Advanced meshing and solution techniques for industrial 3-D electromagnetic simulations

Shchetnikava, V.

Award date:
2009

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER THESIS

Advanced meshing and solution techniques for industrial 3-D electromagnetic simulations

by
Volha Shchetnikava

Supervisors:
Prof. Dr. W.H.A. Schilders (TU/e)
Dr. ir. W. Schoenmaker (Magwel)

August 27, 2009

Abstract

Today's integrated circuits (IC) can contain several hundred millions of nanometer-sized elements packed onto chips. Simulating and verifying these elements before fabricating a real IC is critical to chip designers, as building and troubleshooting a prototype are very expensive, time consuming and difficult processes. In short terms, simulation can be expressed as experiments with a model. Thus, numerical simulation in the Chip Industry replaces time consuming and expensive experiments and saves resources.

The software provided by MAGWEL aims at helping design engineers and scientists to gain an in-depth understanding of the electromagnetic fields and the currents and charge distributions in an environment consisting of different types of materials simultaneously. Thus, at the same instance insulators, semiconductors and metallic regions can be present in the computational domain. Typical applications are parts of integrated circuits consisting of doped semiconducting domains together with the selected part of the interconnect structures, possibly extended with integrated on-chip passive elements. The software considers three dimensions in space.

The sheer density of the latest chips force the modern simulation tools, and also MAGWEL's software, to develop new numerical techniques and algorithms in order to decrease memory usage and computation time of simulations.

In the first part, a simplification of the dual mesh algorithm for a particular case is presented. This method has decreased the time of the dual mesh constructing process 7 times.

The second part of this thesis concentrates on developing an efficient direct solver for sparse linear systems of equations, arising from the numerical discretization of partial differential equations. We present the Crout version of the LU factorization that reduce the solution time by approximately 50 percent in comparison with current using iterative methods.

Preface

This Thesis is the final part of my Erasmus Mundus Master program in Industrial Mathematics at TU Eindhoven, The Netherlands and TU Kaiserslautern, Germany. This research was done at MAGWEL company in Leuven, Belgium.

I am more than grateful to the people and institutions that have made this possible. First and foremost I would like to thank my supervisors Wil Schilders and Wim Schoenmaker, who have guided, supported, inspired, and helped me throughout the project. I am also greatly thankful to my colleagues Peter Meuris and Gabor Bella for always taking out time to listen to the challenges faced by me and to try and look for various solutions.

Surrounding it all I would like to thank my parents for always believing in me and for supporting and standing by me for my every decision, especially to come and study in EU. Last but not the least I would like to thank my colleagues in MAGWEL and my friends who constantly took me away from my work and kept me amused with interesting discussions, movies, parties, food, music, and their terrible sense of humor.

Volha Shchetnikava

Eindhoven, The Netherlands

Contents

Chapter 1

Introduction

Integrated circuits (ICs) are part of virtually every electronic component in the world today, such as cell phones, microwave ovens, digital watches, personal computers, printers, automobiles, and so on.

But what is actually an integrated circuit? IC, also called a microchip, is a single, miniature circuit with many electronically connected components etched onto a small piece of silicon or some other semiconductor material. The components etched onto a microchip include transistors, diodes, capacitors, and resistors. A transistor is a device capable of amplifying and switching electrical signals. A capacitor temporarily stores electrical charges, while a resistor controls current by providing resistance. The diode stops electricity under some conditions and allows it to pass only when these conditions change.

The first integrated circuit was demonstrated by Texas Instruments employee Jack Kilby in 1958. This prototype, measuring about 11.1 by 1.6 mm, consisted of a strip of germanium and just one transistor. Following this new concept, the IC industry started in the late 1960s and early 1970s with a ten micron technology. This technology node was identified by the minimum geometry that could be printed on the chip at that time. The major advance in this technology has been the continued reduction of size, or scaling-down, of the devices with time. This evolution was stated by G.Moore in 1971 and is known as *Moore's law*. Every 18 months or so, technology dimension were scaled by a factor s, which has historically been found to be equal to 0.7. This meant that a chip that was $1\text{mm} \times 1\text{mm} = 1\text{mm}^2$ could be reduced to $0.7\text{mm} \times 0.7\text{mm} = 0.5\text{mm}^2$ in the next generation. In other words, twice the number of transistors could be integrated on the same 1mm^2 chip as

compared to the previous technology node.

In such a way, during the 1970s, the ability to integrate thousands of gates on a single chip became feasible. Integrated circuits began to show up in pocket calculators, computers, and television sets. Around 1980, a revolution began in the microelectronics industry with major advances in IC processing technology, chip design, memory design, and computer-aided design (CAD). The ability to place 1 million transistors on a single chip was reached.

Through out the 1980s, available technologies were mostly in the 5μm to 1μm range. This dimension nominally refers to the *channel length* of the transistor or to the minimum resolvable geometry on a given layer, specially, the metal line width. Advances in photolithography, the key process that defines the minimum dimension in a technology, eventually led to the size below 1μm. In the middle of the 1990s the ranges from 0.5μm to 0.35μm were achieved. At the same time, the number of layers of metal continued to increase. Metal layers composed of aluminium and tungsten were used to connect transistors. The industry began with only one layer of metal for all connections. As the number of transistors increased, there was a need to increase the number of metal layers so that all required connections could be made.

Many argued that 0.35μm technology would be a physical limit for photolithography since the wavelength of light is approximately equal to this value. However, further advances allowed scaling below this barrier. The scaling continued its relentless pace to 0.25μm, 0.18μm, 0.15μm, 0.13μm, and by the year 2009, to $90n$m [12].

So, it is clear, that the on-chip interconnect structure in modern ultra large scale integrated circuits is a highly complicated electromagnetic system. Design of these ICs would not be possible without software assistance at every stage of the process, as IC technology requires now more accurate circuit analysis methods. As a result, technology CAD (TCAD) that accurately predicts the process and device characteristics is indispensable for future IC fabrication technology and device development. Computer experiments based on suitable, adjusted models can be carried out much faster than manufacturing expensive test structures and they enable to find many properties of the devices in an early stage of the development process of new technologies. Furthermore the understanding of the physical processes which occur during the fabrication and the operation of the devices is deepened. Hence, using predictive simulations, devices can be optimized in an

early phase and the manufacturing processes can be improved not only with respect to the quality of the resulting devices, but also with respect to manufacturing throughput.The goals of TCAD start from the physical description of integrated circuit devices, considering both the physical configuration and related device properties, and build the links between the broad range of physics and electrical behavior models that support circuit design.

The MAGWEL software helps design engineers to simulate a physical behavior of an IC or parts of an IC, where the IC layout consists of compositions of layers and in which the different layers may contain bricks of different materials. Typical applications are parts in integrated circuits consisting of doped semiconducting domains (the "frontend" part) together with the selected part of the interconnect structures (the "backend" part) possibly extended with integrated on-chip passive elements.

Whereas there are numerous simulation tools around, the MAGWEL solver has some unique features built in : first of all, contrary to most electromagnetic solvers, the MAGWEL tools solve the electromagnetic potentials i.e. the scalar potential V and the vector potential $\mathbf{A}$. This is quite important when "non-linear" materials are considered. Non-linear materials such as semiconductors have a current-field constitutive relation that is much more complicated than a simple Ohmic relationship that usually can be applied for metals.

MAGWEL is capable of dealing with simulation problems that are composed of different types of materials simultaneously. Thus at the same instance insulators, semiconductors and metallic regions can be present in the computational domain. In the metallic regions, Ohms law is applied to obtain the current densities from the fields. In the semiconducting regions, the drift-diffusion model is solved. In all regions the Maxwell equations are also solved.

The MAGWEL solver is very much inspired by the way how a device simulator operates. Device simulators solve Poissons equation and the current-continuity equations for the carriers in semiconductors in a self-consistent manner. This is also what the MAGWEL solver does. However, it goes a few steps further : it not only deals with semiconducting and insulating regions but also allows metallic domains to be included. Moreover it solves also the Maxwell equations. All this is done in a self-consistent manner.

The structure under study in this thesis concerns a powerMos transistor. This type of transistor is able to act as an accurate and fast current/voltage controller, whose internal energy dissipation in minimal yet large amounts

of current ($> 10^2$ A) can be controlled without switch loss. PowerMOS devices are basically hundreds of transistors operating in parallel and the design challenge consists of avoiding hot spots and current crowding effects in the interconnects patterns that are needed for connecting all the transistor source and drain contact to the fat bond pads to which the external wires are attached.

Finally, we will obtain a system of partial differential equations, which cannot be solved explicitly in general. Therefore, the solution must be calculated by means of numerical approaches. This will result in a linear system of algebraic equations for such structures. Today we can deal with power-MOS design for which the computations need to solve 10^7 linear equations. The coefficient matrices of these systems are said to be sparse, because there are only very few non-zero elements in them. Although there exist several methods for solving linear systems of equations, it is not trivial to solve a large system where the coefficient matrix is sparse. We need a method which can solve such system in a reasonable amount of time.

Methods for solving large sparse linear systems can be divided into two classes: *direct methods* and *iterative techniques*. Now the MAGWEL software uses the iterative methods from the NAG library. The problem is that for the large testcases computation time is rather high. We came up with the idea, that probably a very specific direct method could increase the performance of computations for that particular case.

The thesis is organized as follows. We start in Chapter 2 with the theory of IC simulation which is used in the MAGWEL software. Chapter 3 does the simplification of the current dual mesh algorithm for the structure under study. In Chapter 4, we discuss direct methods for sparse linear systems of equations, the LU factorization and, finally, the Crout method. The fill-reducing ordering is used for refinement of the LU factorization. This algorithm is described in Chapter 5. Chapter 6 contains the information about the LDL package, which is a direct solver for a sparse systems of linear equations . Finally, Chapter 7 brings the thesis to a conclusion by summarizing the main results of all methods.

Chapter 2

Integrated circuit simulation

An integrated circuit is a miniaturized electronic circuit that has been manufactured in the surface of a thin substrate of semiconductor material. They are composed of many overlapping layers, each defined by photolithography, and normally shown in different colors. Some layers mark where various dopants are diffused into the substrate (called diffusion layers), some define where additional ions are implanted (implant layers), some define the conductors (polysilicon or metal layers), and some define the connections between the conducting layers (via or contact layers). All components are constructed from a specific combination of these layers. In that way we can split the simulation in two parts: in-Silicon and on-Silicon.

2.1 Introduction

In-Silicon technology deals with the design and fabrication of devices and structures that are found in the Silicon part of the full architecture. This part of the integrated circuit is also known as 'front-end technology' and the design is traditionally supported by TCAD (Technology computer Aided Design). TCAD naturally falls in two segments: The first part deals with simulation of all fabrication steps such as implanting, diffusion, deposition and etching. The second part deals with the simulation of the devices. In the second part it is assumed that devices are connected to the outside world through the contacts or device ports that provide bias voltages or currents.

On-Silicon Technology deals with the design and fabrication of the metallic wires that connect the sea of transistors in a single integrated circuit. Its

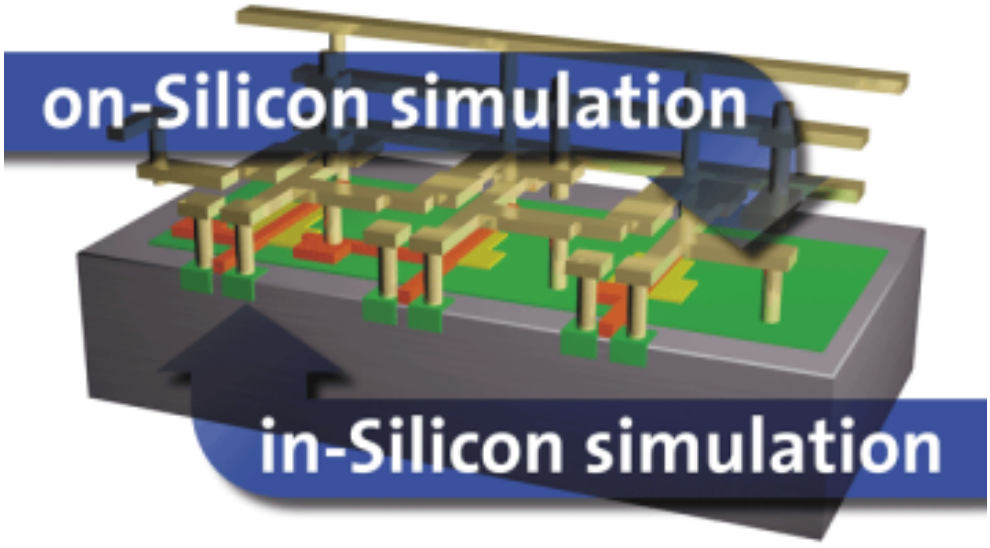

Figure 2.1: The 3D semiconductor-metal interconnect structure

design is supported by a hierarchy of tools. At chip level, the focus is on routing and placing functional blocks. At device level one deals with the physical simulation of the interconnect schemes. Physical simulation can still be fine-grained into several levels of sophistication. At low frequencies, it often suffices to extract lumped-element characteristics, such as the resistance and capacitance the building blocks of the wiring. At higher frequencies, more refined simulations are needed to extract the effects of rapidly changing electro-magnetic fields. In-depth understanding of the high-frequency effects can be achieved by using EM (electromagnetic) field solvers.

High-frequency effects show up in various ways. Besides the well-known skin effect that forces currents to be located only at the surface regions of the interconnect wires, there are effects such as the proximity effect that redistributes the currents in neighboring wires due to exchange of electromagnetic energy. One form of substrate coupling deals with the electromagnetic interaction of different parts of the integrated circuit due to electromagnetic energy that is injected into the Silicon part of the chip. Whereas at low and moderately-low frequencies, the devices in the Silicon and the wires on the Silicon could be characterized separately, we have now entered the era in which an integrated approach is needed. At higher frequencies, the electric fields are for a large part inductive, i.e. a major contribution originates from the rapidly changing magnetic fields. The details of the substrate will then determine the interconnects characteristics, and vice versa the in-Silicon device

characteristics will be strongly influenced by the on-Silicon structures (interconnects) that are around. The border that separates the Silicon from the layers above the Silicon dissolves when a characterization must be achieved at high-frequencies: An integrated characterization by simulation of the front-end and back-end structures is needed at high frequencies.

In traditional combinations of TCAD tools and EM simulators, currents and charges are solved independently, and then used as source for the electromagnetic fields. That these electromagnetic fields on their turn alter the sources is not taken into account, and therefore an inconsistency is incorporated in the solution. The solutions that provide a decoupled simulation of semiconductors and metal are no longer adequate for process nodes of 65nm and below since their accuracy can be easily off by 30 % or more.

MAGWEL offers an approach to accurately simulate in-Silicon devices and on-Silicon structures by simultaneously solving the full set of Maxwell equations that describe the electromagnetic fields, the drift-diffusion equations that describe the semiconductor physics and Ohm's law that describing the currents in metallic domains. The electromagnetic fields are determined by the current and charge distributions and vice versa, the currents and charge distributions are determined by the electromagnetic field. A reliable solution is obtained if all equations are satisfied with a single set of variable values. This is known as the self-consistent solution. It is common practice in TCAD that only the self-consistent solution is compliant with experiment.

2.2 The Maxwell and constitutive equations

The starting set of equations that MAGWEL addresses are the Maxwell equations:

$$\text{Gauss law} : \quad \nabla \cdot \boldsymbol{D} = \rho,$$

$$\text{No magnetic monopoles} : \quad \nabla \cdot \boldsymbol{B} = 0,$$

$$\text{Maxwell-Faraday} : \quad \nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t},$$

$$\text{Maxwell-Ampere} : \quad \nabla \times \boldsymbol{H} = \boldsymbol{J} + \frac{\partial \boldsymbol{D}}{\partial t},$$

where $\boldsymbol{D}$, $\boldsymbol{E}$, $\boldsymbol{B}$, $\boldsymbol{H}$, $\boldsymbol{J}$ and ρ denote the electrical induction, the electric field, the magnetic induction, the magnetic field, the current density and the

charge density, respectively. The following constitutive equations relate the inductances to the field strength:

$$\boldsymbol{B} = \mu \boldsymbol{H}, \qquad \boldsymbol{D} = \epsilon \boldsymbol{E}.$$

The permittivity ϵ, and the permeability μ and the constitutive equation that relates the current $\boldsymbol{J}$ to the electric field as well as the carrier densities are determined by the medium under consideration. Besides Maxwell equations, appropriate constitutive laws must be considered. For dielectrics, no additional relations are needed. For a conductor the current $\boldsymbol{J}$ is given by Ohm's law:

$$\boldsymbol{J} = \sigma \boldsymbol{E},$$

where the current density satisfies the current-continuity equation which guarantees the conservation of electric charge

$$\nabla \cdot \boldsymbol{J} + \frac{\partial \rho}{\partial t} = 0.$$

We will consider a dielectric whose lossy effects can be neglected. Therefore, no current continuity equation needs to be solved in the dielectric materials. In the semiconductor regions, the current $\boldsymbol{J}$ consists of negatively and positively charged carrier currents obeying the current continuity equations

$$\begin{aligned} \nabla \cdot \boldsymbol{J}_n - q\frac{\partial n}{\partial t} &= U(n,p), \\ \nabla \cdot \boldsymbol{J}_p + q\frac{\partial p}{\partial t} &= -U(n,p). \end{aligned}$$

Here, the charge and current densities are

$$\begin{aligned} \rho &= q(p - n + N_D + N_A), \\ \boldsymbol{J}_n &= q\mu_n n\boldsymbol{E} + kTq\mu_n \nabla n, \\ \boldsymbol{J}_p &= q\mu_n p\boldsymbol{E} + kTq\mu_p \nabla p, \end{aligned}$$

and $U(n,p)$ is the generation/recombination rate of charge carriers. The present release of the MAGWEL solver considers only the recombination in the semiconducting regions. The current continuity equations provide the solution for the variables n and p [20].

