MASTER

Computer simulation with PLASIMO of a Technological Microwave Plasma for Manufacturing Optical Glass Fibres

van den Donker, M.J.

Award date:
2004

Link to publication
COMPUTER SIMULATION WITH PLASIMO
OF A TECHNOLOGICAL MICROWAVE PLASMA
FOR MANUFACTURING OPTICAL GLASS FIBRES

Michiel van den Donker

EPG 04-04
February 2004

Supervisors: dr. J.J.A.M. van der Mullen
dr. ir. A. Hartgers, dr. ir. M.J.N. van Stralen
Abstract

Microwave plasmas are often used in technological applications. One of the many industrial applications is that of optical glass fibre production. The core stage in manufacturing of optical glass fibres at Draka Comteq is the Plasma Chemical Vapour Deposition (PCVD) process. SiCl₄ and O₂ gases (and possible dope gases) are ionised in a substrate tube by electromagnetic (EM) waves. By choosing the geometry carefully, power can be coupled in very efficiently. A reactive plasma is formed that deposits (doped) glass (SiO₂) on the inside of this tube. To better understand this process and to be able to optimise it eventually, a computer simulation is made of the plasma. The Plasma Simulation Model (PLASIMO) is extended for this purpose. Three important modelling aspects are the transport of species inside the plasma, the interaction between EM waves and plasma and the chemical composition. These aspects depend on each other. PLASIMO uses a discretised grid and solves the continuity equations (ϕ-equations) in each grid cell as well as the fluxes between cells to find a solution. The EM wave propagation is calculated by using discretised Maxwell equations with the proper boundary conditions (similar to the so-called "Yee scheme").
Contents

1 Introduction 4
   1.1 Technology Assessment ........................................ 5
   1.2 PCVD Process ................................................ 5
   1.3 Microwave Plasma Simulations .............................. 6

2 Plasmas 8
   2.1 Energy Source ................................................ 8
   2.2 Plasma Characterisation .................................... 9
      2.2.1 Thermal Equilibrium .................................. 10
      2.2.2 Saha-Boltzmann Relations ............................. 11
   2.3 PCVD Characterisation ..................................... 12

3 Plasma Chemistry 13
   3.1 Required Data for PLASIMO .................................. 13
   3.2 Model Subset of PCVD Reactions .......................... 14
      3.2.1 Deposition ............................................... 15
   3.3 Ar Plasma ................................................... 16

4 Fluid model 17
   4.1 $\varphi$-equations ............................................ 17
   4.2 Grid Stencil .................................................. 19
      4.2.1 Boundary Conditions .................................. 20
   4.3 Final Remarks ................................................ 21

5 Electromagnetic Power Incoupling 22
   5.1 Description of the Configuration ........................... 22
   5.2 Electromagnetic Field ....................................... 24
      5.2.1 The Maxwell Equations ................................ 24
      5.2.2 Conductivity and Complex Permittivity ............. 25
      5.2.3 The Maxwell Equations in Cylindrical Coordinates 27
      5.2.4 Normalisation .......................................... 27
   5.3 Discretisation of the Maxwell Equations ............... 28
   5.4 Boundary Conditions ........................................ 29
      5.4.1 Internal Interface Boundary Conditions ............. 29
      5.4.2 Symmetry Axis .......................................... 31
      5.4.3 Boundary Conditions at Inlet and Outlet Sides .... 32
      5.4.4 Boundary Conditions at Resonator Wall, Slit and Chokes 32
   5.5 Matrix Structure ............................................ 35
<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6 Power Absorption</td>
<td>40</td>
</tr>
<tr>
<td>5.6.1 Power Check</td>
<td>41</td>
</tr>
<tr>
<td>5.7 Skin Depth</td>
<td>42</td>
</tr>
<tr>
<td>5.8 Grid Stretching</td>
<td>43</td>
</tr>
<tr>
<td>6 Simulations</td>
<td>44</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>44</td>
</tr>
<tr>
<td>6.2 EM Module Tests</td>
<td>45</td>
</tr>
<tr>
<td>6.2.1 Chokes</td>
<td>45</td>
</tr>
<tr>
<td>6.2.2 Conductivity</td>
<td>45</td>
</tr>
<tr>
<td>6.3 Skin Depth Comparison</td>
<td>47</td>
</tr>
<tr>
<td>6.4 PCVD Simulation</td>
<td>50</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>57</td>
</tr>
<tr>
<td>References</td>
<td>58</td>
</tr>
<tr>
<td>A Plasimo Cookery Book</td>
<td>63</td>
</tr>
<tr>
<td>A.1 Introduction</td>
<td>64</td>
</tr>
<tr>
<td>A.1.1 Modularity</td>
<td>64</td>
</tr>
<tr>
<td>A.2 Model syntax</td>
<td>64</td>
</tr>
<tr>
<td>A.2.1 Node description</td>
<td>64</td>
</tr>
<tr>
<td>A.2.2 LUT, boundary conditions and CRM</td>
<td>66</td>
</tr>
<tr>
<td>A.3 Running your model</td>
<td>67</td>
</tr>
<tr>
<td>A.3.1 Diverging models</td>
<td>67</td>
</tr>
<tr>
<td>A.4 Expandability</td>
<td>67</td>
</tr>
<tr>
<td>A.5 Tutorials</td>
<td>68</td>
</tr>
<tr>
<td>B Writing a Plugin</td>
<td>71</td>
</tr>
<tr>
<td>B.1 Introduction</td>
<td>72</td>
</tr>
<tr>
<td>B.2 Example 1: the Mach-1 Velocity Boundary Condition</td>
<td>72</td>
</tr>
<tr>
<td>B.3 Finishing touches</td>
<td>76</td>
</tr>
<tr>
<td>B.4 Summary</td>
<td>78</td>
</tr>
<tr>
<td>C EM code</td>
<td>80</td>
</tr>
<tr>
<td>C.1 Draka EM Skin Plugin</td>
<td>80</td>
</tr>
<tr>
<td>C.2 Draka EM Plugin</td>
<td>85</td>
</tr>
<tr>
<td>C.3 Surfatron EM Plugin</td>
<td>114</td>
</tr>
<tr>
<td>D Conductivity Code</td>
<td>118</td>
</tr>
<tr>
<td>E Cascaded Arc Mach 1 Boundary Condition Plugin</td>
<td>122</td>
</tr>
<tr>
<td>F Draka Inlet Plugin</td>
<td>124</td>
</tr>
<tr>
<td>G Multi Point Grid Stretch Code</td>
<td>127</td>
</tr>
<tr>
<td>H Empedocles</td>
<td>130</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In ancient Greece, Empedocles [1] thought that everything was built up of 4 elements, namely earth, water, air and fire. He was proven wrong, but there is an interesting parallel with the aggregation states of matter a species can be in. Under normal external conditions (atmospheric pressure, room temperature) earth is a solid, water a fluid, air a gas and fire a plasma. There are many definitions of a plasma, in our case the following is relevant. A plasma is a hot gas that contains so many free electrons and ions that its behaviour is dominated by these particles.

It is often said, that plasmas are everywhere. Indeed, over 99% of the visible material in our universe is in a plasma state. There is a large diversity in cosmic plasmas. The most common cosmic plasmas in nature range from dense stars to low density nebulae and anything in between. In technological plasmas there is also much variety in size, energy density, pressure and the way of energy coupling. However, these plasmas are often much richer in their chemical composition than the cosmic ones.

When plasmas are created in a controlled environment, they can be applied in many useful processes. Some examples of these are deposition or etching of certain materials, material cleaning, the generation of light in a tube lamp (TL), flat screen television or in extreme ultraviolet lithography, power generation in a nuclear fusion reactor and even propulsion in space.

There are several ways to couple power into a plasma in order to create and maintain it. One of these, is by electromagnetic (EM) fields. The basic principle is, that charged particles (mainly electrons as those are the most mobile particles) move as a result of the fields, collide with other particles and consequently heat up the plasma.

One of this type of plasma is the microwave plasma (see Chapter 2 for the other types). Rapidly alternating fields generate electromagnetic waves, which ionise the gas. Since the dimensions of the microwave induced plasma (MIP) are the same order as the wavelength of these EM waves, special care must be taken with the geometry to ensure a good power incoupling. When the right geometry is chosen, this incoupling can be very efficient.

