag{90}$$

At this point the duality ends, because we need an admittance expression in both cases, so the final time-step equation for the inductors is

$$i(t_{k+1}) = \frac{h}{L}v(t_{k+1}) + i(t_k). \tag{91}$$

In each time step, inductors are thus replaced by the impedance $g_l = \frac{h}{L}$ in parallel with the independent current source $i_l = i(t_k)$.

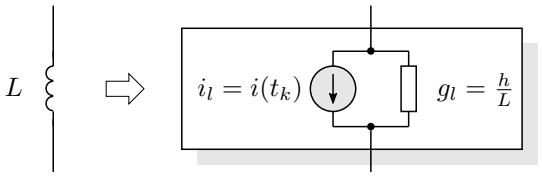

Figure 38: The model of an inductor in time-domain analysis.

In summary, we are able to compute a transient response of a dynamic nonlinear circuit by successively solving resistive nonlinear networks for each individual time step. Actually, we have just reinvented the well-known backward Euler integration algorithm. The program flow is illustrated in figure 39.

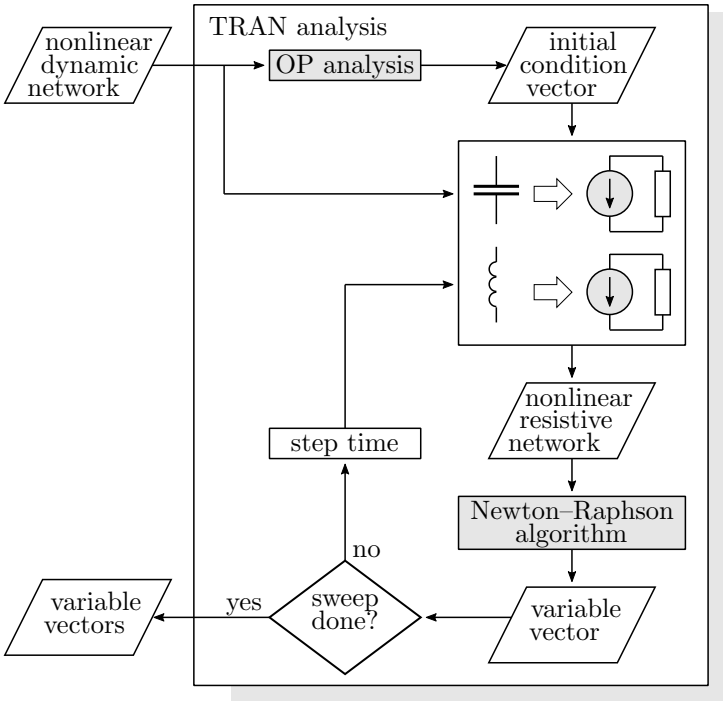

Figure 39: Time-domain transient analysis.

Each step $k + 1$ is computed based on the previous one, k, under the assumption of constant currents through capacitances and constant voltages across inductances. But what about the first step? We need some initialization for the integration algorithm. SPICE OPUS always preforms an operating point analysis prior to any integration steps. The results of the operating point analysis are then used as the initial time step $k = 0$, as depicted in figure 39.

~~~

Before we go into any more details, let us review the basic backward Euler integration on a very simple example. Consider the circuit in figure 40 with a 1K resistor, a 10μF capacitor, and a 5mA step excitation at $t = 0$.
~~~

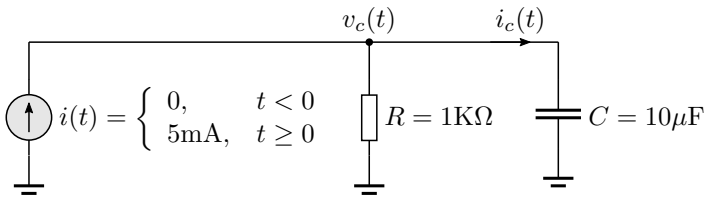

Figure 40: Example circuit demonstrating integration algorithms.

The case is linear and simple enough so that we can guess the transient. We expect a 5mA surge in the capacitor current $i_c(t)$ dying exponentially while the voltage across the capacitor $v_c(t)$ will rise exponentially toward 5V.

The initial operating point is trivial as the excitation at $t = 0$ is zero. The selected time step is intentionally large, $h = 10\text{ms}$, which is actually too large for the circuit time constant. In this way we exaggerate the inexact behavior of the integration algorithm. The observed time interval is 50ms, so we have five integrations steps besides the initial operating point.

The resulting current through the capacitor and the calculated voltage across it are displayed in figures 41 and 42, respectively. The integration steps are clearly marked. Each figure also shows the exact exponential solution for comparison. In both figures we have explicitly represented the backward Euler approximations between the data points, that is, the rectangular current transient and its trapezoidal voltage integration.