In order to solve the drift-diffusion equations self-consistently with the Maxwell equations, one needs to address the Maxwell equations in their potential formulation. The reason is that the carrier concentrations are given by the Boltzmann distribution and the latter are determined by the carrier energies and quasi-fermi levels. We introduce the electric scalar potential V and the magnetic vector potential $\boldsymbol{A}$. The magnetic induction $\boldsymbol{B}$ is given by

$$\boldsymbol{B} = \nabla \times \boldsymbol{A}.$$

Then, 'No magnetic monopoles' equation is automatically satisfied since

$$\nabla \cdot \boldsymbol{B} = \nabla \cdot (\nabla \times \boldsymbol{A}) = \nabla \times \nabla \cdot \boldsymbol{A} = 0.$$

Now the Maxwell-Faraday law can be rewritten by using the vector potential

$$\nabla \times (\boldsymbol{E} + \frac{\partial \boldsymbol{A}}{\partial t}) = 0.$$

This means that one can write the electric field $\boldsymbol{E}$ as

$$\boldsymbol{E} = -\nabla V - \frac{\partial \boldsymbol{A}}{\partial t}.$$

Thus the four coupled first-order partial differential equations are converted to two homogeneous equations that are satisfied identically, and two inhomogeneous, coupled, second-order equations[11]:

$$-\nabla \cdot (\epsilon \nabla V + \epsilon \frac{\partial \boldsymbol{A}}{\partial t}) = \rho(V, \boldsymbol{A}), \tag{2.1}$$

$$\nabla \times (\nu \nabla \times \boldsymbol{A}) = \boldsymbol{J}_{TOT}. \tag{2.2}$$

The current is the sum of the conduction and the displacement current $\boldsymbol{J}_{TOT} = \boldsymbol{J}_c + \boldsymbol{J}_d$, and $\nu = \frac{1}{\mu}$.

The origin of gauge invariance in classical electromagnetism lies in the fact that the potentials $\boldsymbol{A}$ and V are not unique for given physical fields $\boldsymbol{E}$ and $\boldsymbol{B}$ which results in a singular matrix representation. Any solution $\boldsymbol{A}$ can be modulated by adding the gradient of an arbitrary scalar field. In order to elevate this arbitrariness an additional condition is imposed, e.g. a gauge condition. The Coulomb gauge is one of many possible gauge conditions which are used in the MAGWEL software. This gauge condition is specified by the following additional constraint on the vector field $\boldsymbol{A}$

$$\nabla \cdot \boldsymbol{A} = 0.$$

2.3 Boundary conditions

The simulation domain consists of an interconnect (sub)system possibly extended with a region of air surrounding it. Therefore, we must make a distinction between boundary conditions for the simulation domain and boundary conditions for the device. For the latter, it is clear that the electric potential V is well defined on the metal terminals provided that voltage boundary conditions are used. The boundary conditions for the simulation domain are more subtle.

The external currents, impinging perpendicular to the boundary surface of the simulation domain, carry their own circular magnetic field. Such a magnetic field is described by a component of the vector potential parallel to the impinging current. Therefore, the boundary condition is that the vector potential be equal to zero for all links that are in the boundary surface, $\partial\Omega$, of the simulation domain Ω, whereas links pointing orthogonally inwards from the enclosing surface are part of the set of the unknown variables that should be solved

$$A_{ij} = 0 \quad i, j \in \partial\Omega. \tag{2.3}$$

The boundary conditions for the scalar potential V are a mixture of Dirichlet and Neumann boundary conditions. At the contacts we assume Dirichlet boundary conditions, whereas at the remaining part of the enclosing surface we assume Neumann boundary conditions. This assumption implies that no static perpendicular electric field exists for these parts of $\partial\Omega$. If a contact is placed on a semiconducting region, we assume that this contact is also ohmic.

2.4 Interface conditions

In general, the structure consists of insulating, semiconducting and metallic regions. As a consequence, there will be four types of interface nodes, i.e

- insulator/metal interface nodes
- insulator/semiconductor interface nodes
- metal/semiconductor interface nodes
- insulator/metal/semiconductor 'triple' points

At the metal/semiconductor interface nodes, we implement the idealized interface Schottky contact condition, as for a boundary condition for a semiconductor region, by setting $\phi_n = \phi_p = V_{metal}$, where V_{metal} is the value of the Poisson potential at the metal side of the interface. The Poisson potential at the semiconductor side of the interface is $V_{semi} = V_{metal} - \delta V$, where δV represents the contact potential between the two materials.

At a metal/semiconductor interface node there is one variable(V_{metal}) that needs to be solved. The equation for this variable assigned to the node i is the current-continuity equation,

$$\sum_j J_{ij} S_{ij} = 0,$$

where J_{ij} is the current density in disretized form for the link(ij).

At metal/insulator interface nodes we assume continuity of the Poisson potential. For these nodes there is, apart from the variables $\boldsymbol{A}$ and χ, one unknown V_i, and the corresponding equation is the current-continuity equation.

At insulator/semiconductor interface nodes there are three unknowns to be determined, V, n and p. The Poisson equation is solved self-consistently with the current-continuity equations for n and p, while V is continuous at the insulator/semiconductor interface.

At triple point nodes, the Poisson potential is triple-valued.

2.5 Mesh generation

The obtained system of partial differential equations cannot be solved explicitly, so a numerical procedure is used for finding an approximate solution. First, the simulation domain has to be partitioned into a finite number of subdomains(usually triangles/tetrahedra or rectangles/bricks), in which the solution can be approximated with a desired accuracy. This process of subdividing the simulation domain into smaller subdomains is called mesh generation. Second, the differential equations have to be approximated in each of the subdomains by algebraic equations which involves only values of the continuous dependent variables at the discrete points in the domain. In that way one obtains a large system, in our case of linear, algebraic equations with unknowns comprised of approximations of the continuous dependent variables

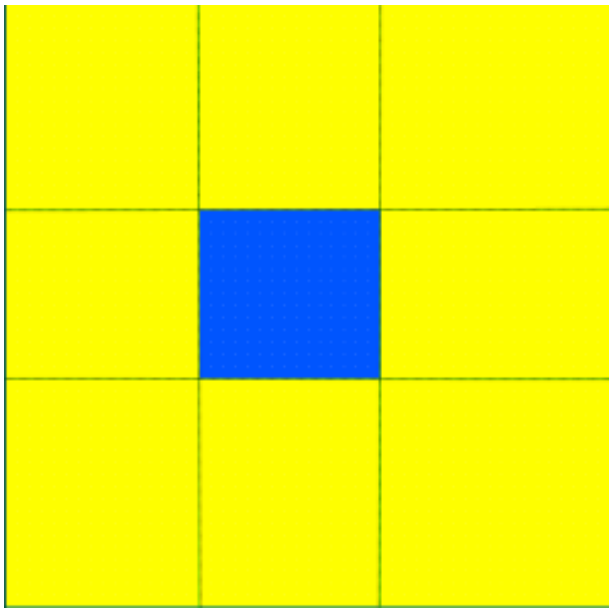

Figure 2.2: A 2D version of a simple structure with the Manhattan mesh

at discrete points. The solution of such system represents a good approximation to the solution of the analytically formulated problem depending upon the fineness of the partitions of the simulation subdomains.

The choice of the mesh is an important part of the simulation work. The MAGWEL software supports 2 different meshing algorithms, each of them has its own characteristics and its own set of applications.

The first algorithm is a Manhattan meshing. This means that the whole simulation domain has a mesh with cartesian mesh lines that run from one border of the simulation domain to the other border of the simulation domain. This mesh can be specified with three arrays : the mesh intersections along the x-axis, those along the y-axis, and those along the z-axis. The main characteristics of this algorithm are :

- Very fast meshing
- Might generate a lot of mesh nodes, resulting into longer simulation times

A simple example of the Manhattan mesh is shown in Fig. 2.2. The structure consists of one oxide layer and an aluminium brick in the middle of it. As a result we obtain 16 grid nodes.

Local Voronoi meshing is an extension of the Manhattan meshing algorithm. The result of this algorithm is still a mesh with cartesian mesh

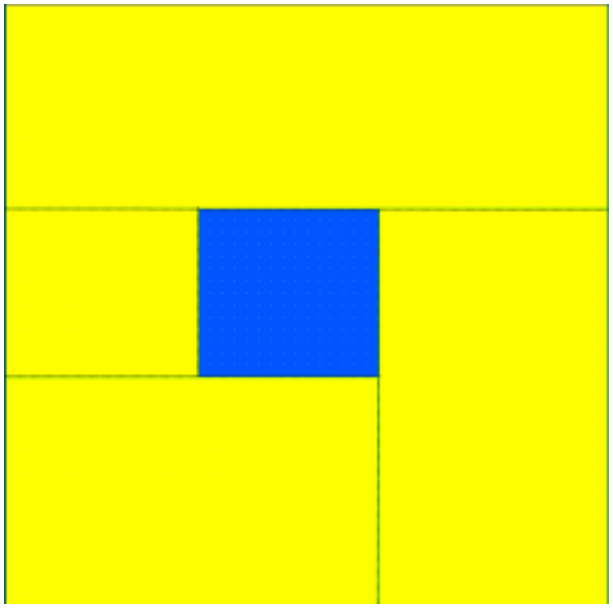

Figure 2.3: A 2D version of a simple structure with the Local Voronoi pre-mesh

lines, but those lines don't have to run onto a border of the simulation domain. They can be ended when the algorithm detects it is appropriate to end them, while maintaining physical correctness of the simulation result. The main characteristics of this algorithm are:

- Generates fewer mesh nodes, resulting into shorter simulation times.
- Calculating this mesh is not a fast process

If the same structure is meshed with the Local Voronoi algorithm, the resulting mesh contains 12 grid nodes, and is shown in Fig. 2.3. We see that some of the mesh bricks have the mesh grid points on their links. Such points are called orphans and additional manipulations should be done with such bricks in order to construct a proper mesh. That is why the method is called the Local Voronoi mesh. We will split each brick with orphans in a several parts by using the Voronoi diagram method. The result is shown in Fig 2.4. Thus, the simulation domain is subdivided into a set of tetrahedral meshing elements, while the Manhattan mesh leads to the rectangular parallepiped elements.

For most industrial applications, the Local Voronoi meshing will be most appropriate as it reduces simulation times dramatically. Manhattan meshing can be preferred in some situations:

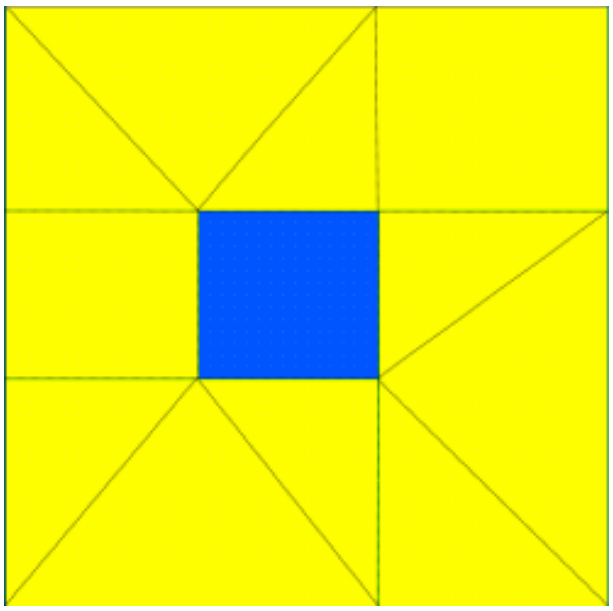

Figure 2.4: A 2D version of a simple structure with the Local Voronoi mesh

- Very detailed information on the electromagnetic field is required in all parts of the structure and a lot of mesh nodes are required.
- The time needed to generate the mesh is of the same order as the time needed to do the simulations.
- The number of nodes generated with the Local Voronoi meshing is close to the number generated with Manhattan meshing.

To make sure a simulation closely matches reality, all variations of the electromagnetic field should be taken into account. This requires meshing of the places where huge variations of those fields will take place. While knowing where those variations will take place requires a simulation, some generic knowledge of electromagnetism allows to predict where those variations will be located. When simulating semiconductor structures, most variations of the electromagnetic fields will take place due to skin effect and due to inversion regions.

2.6 Discretization of operators

The MAGWEL software uses the finite-integration technique (FIT) for setting up the discrete equations[19, 21]. Whereas finite-element method approximates a continuous function over the simulation domain by a piecewise

continuous function, FIT does not provide knowledge of the function between the grid points. The finite-integration method considers the values at the grid points as 'fair' representatives of the values in the surrounding of the grid points, no further assumptions are made what the precise values must be at locations where no grid point is placed. A consequence of the viewpoint of FIT is that discretized equations are derived from balance equations, where fluxes from one volume element into neighboring ones must satisfy conservation laws. Thus, FIT has a built-in guarantee that charge is conserved, i.e. the net currents entering the simulation region at the ports are zero. This is the main motivation to pursue the finite-integration method for computational electrodynamics.

The naive substitution of a differential by a finite difference leads in many circumstances to error-prone results. The reason is that this discretization approach ignores important a-priori knowledge of the system that is usually expressed in conservation laws. Ignoring this knowledge can lead to numerical solutions that create energy, matter or charge from the accumulation of numerical discrepancies. The discretization should be performed in such a way that the a-priori knowledge is exactly respected in the transition in going from the continuum to the discrete formulation. One method to find the correct discretized equations is to identify the geometrical meaning of the variables and operators. To illustrate this statement we consider a few examples.

- **Scalar fields**

  A scalar field $\phi(x,t)$ is a variable that depends on space and/or time. For each space point and for each time instant a single value characterizes the field. The discretization of a scalar field occurs in two steps. First the space/time domain is subdivided in cells that are labeled by a grid node (x_i, t_i), next to each grid node associated field variable $\phi(x,t) \rightarrow \phi_i \backsimeq \phi(x_i, t_i)$. A scalar is "node-associated" to the grid and we say it is of the 0-form[6].

- **Vector fields**

  A vector field, $\boldsymbol{A}(x,t)$ is a variable that assigns to each space point and for each time instant a quantity that has a size and a direction. Depending on the physical meaning of the vector field, the vectors can be one forms or two forms. The exerted forces are an example of a 1-form and should be associated to the links of the computational grid.

One may conclude that forces are "link-associated" to the grid. A 2-form corresponds to flows or fluxes and describes the matter that passes through a surface element. Therefore, a 2-form should be associated to the plaquettes of the grid.

- **Densities**

  A density, $\sigma(x,t)$ expresses the amount of an observable per unit volume. Therefore a density, is associated to the volume elements of the computational grid. Densities are geometrically identified as 3-forms, and should as such be distinguished from scalar fields that are 0-forms

After having identified the geometrical association of the physical variables, it becomes necessary to characterize the physical meaning of the differential operators. Once again this is best illustrated with a few examples.

- **Gradients**

  Acting with the differential operator, ∇, on a scalar field $\phi(x,t)$ results into a vector $\mathbf{v}$ that points into the direction of maximal variation of the scalar field. This vector is a 1-form and after discretization one may assign to each link of the grid a number v_{ij} that corresponds to the projection of $\mathbf{v} = \nabla\phi$ on the link direction $\mathbf{e}_{ij}$, where i and j are the end points of the link,

$$v_{ij} = \mathbf{e}_{ij} \cdot \mathbf{v} = \frac{\phi_j - \phi_i}{h_{ij}}. \tag{2.4}$$

  The discretization of the gradient of a scalar field is rather straightforwardly equal to the results that is obtained by applying a finite difference technique. This seems to contradict what was said earlier. However, this case is the only example where the outcome will be equal. The next example already contains subtleties that prevent such an identification.

- **Divergences**

  The divergence describes the change of a vector field in going from one location to another. Physically, the divergence expresses the conservation of an observable. Considering a volume in space, we may consider

the integrated flux through the surface of the volume. The change of the integral describes the change in time of the integrated observable.

$$\frac{dQ}{dt} = \frac{d}{dt}\int_V dv\rho = -\int_V dv\nabla\cdot\mathbf{J} = -\oint_{\partial V} d\mathbf{a}\cdot\mathbf{J}. \tag{2.5}$$

First of all, we notice that the divergence acts on a 2-form $\mathbf{J}$. Therefore, acting with the divergence on a 1-form is disputable. Indeed, usually 1-forms need to be converted to 2-forms by a Hodge operation[19] before the divergence can be applied. Well-known Hodge operators, although not so well known under this name, are the dielectric constant, ϵ, that converts the electric field $\mathbf{E}$ that is a 1-form into the displacement field $\mathbf{D}$ that is a 2-form, according to $\mathbf{D} = \epsilon\mathbf{E}$, and the magnetic permeability μ that converts the magnetic induction $\mathbf{B}$ into the magnetic field $\mathbf{H}$ according to $\mathbf{B} = \mu\mathbf{H}$. The discretization of the divergence is based on applying Gauss theorem for each volume element of the grid. This discretization is known as the box-integration method or the finite-integration technique.

- **Circulations**

  In order to address circulations or the curl operator, we consider the circulation of a vector field (1-form). The curl is an exterior derivative that increments the form index by one. Therefore, if $\mathbf{A}$ is a 1-form vector field, the circulation $\mathbf{B} = \nabla\times\mathbf{A}$ is a 2-form vector field. Just as with the divergence, a proper discretization can be obtained by applying the appropriate integral theorems. Here, we need to apply Stokes theorem in order to obtain the discretized expressions on the computational grid. In particular, for each link we can identify a perpendicular surface. Integration of the 2-form (flux) over this surface gives

$$\oint_S d\mathbf{a}\cdot\nabla\times\mathbf{A} = \int_{\partial S} dl\cdot\mathbf{A}. \tag{2.6}$$

  It is an interesting fact to note that the Stokes' theorem is a realization of the fundamental theorem of differential geometry

$$\int_\Omega d\omega = \int_{\partial\Omega}\omega, \tag{2.7}$$

where $\omega = \mathbf{A} \cdot \mathbf{dl}$, $\partial\Omega$ is the enclosing 1-D contour of a 2-D surface Ω and $d\omega = \nabla \times \mathbf{A} \cdot d^2\mathbf{S}$.