It is the aim of this graduation report to describe a model of one of these MIPs.
1.1 Technology Assessment

This report describes a microwave plasma that is used in an industrial process. At Draka Comteq in Eindhoven, a plasma is used in the manufacturing process of optical glass fibres. These fibres are used in digital communication applications, like telephony and computer networks (for example the Internet).

One of the steps in manufacturing optical glass fibres is the creation of the so-called preform. Gases are pumped through a hollow quartz substrate tube. Electromagnetic energy creates and maintains a plasma inside this tube. In the plasma, certain chemical reactions take place, which eventually result in the deposition of material on the inside of the quartz tube. By modifying the flow of these gases, the composition of the deposited material can be changed.

1.2 PCVD Process

The resonator that feeds electromagnetic energy to maintain the plasma, moves slowly back and forth along the tube. Consequently, the plasma also moves back and forth along the tube, depositing a thin layer of material on each pass and smoothing the axial deposition profile. By changing the gas flow of certain dope elements on each pass, every layer has its own refractive index. This allows the creation of a predetermined radial profile for the refractive index of the tube. This process can be seen in Figure 1.2.

Subsequently, this hollow tube is collapsed into a massive preform by heating it. Subsequently this massive preform is heated up to the melting point and a fibre can be drawn from it, as can be seen in Figure 1.1. The radial profile of the refractive index of the fibre is a direct result of the profile of the preform. Thus by controlling the deposition process, certain properties of the fibre can be controlled. Undesired optical effects can be compensated (at the wavelengths of the used telecommunication lasers). During the drawing process, a coating is put on the outside of the fibre.

In this report, only the PCVD process is under investigation. For a more in-depth review of the other processes, see [3].

Figure 1.1: Schematic representation of several processes in the creation of an optical fibre. On the left-hand-side, the PCVD deposition is shown, in the middle the collapsing and on the right-hand-side the drawing of the fibre.
Figure 1.2: Representation of the setup used at Draka Comteq. A resonator, fed by a magnetron via waveguides, moves along the tube. The plasma inside the resonator deposits glass on the inside of the quartz substrate tube.

1.3 Microwave Plasma Simulations

Plasmas can be so complex, that a numerical model is needed to get a better understanding. For this purpose, the Plasma Simulation Model (PLASIMO) is developed at the plasma groups at the Eindhoven University of Technology (TU/e). The project's aim is to create a comprehensive set of tools allowing many different types of plasmas to be simulated. The program is constructed in a modular way, so it can be extended. Spatially discretised simulations can be done. Three important aspects of such a simulation are the (non-turbulent) flow, the plasma chemistry and the energy incoupling.

The description of the electromagnetic waves for the simulation of a PCVD plasma such as present at Draka is a key subject of this report. The discretisation scheme (similar to the Yee algorithm [21]) is extensively discussed. The EM part of the model is based on that by Janssen [2]. The main changes are the following:

- Complete rewrite of the model for the C++ object oriented version of PLASIMO. (This is done because the old C code did not work anymore with the newest versions of PLASIMO.)

- Optional use of higher order modes (see Chapter 5).

- The numerical grid of the electromagnetic calculation coincides with the PLASIMO grid (for the plasma simulation) better on the symmetry axis (see Chapter 5).

- By not assuming a fixed grid size there is the possibility to stretch the numerical grid. (In the radial direction this can reduce the number of points (and thus calculations) that are needed for a sufficiently detailed
description of the problem. In the axial direction, this could in the future be used to resize the grid easily without affecting the geometry.)

• Implementation and testing of chokes.

This report is organised as follows. In Chapter 2, plasma equilibria and plasma characteristics are discussed. The chemistry of the simulation is discussed in Chapter 3. Chapter 4 treats the fluid model and its implementation in PLASIMO. The electromagnetic power incoupling, which is the key part of this research, is treated extensively in chapter 5. Finally, the results of the models which are created, are given in Chapter 6 and briefly discussed in Chapter 7.
Chapter 2

Plasmas

In order to classify the plasma at Draka, it is needed to differentiate between the multitude of types of plasmas. To do so, the properties of plasmas have to be specified.

One important aspect of the plasma state is that material properties are determined by the presence of charged particles. This immediately implies that electromagnetic effects play an important role. Therefore, the principles of electromagnetic power incoupling are discussed at first. Moreover, the presence of charges is closely related to departure from equilibrium, which is discussed next. Finally, the PCVD plasma at Draka is classified.

2.1 Energy Source

A plasma is created and maintained by adding energy in some way. One can name many ways to heat a plasma. For example, a fire in a candle is maintained by several exothermal chemical processes, whereas the sun's plasma is fuelled by nuclear fusion. Electromagnetic (EM) radiation can also add energy to a plasma. The latter method is used in the majority of the technological plasmas.

A time-variant EM-field (such as in the plasma discussed in this report) generates a motion of charged particles. This motion does not dissipate power, but collisions of these charged particles with other particles do. The number of collisions is dependent on both the collision cross-sections and the path that the particles travel during one period of an oscillation. Electrons have a much smaller mass than ions, while the EM forces that act on them are of the same order. Hence, only a negligible amount of collisions result from the movement of heavy particles directly induced by the EM fields. Because of this, the electrons dominate the EM energy incoupling. Part of the energy supplied to the electrons then trickles down to the heavy particles via elastic and inelastic collisions.

There are many ways of supplying EM energy to a plasma. With increasing frequency there are (quasi-)direct current, capacitively and inductively, microwave and laser coupled plasmas respectively.

In the case of direct current coupling, a constant voltage difference is supplied over two opposing plates or electrodes. The charged particles move towards the plate which is oppositely charged to them.

In capacitively coupled plasmas, two opposing plates (or electrodes) can be
supplied with an alternating voltage. Again, the charged particles move towards the plate which is oppositely charged to them at that moment. A while later, the plates have changed polarity and the charged particles move in the opposite direction. Collisions between all the moving and neutral particles result in energy dissipation.

By using magnetic coils in the right configuration, the plasma acts as a secondary coil of a transformer. This electrode-less method is referred to as inductive heating. An advantage of an inductively coupled plasma (ICP) is that there is no contact between plasma and electrodes and thus no damage to electrodes. Therefore, in some cases, less maintenance is needed. Moreover, due to the absence of electrodes no electrode material can contaminate the plasma.

An often employed way to heat laboratory plasmas is by employing high frequency EM energy. A big advantage of this method is that the energy source can be remote because a negligible amount of power is dissipated in the waveguides that lead the waves to the plasma. Dielectrics like quartz are largely transparent for the EM energy at this frequency range and can be used to confine gases. If the wavelength is of the same order as the geometry, the latter can also be optimised to confine the EM energy, which results in a specific plasma shape. Because of the above reasons, heating by microwaves can be a very efficient way of remote power incoupling, almost all power ends up in the plasma. Therefore microwave plasmas are very important in technology.

In these high frequency EM fields, the electron motion can be quite complex and even out of phase with the original fields. Also because of the fields generated by the electrons themselves, the supplied fields can be partially shielded. This is called the skin effect as most power is dissipated in a small region (skin) near the outer boundaries of the plasma. This effect is discussed in more detail in Chapter 5.

There are three main types of these microwave induced plasmas (MIP), these are created by microwave torches, microwave launchers (surfatron) and inside microwave resonators [4]. The industrial setup at Draka uses a travelling microwave resonator.

The final category of EM heating is the laser produced plasmas. Electrons can move in the wake of a high energy laser pulse and dissipate energy that way. Many other effects [5] [6] can also happen because of the interaction between a high energy laser pulse and a plasma, but those will not be discussed in this report.

2.2 Plasma Characterisation

Densities and energy distributions of particles play an important role in the description of plasmas.

Sometimes the density of other species can be calculated from that of the electrons $n_e$ but in many cases densities like that of the heavy particle species and their ionisation stages and excited states thereof are also needed.

The number of independent parameters (densities and energy distributions) that is needed to describe a plasma, depends on the degree of departure from equilibrium. The lower the degree of equilibrium departure, the smaller the required number will be.