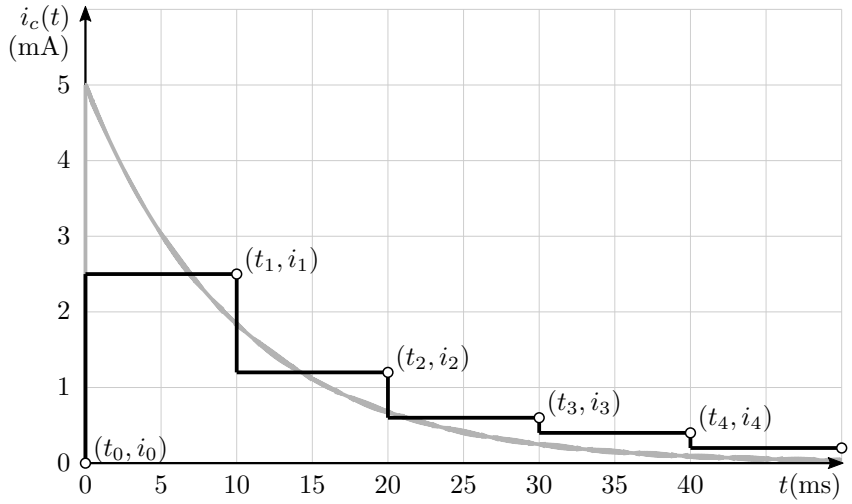

Figure 41: Backward Euler integration current through capacitor.

Let us observe the current step function in figure 41. Because of the huge time step, it is way off the correct solution. The absolutely worst part is the beginning, because the assumption that nothing much is happening inside the time slice is most seriously violated at the beginning. This was to be expected as we have a discontinuous excitation at $t = 0$. Unfortunately, this is a typical scenario in real circuits. So ideally, instead of a constant integration time step h we would rather have an intelligently adapting time step.

There is another interesting observation to be made in figure 41. By considering the fact that each integration step is numerically based on the results of the previous calculations (87) and (91), one would think that the calculation errors will have a cumulative nature. In other words, calculations based on erroneous data cannot be expected to produce results with improved accuracy. However, by looking at the current steps we observe the surprising fact that toward the end of the transient the integration algorithm comes much closer to the correct solution than it started out. This is by no means a rule, but neither is it very exceptional.

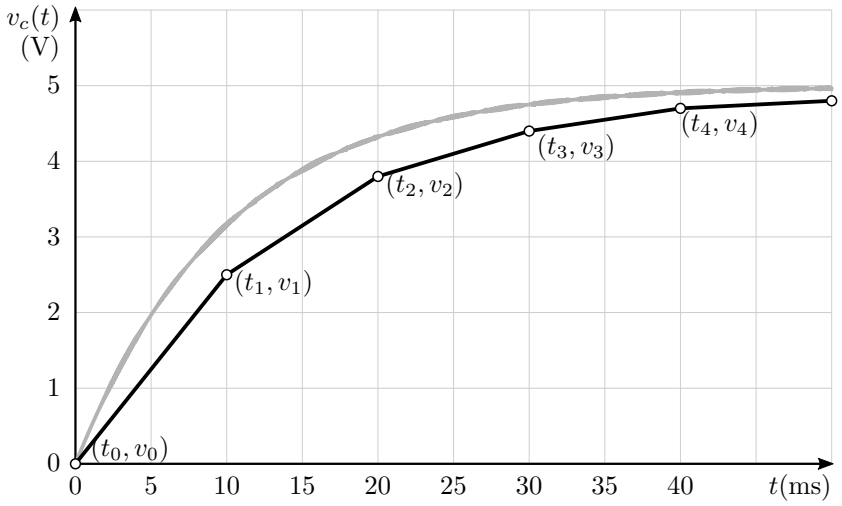

Figure 42: Backward Euler integration voltage across capacitor.

Turning our attention to the voltage transient in figure 42, we find one more typical effect. A rough glance at the characteristics reveals that the voltage transient is more accurate than its current counterpart. If we were observing an inductor instead of the capacitor, we would find the reverse situation. The reason for this is that SPICE controls the numerical error by keeping an eye on the circuit variables that are subject to numerical integration (voltages for capacitors and currents for inductors). Other variables may exhibit larger numerical errors.

With a smaller time step h, the presented backward Euler integration could produce useful results with real circuits.

In SPICE OPUS backward Euler integration is selected by setting the `method` simulator parameter to `trap` and the `maxord` simulator parameter to 1. Besides backward Euler integration, SPICE OPUS also provides more advanced algorithms.