Given a particular grid, the 2-form $\mathbf{B}$ is allocated to the links of the dual grid. These links point through the center of the plaquettes of the primary grid.

2.6.1 Discretization Scheme

Because of the specific geometry of the problem, we will discretize the set of equations on a regular Cartesian grid having N nodes in each direction. The total number of nodes in dimensions D is $M_{nod} = N^D$. To each node we may associate D links along the positive directions and, therefore, the grid has roughly DN^D links. We say roughly because nodes at side walls will have less contributions.

2.6.2 Discretization of the Curl-Curl Operator

For determining the discretized equation for a particular $\mathbf{A}$ on a link we integrate the Maxwell-Ampere equation over the surface of the dual mesh that the link intersects.

$$\int_S \nabla \times (\upsilon \nabla \times \mathbf{A}) \cdot d\mathbf{S} = \int_S \mathbf{J} \cdot d\mathbf{S} \approx |S|\mathbf{J}. \tag{2.8}$$

Applying Stokes' law this becomes

$$\oint_{\partial S} \upsilon \nabla \times \mathbf{A} \cdot dl = \int_S \mathbf{J} \cdot d\mathbf{a} \approx |S|\mathbf{J}. \tag{2.9}$$

Replacing the integration by a finite summation we obtain

$$\sum_{i=1}^{4} \upsilon \mathbf{B}_i l_i \approx |S|\mathbf{J}. \tag{2.10}$$

The relation $\mathbf{B} = \nabla \times \mathbf{A}$ can be discretized in the same manner by integrating over the surfaces in the primary mesh and applying Stokes law again. The method is illustrated in Fig. 2.5, where the thick lines represent the $\mathbf{B}$ - fields piercing through the plaquettes at the black dots.

We will now give the detailed discretized equation for the middle link in z-direction in Fig. 2.5. In the notation below the variables x_i and y_i represent

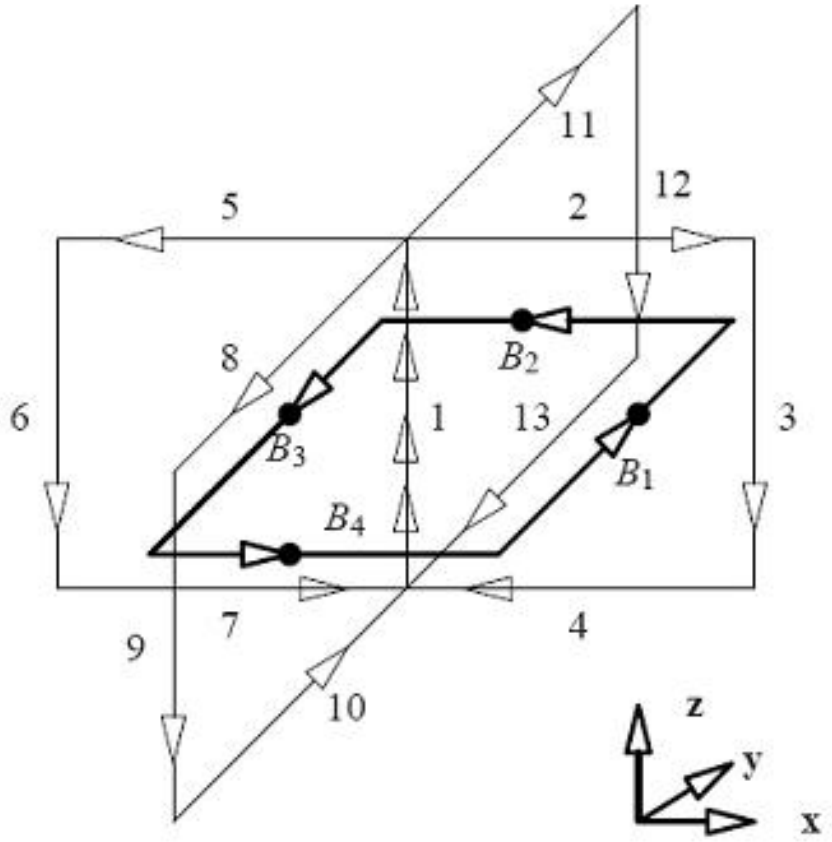

Figure 2.5: Links involved in the discretization of the curl-curl operator for the central link 1.

the lengths of the links that point towards the positive ($i = +$) or negative ($i = -$)x and y directions. The variable z_0 represents the length of the link (1) under consideration. Equation (2.10) then becomes :

$$B_1\frac{y_- + y_+}{2} - B_2\frac{x_- + x_+}{2} - B_3\frac{y_- + y_+}{2} + B_4\frac{x_- + x_+}{2} = \mu J_1\frac{(x_- + x_+)(y_- + y_+)}{4}, \tag{2.11}$$

with

$$
\begin{aligned}
B_1 &= \frac{1}{x_+ z_0}(A_1 z_0 + A_2 x_+ - A_3 z_0 - A_4 x_+),\\
-B_2 &= \frac{1}{y_+ z_0}(A_1 z_0 + A_{11} y_+ - A_{12} z_0 - A_{13} y_+),\\
-B_3 &= \frac{1}{x_- z_0}(A_1 z_0 - A_5 x_- - A_6 z_0 - A_7 x_-),\\
B_4 &= \frac{1}{y_- z_0}(A_1 z_0 - A_8 y_- - A_9 z_0 - A_{10} y_-).
\end{aligned}
$$

We may substitute the latter expressions into (2.11) and obtain the algebraic relation between the vector field variables $(A_1, \ldots, A_{13})$ and the current density J_1 on link 1. The result is:

$$
\sum_{k=1}^{13} \Lambda_k A_k = \mu J_1 \frac{1}{2}(x_- + x_+)(y_- + y_+). \tag{2.12}
$$

2.6.3 Discretization of the Divergence Operator

The discretization of the equation $\nabla \cdot \mathbf{A} = 0$ is performed by applying the Gauss law over a volume element on the dual mesh and replacing the surface integration over $\mathbf{A}$ by finite summation. The method is illustrated in Fig. 2.6. In this figure, the lengths of the links that are involved in the finite summation also denoted.

By defining the variables $A_i = \mathbf{e}_i \cdot \mathbf{A}$ we obtain

$$
\begin{aligned}
&\frac{1}{4}A_1(y_- + y_+)(z_- + z_+) - \frac{1}{4}A_2(y_- + y_+)(z_- + z_+)\\
&+\frac{1}{4}A_3(x_- + x_+)(z_- + z_+) - \frac{1}{4}A_4(x_- + x_+)(z_- + z_+)\\
&+\frac{1}{4}A_5(x_- + x_+)(y_- + y_+) - \frac{1}{4}A_6(x_- + x_+)(z_- + z_+) = 0.
\end{aligned} \tag{2.13}
$$

By grouping the links in the left-hand side of equation (2.12), it can be rewritten as

$$
\sum_{l=1}^{3} \sum_{G_l} \Lambda_k A_k = \mu J_1 \frac{1}{2}(x_- + x_+)(y_- + y_+), \tag{2.14}
$$

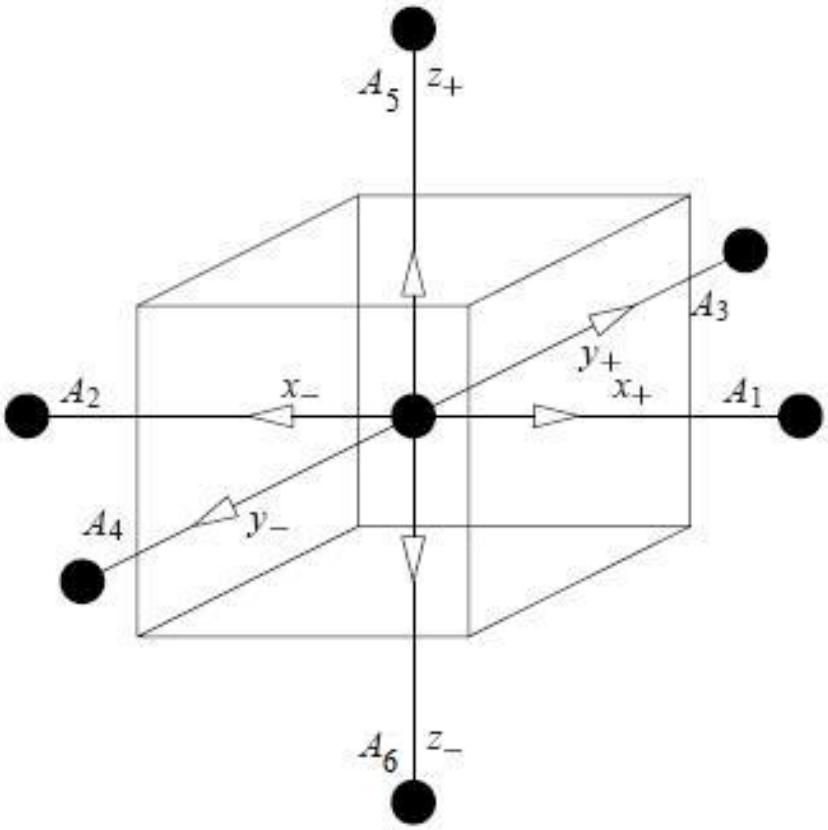

Figure 2.6: Links involved in the discretization of the divergence operator.

where $G_1 = \{1, 3, 6, 9, 12\}$, $G_2 = \{2, 5, 8, 11\}$ and $G_3 = \{4, 7, 10, 13\}$. Using $\nabla \cdot \mathbf{A} = 0$, we find that

$$\sum_{G_2} \Lambda_k A_k = \frac{(x_- + x_+)(y_- + y_+)}{z_0(z_0 + z_+)}(A_1 - A_{15}), \tag{2.15}$$

and

$$\sum_{G_3} \Lambda_k A_k = \frac{(x_- + x_+)(y_- + y_+)}{z_0(z_0 + z_-)}(A_1 - A_{14}). \tag{2.16}$$

This finally leads to a representation of the curl-curl operator that does not contain the links from the sets G_2 and G_3 anymore. In other words: only links parallel to the starting link and being neighbor contribute to the discretization. The final result is illustrated in Fig. 2.7.

Equation (2.12) can be rewritten as

$$\sum_{k=1'}^{7'} \lambda_k A_k = \mu J_{1'} \frac{1}{2}(x_- + x_+)(y_- + y_+), \tag{2.17}$$

where the summation runs over the primed labeling of the links in Fig. 2.7.

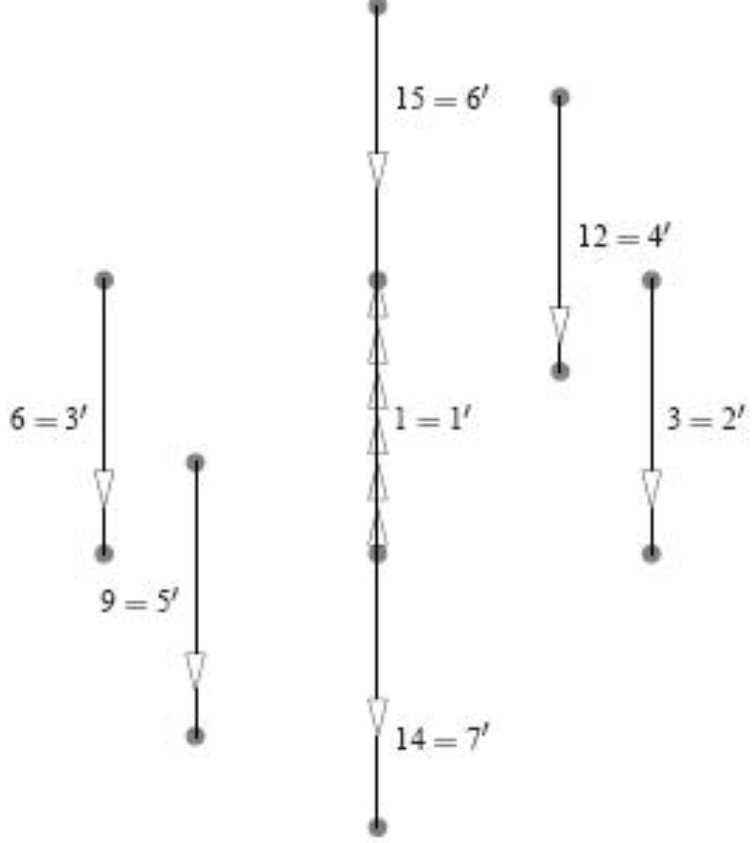

Figure 2.7: Links involved in the discretization of the curl-curl operator together with divergence.

This representation leads to an assembling of the discretized full set of equations with a considerably narrower band width. The width of the matrix stencil reduces from 13 to 7. Moreover, since the gauge condition is applied implicitly, the resulting matrix is also regular.

2.6.4 Discretization of the Poisson-type Operators

The discretization of the Poisson-type equations for V is constructed by replacing the gradients in the Laplace operator by finite differences, integrating this over a volume-element, applying Gauss law, and replacing the surface integrals by finite summation. As the values of the scalar potential V are prescribed in the nodes of the primary grid, the volume-element is selected to be the dual volume of the node.

$$\int_{V_i} \nabla \cdot (\epsilon \nabla V) dv = \int_{\partial V_i} \epsilon \nabla V \cdot dS \backsim \sum_{k}^{6} S_{ik} \epsilon_{ik} \frac{V_k - V_i}{h_{ik}}. \tag{2.18}$$

Such type of the discretization is done for the every node v of the mesh structure.

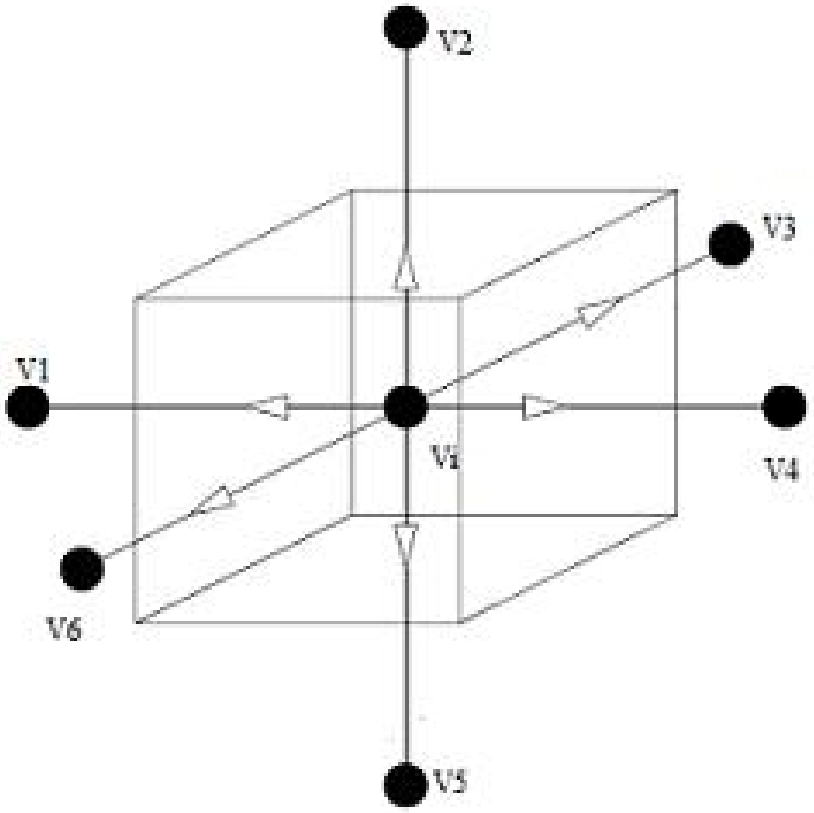

Figure 2.8: Nodes involved in the discretization of the Poisson operator.

Chapter 3

The dual mesh algorithm

The first step in a numerical procedure for approximating solutions to mathematical models is to choose a representative set of points in the problem area. This is the problem of mesh generation. Once we have a discretization or sometimes we just say a mesh, we are able to construct a finite set of equations, the solution of which must be an approximation to the required solution.

In other words, mesh generation is a process of subdivision of a simulation domain into simple-shaped sub-volumes. These sub-volumes are usually referred to as mesh elements or cells. Elements of a mesh are connected to each other, they do not intersect with each other and cover the entire domain.

3.1 Manhattan mesh

The easiest and most straightforward way to construct mesh is the Manhattan mesh. It will divide the simulation domain into rectangular parallelepipeds (bricks), where all links of those bricks are parallel to one of the x, y or z axes. The mesh is constructed in such a way, that every mesh line starts and finishes at the boundary of the domain. The reverse side of this simplicity is obtaining the useless extra nodes. This happens because of local details that need to be captured, then the mesh lines propagate through the simulation domain all the way to the boundary. As this process affects all three dimensions, an exponential growth in the number of points are observed. The result is that a large matrix is given to the solver, which makes the use of Manhattan meshes relatively inefficient. However, such structured

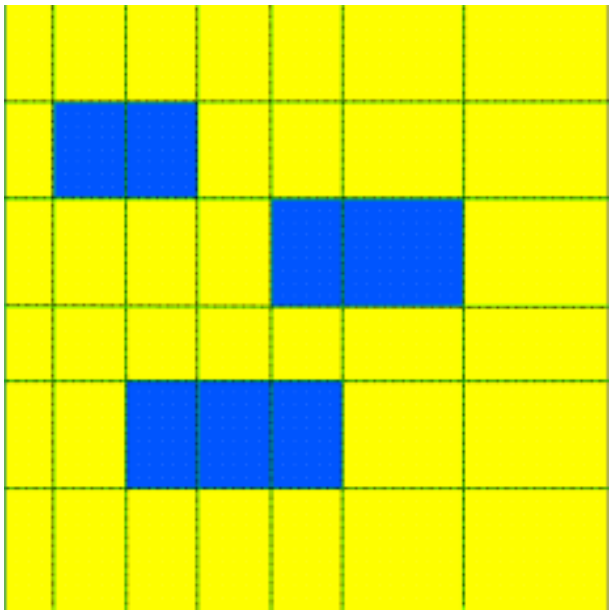

Figure 3.1: A 2-D version of simple structure with a Manhattan mesh

grids are much easier to generate and manipulate. Fig. 3.1 shows a 2-D simple structure with the relatively coarse Manhattan mesh defined by corners of bricks inside it. This structure is an oxide layer with 3 aluminium bricks in it. In Fig 3.2 the 3-D version of the same structure is presented.