Every plasma has equilibrium and non-equilibrium aspects. To find the
boundary between partial equilibrium and equilibrium departure, the frequencies of equilibrium restoring processes need to be close to that of equilibrium disturbing processes ("leaks" of the balance). A balance is proper if the backwards reaction is the same as the forwards reaction. If one species in the balance is destroyed by another reaction than it is created by, the balance is improper. Specific examples of such balances are given in the next two subsections. This method of comparing those frequencies is called disturbed bilateral relations (DBR) and is more thoroughly treated in [7]. Especially the pressure in a plasma plays a large role in the competition of frequencies of the processes, and thus often determines whether some of these equilibria hold or not.

If all processes are in equilibrium, there is thermodynamic equilibrium (TD). This would mean, that no radiation escapes the plasma though. In industrial plasmas, this is usually not the case. If all material reactions are in equilibrium, there is local thermodynamic equilibrium (LTE).

2.2.1 Thermal Equilibrium

Material equilibria are related to collisions between particles in a plasma. When all material particles have the same energy distribution, there is thermal equilibrium.

One aspect of thermal equilibrium pertains to collisions between particles of the same type while another aspect results from collisions between particles of a different type.

When particles of the same type collide with each other often, a so-called Maxwellian energy distribution results [8] and a typical temperature is defined, which is the average of that distribution. For example, when enough electron to electron collisions occur, there is a typical electron temperature $T_e$ and the electron energy distribution function (EEDF) is Maxwellian. This is the first aspect of thermal equilibrium. Often, no distinction is made between different types of heavy particles, as their mass difference is usually much smaller than that between electrons and atoms. Therefore only one heavy particle temperature $T_h$ is defined, which can be different from the electron temperature. This two temperature model is often a valid way to describe a plasma and can also be used for the description of the plasma at Draka.

The other aspect of thermal equilibrium is when the average electron temperature $T_e$ is equal to the temperature of the heavy particles $T_h$. Because of the large difference in mass between heavy particles ($M$) and electrons $m_e/M \ll 1$, the energy transfer between these particles is not very efficient. So in order to satisfy this second condition of TE, a high number of collisions need to take place. The following balance is regarded

\[ \{e\} \leftrightarrow \{h\} \]

where the $e$ and $h$ between the braces are the energy distribution of electrons and heavy particles respectively. If this condition is satisfied and leaks on both sides are small, only one plasma temperature $T = T_e = T_h$ is needed to describe the plasma.

In many technological plasmas the following schematic can be made for energy transfer however

\[ EM \rightarrow \{e\} \rightarrow \{h\} \rightarrow \text{wall} \]
In such a case, this second aspect of thermal equilibrium is not satisfied as the balance is improper. There are big leaks on both sides, the EM energy heats the electrons (although this adds energy to the electrons this is called a leak of the balance) while the heavy particles get cooled by collisions with the wall. Both effects are much greater than the energy transfer from heavy particles to electrons. Thus two temperatures are needed to describe such a plasma and thermal equilibrium does not hold, the plasma is not in LTE. Generally only high pressure plasmas (many collisions) or plasmas which are very large (few losses to the wall) can be described by one temperature.

### 2.2.2 Saha-Boltzmann Relations

The Saha equilibrium describes the distribution of the ionisation stages and the Boltzmann equilibrium describes the distribution of excited states. Both aspects of the Saha-Boltzmann equilibrium are explained in some more detail.

The Saha equilibrium describes the balance between two ionisation stages. This is a reaction of the following type for an atom $A$

$$A^{(Z-1)+} + e \leftrightarrow A^{Z+} + e + e \quad (2.1)$$

Here the reaction from left to right is collisional ionisation by electrons and the reaction from right to left is three particle recombination. Of course it is clear that reactions which require 3 particles to be near each other only occur at very high electron densities. To see if this balance is valid, again the frequencies of the equilibrium disturbing effects need to be compared to those of the equilibrium restoring ones (DBR).

If Saha equilibrium is valid, it is possible to use the following equation to describe the population density of an ionised stage $n_{z+}$ of a plasma relative to the density of the ionisation stage below it (often the ground state)

$$\frac{n_{z+}}{n_z} = \frac{2g_{z+}}{g_z} \left( \frac{2\pi m_e kT}{\hbar^2} \right)^{3/2} \exp \left( -\frac{E_i}{kT} \right) \quad (2.2)$$

where $g_{z+}$ and $g_z$ are the degeneracies of the ionisation stage and of the stage below it respectively. $m_e$ is the rest mass of an electron, $k$ is the Boltzmann constant and $\hbar$ is the Planck constant.

The Boltzmann equilibrium treats the excitation and de-excitation by electrons

$$A^{Z+,m*} + e \leftrightarrow A^{Z+,n*} + e \quad (2.3)$$

Where $A^{Z+,m*}$ is the $m$'th excited state of the ion $A^{Z+}$. With the following equation the density distribution over the excited states $n_{z*}$ can be calculated

$$\frac{n_{z*}}{n_z} = \frac{g_{z*}}{g_z} \exp \left( -\frac{\Delta E}{kT} \right) \quad (2.4)$$

where $g_{z*}$ is the degeneracy of the excited state and $\Delta E$ is the energy difference between the two excited states.

We deal with a non-LTE plasma for which the Boltzmann and Saha balance are out of equilibrium. Thus, equation 2.2 and 2.4 can not be used. To calculate the atomic state distribution function (ASDF) and ionisation degree, a so-called collisional radiative model (CRM) [10] can be used. A CRM takes into account
all possible collisions between electrons and heavy particles. This requires a lot of input data (collision crosssections) which is often hard to acquire. In PLASIMO both LTE and non-LTE plasmas can be simulated. A CRM is one of the built-in tools of Plasimo.

The Saha equations are not sufficient to describe the absolute densities of species and only give the relative densities. If the total density of matter and the total charge are also known (usually plasmas are globally charge neutral), absolute densities can be calculated.

### 2.3 PCVD Characterisation

In an industrial situation, the electron temperature and density can not be changed directly. The most important parameters that influence the nature of a plasma are pressure, size, chemical composition and energy density. The electron and heavy particles temperature and density are dependent on these control parameters. There are simple models [11] [12] that give reasonable estimates of the plasma properties, although a full simulation is needed to see the exact function of the control parameters.

Since the pressure in the plasma at Draka is reasonably low \((10^3 \text{Pa})\), it is expected that the equality \(T_e = T_i\) does not hold. Therefore a two temperature model has to be used. This can be illustrated by comparing our plasma to an ICP. The ICP has about the same energy density, but a much higher pressure. But in an ICP the difference between electron and heavy particle temperature is typically \(2000\text{K}\) already. It is expected that these temperatures are even further apart for our plasma.

Moreover, due to the relatively low pressure it is expected that there is also a relatively large diffusion coefficient. The strong outward fluxes of large particles disturb the Saha balance. Therefore the ionisation degree does not satisfy the Saha equation and a CRM has to be used. Of course this means a lot of data (collision crosssections) is required about the species which are present.

The most important overall reaction in the plasma is \(\text{SiCl}_4 + \text{O}_2 \rightarrow \text{SiO}_2 + \text{Cl}_2\) but this most likely occurs through many intermediate steps. To keep the simulation of the chemistry reasonably simple, it is done with only argon and its ions instead. It is important to understand the flow properties of our plasma. So the simulation needs to be done in multiple dimensions. This is done by discretising the plasma volume. To simulate this plasma is a very complex task. There are several problems converging in this simulation. The EM structure has to be carefully described, the fluid part is important and the chemical reactions between species. All these components are strongly coupled with each other so need to be regarded at the same time. The next chapters will discuss these topics extensively.
Chapter 3

Plasma Chemistry

The plasma composition is one of the crucial elements of the simulation. The chemical composition has a great influence on all the other parts of the plasma simulation. For instance the ionisation and dissociation degree will largely determine the conductivity which in turn has a big impact on the EM incoupling. The plasma composition also has influence on the flow. The flow and EM incoupling are discussed in the next two chapters, however. As seen in the previous chapter, a plasma composition in LTE is much easier to determine than a plasma composition in non-LTE. However, it is also shown that LTE does not hold in our case. In non-LTE, a complete collisional radiative model (CRM) is needed to find the (relative) densities of the species. A large departure from LTE will significantly increase the calculation time (especially when many different species are involved) as well as require a lot more data about the collisions and the species themselves. Often reliable data is hard to find in literature or not available at all. A short overview of data that is needed for a non-LTE simulation is given first. Then, a subset of the chemical reactions that occur in the plasma at Draka is described. However, the eventual simulations are done with argon, this is treated last.