6.2 Trapezoidal Integration

For trapezoidal integration, instead of assuming constant currents through capacitors and constant voltages across inductors with a time slice, we assume respective linear functions. Instead of having to integrate rectangles, we must now integrate a piecewise linear function with each segment having a trapezoidal form as seen in figure 43.

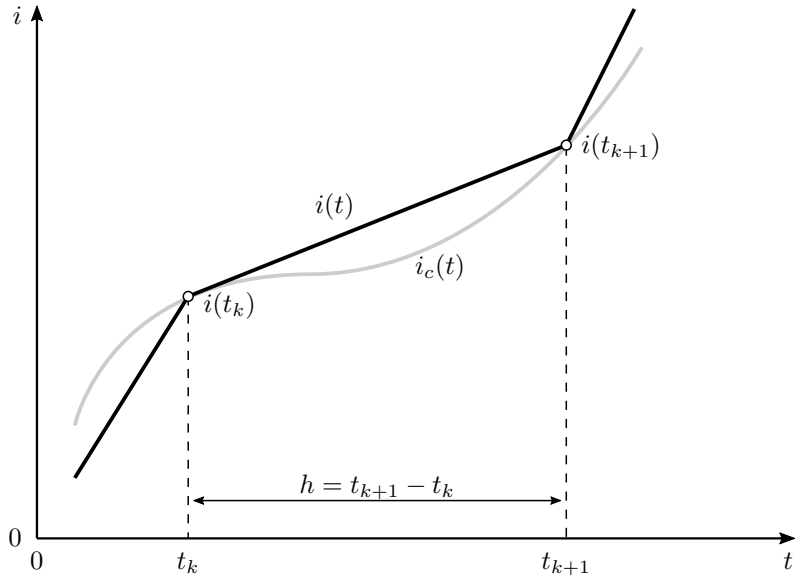

Figure 43: Trapezoidal integration.

We will examine the situation on capacitors first. The expression will be a bit more complicated this time, so we will simplify the notation. We drop the explicit functional dependence on time and use time-step indices directly on current and voltage symbols. So instead of $i(t), i(t_k), v(t_k)$ we will use i, i_k, v_k, respectively.

Consequently, a constant current (84) is now replaced by the linear ramp

$$i = \frac{(i_{k+1} - i_k)}{(t_{k+1} - t_k)}(t - t_k) - i_k. \tag{92}$$

We obtain the corresponding voltage transient by integration, resulting in a parabolic characteristic,

$$v = \frac{1}{C}\int i\mathrm{d}t = \frac{1}{C}\left[\frac{(i_{k+1} - i_k)}{(t_{k+1} - t_k)}(\frac{t^2}{2} - t_k t) + i_k t\right]. \tag{93}$$

The expression is somewhat complicated, but as we are only interested in the voltage increment, we can calculate the difference $v_{k+1} - v_k$ between time steps t_{k+1} and t_k using (93),

$$v_{k+1} - v_k = \frac{1}{2C}(t_{k+1} - t_k)(i_{k+1} + i_k). \tag{94}$$

We now resolve the expression to admittance form and substitute h for the time step

$$i_{k+1} = \frac{2C}{h}v_{k+1} - (\frac{2C}{h}v_k + i_k). \tag{95}$$

Again we arrive at an admittance $g_c = \frac{2C}{h}$ in parallel with an independent current source $i_c = -\frac{2C}{h}v(t_k) - i(t_k)$. Comparing (95) to (87), we can see that this time the current source depends not only on the voltage from the previous

step but also involves the current. This is very important. As the trapezoidal integration algorithm proceeds through the time steps, it takes more information from the signal history than backward Euler integration.

The same can be observed with inductors. First we assume a linear voltage ramp across the inductor,

$$v = \frac{(v_{k+1} - v_k)}{(t_{k+1} - t_k)}(t - t_k) - v_k. \tag{96}$$

After integration we get

$$i = \frac{1}{L}\int v\mathrm{d}t = \frac{1}{L}\left[\frac{(v_{k+1} - v_k)}{(t_{k+1} - t_k)}(\frac{t^2}{2} - t_k t) + v_k t\right]. \tag{97}$$

The resulting current step now is

$$i_{k+1} - i_k = \frac{1}{2L}(t_{k+1} - t_k)(v_{k+1} + v_k), \tag{98}$$

and the admittance form with the h substitution finally yields

$$i_{k+1} = \frac{h}{2L}v_{k+1} + (\frac{h}{2L}v_k + i_k). \tag{99}$$

Inductors are thus replaced by the impedance $g_l = \frac{h}{2L}$ in parallel with the independent current source $i_l = \frac{h}{2L}v(t_k) + i(t_k)$. In comparison with (91) we again notice the additional dependence on $v(t_k)$.