We first describe briefly the Manhattan mesh generation method. Here, we use methods that are based on the cubetree approach of generating a suitable representation of objects (simulation domain or simulated device). Cubetree is a rooted tree in which every internal node has several children. Every node in the cubetree corresponds to a rectangular parallelepiped (brick). This implies that the bricks of the leaves together form a subdivision of the brick of the root. It is obvious, that the root brick is a simulation domain contour.

Before we continue, we introduce some terminology related to elements of mesh modeling and cubetree subdivision. Objects created with the Manhattan mesh must store different types of elements. These include vertices, edges (links), surfaces and bricks. A vertex v is a position along with other additional information. We store it as the coordinate vector $v = (x, y, z)$. A link v is a connection between two vertices v_1 and v_2. In order to operate with it, we just need to store coordinates of vertices v_1 and v_2. Every brick is formed by six surfaces, each of which is a rectangle. This surface could be stored in very economic way by keeping only the values of vertices in the lower-left and upper-right corners (Fig 3.3). The same trick is done for the

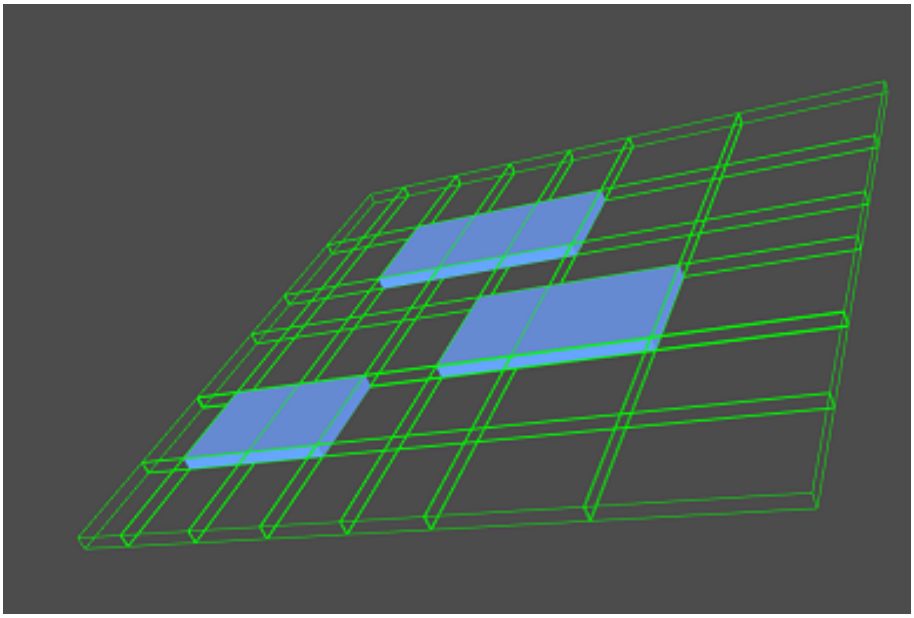

Figure 3.2: A 3-D version of simple structure with a Manhattan mesh

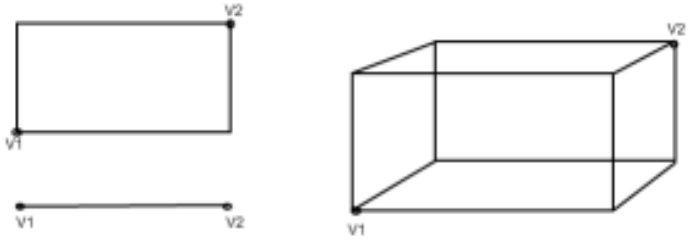

Figure 3.3: Elements of the Manhattan mesh modeling

brick, instead of storing 8 vertices we keep only 2: the lower-left at the front surface and upper-right at the back face (Fig. 3.3).

The Manhattan mesh generation method has the following main steps:

- The root of the cubetree structure is determined by the simulation domain, which is also a brick.
- Every structure consist of several layers, thus, we subdivide the root brick into several layers of bricks.
- Collect divisions from the rectangular contacts. Each of them divide a layer brick also in several bricks.

- Then, we collect divisions from all bricks in all existing layers.
- At the border of two materials we will add extra mesh points. This is very useful in metals, when the skin effect is pronounced or in junction regions in semiconductor materials.
- A uniform mesh can be specified for the whole simulation domain in addition to the Manhattan mesh. Anyway, it will result into mesh elements which are also bricks.
- Some additional features can also be used to refine existing mesh locally.

As we have already mentioned, the Manhattan mesh leads to the mesh elements which are rectangular parallelepipeds. The next step, would be to generate a dual mesh for the primary Manhattan mesh and extract the dual information from it. This needs to be done because of the discretization of the Curl-Curl operator, discussed in the previous chapter.

3.2 The dual mesh

The MAGWEL software is implemented in such a way, that the dual mesh function is used by the Manhattan and the Local Voronoi meshes. This function is a part of the Qhull library and besides the generation of a dual mesh, it divides all bricks with orphans into several parts for the Local Voronoi mesh. As we said before, the Local Voronoi mesh results into the tetrahedral mesh elements, which are more complicated than the bricks from the Manhattan mesh. In such a way, this dual mesh function is more complex and thus time consuming for the Manhattan mesh. Instead of using the Qhull library, the Dual Mesh Algorithm was created.

The dual information for the Manhattan mesh consists of three parts:

- For a given brick and given facet, the length of the corresponding dual link.
- For a given brick and given edge, the area of the corresponding dual surface.
- For a given brick and given node, the volume of the corresponding dual volume.

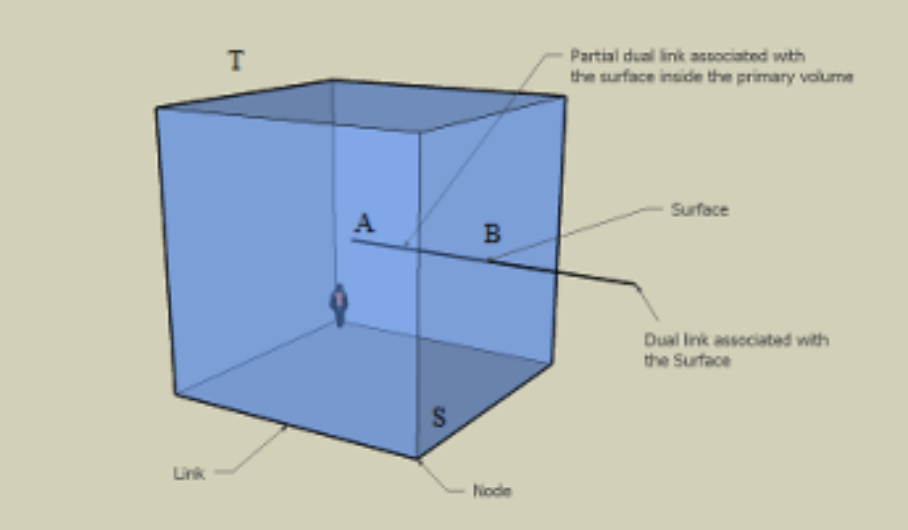

Figure 3.4: Dual-link for the primary surface

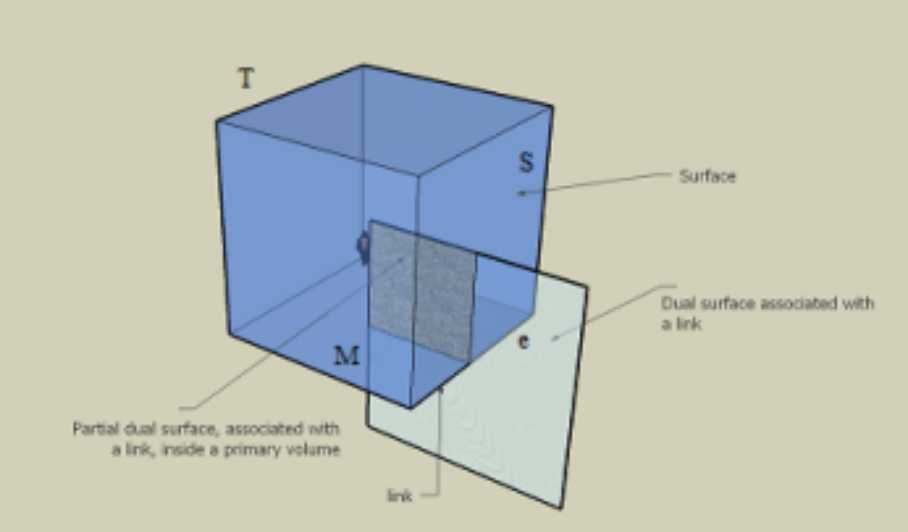

Figure 3.5: Dual-surface for the primary link

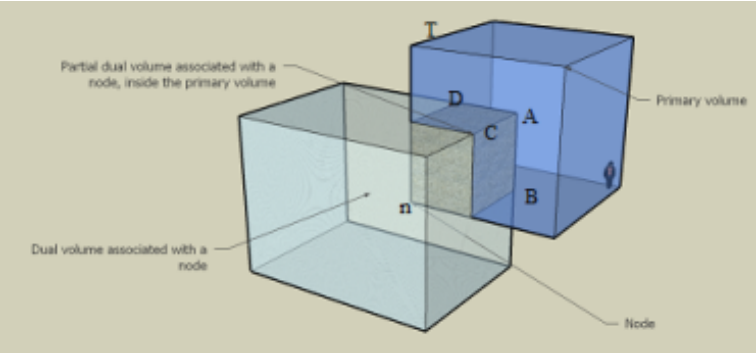

Figure 3.6: Dual-volume for the primary node

Consider the cube T in Fig. 3.4 and its right facet S. Here, the point A is the circumcenter of the cube, and B is the circumcenter of the facet S. The corresponding dual link is defined as the line between A and B. As we work only with bricks, point A will be always the center of the brick, while point B will be the center of the corresponding facet. The length of the dual link is always positive, as it lies inside the brick.

Second, we define the partial dual surface and its area. In Fig. 3.5 the cube T is given, also an edge e of T, and a facet S of T, S being incident to e. The corresponding partial dual surface is defined as the rectangle with the vertices in the center of the edge e, the circumcenters of S and T. The area of the partial dual surface is an area of a rectangle which forms it.

In Fig. 3.6, at the given cube T, the node n is the intersection of three facets, whose circumcenters are points B, C and D. A brick with vertices in points n, A, B, C and D is called the partial dual volume with the volume equal to the volume of that brick.

3.3 Algorithm Overview

Consider the Manhattan mesh of some structure resulting into a set of bricks. In our terminology, every brick is just a pair of two nodes. For our convenience, we restore the coordinates of 6 vertices of the brick and then form 12 links with their help.

After that, we calculate the coordinate of the center point A for every brick and the middle point B for every facet. Now, the dual link length for any surface is simply the distance between points A and B. It is easily seen, that the partial dual surface of the link e is formed by the dual links of two conjugate surfaces S and M. Thus, the area of the partial dual surface is the product of the corresponding dual links. At the same time, the volume of the partial dual volume of the node n is defined as $V = AC * AB * AD$, where AC, AB and AD are the dual links of the conjugate surfaces.

Algorithm 1 The Dual Mesh Algorithm

```
Generate the primary mesh with n bricks
for k = 1 to n do
  Get the coordinates of the vertices v_1, ..., v_2
  Form the links l_1, ..., l_12 and the surfaces s_1, ..., s_6
  Find a center point c of the current brick
  for i = 1 to 6 do
    Find a center point c_i of a surface s_i
    dlink_i = |c_i - c|
  end for
  for i = 1 to 11 do
    Find conjugate surfaces cs_1, cs_2 to the link l_i
    dsurf_i = 1
    for j = 1 to 2 do
      Determine the dual link dlink of the cs_j surface
      dsurf_i = dsurf_i * dlink
    end for
  end for
  for i = 1 to 8 do
    Find conjugate surfaces cs_1, cs_2,cs_3 to the node v_i
    dvolume_i = 1
    for j = 1 to 3 do
      Determine the dual link dlink of the cs_j surface
      dvolume_i = dvolume_i * dlink
    end for
  end for
end for
```

3.4 Numerical experiments

The experiments concerning the refinement of the dual mesh function for the Manhattan mesh were done. We use 4 different structures. Table 3.1 compares the computational time of the old function from the Qhull library and the new Dual Mesh algorithm.

Table 3.1: Results of NAG, LDL and Crout for the matrix $mx6$

Method	$mx1$	$mx2$	$mx3$	$mx4$
Qhull	143.75	462.00	1115.16	3257.20
Dual Mesh	20.42	61.72	157.43	453.54

The new method has much better performance than the old one. We reduced the computational cost in seven times.

Chapter 4

Direct Methods for Sparse Linear Systems of Equations

The discretization of partial differential equations leads to the problem of solving a large linear system Ax=b. The process of solving the unknowns from this linear system involves much computational work. A lot of different techniques and methods are used for solving such problems. The trick is to find the most effective method for your own problem. Unfortunately, the method which works well for one type of problem could be completely useless for another one. In our case, a matrix A is squared, sparse, positive definite and symmetric. First, we will focus on the direct methods in cooperation with the AMD ordering technique. They theoretically give us an exact solution in a finite number of steps.

4.1 Introduction

The actual computations are restricted to values on a previously constructed grid, where the number of grid points (nodes) determines the dimension of the linear system. Each grid point is associated with one or more unknowns and with equations that describe how these unknowns are related to unknowns from the neighboring grid points. These relations are dictated by the physical model. Because many grid points are necessary in order to have a realistic model of a problem, we will as a consequence have about million of equations. A nice property of these linear systems is that each equation contains only

few unknowns. Hence, the number of non-zero elements is much smaller than the size of the matrix.

In our case, the pair of equations (2.17) and (2.18) results in the matrix equation $Ax = b$. The vector of unknowns is formed by discrete values of the electric scalar potential V and the magnetic vector potential $\mathbf{A}$, in such a way that $x = [V_1, \ldots, V_n, \mathbf{A}_1, \ldots, \mathbf{A}_m]^T$, where n is the number of nodes and m is the number of links in the constructed grid. It is easy to see from the equations (2.17) and (2.18) that the amount of non-zero elements in each row is not more than 7. For the structure under study, matrix A is sparse, regular, square, positive definite and symmetric.

Nowadays, the MAGWEL software uses iterative solvers for getting a fast and efficient solution. After testing a number of different linear solvers with or without preconditioning, it turned out that the most robust method is the symmetric successive-over-relaxation (SSOR) preconditioner, combined with the conjugate-gradient (CG) iterative solver.

4.2 Sparse matrix storage format

If the matrix A is sparse, the linear system Ax=b can be solved in a more efficient way when we store only the nonzero elements. As there is no reason to operate on a huge number of zeros, such compression will give us significant savings in memory usage. This, of course, requires a special scheme for knowing the exact position of each element in the initial matrix. In that way, we are using the most general sparse matrix representation formats - Compressed Row Storage (CRS). It makes absolutely no assumptions about the sparsity structure of the matrix and on the storage of any unnecessary elements. However, it is not very efficient, because it needs an indirect addressing step for every scalar operation with matrix elements.

The matrix A is represented with three vectors: matrix values AN, column values JA and rows IA. A vector AN is constructed in such a way, that contains all the values of the non-zero elements taken row by row from the initial matrix A. The next vector JA has the same size as AN and contains the original column indices of the corresponding elements in AN. The first element of this vector is always one. The IA vector stores the locations in the AN vector that start a row.

As an example, consider the nonsymmetric matrix A defined by

$$A = \begin{pmatrix} 5 & 0 & 1 \\ 3 & 2 & 0 \\ 7 & 1 & 3 \end{pmatrix}$$

.

The CRS format for this matrix is specified then by the following arrays:

$$\begin{aligned} AN &= \{5, 1, 3, 2, 7, 1, 3\}, \\ JA &= \{1, 3, 1, 2, 1, 2, 3\}, \\ IA &= \{1, 3, 5, 7\}. \end{aligned}$$

Often, the storage scheme used arises naturally from the given problem. For instance, if the matrix is symmetric, we need only store the upper (lower) triangular part of the matrix A. Or, if it's necessary to work with the diagonal elements, we can extract them into a separate vector, and the rest will be stored as the strictly upper (lower) triangular matrix.

4.3 Sparse Direct Solution Methods

If a problem is solved with a finite number of elementary operations, then the corresponding method is said to be direct. Most direct methods for sparse linear systems rely on the LU factorization of the coefficient matrix A, generally exploiting graph models to try to minimize both the storage usage and work performed. Such solvers involve much more complicated algorithms than for dense matrices, due to the need for efficient handling the fill-in in the factors L and U.

A typical sparse direct solver for the positive definite matrices consist of four phases [16, 19]:

1. Preordering: This phase determines a reordering of the original matrix such that fill-in is reduced in the L and U factors

2. Symbolic factorization: Determine the structure of factors and set up a compact data structure for storing non-zeros of L and U.

3. Numerical factorization: In this phase, the actual values of L and U are computed. In most cases this phase is by far the most expensive in terms of serial operation count.