3.1 Required Data for PLASIMO

The required chemistry data can be split in 3 separate parts: Data of the particles themselves, transport properties and data of the chemical reactions between particles (and data needed for the CRM). For each particle that plays a role in the plasma, the following data is important:

- Charge number
- Hard sphere radius
- Excited state statistical weight
- Excited state energy

Aside from the properties of the species, it is also important to know which chemical reactions occur in the plasma, and what the rate of these reactions is.
These rates are often given in a standardised form, the so-called Arrhenius rates

\[ f = CT_q^q \exp \left( -\frac{E_{th}}{kT_e} \right) \]  

(3.1)

where often \( q = 1/2 \) is taken. The value for \( C \) should be deduced from experimental or literature data. The exponential factor \( E_{th} \) is found from the species types (the energy of the resultant species minus the energy of the reactants). When doing a simulation, it can be a good thing to reduce the number of reactions that are simulated and only take dominant reactions into account. Note that the Arrhenius equation does not always give an optimal fit for the temperature dependence over a large (temperature) range of a reaction. Often, the rates of reaction paths where a species goes through several excited states before the ionisation are much higher than those of a direct ionisation. Therefore, it is very important to include metastable excited states in the species types.

When collision cross-sections, reactions and species properties are known, the collisional radiative model (CRM) in PLASIMO can solve the composition (or in the case of LTE, the Saha and Boltzmann relations can be used instead). For a known composition, mixture properties can then be calculated that affect the flow and electromagnetic coupling, for example:

- Thermal conductivity of the heavy particles
- Viscosity of the heavy particles
- Electrical conductivity

3.2 Model Subset of PCVD Reactions

In the plasma reactor at Draka, several chemical reactions take place. During this study, preparation work is done to add a non-LTE chemistry to the model of this reactor.

The overall bulk process (not regarding the trace gases for the moment) can be summarised as

\[ \text{SiCl}_4(g) + O_2(g) \rightarrow \text{SiO}_2(s) + 2\text{Cl}_2(g) \]  

(3.2)

where the aggregation phase of these species is added in brackets, \((g)\) for gas and \((s)\) for solid. The real reaction path is in general much more complicated. The \( \text{SiO}_2 \) formed in the plasma is not usable since this will give glass dust, also referred to as “soot”. Thus, \( \text{SiO}_2 \) has to be formed at the wall to give a growth of the tube.

First of all, reactions are regarded which are likely to form \( \text{SiO}_2 \) at the wall instead of in the plasma. In the industrial setup, the walls are saturated with oxygen. If \( \text{SiO} \) or one of its ions is formed in the plasma, this will react at the wall and form \( \text{SiO}_2 \) there. Thus reactions that form \( \text{SiO} \) in the plasma may determine the deposition rate.

The amount of possible reactions that can occur in a plasma to form \( \text{SiO} \) or its ions is tremendous. Therefore, it is useful to develop a system to find the dominant reaction path and thereby reduce the number of reactions. It is important that no crucial reactions are removed.
In [17] the following possible reactions are described

\[
\begin{align*}
\text{SiCl}_4 + \text{O} & \rightarrow \text{SiO} + 2\text{Cl}_2 \\
\text{Si} + \text{O} & \rightarrow \text{SiO} \\
\text{SiCl}_2 + \text{O} & \rightarrow \text{SiO} + \text{Cl}_2
\end{align*}
\]

as well as the following etching reaction

\[
\text{SiO} + e \rightarrow \text{Si} + \text{O} + e
\]

Another method that can be used to reduce the number of reactions is by taking the (electron and heavy particle) temperature(s) of the plasma as a criterion. The plasma is hot enough to almost immediately break the very large molecules \(\text{SiCl}_4, \text{SiCl}_3\), and \(\text{SiCl}_2\) into smaller particles. Measurements [18] also show that mainly single- and diatomic molecules and their ions exist in the plasma, which backs up this assumption.

Note that it can be dangerous to assume that a particle is not important if its concentration is difficult to measure. It could be extremely important if a reaction which destroys this particle has a very high reaction rate, so when the particle is formed it immediately reacts. Because of that, the effective reaction rate of another reaction might have to be increased significantly to compensate. So even though such a particle is never measured it may act as an important intermediate species. Such species can be called locally chemical (LC) determined. This means transport of such particles is not important as they immediately react when formed.

If the large molecules are discarded, only the second one of the deposition reactions (3.4) can occur at the wall, because the other particles are hardly present. Thus, all reactions with the three big molecules \(\text{SiCl}_2, \text{SiCl}_3\) and \(\text{SiCl}_4\) and their ions are discarded under this assumption. The species that are left over are \(\text{O}, \text{Cl}, \text{Si}, \text{O}_2, \text{Cl}_2, \text{SiCl}, \text{SiO}\) and their ions and metastable states.

To simulate this plasma, Arrhenius rates are needed for all reactions that can occur between these species. Also, all transport coefficients need to be calculated. One of the important type of reaction balances is thus

\[
\text{Si} + \text{O} + e \leftrightarrow \text{SiO} + e
\]

Where instead of \(\text{Si}, \text{O}\) and \(\text{SiO}\), their ions could be substituted for equivalent reactions (as long as the total charge is conserved). Arrhenius rates from the ionisation and the recombination of some of these species are known from other simulations. But a large part of this still complex chemistry is hard to find, especially data about the transport properties and sensible reaction rates for the temperature ranges in the plasma. In the future, models [13] may be used to calculate data for the transport properties from the spatial geometry of a molecule.

Since the main focus of this research is the electromagnetic module, a much simpler chemistry of argon is used instead. Later, this can be replaced by simulations for an oxygen plasma and eventually a subset of the full chemistry, once all data is available.

### 3.2.1 Deposition

To model the deposition, there are several important aspects that need to be looked into. First of all, which reactions are important and which species are
defining for the speed of the reaction. For example, it is assumed that atoms or ions containing SiO react with oxygen based atoms or ions at the wall to form SiO$_2$ (or charged variants thereof). In the industrial setup, there is a surplus of oxygen at the wall so the SiO flux will likely be limiting for the rate of formation of SiO$_2$. Even this simple relation does not reveal yet if it is the SiO$^+$ or SiO$^-$ ions or the neutral SiO which is important in the reaction. Also, since these can all be formed in a multitude of reactions, it is very hard to predict a dominant reaction path for the deposition reaction. An additional difficulty can be the impact of wall reactions on the chemistry inside the plasma. To model this, one needs a complete interaction between these 2 parts (wall chemistry and plasma chemistry). Other aspects can include sputtering (which becomes important at very high ion velocities to the wall, and thus high energy tails of a plasma with a high heavy particle temperature) or etching (think about fluorine, which can etch SiO$_2$). In the model, a fixed sticking coefficient is assumed which denotes the chance that a particle reacts with the wall when it impinges on it. With this, it is in principle possible to calculate a deposition profile. This is not implemented yet in the model, as it becomes important after the real chemistry can be calculated.

3.3 Ar Plasma

As stated before, this graduation report is mainly devoted to the modelling of the EM part. In order to validate the results, the chemical composition is significantly simplified. Instead of the chemistry described above, an argon plasma is simulated. Argon data is widely available since this type of plasma is often used in physical experiments and the chemistry is extremely simple. The argon gas forms only 3 component species when a plasma is created with it, namely Ar, Ar$^+$ and e$^-$. Most excited levels are treated as local chemistry levels. They serve to facilitate the stepwise ionisation and recombination. For the argon ion there is an excited state of 15.759 eV with a degeneracy of 4 and a state of 15.936 eV with a degeneracy of 2. These data are taken from other simulations.