In conclusion, we expect the assumption of piecewise linear derivatives of reactances to give us a better approximation of real transients than the simplistic step model in backward Euler integration.

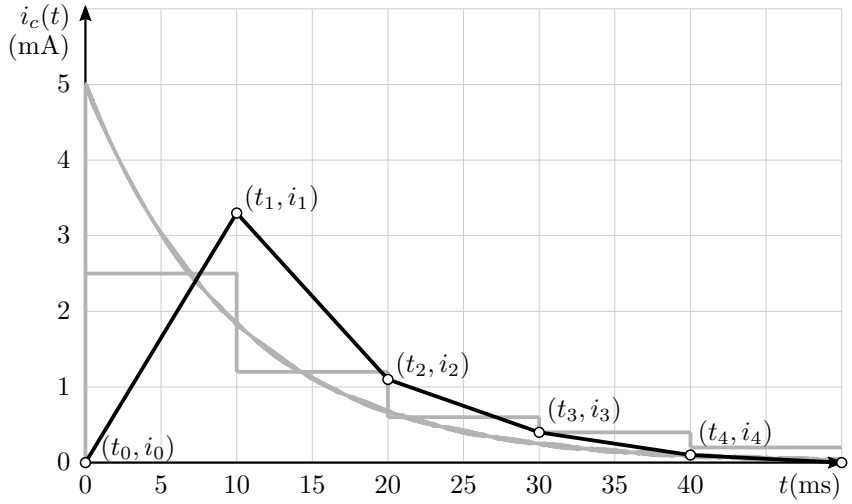

Figure 44: Trapezoidal integration current through capacitor.

Let us repeat the five integration steps for the example in figure 40. For comparison, we have plotted the trapeziodal current steps against the correct current transient as well as the results from backward Euler integration in figure 44.

Interestingly, the first step is just as badly off the correct current surge as in backward Euler integration. Actually it is even worse, because the average current in the first time slice is even less than that in the backward Euler step. The remaining steps, on the other hand, catch up with the correct signal nicely.

This becomes even more evident by comparing the voltage responses in figure 45. Not only is the trapezoidal algorithm with its parabolic segments smoother than the backward Euler integration curve, it also approximates the correct voltage trace much better.

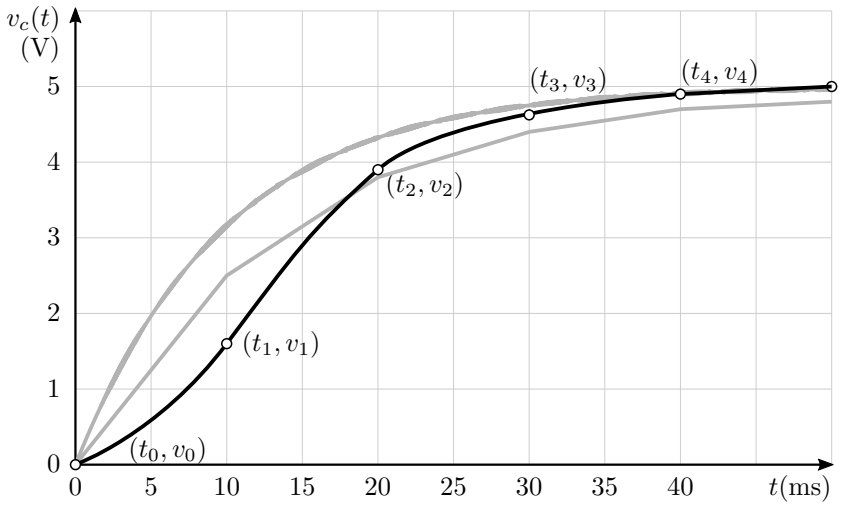

Figure 45: Trapezoidal integration voltage across capacitor.

The discontinuous nature of the circuit excitation does not match well with trapezoidal integration. The more an integration algorithm is drawing information from previous time slices, the worse it reacts to discontinuous phenomena. Unfortunately, in electrical engineering discontinuous signals are quite frequent.

SPICE OPUS uses trapezoidal integration if the `method` simulator parameter is set to `trap` and the `maxord` simulator parameter is set to 2.

By default, SPICE OPUS utilizes trapezoidal integration. In everyday use this integration method has proved itself as the most efficient. Backward Euler integration is used only when the simulator encounters a discontinuity or when the transient analysis is started. However, for special cases the user can manually switch to backward Euler integration.