4. Forward/Back Solve: Solve the corresponding triangular systems by using forward and backward substitutions. This solve phase may include iterative refinement. Compared to the factorization, the cost of this phase is low, perhaps by a factor of 10-100 or more.

The main advantage of splitting up the factorization is flexibility. For example, one may need to solve many problems with the same structure of A but different numerical values, or many problems differing only in their right-hand side. Then phases 1 and 2 need to be done only once. In a solver for the simplest systems, symmetric and positive definite, all four phases can be well separated. But in spite of that fact, we will combine phases 2 and 3.

Compared to iterative linear solvers, direct solvers tend to be more reliable and accurate, but they sometimes require significantly more time and memory. In general, direct solvers are preferred when iterative methods fail or converge too slowly, or when many linear systems with the same coefficient matrix must be solved.

4.3.1 Gaussian Elimination

Dense linear systems, and sparse systems with a suitable nonzero structure, are most often solved by a direct method, such as Gaussian elimination. The main idea behind this method is to reduce the general linear system $Ax = b$ to an equivalent system of equations $Ux = y$ whose coefficient matrix is upper triangular. The system $Ux = y$ can then be solved easily by backward substitution. As an example we consider the small linear system:

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

.

The very basic idea of the Gaussian elimination is to use the first equation to eliminate the first unknown from the rest of equations, then the new second equation is used for elimination of the second unknown from the rest

of equations. In that way, we subtract $\frac{3}{5}$ times the first row from the second row and then $\frac{7}{5}$ times the first row from the third one[13]. This leads to

$$\begin{bmatrix} 5 & 0 & 1 \\ 0 & 2 & -\frac{3}{5} \\ 0 & 1 & \frac{8}{5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -4 \cdot \frac{3}{5} - 1 \\ -4 \cdot \frac{7}{5} + 2 \end{bmatrix}$$

.

Now we multiply the second row by $\frac{1}{2}$ and subtract it from the third row, one obtains

$$\begin{bmatrix} 5 & 0 & 1 \\ 0 & 2 & -\frac{3}{5} \\ 0 & 0 & \frac{19}{10} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -4 \cdot \frac{3}{5} - 1 \\ -4 \cdot \frac{11}{10} + 1 \cdot \frac{1}{2} + 2 \end{bmatrix}$$

.

Here, the coefficient matrix is an upper triangular and hence, the back substitution can be used to solve this new linear system. Using the third equation, one has

$$x_3 = -1.$$

The second equation yields

$$2x_2 = -\frac{17}{5} + \frac{3}{5}x_3 = -4.$$

Finally, from the first equation we can find easily the value of x_1

$$x_1 = \frac{1}{5}(4 - x_3) = 1.$$

Therefore, the solution of the original system is

$$x = \{1, -2, -1\}.$$

There is one more important thing to mention here. We can rewrite the obtained system in a matrix form as following:

$$Ux = Mb, \tag{4.1}$$

where U is an upper triangular matrix

$$U = \begin{pmatrix} 5 & 0 & 1 \\ 0 & 2 & -\frac{3}{5} \\ 0 & 0 & \frac{19}{10} \end{pmatrix}$$

and M is a lower triangular

$$M = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{3}{5} & 1 & 0 \\ -\frac{11}{10} & -\frac{1}{2} & 1 \end{pmatrix}$$

.

Overwriting the original matrix A, the corresponding algorithm can be described as follows[2]:

Algorithm 2 The Gaussian elimination

---

a) Forward elimination

for $k = 1$ to n-1 **do**
  for $i =$ k+1 to n **do**
    $m_{ik} = a_{ik}/a_{kk}$
    for $j =$ k+1 to n **do**
      $a_{ij} = a_{ij} - m_{ik} * a_{kj}$
    end for
    $b_i = b_i - m_{ik} * b_k$
  end for
end for

b) Backward substitution

$x_n = b_n/a_{nn}$
for $i =$ n-1 to 1 **do**
  $x_i = (b_i - \sum_{k=i+1}^{n} a_{ik}x_i)/a_{ii}$
end for

The numbers m_{ik} are sometimes called *multipliers*.

The Gaussian Elimination is known to be stable for only a few classes of matrices, such as diagonal dominant and positive definite. It is clear, that the procedure requires division by the diagonal element a_{kk}. This element is called the *pivot*. But the algorithm could fail at the kth step if the pivot a_{kk} is equal to zero. Furthermore, if the pivot is really small, the elimination procedure could be unstable. These problems are generally alleviated by using one of the pivoting strategies: partial pivoting or full pivoting[9].

4.3.2 The LU factorization

There is another way to view Gaussian elimination that is useful for the purposes of programming and handling special cases. If we multiply both sides of (4.1) by the inverse of the lower triangular matrix M, we get the system

$$LUx = b, \tag{4.2}$$

where L is also a lower triangular matrix

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{3}{5} & 1 & 0 \\ \frac{1}{5} & \frac{1}{2} & 1 \end{pmatrix}$$

.

It is easy to see that the elements in L are the multipliers and the matrix U is the final upper triangular matrix obtained from the Gaussian elimination.

In theory, if Gaussian elimination is using without row exchanging (pivoting) on the nonsingular matrix A, resulting in the upper triangular matrix U, and if L is the unit lower triangular matrix whose entrance below the diagonal are the multipliers m_{ij}, then the matrix A is factored:

$$A = LU. \tag{4.3}$$

Once the matrices L and U have been computed, solving the linear system (4.2) could be done in two steps:

1. Find y from $Ly = b$.
2. Find x from $Ux = y$.

Owing to the triangular form of the matrices L and U these equations are easy to solve. First we compute vector y by forward substitution:

$$\begin{aligned} y_1 &= \frac{b_1}{l_{11}}, \\ y_i &= \frac{1}{l_{ii}}(b_i - \sum_{j=1}^{i-1} l_{ij} y_j) \qquad i = 2, \ldots, n. \end{aligned} \tag{4.4}$$

The solution vector x is then obtained by back substitution

$$\begin{aligned} x_n &= \frac{y_n}{u_{nn}}, \\ x_i &= \frac{1}{u_{ii}}(y_i - \sum_{j=i+1}^{n} u_{ij}x_j) \qquad i = n-1, \ldots, 1. \end{aligned} \tag{4.5}$$

The general LU decomposition is used when the coefficient matrix A of the system is dense. Namely, the values of the elements in L and U are determined by comparing the two matrices, A and LU, what will bring us to the step where we need to solve N^2 equations.

This is illustrated using an example of a 4x4 matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} =$$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} & l_{21}u_{14} + u_{24} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} & l_{31}u_{14} + l_{32}u_{24} + u_{34} \\ l_{41}u_{11} & l_{41}u_{12} + l_{42}u_{22} & l_{41}u_{13} + l_{42}u_{23} + l_{43}u_{33} & l_{41}u_{14} + l_{42}u_{24} + l_{43}u_{34} + u_{44} \end{bmatrix}$$

For the general case, these values can be formulated as follows[15]:

$$\begin{aligned} u_{kj} &= a_{kj} - \sum_{p=1}^{k-1} l_{kp}u_{pj} \qquad\qquad j \geq k \geq 2, \\ l_{ik} &= \frac{1}{u_{kk}}(a_{ik} - \sum_{p=1}^{k-1} l_{ip}u_{pk}) \qquad j > k \geq 2. \end{aligned} \tag{4.6}$$

The algorithm to compute the LU decomposition can be represented in one of the forms KIJ, JKI, JIK, IKJ[9]. Since L is a lower triangular matrix with ones on the diagonal and U is an upper triangular, it is possible to store the LU factorization directly in the same memory area that is occupied by A. More precisely, U is stored in the upper part of A, whilst L occupies the lower triangular portion of A. Then, for instance, the KIJ form for dense matrces will be the following:

Algorithm 3 The LU factorization, KIJ version

for $k = 1$ to n-1 **do**
  for $i =$ k+1 to n **do**
    $a_{i,k} = a_{i,k}/a_{k,k}$
    for $j =$ k+1 to n **do**
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
    end for
  end for
end for

This implementation of the factorization algorithm is called the KIJ version, due to the ordering of three loops. In this form rows are updated as soon as the multipliers are available, we call it *immediate update* algorithm. The IKJ form of LU factorization is analogous to the KIJ form except that updates to subsequent rows of A are delayed until that row receives final processing. We call such form *delayed update* algorithm.

Algorithm 4 The LU factorization, IKJ version

for $i = 2$ to n **do**
  for $k = 1$ to i-1 **do**
    $a_{i,k} = a_{i,k}/a_{k,k}$
    for $j =$ k+1 to n **do**
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
    end for
  end for
end for

Here, at the ith step, all updates are performed on the ith row of A to produce the ith row of L and U at the same time. Thus, the ith row of U is computed as a linear combination of the previous rows of U, while the elements of L give the multipliers in this combination. It is significant that at the step i all the previous rows 1,...,i-1 are accessed, but not modified. This is illustrated in Figure 4.1.

In theory, all these different implementations of LU factorization yield the same. In order to have a kind of symmetry, the upper triangular matrix U, can be transformed to the unit upper triangular one by putting the diagonal elements of the original matrix U into a diagonal matrix D [7]. Thus, we

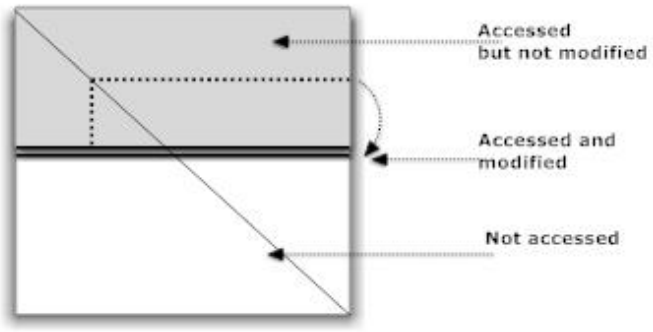

Figure 4.1: IKJ form of the LU factorization

will obtain the LDU factorization, where both matrices L and U have 1s on their diagonals.

$$A = LDU. \tag{4.7}$$

If the matrix A is symmetric, then U is a transpose of L. In this case, we can write A as

$$A = LDL^T. \tag{4.8}$$

Thus, the solution of the system will be obtained in the following way:

1. Find y from $Ly = b$.
2. Find x from $DL^T x = y$.

If A is positive-definite and symmetric, the diagonal elements are all positive initially and will remain positive throughout the Gaussian elimination process. Thus, when the LDU decomposition of A is formed, the diagonal matrix D consists of positive elements. Then we can write:

$$A = LDL^T = (LD^{\frac{1}{2}})(D^{\frac{1}{2}}L^T) = (LD^{\frac{1}{2}})(D^{\frac{1}{2}}L)^T = L_1 L_1^T. \tag{4.9}$$

Equation (4.9) is called the Cholesky decomposition of a positive-definite matrix and it will be discussed later.

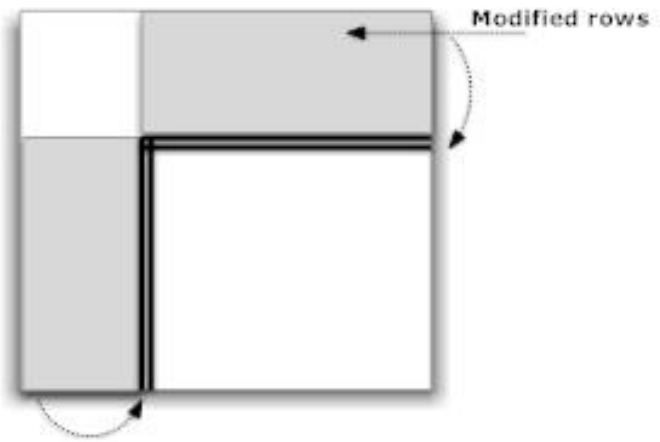

Figure 4.2: The computational scheme for the Crout factorization

4.4 The Crout factorization

The Gaussian elimination has been shown to be one form of the LU decomposition. There are several remarkable variants of this fundamental approach: the Crout factorization and Doolittle factorization[15]. They are known also as compact forms of the Gaussian elimination method (GEM), due to the fact that these approaches have eliminated much of the intermediate storage and reordering requirements than the standard GEM to generate the factorization of A.

We are using the Crout version of the GEM, which is a combination of the IKJ algorithm shown above to compute the U matrix and a transposed version to compute the L part. This method is applicable if the rows and columns are so arranged that no pivoting is required in the GEM (as in the case of symmetric, positive definite matrices). The kth step will therefore compute the pieces $U(k; k : n)$ and $L(k : n; k)$ of the decomposition. Unlike the IKJ version, all elements that will be used to update $L(k : n; k)$ and $U(k; k : n)$ at the kth step in the Crout version have already been calculated, i.e., the fill-ins will not interfere with the row updates at the kth step. To see how the Crout method works, it is convenient to consider the coefficient matrix A to be divided into three parts; part one being the region which is fully modified, part two is the region which is currently modified with the use of part one, and part three that contains the original not modified coefficients. The computational pattern for the factorization is shown in Figure 4.2

In order to describe the algorithm in a simple way, it is convenient to introduce the following definition. The inner product i of row into a column, each containing n elements, is defined as the sum of the i products of the first i corresponding elements, the elements of a row being ordered from left

to right, and the elements of a column from top to bottom.

Again, the matrix U is stored in the upper part of A, and L is stored in the strict lower triangular portion of A. From this stage, the elements of A are calculated, in the order specified above, according to two rules[18]:

1. Each element $a_{i,j}$ on or above the diagonal in A is obtained by subtracting from the original element the inner product *i-1* of column j and row i. This step is analogues to the calculation of the matrix U in the IKJ form of LUfactorization algorithm.

2. Each element $a_{i,j}$ below the diagonal is calculated by the same procedure with the use of the inner product *j-1*, followed by the division by the diagonal element $a_{i,i}$. This step calculates the pivot element for each row as it changes during the first procedure rule.

Algorithm 5 The LU factorization, Crout version

for $k = 1$ to n **do**
  for $i = 1$ to k-1 **do**
    if $a_{k,i} \neq 0$ **then**
      $a_{k,k:n} = a_{k,k:n} - a_{k,i} * a_{i,k:n}$
    end if
  end for
  for $i = 1$ to k-1 **do**
    if $a_{i,k} \neq 0$ **then**
      $a_{k+1:n,k} = a_{k+1:n,k} - a_{i,k} * a_{k+1:n,i}$
    end if
  end for
  $a_{i,k} = a_{i,k}/a_{k,k} \qquad i = k+1, \ldots, n$
end for

There are two main difficulties in the sparse implementation of algorithm:

1. Only the section $(k : n)$ of the i-th row of U is required, and similarly, only the section $(k : n)$ of the i-th column of L is needed. But extracting these parts form the entire row or column is a time consuming operation.

2. It is more convenient to store the matrix L by columns and the matrix U by rows, but then according to the Crout algorithm, an easy access to rows of L and columns of U is needed.

All elements in the factor U are stored in order by column number. At the kth step of the Crout algorithm we need to update the current row k with the use of previous modified rows. As the length of that rows is longer than the length of the current row k, a pointer for each row j, with $j < k$ will be helpful. It can be used to signal the starting point of row j needed to update the current row k. The pointers for each row are stored in a pointer array called *Ufirst*. This pointer array is updated after each elimination step by incrementing each pointer to a position of the next nonzero element in the row, if necessary. A pointer for row k is also added after the kth step.The similar pointer array is used for the L factor and is called *Lfirst*.

For the second difficulty, consider again the factor U, and the need to traverse column k of U, although U is stored by rows. For that reason, a linked list for the non-zeros elements in column k of U is used, called $Ulist$. $Ulist(k)$ contains the non-zero element in column k of U. It's not necessary the first non-zero element. Then $Ulist(Ulist(k))$ contains the other non-zero element of that column. The updates to the kth row of U can be made in any order, i.e, there is no difference which modified row to use first or last. Thus, to keep the order of non-zeros in the column k is not necessary. At the end of step k, $Ulist$ is updated so that it becomes the linked list for column $k+1$ It is obligatory, that $Ulist$ is updated after the $Ufirst$ is updated. For the L factor, there is a linked list called $Llist$.

In summary, four length n arrays are used: $Ufirst, Ulist, Lfirst$, and $Llist$:

1. $Ufirst(i)$ points to the first entry with column index greater than or equal to k in row i of U, where $i = 1; \ldots; k-1$;

2. $Ulist(k)$ points to a linked list of rows that will update row k;

3. $Lfirst(i)$ points to the first entry with row index greater than or equal to k in column i of L, where $i = 1; \ldots; k-1$;

4. $Llist(k)$ points to a linked list of columns that will update column k.

Our goal is to improve the robustness of the Crout algorithm for symmetric systems. For this reason, we use the revised version of the previous

algorithm, and compute the factorization $A = LDL^T$ instead of $A = LU$. This will allow us to reduce the memory usage during the implementation and also, it's enough to store only the L and D part of $A = LDL^T$ factorization. The first step is to show the process of calculation the LDU factorization for the general case, which was suggested by Y. Saad[17]. The corresponding algorithm can be described as follows:

Algorithm 6 The LDU factorization, Crout version

$d_{k,k} = a_{k,k}, k = 1 : n$
for $k = 1$ to n **do**
  Initialize row: $z : z_{1:k} = 0, z_{k+1:n} = a_{k,k+1:n}$
  for $i = 1$ to k-1 **do**
    if $l_{k,i} \neq 0$ **then**
      $z_{k+1:n} = z_{k+1:n} - l_{ki} * d_i * u_{i,k+1:n}$
    end if
  end for
  Initialize column: $w : w_{1:k} = 0, w_{k+1:n} = a_{k+1:n,k}$
  for $i = 1$ to k-1 **do**
    if $u_{i,k} \neq 0$ **then**
      $w_{k+1:n} = w_{k+1:n} - u_{i,k} * d_i * l_{k+1:n,i}$
    end if
  end for
  $u_{k,:} = z/d_k \qquad u_{k,k} = 1$
  $l_{:,k} = w/d_k \qquad l_{kk} = 1$
  for $i =$ k+1 to n **do**
    if $u_{ki} \neq 0$ and $l_{ik} \neq 0$ **then**
      $d_i = d_i - l_{ik} * d_k * u_{ki}$
    end if
  end for
end for

It was already mentioned that in the case when the coefficient matrix A is symmetric, the factorization LDL^T can be obtained in an easier way, as only the factor L or L^T is necessary for the further calculations. In such a way, we reduce twice the number of mathematical operations in the Algorithm 6. Also, instead of using auxiliary arrays $Ufirst, Ulist, Lfirst$, and $Llist$, only $Ufirst$ and $Ulist$ are needed.