For the Arrhenius rates in an argon plasma the reaction rate for the reaction

$$\text{Ar} + e^- \leftrightarrow \text{Ar}^+ + 2e^- \quad (3.8)$$

is given by

$$f = 7.34 \cdot 10^{-15} \text{s}^{-1} T_e^{0.5} \exp \left( - \frac{E_{\text{th}}}{kT_e} \right) \quad (3.9)$$

The transport properties for viscosity and thermal conductivity are calculated with a built-in function using the Chapman-Ensikog-Burnett relations [14]. The electrical conductivity is calculated by the EM-module described in Chapter 5.
Chapter 4

Fluid model

This chapter is devoted to the fluid model, which describes the transport phenomena. To simulate a non-uniform plasma numerically, the plasma region is divided by a grid into many small volumes, the so-called cells. Physical quantities like densities and temperatures on a grid cell are averaged and defined at so called nodal points. These points are located in the middle of a grid cell (in case of a uniform grid). Fluxes however, are defined at the boundaries between two cells. For each grid cell several physical conservation laws are solved using the physical dimensions of this cell, fluxes at the boundaries and the plasma properties defined at the nodal point.

Besides transport by fluxes at the boundaries between cells, there can be transport of energy between cells that are not adjacent to each other. This is for example the case with radiative transport [15]. However, in the simulations in this report, such transport is not regarded.

The macroscopic behaviour of the plasma can be described by a fluid mechanical model [16] for which these assumptions are made:

- The flow is laminar.
- The flow is in a steady-state.

4.1 \( \phi \)-equations

The conservation equations for mass, momentum and energy can all be written in the same form, the so called canonical transport or \( \phi \) equations. Derivations of these can be found in [2] [15] [11]. Since only one form of these partial derivative equations (PDEs) is used, it is possible to focus extensively on it and develop an efficient way to solve them. In these equations \( \phi \) can be substituted by the conserved variable, for instance a particle's velocity, temperature or density. The following form of \( \phi \)-equation is used in PLASIMO [10]

\[
\frac{\partial (ce\phi)}{\partial t} + \nabla \cdot (U\phi) - \nabla \cdot (\Gamma \nabla \phi) = S_c + S_\phi \phi
\]

(4.1)

The first term on the left-hand-side is the time-dependent term, the second term the convection term and the third term the diffusion term. The first term on the
right hand side is the constant part of the source term, the second term on the right-hand side is the linear part of the source term. \( c, \Gamma \) and \( \vec{U} \), the constant source term \( S_c \) and linear source term \( S_p \) are replaced with the appropriate expressions to get the specific balances.

Since one of the assumptions in the flow model is the steady-state approximation, the first term in the \( \phi \) equation is always equal to zero in this study, but for completeness it is included in the equations. The first example of the \( \phi \) equation is the specific mass balance for species \( i \).

\[
\frac{\partial n_i}{\partial t} + \vec{\nabla} \cdot (n_i \vec{U}) - \vec{\nabla} \cdot (D_i \vec{U}^2) = S_i = P - n_i \text{Dest}
\]  

(4.2)

where \( D_i \) is the diffusion coefficient of species \( i \), \( n_i \) its density and \( \vec{U} \) is the plasma bulk velocity. The source term \( S_i \) is the net production of species \( i \) by chemical and collisional-radiative processes. \( P \) is the production of the species by chemical reactions (which generally does not depend on the density of species \( i \), but to that of another species) while \( \text{Dest} \) is the destruction term.

Note that aside from the electron, 4.2 also holds for all types of heavy particles. The larger the equilibrium departure (see Chapter 2), the more species have to be described and the more of these equations are necessary. Since these equations are solved on each iteration for each species on each grid cell, a simulation with a complex chemistry and a large grid can take a long time to converge.

The next application of the \( \phi \) equation is the continuity equation for the bulk properties of the plasma, which is especially needed when a flow is present, like in our model. It is obtained by summing the specific mass balances over all species \( i \) and defining the bulk density \( \rho = \sum_i n_i \) and the bulk velocity \( \vec{u} = \sum_i \frac{\rho}{\rho} \vec{u}_i \).

\[
\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \vec{U}) = 0
\]  

(4.3)

Another \( \phi \) equation is the specific momentum balance for species \( i \).

\[
\frac{\partial (\rho \vec{u}_i)}{\partial t} + \vec{\nabla} \cdot (\rho \vec{u}_i \vec{u}_i) = -\vec{\nabla} \cdot \vec{P}_i + \vec{u}_i m_i \dot{S}_i + \vec{F}_i + \vec{F}_i
\]  

(4.4)

where \( \rho_i = n_i m_i \) is the mass density, \( m_i \) the mass and \( \vec{u}_i \) the velocity of species \( i \). \( \vec{P} \) is the pressure tensor [15], \( \vec{F}_i \) the body force acting on species \( i \) and \( \vec{F}_i \) is the force caused by collisions with other species. For the bulk momentum equation we get

\[
\frac{\partial \rho \vec{u}}{\partial t} + \vec{\nabla} \cdot (\rho \vec{u} \vec{u}) = -\nabla \cdot \vec{P} + \sum_i \rho_i \vec{F}_i
\]  

(4.5)

The equation of state is given by

\[
p = \sum_o p_o
\]  

(4.6)

with \( p_o = n_o k_b T_o \). Another equation we need to define the model is the energy balance for particles of species \( i \)

\[
\frac{\partial (n_i \epsilon_i)}{\partial t} + \vec{\nabla} \cdot (n_i \epsilon_i \vec{u}_i) + \vec{P}_i \vec{u}_i + \vec{F}_i : \vec{u}_i + \vec{q}_i = Q_i
\]  

(4.7)

18
where $e_i$ is the internal energy per particle of species $i$, $Q_i$ the energy gain or loss through either elastic or inelastic reactions with other particles and chemical reactions between particles and the energy gain by electromagnetic power in-coupling. It is assumed that the heat flux satisfies Fourier's law $q_i = \lambda_i \nabla T_i$, where $\lambda_i$ is the thermal conductivity of the species. Since the two temperature plasma balance holds, only two of these equations are needed. One for the electrons and one for the heavy particles (and not for every separate type of heavy particle).

$Q_i$ is only affected by EM fields in case the species $i$ are the electrons. The part of this term that is a result of the electromagnetic in-coupling is calculated by the EM module described in Chapter 5.

A modified version [10] of the Semi-Implicit Method for Pressure-Linked Equations Revised (SIMPLER) [16] is used to create a self-consistent solution to all these $\phi$-equations.

### 4.2 Grid Stencil

Since the simulations of a non-LTE plasma can be very complex and are thus unstable by nature, methods are needed to reduce the complexity. One possibility is to use only two dimensions by exploiting a carefully chosen symmetry. In our case this is a cylindrical symmetry. The plasma is divided into several cells which are also called control volumes. The number of cells in each direction (and their spacing) can be specified in the simulation. The boundaries of this grid are designated by the "points of the compass". The top, which corresponds to the outer boundary of the plasma, is called the north boundary. Similarly inlet is west, outlet is east and the symmetry axis is south. The east-west direction is called the $x_1$ direction and the north-south direction is $x_2$.

Nodal points are defined in the middle of a grid cell, most quantities like densities are given at these points. Fluxes are defined on the north, east, west and south faces of a cell. The grids they are defined on, are called east-west and north-south grids. Thus the points of such grids lie in between two adjacent cells, on the faces. Not all grid cells have to be of the same size, in case the grid is "stretched" the north-south and east-west grid points do not have to lie halfway between two nodal points. Stretching the grid in places where steep gradients can be expected can reduce the number of points needed to simulate a plasma accurately enough.

![Grid Stencil Diagram](image)

Figure 4.1: Several internal grid cells and their correspondence to the plasma volume. Nodal points are noted with a cross, east-west with a square and north-south points with a circle.
The \( \phi \) equations are discretised by integration on these cells. In case of derivatives a certain stencil is used to create a numerical derivative. For example the numerical derivative in the \( x_1 \) direction of the electron density flux \( n_e u_1 \) is given by

\[
\frac{\partial (n_e u_1)}{\partial x_1} \approx \frac{(n_e u_1)^{\text{east}} - (n_e u_1)^{\text{west}}}{\Delta x_1}
\]

where \( \Delta x_1 \) is the width of the cell in the east-west direction and \( u_1 \) is the speed in the \( x_1 \) direction.

When more grid cells are regarded, values of \( n_e u_1 \) can be stored in a vector and the numerical derivative can be written in a matrix form. In Chapter 5 examples are given on how derivatives can be written in matrix form.

### 4.2.1 Boundary Conditions

One of the most important aspects of a fluid simulation is choosing the right boundary conditions. In our simulation the same conditions are chosen as by Janssen [2]. Two basic types of boundary condition can be distinguished, namely Neumann and Dirichlet. The Dirichlet condition fixes the value of the variable at a certain value, while the Neumann condition fixes the first derivative of the variable at a certain value. This value must be physically sensible for the variable in question (\( T_e, T_h, \) densities, pressure, velocities, etc.).

The most simple variants are the homogeneous Dirichlet conditions which fixes the value of the variable at zero and the homogeneous Neumann condition which fixes the value of the first derivative of the variable at zero.

An example dirichlet condition, is the no-slip boundary condition for the velocity on the north wall in a cylindrical symmetric simulation. This simply means that at the wall \( v_r = 0 \) (because no fluid flows beyond the wall) and \( v_z = 0 \) (the real no-slip condition). Since the values are zero, these conditions are homogeneous dirichlet conditions.

In the middle of the tube inlet \( v_r = 0 \) (because if \( v_r \neq 0 \), a discontinuity in the velocity would result around the symmetry axis). This is also a homogeneous dirichlet condition.

An example of a (homogeneous) Neumann condition on the inlet wall is \( \partial_z v_z = 0 \). Together with the above three conditions this results in a laminar profile in a tube.

The Neumann and Dirichlet boundary conditions can be used as a primitive for more complicated boundary conditions, for example a radial profile of the boundary condition of a variable can be created by modifying the Dirichlet condition.

An example of a modified boundary condition is the velocity boundary condition that is created for the inlet of the tube in the simulation of the plasma at Draka. Here the no-slip condition is used at the walls, while the profile has a maximum speed at the centre. The shape of this profile can be chosen to be parabolic or of higher order. The inlet velocity is dependent on the gas flow input parameter which can be given by the user. In this implementation, the volume flow (in scc/s) and the composition of the gas need to be supplied. With this, the velocity in the \( x_1 \) direction is calculated and used for the boundary condition. The code of this boundary condition can be seen in Appendix F.

Another example of a modified boundary condition that is used to model plasma expansion is the Mach 1 condition. Note that this condition is not used.
in the simulation of the plasma at Draka, but for the poisseulle flow which is described in ?? . A Dirichlet profile is created for the velocity in the axial direction on the outlet of a tube corresponding to the speed of sound, which changes according to pressure $p$ and mass density $\rho$ at the outlet. When the expansion in the tube is adiabatic and the ionisation degree is constant, the maximum speed is given by

$$u_{\text{out}}^{\text{max}} = \sqrt{\frac{5p}{3\rho}}$$

(4.9)

while the velocity profile along a wall with radius $r = R$ can be given by

$$u_{z,\text{out}} = u_{\text{out}}^{\text{max}} \left[ 1 - \left( \frac{r}{R} \right)^2 \right]$$

(4.10)

where $u_{\text{out}}^{\text{max}}(R) = 0$. Thus a radial profile for the velocity in the axial ($x_1$) direction is created that goes from the maximum value at the centre to zero at the walls (because of the friction there). The code of this boundary condition is given in Appendix E.

The custom boundary conditions were created as so called plug-ins. In Appendix B more is said on how to write such a plugin. In Chapter ?? the exact boundary conditions of the respective simulations are given.

The power of the modularity that PLASIMO offers becomes apparent by the flexibility with which these boundary conditions can be applied.

### 4.3 Final Remarks

Because of the many non-linear interdependencies of these models, an iterative solution method is needed. Even when a model is fully defined, by the conservation laws inside its cells as well as the communication between cells, it can often still not be simulated correctly. To use a model, starting conditions have to be supplied which give a reasonable physical estimate of the problem. One very important aspect which is often overlooked are the under-relaxation factors (URFs), which are used to reduce sensitivity of certain variables so that convergence becomes smoother. If either the starting conditions or the URFs are not optimally chosen the simulation can take a long time to converge, arrive at the wrong solution or even diverge altogether.
Chapter 5

Electromagnetic Power Incoupling

Besides the fluid model and chemistry module, a model for the power incoupling is needed to get a complete plasma description. In our case, this is done by means of electromagnetic (EM) waves. Modelling this is a key objective of this research.

This chapter starts in Section 5.1 by giving the configuration of the industrial setup as well as its equivalent in the model. Subsequently, in section 5.2 the Maxwell equations are treated which describe the EM waves. Also, the influence of the plasma properties on the EM field via the complex permittivity is explained. After this general treatment, two methods to model EM waves are investigated.

The first one is an elaborate model, similar to the Yee method [21], which is thoroughly treated. Due to the symmetry of the configuration under investigation, the number of Maxwell equations which are necessary to solve can be reduced.

The discretisation of these equations is then explained in Section 5.3. The boundary conditions of the model, which can be very complex, are also discussed. Then, the way this is solved in a matrix form is given in Section 5.5. After this the power flow is discussed in Section 5.6, which is the link from the EM calculation to the plasma.

Finally in Section 5.7, the second model, skin depth is described. Since this model is much simpler and thus takes much less calculation time than the complete description, it is used to initialise the more complete Yee model by generating an approximate solution to the plasma flow. A variant of this skin depth model is used in the preliminary simulation of a surfatron plasma. The chapter ends with a short mention on the stretching of the grid.

5.1 Description of the Configuration

A schematic representation of the industrial setup under investigation is given in Figure 5.1.

A microwave resonator moves axially along a quartz tube. Electromagnetic energy generated by a magnetron is guided to the resonator by means of a microwave waveguide circuit. EM waves pass through a (rotationally extended)
Figure 5.1: Representation of the resonator in the setup used at Draka Comteq. A resonator, fed by a magnetron via waveguides, moves along the tube. The plasma inside the resonator deposits glass on the inside of the quartz substrate tube.

Figure 5.2: The 2-D grid structure used for the simulation of the EM-field. The problem is rotationally symmetric. The slit is a circular hole in the resonator through which the EM waves enter the region within. These waves enter via the air and glass into the plasma and heat it. At the same time, gas is flowing in from the inlet to the outlet. Note that the EM grid structure is larger than the grid used for the simulation of the plasma flow and chemistry. In the northeastern most grid cell, the structure of the grids and the position of the field components is shown.
slit on the inner side of the resonator and enter the plasma through the quartz substrate tube. Here, EM waves interact with the species that flow through the substrate tube. Consequently, a plasma is created and heated. Because the EM waves are mostly confined by so-called chokes, the plasma is mostly situated inside the resonator. The plasma deposits (doped) glass on the inside of the tube and the remaining species leave the tube through the outlet. A furnace is used to keep the walls of the substrate tube at a temperature of 1200°C.

The frequency of the EM waves is 2.46 GHz, which corresponds to a wavelength of 12.2 cm. This is in the same length scale as the dimensions of the setup. Therefore, an accurate description of the waves is necessary.

Since the motion of the resonator along the tube (0.3 m/s) is negligible compared to the gas flow speed (> 10 m/s), a fixed resonator can be assumed in the simulations. Thus, a steady-state situation is simulated.

An example of a grid on which this problem can be simulated is shown in Figure 5.2. This is a 2-dimensional grid that encompasses both the plasma region and the other regions inside the resonator. The grid is cylindrically symmetric so each of the squares represents a ring (except the ones in the bottom, which are small cylinders). The flow and plasma calculations encompass only the plasma region (the southernmost part of the computational region).