Recall that for the programming part all sparse matrices are stored in the

Compressed Row Storage format. However, if we calculate just the factor L^T for the LDL^T factorization one can say that the matrix L is exactly the same as L^T, but is stored in the Compressed Column Storage format. And viceversa, if we get the factor L from the algorithm, it is more convenient to keep it in the Compressed Column Storage format, then L is the same matrix, but in the Compressed Row Storage format.

The adapting Crout algorithm for the symmetric case will be the following:

Algorithm 7 The LDU factorization for the symmetric matrices, Crout version

$d_{k,k} = a_{k,k}, k = 1 : n$
for $k = 1$ to n **do**
  Initialize row: $z : z_{1:k} = 0, z_{k+1:n} = a_{k,k+1:n}$
  for $i = 1$ to k-1 **do**
    if $u_{ik} \neq 0$ **then**
      $z_{k+1:n} = z_{k+1:n} - u_{ik} * d_i * u_{i,k+1:n}$
    end if
  end for
  $u_{k,:} = z/d_k \qquad u_{k,k} = 1$
  for $i =$ k+1 to n **do**
    if $u_{ki} \neq 0$ **then**
      $d_i = d_i - u_{ki} * d_k * u_{ki}$
    end if
  end for
end for

4.5 Cholesky Factorization

As mentioned earlier in this chapter, if A is a symmetric, positive-definite matrix, then it has a unique Cholesky factorization,

$$A = LL^T$$

in which L is lower triangular with a positive diagonal. The components of L^T are of course related to those of L by

$$L_{ij}^T = L_{ji} \tag{4.10}$$

The factor L is found simply by writing $(n + n^2)/2$ equations expressing each element in the lower triangular of A in terms of the elements in L. This procedure will establish the algorithm:

Algorithm 8 The Cholesky factorization

for $j = 1$ to n **do**
  $l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$
  for $i =$ j+1 to n **do**
    $l_{ij} = \frac{1}{l_{jj}} \sqrt{a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk}}$
  end for
end for

The amount of computations is reduced half compare to the LU factorization. Also, the Cholesky decomposition is extremely stable numerically, without any pivoting at all.

Chapter 5

Fill-reducing ordering

There are several methods for reducing the number of nonzero elements in the factor L. Their common feature is to permute all rows or/and columns of matrix A before applying the LDL^T factorization. We use an approximate minimum degree (AMD) ordering algorithm which is usually much faster than the other ordering methods.

5.1 Effect of Reorderings

When a sparse matrix is factored, the upper or lower triangular factors may not reflect the sparsity pattern of the original sparse matrix A. This phenomenon is called *fill-in* which corresponds to the additional non-zero elements generated during factorization. A large number of fill-ins is undesirable for two reasons:

1. Additional storage must be allocated for the extra non-zeros.
2. Number of operations needed for factorization increases with the increasing of fill-ins.

Since the amount of fill-ins depends on the row or/and column permutation, a convenient ordering of the matrix will dramatically reduce the computation time and storage requirements of the factorization. In general, it is extremely difficult to find an optimum ordering which will guarantee the smallest possible fill-in, so the existing algorithms usually find an ordering

for which the fill-in is low, but not necessarily the true minimum[19]. As an example, we take an "arrow matrix"

$$A = \begin{bmatrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{bmatrix}$$

Then, the factor L in the Cholesky factorization will have the following form and have the maximum fill-in:

$$L = \begin{bmatrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ x & x & x & x & 0 \\ x & x & x & x & x \end{bmatrix}$$

Now, if the first row and column permute with the row and column number five, and the second row and column with row and column number four, the permuted matrix A will be:

$$A = \begin{bmatrix} x & 0 & 0 & 0 & x \\ 0 & x & 0 & 0 & x \\ 0 & 0 & x & 0 & x \\ 0 & 0 & 0 & x & x \\ x & x & x & x & x \end{bmatrix}$$

Corresponding to this matrix, the factor L will change its form and have reduced the number of fill-in as low as possible. In that simple case, there is no fill-in created at all:

$$L = \begin{bmatrix} x & 0 & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ x & x & x & x & x \end{bmatrix}$$

We have now reached an important conclusion for sparse matrix factorization. The amount of fill-in depends upon the preordering process. As

we focus on fill reduction for symmetric matrices, the symmetry of the permuted matrix has to be preserved. If rows i and j are interchanged then subsequently column i and j are also exchanged. In terms of permutation matrix P, reordering can be represented as

$$\hat{A} = PAP^T.$$

Thus, preordering preserves symmetry of the matrix $\hat{A}$. As both the row and column have been interchanged, the elements of vector b as well as the elements of vector x have to be reshuffled in the same order. It can be written as follows:

$$Ax = b \quad => \quad PAP^T(Px) = Pb \quad => \quad \hat{A}\hat{x} = \hat{b}$$

Unfortunately, the problem of determining the permutation matrix P is NP-complete, so heuristics are used that attempt to reduce fill-in. We are using the Approximate Minimum Degree ordering heuristic[1]. The AMD algorithm is typically much faster than other ordering methods and minimum degree ordering algorithms that compute an exact degree.

5.2 Graph representation of sparse matrices

Strategies for the reduction of fill-in have their origins in graph theory. A useful way to represent the structure of a sparse symmetric matrix is by an undirected graph $G = (V, E)$, consisting of a set of nodes V and a set of edges E. A graph is ordered if its nodes are labeled. The ordered graph $G(A) = (V, E)$, representing the structure of a symmetric matrix $A \in R^{n \times n}$, consist of nodes labeled $1, \ldots, n$ and edges $(i, j) \in E$ if and only if $a_{ij} = a_{ji} \neq 0$. The diagonal element a_{ii} corresponds to a loop and is always present for a nonsingular matrix[3]. Thus there is a direct correspondence between non-zero elements and edges in its graph. For example, a matrix

$$A = \begin{bmatrix} x & x & & x & x & & \\ x & x & x & & & & \\ & x & x & & & x & x \\ x & & & x & & & \\ x & & & & x & & \\ & & x & & & x & \\ & & x & & & & x \end{bmatrix}$$

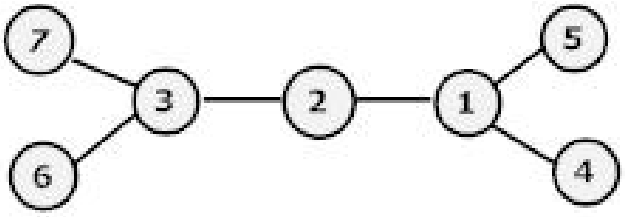

Figure 5.1: The labeled graph of matrix A

has the undirected graph which is shown in Fig. 5.1.

Two nodes, i and j, are said to be adjacent if there is an edge $(i, j) \in E$. The adjacency set of i in G is defined by

$$Adj_G(i) = \{j \in V \mid (i, j) \in E\}.$$

The number of nodes adjacent to i is denoted by d_i, and is called the degree of i

$$d_i = |Adj_G(i)|.$$

An ordering π is a bijection $V \mapsto \{1, \ldots, n\}$. Any ordering π of the nodes V induces a permutation matrix P. Therefore, a fill-reducing permutation matrix P can be obtained by finding good ordering of the nodes in V.

The symmetric Gaussian elimination can be interpreted as generating a sequence of elimination undirected graphs $G^i = (V^i, E^i), i = 0, \ldots, n-1$. The elimination graph $G^k = (V^k, E^k)$ is used to describe the non-zero pattern of the submatrix yet to be factorized after the first k pivots have been chosen. The next elimination graph $G^{k+1} = (V^{k+1}, E^{k+1})$ is formed in the following way:

1. Choose a pivot node p from V^k according to some criterion.

2. Add edges to E^k to make the nodes adjacent to i in G^{k+1} a clique. These new edges correspond to fill-in caused by the current step of the factorization.

3. Remove a pivot node p and its incident edges from G^{k+1}.

The sequence of nodes $\{v_0, v_1, \ldots\}$ selected by the algorithm defines the ordering.

The elimination graphs for the matrix in Figure 5.1 are pictured in Figure 5.2. It can be verified that the number of fill-in elements (edges) is ten.

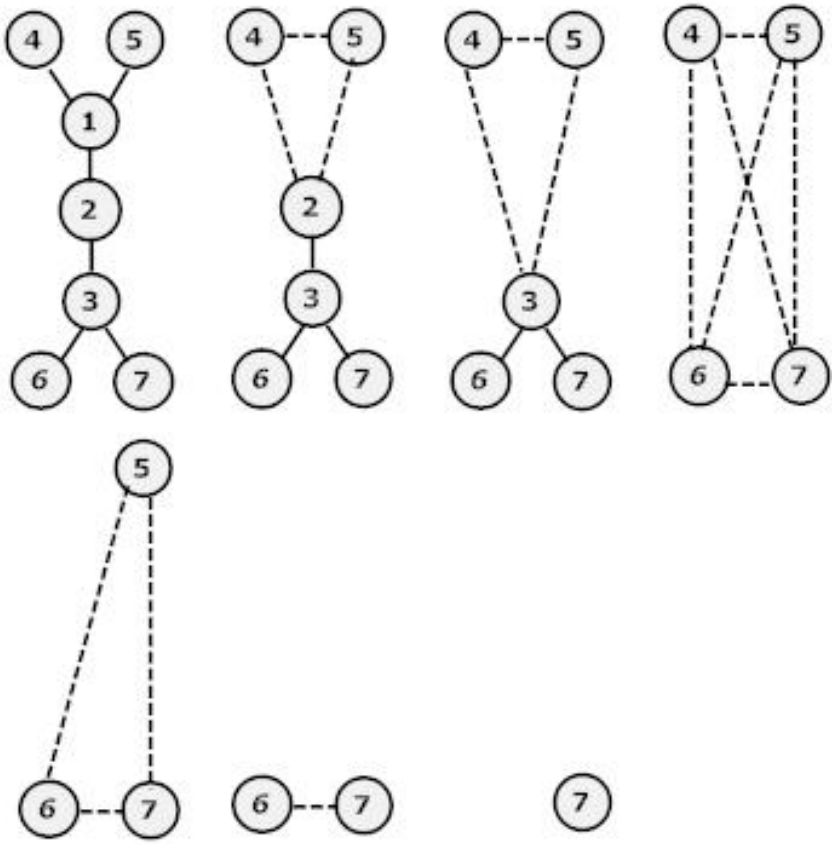

Figure 5.2: Sequence of elimination graphs of the matrix A

5.3 Minimum degree reordering

The minimum degree ordering is one of the most effective algorithms. This algorithm determines a new ordering π of the nodes in G^0 by generating a special sequence of elimination graphs[8]. In each iteration G^k is constructed from G^{k-1} by choosing a pivot node p such that the degree of p, $d_p = |Adj_{G^{k-1}}(p)|$ has the minimum value. This will minimize the number of entries that will be modified in the next elimination step, and hence tend to minimize the amount of fill-in that occurs in this step. Selecting p as pivot creates at most $(d_p^2 - d_p)/2$ new edges in G^k and the degree of some nodes in G^k must be recomputed.

However, the idea behind the algorithm is very simple, but conceals a great deal of complexity. The first complication is in its storage requirements. The second complication comes from redundant computations.

First, Algorithm 9 is an outline of the minimum degree algorithm, based on the elimination graph. This algorithm computes the degrees of the nodes at each step, so that the pivot can be chosen for the next step. Only the degrees of the nodes in $Adj_{G^{k-1}}(p)$ need to be updated after p is selected.

All other degrees are unchanged. The algorithm explicitly adds edges to G, causing the storage requirements to be unpredictable.

Algorithm 9 Minimum Degree Ordering, based on elimination graph

$V^0 = \{1 \dots n\}$
$E^0 = \{(i,j) : a_{ij} \neq 0 \text{ and } i \neq j\}$
for $i = 1$ to n **do**
  $d_i = |Adj_{G^0}(i)|$
end for
for $k = 1$ to n **do**
  Select kth pivot, $p \in V^{k-1}$ with the minimum value of d_p
  for $i \in Adj_{G^{k-1}}(p)$ **do**
    $Adj_{G^k}(i) = (Adj_{G^{k-1}}(i)) \cup Adj_{G^{k-1}}(p)) \setminus \{i, p\}$
    $d_i = |Adj_{G^k}(i)|$
  end for
  $V^k = V^{k-1} \setminus \{p\}$
end for

It was mentioned already that all new edges in each elimination step are created and stored explicitly. In other words, a new edge must be created for every instance of fill-in. For some problems this can mean millions of newly created edges and the storage requirements of the algorithm become infeasible. To remedy this, the algorithm stays the same but quotient graphs replace elimination graphs.

The strategy is to introduce a special type of vertex. We no longer remove pivot node p during the elimination iteration. Instead, it will be labeled as eliminated. Such nodes are referred to as *elements*. We use the term *variable* to refer to unlabeled nodes.

The quotient graph, $\mathcal{G}^k = (V^k \cup \overline{V}^k, E^k \cup \overline{E}^k)$, represents implicitly the elimination graph G^k, and the initial quotient graph $\mathcal{G}^0$ is the same as the initial elimination graph G^0. Now, the nodes in $\mathcal{G}$ consist of variables (V), and elements ($\overline{V}$). At the same time, the edges in $\mathcal{G}$ are divided in two sets: edges between variables E, and between variables and elements $\overline{E}$. The elements can not be connected between each other as they are removed by element absorption(described below).

For the sake of simplicity, we will introduce the following notation. The adjacency list for an unlabeled node i is split into two sets: $\mathcal{A}_i^k$ is a list of

variables (original non-zeroes a_{ij}) adjacent to variable i in $\mathcal{G}^k$, and $\mathcal{E}_i^k$ is a set of elements adjacent to variable i. In other words,

$$\mathcal{A}_i = \{j : (i,j) \in E\}, \tag{5.1}$$

$$\mathcal{E}_i = \{e : (i,e) \in \overline{E}\}, \tag{5.2}$$

and

$$Adj_G(i) = \mathcal{A}_i \cup \mathcal{E}_i. \tag{5.3}$$

Let $\mathcal{L}_e$ be a set of variables adjacent to element e in $\mathcal{G}$. It means, if $e \in \overline{V}$, then

$$\mathcal{L}_e = \{i : (i,e) \in \overline{E}\}, \tag{5.4}$$

and

$$Adj_G(e) = \mathcal{L}_e. \tag{5.5}$$

The following formula shows the connection between the quotient graph $\mathcal{G}$ and the elimination graph G in the sense of adjacency of node $i \in G, \mathcal{G}$

$$Adj_G(i) = \mathcal{A}_i \cup (\bigcup \mathcal{L}_e) \setminus \{i\}. \tag{5.6}$$

When a pivot node p is selected on the kth iteration step, the variable p is moved from set V to $\overline{V}$ and the element p is created. Next step is to find the list of new adjacent variables for the element p $\mathcal{L}_p$ by using Equation (5.6). When node p is eliminated, all elements $e \in \mathcal{E}_p$ are no longer required to represent the non-zero pattern of factor L, they are absorbed into the new element p, and deleted. After that, we add element p to all sets $\mathcal{E}_i$, where i adjacent to variable p. Also, we delete absorbed elements $e \in \mathcal{E}^p$ from all elements lists, and delete sets $e \in \mathcal{A}_p$, $e \in \mathcal{E}_p$ and $e \in \mathcal{L}_e$ of that elements. Finally, any entry j in $\mathcal{A}_i$, where both i and j are in $\mathcal{L}_p$ is deleted.

When a node is eliminated there is often a subset of nodes that can be eliminated in the same elimination step. We say that two nodes i and j are indistinguishable if

$$Adj_G(i) \cup \{i\} = Adj_G(j) \cup \{j\}.$$

It is obvious that if two nodes are indistinguishable, their degrees must be the same. If two nodes i and j become identical ($\mathcal{E}_i = \mathcal{E}_j$ and $\mathcal{A}_i = \mathcal{A}_j$), they will remain identical until one of them is eliminated, they will have identical neighborhood sets (and hence the same degree). Furthermore, if i is selected, then j can be selected next without casing any additional fill-in. We merge indistinguishable nodes nodes together and treated as one, which we will call a supernode. In this way, we need only to consider one representative from each group of indistinguishable nodes. The advantage of using indistinguishable nodes should be clear. We need to perform the degree update only on the representative nodes. This reduces the operating size of the current elimination graph in terms of both nodes and edges. This process is called *mass elimination.*

We denote the set of simple variables in the supernode i as $\mathcal{V}_i$ and define $\mathcal{V}_i = \{i\}$ if i is a simple variable. We use the set notation $\mathcal{A}$ and $\mathcal{L}$ for the simple variables, and Principal($\mathcal{A}$) and Principal($\mathcal{L}$) for the principal variables in $\mathcal{A}$ and $\mathcal{L}$. When p is selected as a pivot node, all variables in $\mathcal{V}_p$ are eliminated, which reduce the computation time.