### 5.2 Electromagnetic Field

#### 5.2.1 The Maxwell Equations

In microwave plasmas, both the electric and the magnetic fields play a key role. The Maxwell equations describe time-dependent electromagnetic fields. Especially the last two Maxwell equations are important.

\[ \nabla \times \vec{E} + \partial_t \vec{B} = 0 \]  \hspace{1cm} (5.1)

\[ -\nabla \times \vec{H} + \vec{J} + \partial_t \vec{D} = 0 \]  \hspace{1cm} (5.2)

where \( \vec{E} \) is the electric field, \( \vec{B} \) is the magnetic flux density, \( \vec{H} \) is the magnetic field, \( \vec{J} \) is the current density and \( \vec{D} \) is the electric flux density, respectively.

These two vector equations are the equivalent of six scalar equations (two equations for each of the three spatial directions) and contain 15 unknowns. Therefore, 9 extra equations are needed to solve the system. These relations, which contain macroscopic media properties, are called the constitutive relations. In this study, the following three constitutive relations are considered. First, the magnetic flux is related to the magnetic field by means of the magnetic susceptibility in vacuum (\( \mu_0 = 4\pi \cdot 10^{-7} \text{H/m} \))

\[ \vec{B} = \mu_0 \mu_r \vec{H} \]  \hspace{1cm} (5.3)

The relative magnetic susceptibility \( \mu_r = 1 \) because there are no magnetically active materials. Secondly, the dielectric displacement is related to the electric field by means of the electric permittivity of vacuum \( \varepsilon_0 \) and the relative electrical permittivity \( \varepsilon_r \) (for a linear medium)

\[ \vec{D} = \varepsilon_0 \varepsilon_r \vec{E} \]  \hspace{1cm} (5.4)
Finally, Ohm's law gives the current density in terms of the electric field and the conductivity of the medium \( \sigma \),

\[
\vec{J} = \sigma \vec{E}
\]  

(5.5)

These three constitutive relations hold for isotropic, non-magnetic, linear, time-invariant, instantaneously reacting and locally reacting media. Note that the conductivity and the relative electrical permittivity can be functions of position since it is assumed that the medium is inhomogeneous in our simulations. In the plasma region, these parameters are dependent on the plasma conditions.

If the source is quasi-stationary harmonic, the field quantities depend harmonically on time with a real angular frequency \( \omega = 2\pi f \). The fields can now be split in a time-dependent part \( e^{j\omega t} \) and a constant pre-exponential term. For the electric and magnetic fields, this time-independent vectorial term is designated with \( \vec{E} \) and \( \vec{H} \), respectively. The Maxwell equations in the frequency domain are found by replacing \( \partial_t \) with \( j\omega \) and dividing by the exponential time factor. With these substitutions, expressions are found for \( \vec{E} \) and \( \vec{H} \), which are vectors of complex numbers.

Substituting Equations (5.5) and (5.4) into Equation (5.2), and rewriting the resulting equation for the frequency domain, yield

\[
\nabla \times \vec{H} - \sigma \vec{E} - j\omega \varepsilon_0 \varepsilon_r \vec{E} = 0
\]  

(5.6)

Collecting terms of \( \vec{E} \) results in

\[
\nabla \times \vec{H} - j\omega \varepsilon_0 \varepsilon_r \vec{E} = 0
\]  

(5.7)

where the relative complex permittivity \( \varepsilon_r \) is introduced as

\[
\varepsilon_r = \varepsilon_r + \frac{\sigma}{j\omega \varepsilon_0}
\]  

(5.8)

This leads to the following simplified form of the Maxwell equations in the frequency domain

\[
\nabla \times \vec{H} - j\omega \varepsilon_0 \varepsilon_r \vec{E} = 0
\]  

(5.9)

\[
\nabla \times \vec{E} + j\omega \mu_0 \vec{H} = 0
\]  

(5.10)

Remember that \( \varepsilon_r, \vec{H} \) and \( \vec{E} \) are complex quantities.

### 5.2.2 Conductivity and Complex Permittivity

This subsection describes how the conductivity \( \sigma \) of the plasma depends on the basic plasma properties. The \( \sigma \) is also called the complex Lorentz conductivity and can be derived from the equation of motion for an electron. Since the relative complex permittivity is related to the conductivity, the relative complex permittivity can also be written as a function of the same basic properties, which is done at the end of this section.

In the plasma of interest, mainly two forces act on an electron. First of all, the Lorentz force \( \vec{F}_L \), acting on a moving electron with velocity \( \vec{v}_e \), is given by

\[
\vec{F}_L = -e(\vec{E} + \vec{v}_e \times \vec{B})
\]  

(5.11)
Secondly, the effective collision force \( \vec{F}_{\text{coll},e} \) results from collisions of the electron with the heavy particles

\[
\vec{F}_{\text{coll},e} = -\nu_{\text{eh}} m_e \vec{v}_e
\]

(5.12)

where \( \nu_{\text{eh}} \) is the effective rate for momentum transfer from electrons to heavy particles and \( m_e \) the rest mass of an electron. The equation of motion reads

\[
m_e \frac{d}{dt} \vec{v}_e = \vec{F}_L + \vec{F}_{\text{coll},e}
\]

(5.13)

It is allowed to discard the magnetic field term because in our plasma, the mean-free-path length of an electron is much smaller than the cyclotron or gyration radius. This leads to

\[
\frac{d}{dt} \vec{v}_e + \nu_{\text{eh}} \vec{v}_e = -\frac{e \vec{E}}{m_e}
\]

(5.14)

Since the electrical field is harmonic, the electron velocity is harmonic as well, which allows the replacement of \( d\vec{v}_e/dt \) by \( j\omega \vec{v}_e \). Consequently,

\[
\vec{v}_e = \frac{-e \vec{E}}{m_e (\nu_{\text{eh}} + j\omega)}
\]

(5.15)

Substituting the current density \(^1\) due to the motion of the electrons

\[
\vec{J} = -e n_e \vec{v}_e = \sigma \vec{E}
\]

(5.16)

in Equation (5.15), leads to an expression for the complex electrical conductivity

\[
\sigma = \frac{e^2 n_e}{m_e (\nu_{\text{eh}} + j\omega)} = \frac{e^2 n_e (\nu_{\text{eh}} - j\omega)}{m_e (\nu_{\text{eh}}^2 + \omega^2)}
\]

(5.17)

Note that for \( \nu_{\text{eh}} \gg \omega \), the conductivity becomes purely real and equals the so-called direct current expression

\[
\sigma = \frac{e^2 n_e}{m_e \nu_{\text{eh}}}
\]

(5.18)

However, in our simulation this approximation fails as there is for example a transition from plasma to glass (in this transition \( \nu_{\text{eh}} \ll \omega \)). Introducing the plasma frequency \( \omega_p \)

\[
\omega_p = \sqrt{\frac{n_e e^2}{m_e \varepsilon_0}}
\]

(5.19)

the conductivity can be rewritten as

\[
\sigma = \frac{\omega_p^2 \varepsilon_0}{\nu_{\text{eh}} + j\omega}
\]

(5.20)

while the relative complex permittivity becomes

\[
\varepsilon_r = 1 - \frac{\omega_p^2}{\omega^2 + \nu_{\text{eh}}^2} \left( 1 + j\frac{\nu_{\text{eh}}}{\omega} \right) = \varepsilon_{r \text{ real}} - j\varepsilon_{r \text{ imag}}
\]

(5.21)

\(^1\) The contribution of ion velocity on the current density can be neglected due to the low mobility.
where
\[ \varepsilon_r^{\text{real}} = 1 - \frac{\omega_p^2}{\omega^2 + \nu_{eh}^2} \] (5.22)

and
\[ \varepsilon_r^{\text{imag}} = \nu_{eh} \left( \frac{\omega_p^2}{\omega^2 + \nu_{eh}^2} \right) \] (5.23)

are both real.

5.2.3 The Maxwell Equations in Cylindrical Coordinates

The region under investigation is rotationally symmetric. It is fed by electromagnetic energy through the resonator slit. Due to the design of the resonator and the sizes of the slit, only rotationally symmetric fields can reach the region of interest. Therefore, it is worthwhile to rewrite the Maxwell equation in cylindrical coordinates and use the rotational symmetry. This analysis results in two independent sets of three equations. These two sets contain transverse magnetic (TM) and transverse electric (TE) fields respectively (transverse with respect to \( r \)). Only one set makes sense, because of the properties of the resonator slit. This remaining set of Maxwell equations describes the TM fields.