Another technique discards the use of (5.6), and uses an approximation degrees instead of true degrees, which is cheaper to compute. External degrees of supernodes are equal to the true degree minus the number of nodes merged into the supernode. Since supernodes form a clique, no internal fill-in will be created, so external degrees form a tighter bound than true degrees. Therefore, the degree update should be

$$d_i = |Adj_G(i) \setminus \mathcal{V}_i| = |\mathcal{A}_i \setminus \mathcal{V}_i| + |(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e) \setminus \mathcal{V}_i|. \tag{5.7}$$

A minimum degree algorithm based on the quotient graph is shown in Algorithm 10. However this algorithm doesn't include to important features from *Multiple Minimum Degree* algorithm: incomplete update and multiple elimination. The idea behind the multiple elimination algorithm is the following, If we eliminate a node p we need to calculate the degree of its neighbors. But if we find another node with the same minimal degree to eliminate that is not a neighbor of p we can postpone updating the degrees. A variable j is *outmatched* if $Adj_G(i) \subseteq Adj_G(j)$. With incomplete degree update, the degree update of the outmatched variable j is avoided until variable i is selected as a pivot. We will discuss the relationship of these two features to the AMD algorithm in the next section.

Algorithm 10 Minimum Degree Ordering, based on quotient graph

$V = \{1 \dots n\}$
$\overline{V} = \emptyset$
for $i = 1$ to n **do**
  $\mathcal{A}_i = \{j : a_{ij} \neq 0 \text{ and } i \neq j\}$
  $\mathcal{E}_i = \emptyset$
  $d_i = |\mathcal{A}_i|$
  $\mathcal{V}_i = \{i\}$
end for
k=1
while $k \leq n$ **do**
  mass elimination:
  select variable $p \in V$ that minimizes d_p
  $\mathcal{L}_p = \mathcal{A}_p \cup (\bigcup_{e \in \mathcal{E}_p} \mathcal{L}_e) \setminus \mathcal{V}_p$
  for each $i \in Principal(\mathcal{L}_p)$ **do**
    $\mathcal{A}_i = \mathcal{A}_i \setminus \mathcal{L}_p$
    element absorption:
    $\mathcal{E}_i = (\mathcal{E}_i \cup \{p\}) \setminus \mathcal{E}_p$
    compute external degree:
    $d_i = |\mathcal{A}_i \setminus \mathcal{V}_i| + |(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e) \setminus \mathcal{V}_i|$
  end for
  supervariable detection, pairs found via hash function
  for each pair i and $j \in Principal(\mathcal{L}_p)$ **do**
    if i and j are indistinguishable **then**
      $\mathcal{V}_i = \mathcal{V}_i \cup \mathcal{V}_j$
      $d_i = d_i - |\mathcal{V}_j|$
      $V = V \setminus \{j\}$
      $\mathcal{A}_j = \emptyset$
      $\mathcal{E}_j = \emptyset$
    end if
  end for
  $\overline{V} = (\overline{V} \cup \{p\}) \setminus \mathcal{E}_p$
  $V = V \setminus \{p\}$
  $\mathcal{A}_p = \emptyset$
  $\mathcal{E}_p = \emptyset$
  $k = k + |\mathcal{V}_p|$
end while

5.4 Approximate minimum degree

The most costly part of the minimum degree algorithm is the re-computation of the degrees of nodes adjacent to the current pivot element p. Instead of calculating the exact degree d_i, the *Approximate Minimum Degree ordering*(AMD)[1] algorithm finds an upper bound $\overline{d_i}$ on the degree that is easier to compute. For nodes of least degree, this bound tends to be tight. Using the approximate degree instead of the exact degree leads to a substantial savings in run time. It has no effect on the quality of the ordering.

This approximate degree is a minimum of three upper bounds for the degree. Let p be a pivot node on the kth elimination step, then the approximate degree is

$$\overline{d_i}^k = min \begin{cases} n-k \\ \overline{d_i}^{k-1} + |\mathcal{L}_p \setminus \mathcal{V}_i| \\ |\mathcal{A}_i \setminus \mathcal{V}_i| + |\mathcal{L}_p \setminus \mathcal{V}_i| + \sum_{e \in \mathcal{E}_i \setminus \{p\}} |\mathcal{L}_e \setminus \mathcal{L}_p| \end{cases} \tag{5.8}$$

The first bound is equal to the number of nodes left in the graph. The second bound is equal to the old degree plus the worst case fill-in. For the third bound we calculate the number of directly adjacent nodes, the number of adjacent nodes through the new element, and the number of nodes through other elements that are not present in the current element. Thus,all nodes adjacent to the current node are counted.

Algorithm 11 computes $|\mathcal{L}_e \setminus \mathcal{L}_p|$ for all elements, where all variables assumed to be simple. The term $w(e)$ is initialized to $|\mathcal{L}_e|$ when the algorithm goes through the element e, otherwise it will be still -1. Then it is decremented once for each variable i in the internal subset $\mathcal{L}_e \cap \mathcal{L}_p$. In the end $w(e) = |\mathcal{L}_e| - |\mathcal{L}_e \cap \mathcal{L}_p|$. In other words,

$$|\mathcal{L}_e \cap \mathcal{L}_p| = \begin{cases} w(e) & \text{if} \quad w(e) \geq 0 \\ |\mathcal{L}_e| & \text{otherwise} \end{cases} \tag{5.9}$$

Algorithm 12 computes $|\mathcal{L}_e \setminus \mathcal{L}_p|$ for all supernodes. This algorithm allows easy reuse of the array w for the next pivot by incrementing w_0 by one plus the maximum size of any element seen so far. At the end of the algorithm, we have

$$|\mathcal{L}_e \cap \mathcal{L}_p| = \begin{cases} w(e) - w_0 & \text{if} \quad w(e) \geq w_0 \\ |\mathcal{L}_e| & \text{otherwise} \end{cases} \tag{5.10}$$

At the start of factorization we initialize w_0 and w to zero and -1, respectively.

Algorithm 11 Computation of $|\mathcal{L}_e \setminus \mathcal{L}_p|$ for all elements

```
assume w(1...n) < 0)
for each variable i ∈ L_p do
  for each element e ∈ E_i do
    if w(e) < 0 then
      w(e) = |L_e|
    end if
    w(e) = w(e) − 1
  end for
end for
```

Algorithm 12 Computation of $|\mathcal{L}_e \setminus \mathcal{L}_p|$ with supernodes

```
assume w(1...n) < w_0)
for each variable i ∈ Principal(L_p) do
  for each element e ∈ E_i do
    if w(e) < w_0 then
      w(e) = |L_e| + w_0
    end if
    w(e) = w(e) − |V_i|
  end for
end for
```

Thus, the Approximate Minimum Degree algorithm is identical to Algorithm 10, except that the exact degree d_i is replaced with the approximate $\overline{d_i}$.

Chapter 6

The LDL Package

Over the years a great deal of research effort has been devoted to finding efficient direct solvers for large sparse symmetric linear systems of equations. Direct methods are important because of their generality and robustness. Indeed, for the tough linear systems arising from some applications, they are currently the only feasible solution methods. In many other cases, direct methods are the method of choice because finding and computing a good preconditioner for an iterative method can be computationally more expensive than using a direct method.

6.1 Overview

The performance of 11 different sparse direct solvers packages was compared in a paper by Nick Gould, Yifan Hu and Jennifer Scott[10]. They used the following packages: BCSLIB-EXT, CHOLMOD, MA57, MUMPS, Oblio, PARDISO, SPOOLES, SPRSBLKLLT, TAUCS, UMFPACK and WSMP, on 87 large symmetric positive definite matrices. It was found that CHOLMOD was the fastest for 42 of the 87 matrices and was in the top of fastest packages for 73 out of 87 matrices. Without any doubt, we can make a conclusion, that CHOLMOD is the best candidate for solving large sparse positive definite systems.

CHOLMOD is a set of ANSI C routines for solving sparse symmetric linear systems that is being developed by Tim Davis from the University of Florida[4]. It includes both a supernodal method and a nonsupernodal up-looking method that does not exploit the BLAS. An up-looking Cholesky

factorization computes L one row at a time, and is much faster than the supernodal method for very sparse matrices. After communication with Tim Davis, it turned out, that that the latter factorization method is also implemented in a more simple way in the LDL package, which was also developed by him.

LDL is a set of short, concise routines that compute the LDL^T factorization of sparse symmetric matrix A, where the lower triangular factor L is computed row-by-row without numeric pivoting[5].

6.2 Algorithm

It was mentioned before, that all sparse direct solvers have a number of common phases. The first phase is the preordering strategy, and towards this end we use the AMD preordering algorithm again. Following the ordering phase, the analysis can be performed in many different ways. The idea of getting the structure of the matrix L was taken from the article of Joseph W. H. Liu[14].

6.2.1 Symbolic factorization

A given symmetric sparse matrix A is represented again by its associated graph $G(A) = (X(A), E(A))$, where nodes in $X(A)$ correspond to rows and columns of the matrix and edges in $E(A)$ correspond to nonzero elements. We shall use u, v to indicate an edge between two nodes u and v.

Let $x_1, x_2, \ldots, x_n$ be the sequence of nodes. Moreover, let $G(F)$ be the associated filled graph, where $F = L + L^T$ is filled matrix of A. It is well known that $G(F)$ has the same set of nodes and is a supergraph of $G(A)$.

Consider the Cholesky factor L and the filled matrix F of A. For each column $j < n$ of L, remove all the non-zeroes in this column except the first non-zero below the diagonal. Let L_t be the resulting matrix and $F_t = L_t + L_t^T$. The graph $G(F_t)$ is a tree structure, and it depends entirely on the structure of the original sparse matrix A and its initial ordering. We use $T(A)$ to denote this tree structure and refer it as the *elimination tree* of A.

The elimination tree $T(A)$ has the same node set as $G(A)$. We define the node x_n to be the root of this tree $T(A)$. The elimination tree structure can be conveniently represented by the $PARENT[*]$ vector of F_t. In terms of the triangular factor L, we have, for $j < n$,

$$PARENT[j] = min\{i > j | l_{ij} \neq 0\}. \tag{6.1}$$

It follows directly from the definition of the $PARENT$ vector that, if x_i is a proper ancestor node of x_j in the elimination tree, then $i > j$.

We want to introduce the subtree notation. We use $T[x]$ to represent the subtree of $T(A)$ rooted at the node x, which includes all descendants of x in the tree T, also $T[x]$ denote the set of nodes in this subtree. Since x_n is the root of the tree, we have $T(A) = T[x_n]$. If $y \in T[x]$, then the node y is a descendent of x, and x an ancestor of y.

Theorem 6.2.1 provides a necessary condition in terms of the ancestor-descendant relation in the elimination tree for an entry to be nonzero in the filled matrix[14].

Theorem 6.2.1 *If $l_ij \neq 0$, then the node x_i is an ancestor of x_j in the elimination tree.*

We now consider a necessary and sufficient condition for an entry to be nonzero in the filled graph. Rose, Tarjan, and Lueker characterize edges in the filled graph based on the special type of paths in the original graph $G(A)$. We quote the following "path theorem" from them [14].

Theorem 6.2.2 *Let $i > j$. Then $l_{ij} \neq 0$ if and only if there exists a path*

$$x_i, x_{p_1}, \ldots, x_{p_t}, x_j$$

in the graph $G(A)$ such that all subscripts in $\{p_l, \ldots, p_t\}$ are less than j.

We now extend this path condition in terms of subtrees in the elimination tree.

Theorem 6.2.3 *Let $i > j$. Then $l_{ij} \neq 0$ if and only if there exists a path*

$$x_i, x_{p_1}, \ldots, x_{p_t}, x_j$$

in the graph $G(A)$ such that a $\{x_{p_1}, \ldots, x_{p_t}\} \subseteq T[x_i]$.

Theorems 6.2.2 and 6.2.3 characterize edges in the filled graph in terms of paths. We extend this properties and provide a different necessary and sufficient condition for entries in the Cholesky factor L to be non-zero.

Theorem 6.2.4 *$l_{ij} \neq 0$ if and only if the node x_j is an ancestor of some node x_k in the elimination tree, where $a_{ij} \neq 0$.*

The result of this theorem can be used to devise efficient algorithms for counting non-zeroes in the Cholesky factor L. Let $n(L_{i*})$ and $n(L_{*j})$ be the number of non-zeroes in the ith row and jth column of the factor L, respectively. These quantities can be computed efficiently using the following algorithm. Here, $marker[*]$ is a working integer vector used in the algorithm to mark nodes that have been considered in the current row structure.

Algorithm 13 Non-zero count for the factor L

```
for j=1 to n do
  n(L_{*j})=1
end for
for i=1 to n do
  n(L_{i*})=1
  marker[x_i] = i
  for k < i=1 and a_{ik} ≠ 0 do
    traverse and mark nodes in the row subtree T_r[x_i]
    j = k
    while marker[x_j] ≠ i do
      n(L_{i*}) = n(L_{i*}) + 1
      n(L_{*j}) = n(L_{*j}) + 1
      marker[x_j] = i
      j = PARENT[j]
    end while
  end for
end for
```

Due to that algorithm, the storage requirement for values of L can be obtained from only the structure of A and the elimination tree. It is easy to verify that Algorithm 13 takes time proportional to the number of non-zeros in the Cholesky factor L and space proportional to the number of non-zeros in the original matrix A.

6.2.2 Numerical factorization

The kth step solves a lower triangular system of dimension $k-1$ to compute the kth row of L and the d_{kk} entry of the diagonal matrix D. Colon notation

is used for submatrices. For instance, $L_{k,1:k-1}$ refers to the first $k-1$ columns of the kth row of L. Similarly, $L_{1:k-1,1:k-1}$ refers to the leading $(k-1)$by$(k-1)$ submatrix of L.

Algorithm 14 LDL^T factorization of an symmetric matrix A

for k=1 to n **do**
(step 1) Solve $L_{1:k-1,1:k-1}y = A_{1:k-1,k}$ for y
(step 2) $L_{k,1:k-1} = (D^{-}_{1:k-1,1:k-1}1y)^T$
(step 3) $l_{kk} = 1$
(step 4) $d_{kk} = a_{kk} - L_{k,1:k-1}y$
end for

Step 1 of Algorithm 14 requires a triangular solve of the form $Lx = b$, where all three terms in the equation are sparse. This is the most costly step of the Algorithm. For simplicity, in the kth step of factorization, we can use the set X which is the non-zero pattern of row k of L. This step requires the elimination tree of $L_{1:k-1,1:k-1}$ and must construct the elimination tree of $L_{1:k,1:k}$ for step $k+1$. Recall that the parent of i in the tree is the smallest j such that $i < j$ and $l_{ji} \neq 0$. Thus, if any node i already has a parent j, then j will remain the parent of i in the elimination trees of all other leading submatrices of L, and in the elimination tree of L itself. If $l_{ki} \neq 0$ and i does not have a parent in the elimination tree of $L_{1:k-1,1:k-1}$, then the parent of i is k in the elimination tree of $L_{1:k,1:k}$. Node k becomes the parent of any node $i \in X$ that does not yet have a parent.

Chapter 7

Numerical experiments

In this section, we compare the performance of the Crout direct solver, the LDL package and the Conjugate Gradient iterative method with SSOR preconditioner from the NAG library. The computational code for the Crout method was written in Fortran, the code for the LDL package is in C, and the NAG library is also written in Fortran. The experiments were conducted on a 2.4GHz Intel Xeon PC with 8 cores, 64GB of main memory and 3072 KB cache.

We tested Crout, LDL, and NAG on 6 sparse symmetric positive-definite matrices. Some generic information on these matrices is given in Table 7.1, where n is the matrix size and nnz is the total number of non-zeros in a full matrix.

Table 7.1: Symmetric positive-definite matrices

Matrix	n	nnz
mx1	78 694	482 350
mx2	354 236	2 196 827
mx3	598 314	3 678 246
mx4	1 295 487	8 165 279
mx5	2 071 004	12 845 046
mx6	3 954 423	24 231 563

The Crout method was implemented in such a way, that we can make a decision which number is considered to be a zero. Hence, by varying the precision of zero (from 10^{-13} to 10^{-16}), the accuracy of the solution could

be changed. This conclusion is quite obvious, as the dropping of some really small elements in the matrix L, make the LDL^T factorization not complete, but a little bit rough.

We will use the following notation:

1. *Zero* is the lower-bound for numbers to consider them non-zeroes in the Crout method.

2. *Lnz* is the number of non-zero elements of the factor L.

3. *Slnz* is the number of non-zero elements of the factor L computed by the symbolic factorization function of the LDL package.

4. *Residue* is the norm of the residual vector $r(x) = (r_1(x), \ldots, r_n(x)) = b - Ax$. The norm is calculated in the following way

$$||r(x)|| = \sqrt{\frac{r_1^2(x)}{max|a_{1j}|} + \cdots + \frac{r_n^2(x)}{max|a_{nj}|}} \quad j = 1, \ldots, n.$$

5. *Cbalance* is a current balance. This value supposed to be close to 0 and less then 1% from the smallest current value in order to consider the solution is correct.

6. *Time* is the computing time.

7. *Memory* is the memory usage during computations.

Table 7.2 shows the results of the Crout method for the matrix $mx1$.

Table 7.2: Crout method for the matrix $mx1$

Zero	*Lnz*	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(MB)
10^{-12}	5 027 253	77.2834	$-8.583\mathbf{e}^{-05}$	6.08	68
10^{-13}	5 509 536	6.2912	$-6.859\mathbf{e}^{-06}$	6.33	84
10^{-14}	5 924 215	0.5383	$-5.546\mathbf{e}^{-07}$	7.23	92
10^{-15}	6 270 958	0.0459	$-4.463\mathbf{e}^{-08}$	8.67	92
10^{-16}	6 556 473	0.0043	$-4.126\mathbf{e}^{-09}$	8.85	95
10^{-17}	6 791 075	0.0019	$-1.007\mathbf{e}^{-09}$	9.43	95

The $Slnz$ value for this matrix is equal to 7 568 304 and the smallest current value is 0.03112 A. This means that all obtained results satisfy the condition of the correct solution and are acceptable. In Table 7.3 we compare results of the LDL package and NAG library. Since the Crout method is able to give us a solution with different accuracies, only the version where numbers less than 10^{-13} are considered to be zeros, is added for the comparison.

Table 7.3: Results of NAG, LDL and Crout for the matrix $mx1$

Method	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(MB)
NAG	7.7991	$1.143\mathbf{e}^{-07}$	20.55	20
LDL	0.0021	$-9.006\mathbf{e}^{-10}$	10.1	106
Crout	6.2912	$-6.859\mathbf{e}^{-06}$	6.33	84

We can see from the table, that if we gain in time one has to pay by increasing the memory usage.

Table 7.4: Crout method for the matrix $mx2$

Zero	*Lnz*	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(GB)
10^{-12}	33 692 043	221.1828	$-4.349\mathbf{e}^{-04}$	68.43	0.4
10^{-13}	36 883 788	16.6203	$-3.241\mathbf{e}^{-05}$	93.40	0.5
10^{-14}	39 465 891	1.2731	$-2.456\mathbf{e}^{-06}$	100.43	0.5
10^{-15}	41 561 973	0.1016	$-1.926\mathbf{e}^{-07}$	105.50	0.6
10^{-16}	43 270 547	0.0121	$-1.920\mathbf{e}^{-08}$	111.56	0.6
10^{-17}	44 663 309	0.0082	$-5.628\mathbf{e}^{-09}$	137.29	0.6

The results of the Crout method for $mx2$ are given in Table 7.4, while the value of $Slnz$ is 49 028 614.

Table 7.5: Results of NAG, LDL and Crout for the matrix $mx2$

Method	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(GB)
NAG	15.2639	$4.334\mathbf{e}^{-08}$	192.94	0.1
LDL	0.0087	$-4.924\mathbf{e}^{-09}$	124.17	0.6
Crout	16.6203	$-3.241\mathbf{e}^{-05}$	93.40	0.5

The smallest current value is 0.1181A and all obtained results for the Crout method are acceptable.

We continue our numerical experiments with the matrix $mx3$. Table 7.6 compares the results of the Crout method for that matrix.

Table 7.6: Crout method for the matrix $mx3$

Zero	Lnz	$Residue$	$Cbalance$(A)	$Time$(s)	$Memory$(GB)
10^{-12}	55 395 033	112.2523	$-2.632\mathbf{e}^{-04}$	97.92	0.8
10^{-13}	60 607 003	9.2691	$-2.068\mathbf{e}^{-05}$	113.54	0.8
10^{-14}	64 973 296	0.7745	$-1.654\mathbf{e}^{-06}$	139.13	0.9
10^{-15}	68 588 549	0.0685	$-1.361\mathbf{e}^{-07}$	147.70	0.9
10^{-16}	71 524 265	0.0089	$-1.439\mathbf{e}^{-08}$	172.83	0.9
10^{-17}	73 854 022	0.0056	$-4.846\mathbf{e}^{-09}$	183.47	1.0

For the matrix $mx3$, the value of $Slnz$ would be 80 198 670. And, in that case, the smallest current value is 0.3184A and solutions are correct.

Table 7.7: Results of NAG, LDL and Crout for the matrix $mx3$

Method	$Residue$	$Cbalance$(A)	$Time$(s)	$Memory$(MB)
NAG	6.8934	$-9.234\mathbf{e}^{-09}$	297.64	0.1
LDL	0.0062	$-4.921\mathbf{e}^{-09}$	172.19	1.0
Crout	9.2691	$-2.068\mathbf{e}^{-05}$	113.54	0.8

In Table 7.8 we see the results of numerical experiments for the matrix $mx4$.

Table 7.8: Crout method for the matrix $mx4$

Zero	Lnz	$Residue$	$Cbalance$(A)	$Time$(s)	$Memory$(GB)
10^{-13}	167 085 542	129.4490	$-3.207\mathbf{e}^{-04}$	630.90	2.3
10^{-14}	181 554 892	10.7933	$-2.639\mathbf{e}^{-05}$	693.67	2.5
10^{-15}	193 549 557	0.9075	$-2.148\mathbf{e}^{-06}$	871.98	2.6
10^{-16}	203 324 402	0.0851	$-1.918\mathbf{e}^{-07}$	879.49	2.8
10^{-17}	211 193 835	0.0322	$-3.812\mathbf{e}^{-08}$	960.04	2.8

Here, the maximum number of fill-in in the factor L is 235 155 462. The experiment where all elements less than 10^{-12} are considered to be zeros, haven't been done. This is because the solution of the direct method for such large matrix was too rough. For the rest, all results were acceptable, as the smallest current value was 0.07466A.

Table 7.9: Results of NAG, LDL and Crout for the matrix $mx4$

Method	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(GB)
NAG	454.7840	$2.063\mathbf{e}^{-06}$	1486.58	0.3
LDL	0.0337	$-2.823\mathbf{e}^{-08}$	876.31	3.0
Crout	10.7933	$-2.639\mathbf{e}^{-05}$	693.67	2.5

Table 7.10 shows the results of the experiments with the Crout method for the matrix $mx5$.

Table 7.10: Crout method for the matrix $mx5$

Zero	*Lnz*	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(GB)
10^{-13}	237 984 984	61.5834	$-2.263\mathbf{e}^{-04}$	739.27	3.0
10^{-14}	266 211 954	5.5592	$-1.989\mathbf{e}^{-05}$	937.43	3.0
10^{-15}	291 376 478	0.4841	$-1.689\mathbf{e}^{-06}$	1048.76	3.3
10^{-16}	313 245 957	0.0471	$-1.535\mathbf{e}^{-07}$	1214.34	3.6
10^{-17}	331 824 160	0.0179	$-2.937\mathbf{e}^{-08}$	1367.80	4.0

For that matrix the value of $Slnz$ is 401 287 004. As the smallest current value is 0.104701A, it is enough to consider all obtained results to be correct.

To compare the linear solvers for the matrix $mx5$ we will look for results in Table 7.11.

Table 7.11: Results of NAG, LDL and Crout for the matrix $mx5$

Method	*Residue*	*Cbalance*(A)	*Time*(s)	*Memory*(GB)
NAG	31.4957	$-2.718\mathbf{e}^{-08}$	1811.29	0.3
LDL	0.0215	$-2.880\mathbf{e}^{-08}$	1414.63	4.0
Crout	5.5592	$-1.989\mathbf{e}^{-05}$	937.43	3.0

Finally, we do the experiments with the biggest testcase, which is matrix $mx6$.

Table 7.12: Crout method for the matrix $mx6$

Zero	Lnz	$Residue$	$Cbalance$(A)	$Time$(s)	$Memory$(GB)
10^{-13}	539 835 090	127.4020	$-6.483\mathbf{e}^{-04}$	2118.89	6.0
10^{-14}	603 948 021	11.6017	$-5.653\mathbf{e}^{-05}$	2721.42	6.0
10^{-15}	660 599 268	1.0092	$-4.781\mathbf{e}^{-06}$	3336.39	8.0
10^{-16}	709 240 699	0.0995	$-4.369\mathbf{e}^{-07}$	3929.74	8.0
10^{-17}	750 047 966	0.0363	$-8.217\mathbf{e}^{-08}$	4437.46	9.0

The value of $Slnz$ is 892 923 742. The smallest current value is 0.58327A, thus all results are correct.

Table 7.13: Results of NAG, LDL and Crout for the matrix $mx6$

Method	$Residue$	$Cbalance$(A)	$Time$(s)	$Memory$(GB)
NAG	39.8080	$9.906\mathbf{e}^{-08}$	6124.17	1.0
LDL	0.0421	$-6.320\mathbf{e}^{-08}$	4683.35	10.0
Crout	11.6017	$-5.653\mathbf{e}^{-05}$	2721.42	6.0

From the experiments, it is clear that for the efficient usage of the Crout method, it is enough to say that all elements which are less than 10^{-13} are considered to be zero. In that case, the current balance is sufficiently small, approximately 0.03% from the smallest current value.

A very interesting observation would be to compare graphically the computational and the memory cost for observing methods. In Figure 7.1 we show the computational time which was spent on solving the linear system $Ax = b$ for six matrices A, listing in Table 7.1. From this figure, we see that the Crout method is more efficient than the LDL package and the NAG library. It solves all linear systems, in average, twice faster than NAG, and 1.5 times faster than the LDL package. The most important result for us is that the Crout method performs stably for all numerical experiments.

Figure 7.2 shows the memory required for computing the solution of the linear system $Ax = b$ for the matrices listed in in Table 7.1. The figure shows that the NAG library is using much less memory than all the other

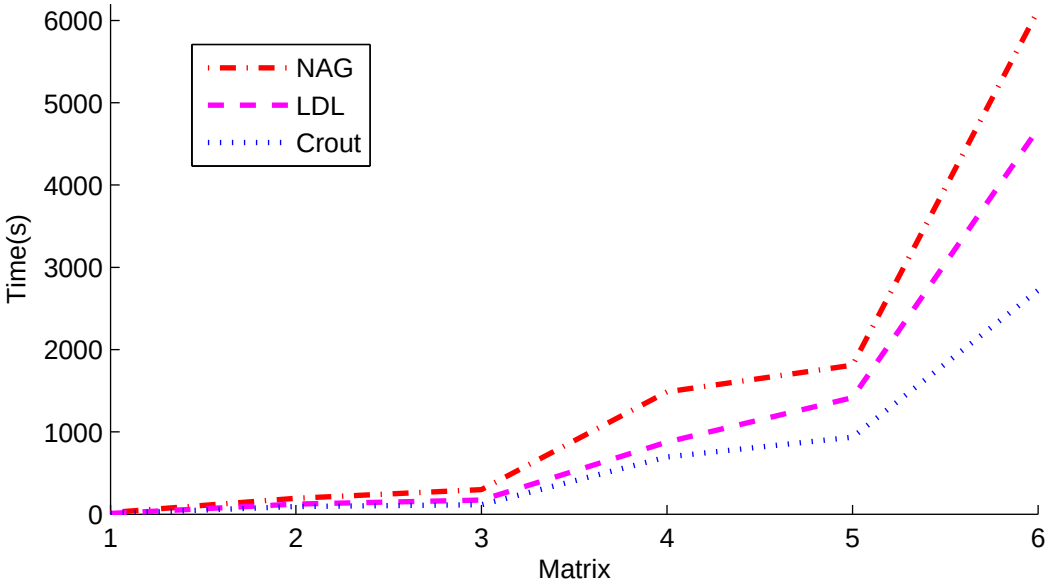

Figure 7.1: Comparison on the computational time for the matrices $mx1$,..,$mx6$

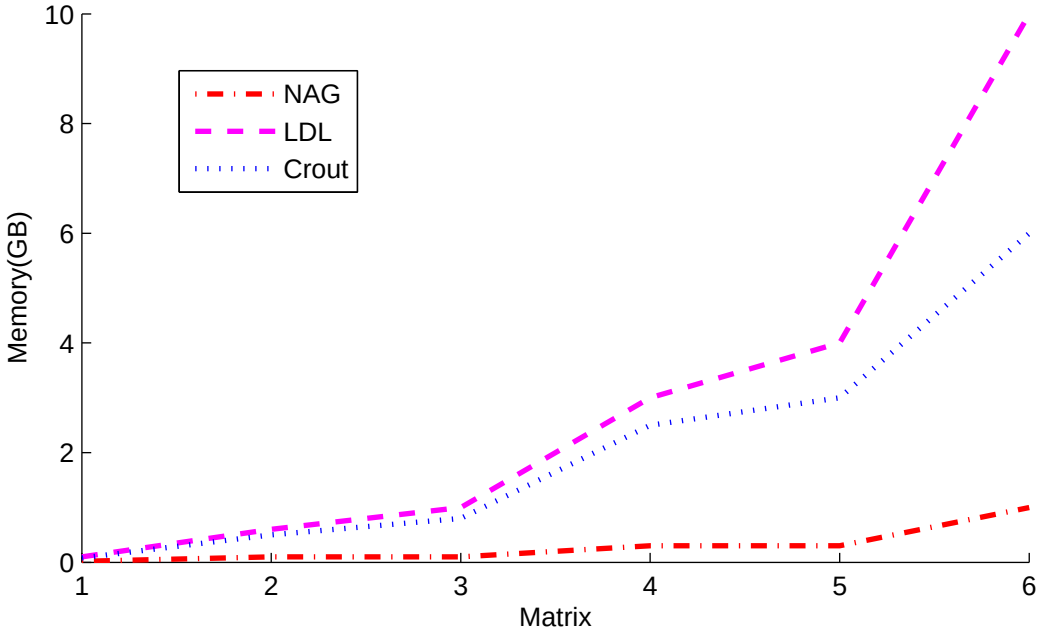

Figure 7.2: Comparison on the memory usage for the matrices $mx1$,..,$mx6$

methods, and that the Crout's memory usage increases more slowly than the LDL one. Moreover, as we see from Figure 7.2, the LDL package perform not very good result for the biggest matrix $mx6$, the used memory is in 10 times bigger than for the NAG library. This fact could be explained by the difference in the number of non-zero elements which this methods need to store. The NAG library use the preconditioner, which is an incomplete LU factorization where most of non-zero elements are dropped. While, the LDL package have to allocate memory for all non-zero elements in the complete LDL^T factorization, also several additional arrays of size N are used for computations. What concerns the Crout method, we saw from the numerical experiments that the actual number of fill-ins is much less than the number of non-zeroes computed by LDL symbolic function. That is the reason why the Crout method uses less memory than the LDL package.

Chapter 8

Conclusions and Future Work

In this chapter, the main results of the thesis are summarized. Furthermore, we specify how these results can be refined in order to form a basis for future research.

With our research we implemented a new approach for complete LU factorization of a coefficient matrix A, in the case when A is sparse, symmetric and positive definite matrix. This approach is based on the idea of the incomplete Crout factorization suggested by Y.Saad. Anyhow, this idea of the Crout complete factorization method was never taken into account seriously before. The code is written in Fortran.

We performed several experiments in order to compare three different methods of solving a large linear systems of equations: the Crout method, the LDL package and the iterative conjugate gradient method from the NAG library. The Crout method successfully improves the performance of the solving part of the MAGWEL software in comparison with the current using the Conjugate Gradient iterative method with SSOR preconditioner from the NAG library. In addition, not a pleasant part of the experiment, the amount of the memory needed for the Crout method is increased by factor 8 (average) in comparison with the current iterative method. However, the required amount of memory is double for the most modern computers used in the industry.

If we compare the LDL package and the Crout method, then the Crout version of the LU factorization is better again. Moreover, the Crout method shows the better results both in computational time and in memory usage. In spite of this, the LDL package also has improved the performance of the MAGWEL software but only in the meaning of time.

Our experiments show that the direct solver together with the Crout factorization can be efficiently and effectively integrated into the MAGWEL software.

The next step in the research area of the fast and efficient solver for the linear system $Ax = b$, could be the implementation of the original idea of Y.Saad. Namely, the incomplete Crout version of the LU factorization can be used as the preconditioner for any iterative method.

Also, the Dual Mesh algorithm for the Manhattan mesh was created. It shows much better performance that the previous method and reduced the computational time in seven times.

Bibliography

[1] P. Amestroy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.

[2] A.Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer, Berlin, 2nd edition, 2008.

[3] A. Bjrck. *Numerical methods for least squares problems*. SIAM, 1999.

[4] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate . *ACM Trans. Math. Software*, 35(3), 2009.

[5] T. A. Davis. Algorithm 849: A Concise Sparse Cholesky Factorization Package. *ACM Trans. Math. Software*, 31(4), 2005.

[6] T. Frankel. *The geometry of physics*. Cambridge University Press, Cambridge, 2nd edition, 1997.

[7] J. E Gentle. *Random number generation and Monte Carlo methods*. Springer, New York, 2nd edition, 2003.

[8] A. George and W.L. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, (31):1–19, 1989.

[9] G. H. Golub and C. F. Van Loan. *Matrix computations*. The Johns Hopkins University press, Baltimore, 3d edition, 1996.

[10] N.I.M Gould, J.A. Scott, and Y. Hu. A Numerical Evaluation of Sparse Direct Solvers for the Solution of Large Sparse Symmetric Linear Systems of Equations. *ACM Trans. Math. Software*, 33(2):300–335, 2007.

[11] M. W. Guidry. *Gauge field theory*. Wiley-VCH, New York, 1991.

[12] David A. Hodges, Horace G. Jackson, and Resve A. Saleh. *Analysis and design of digital integrated circuits*. McGraw-Hill Science, Boston, 3d edition, 2003.

[13] D. Kincaid and E.W. Cheney. *Numerical Analysis*. American Mathematical Society, Providence, 2nd edition, 2009.

[14] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Applic.*, 11(1):134–172, 1990.

[15] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*. Springer, Berlin, 2nd edition, 2007.

[16] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, 2nd edition, 2003.

[17] Y. Saad and N. Li. Crout version of the ILU factorization with pivoting for sparse symmetric matrices. 2002.

[18] Y. Saad, N. Li, and E. Chow. Crout version of ILU for general sparse matrices. 2002.

[19] W. H. A. Schilders, P. G. Ciarlet, J. L. Lions, and E. J. W. ter Maten. *Handbook of Numerical Analysis: Special volume: numerical methods in Electromagnetics*. Elsevier Science, 3d edition, 2005.

[20] W. Schoenmaker and P. Meuris. Electromagnetic Interconnects and Passives Modeling: Software Implementation Issues. *IEEE Trans. on Computer-Aided Design*, 21(5):534–543, 2002.

[21] W. Schoenmaker, P. Meuris, E. Janssens, K.-J. van der Meijs, and W.H.A. Schilders. Maxwell Equations on Unstructured Grids Using Finite-Integration Methods. *Sim. of semiconductor proc. and devices*, 12, 2007.