The Maxwell equations, for TM fields in cylindrical coordinates, are now reduced from 9 to the following 3 equations.

\[ j\omega \mu_0 H_\phi + \partial_z E_r = -\partial_r E_z = 0 \] (5.24)
\[ \delta_r H_\phi + j\omega \varepsilon_0 \dot{\varepsilon}_r E_r = 0 \] (5.25)
\[ -r^{-1} \partial_r r H_\phi + j\omega \varepsilon_0 \dot{\varepsilon}_r E_z = 0 \] (5.26)

5.2.4 Normalisation

To make sure that the magnetic and electric field quantities are numerically in the same order, the \( H_\phi \) field is scaled with the pure imaginary resistance \( Z = -j\sqrt{\varepsilon_0/\mu_0} \). The scaled magnetic field is noted as \( \hat{H}_\phi \)

\[ H_\phi = -j\sqrt{\varepsilon_0/\mu_0} \cdot \hat{H}_\phi \] (5.27)
This is beneficial for the numerical stability of the problem. The corresponding Maxwell equations of the TM-field become

\begin{align}
\dot{H}_\phi + k_0^{-1} \partial_z E_r - k_0^{-1} \partial_r E_z &= 0 \\
-k_0^{-1} \partial_z \dot{H}_\phi + \epsilon_r E_r &= 0 \\
-k_0^{-1} r^{-1} \partial_r r \dot{H}_\phi + \epsilon_r E_z &= 0
\end{align}

(5.28) (5.29) (5.30)

where

\[ k_0 = \omega \sqrt{\epsilon_0 \mu_0} \]

(5.31)

is the wave number for vacuum.

5.3 Discretisation of the Maxwell Equations

![Figure 5.4: The location of variables on the discretised grid. \( H_\phi \) is located in the middle of a grid cell, coinciding with a nodal point. \( E_z \) is located to the north and south of this, while \( E_r \) is located on the east and west. The axial (z) direction points along the east-west direction and the radial (r) axis along the north-south direction.](image)

To numerically solve the Maxwell equations in complex media such as a plasma, a discretisation is needed. We choose for a finite difference (FD) method. One advantage of this method is its simplicity. A staggered grid is chosen to discretise the volume under investigation. This grid encompasses everything inside the resonator. Thus, the plasma region as well as the glass tube and air shell are included. The grid has exactly the same cell sizes at the plasma region, as the cells of the flow simulation. Because of this, there is an easy correspondence between the two grids. \( H_\phi \) is defined at the nodal points of the grid (see Figure 5.4). Since derivatives of \( E_r \) are needed in the z-direction, this variable is defined at the east-west gridpoints. \( E_z \) is defined at north-south points as its derivative is needed in the radial direction.
The operator matrix notation of the Maxwell equations that need to be solved is

\[
\begin{pmatrix}
1 & k_0^{-1} \partial_z & -k_0^{-1} \partial_r \\
-k_0^{-1} \partial_z & \varepsilon_r & 0 \\
-r^{-1}k_0^{-1} \partial_r & 0 & \varepsilon_r
\end{pmatrix}
\begin{pmatrix}
\vec{H}_\theta \\
\vec{E}_r \\
\vec{E}_z
\end{pmatrix}
= \begin{pmatrix}
0 \\
0 \\
0
\end{pmatrix}
\quad (5.32)
\]

The discretisation scheme replaces the derivation operators \( \partial_z E_r \) and \( \partial_r E_z \) by their matrix approximation. Since \( E_r \) is defined on the north-south grid-points, and \( E_z \) is defined on the east-west grid, no interpolation is required to find these derivatives. As an example,

\[
\partial_z E_r = \frac{E_{r, \text{east}} - E_{r, \text{west}}}{z_{\text{east}} - z_{\text{west}}} \quad (5.33)
\]

The numerical derivative in the \( r \)-direction is discretised in a similar way

\[
\partial_r E_z = \frac{E_{z, \text{north}} - E_{z, \text{south}}}{r_{\text{north}} - r_{\text{south}}} \quad (5.34)
\]

These are just two derivatives for one point (namely at a nodal point). The total matrix structure will be given in Section 5.5, after the boundary conditions of the problem are discussed.

5.4 Boundary Conditions

To build the complex matrix equation, boundary conditions are needed. A distinction can be made between internal and external boundary conditions. Since the EM grid encompasses several material regions, there are internal boundaries between glass, air and plasma. These internal boundaries are treated first. Since these regions can have different complex relative permittivities, it is important to interpolate those correctly.

After that, the external boundary conditions are treated, which give expressions for the components of \( \vec{E} \) at the boundaries. The south boundary condition arises from the symmetry axis located there. There are two simple boundaries on the east and west side (on the sides of the gas outlet and inlet respectively). Finally, the north boundary condition is treated, which is that of the resonator wall, chokes and the slit. These are the most complex boundary conditions.

5.4.1 Internal Interface Boundary Conditions

As stated before, the EM grid is an extension in the radial direction of the plasma grid. In addition to the plasma area, it also includes other regions like the quartz substrate tube and the air shell. In the plasma region, the fluid and electromagnetic grids exactly coincide. The relative complex permittivity is calculated from chemical composition quantities that are known at the nodal points. Therefore, \( \varepsilon_r \) is also defined at the nodal points. However, in the discretised Maxwell equations the relative complex permittivity is needed at the east-west and north-south grids. This means, that interpolation has to be done. The glass (or quartz) medium has a uniform \( \varepsilon_r \), so interpolation is exact within that region. The same holds for the air medium. Especially in the case of a
stretched grid, it is important that interpolation is done correctly. In the plasma, interpolation is accurate as long as the grid is refined enough. Since there are no discontinuities in the permittivity, values between adjacent gridpoints will not differ too much. But care must be taken at the boundaries between two regions as discontinuities do exist there. Because the values can differ much there, care must be taken when grid sizes are different in the plasma region and the glass region.

These boundaries always contain points located on the north-south grid, thus $E_r$ is defined there. The complex relative permittivity at such a point is undefined (because there is a jump). Therefore the property that the field components are continuous across the interface is used. This leads to a scheme in which the permittivity at the interface is set to a specific value.

In such a transition between two media, the electric-field component perpendicular to the boundary ($E_r$) is discontinuous while the component along the boundary ($E_z$) is continuous. Because there are no active magnetic media, the magnetic-field component ($H_\phi$) is also unaffected by the boundary.

![Figure 5.5: Internal boundary conditions](image)

Regard the example in Figure 5.5. The grid cell north of the boundary is called $N$, while the one south of the boundary is called $S$. The radius at the boundary is $R$, at the nodal point north of the boundary $R^N$ and at the nodal point south of the boundary $R^S$. Similarly, the value of $H_\phi$ at the nodal point south of the boundary is called $H_\phi^S$ and the one north $H_\phi^N$. The value at the boundary itself is merely called $H_\phi$ because it is continuous across the interface. This variable is only known at the nodal points and not on the north-south grid. However, this is no problem since it will vanish from the final equation.

In this example, the boundary is between plasma (south) and glass (north). The following Maxwell equation is considered

$$-r^{-1}\partial_r(r\hat{H}_\phi) - k_0\varepsilon_r E_z = 0$$

and it is discretised with a left FD scheme on half the grid cell above the
and on half the grid cell below the boundary

\[-R^{-1} \frac{R_N^N H_N^N - R H_\phi}{R_N - R} + j \omega \epsilon_{\text{glass}} E_z = 0 \quad (5.36)\]

One finds values for $E_z$ at the interface because $E_z$ is also continuous across the interface.

The first equation is multiplied by $(R_N - R)/(R_N - R_S)$ and the second one by $(R - R_S)(R_N - R_S)$. Added together, this gives

\[-R^{-1} \frac{R N^N H_N^N - R_S H_S^N}{R_N - R_S} + j \omega \epsilon_{\text{glass}} E_z = 0 \quad (5.37)\]

Where the $\epsilon_{\text{avg}}$ is given by

\[\epsilon_{\text{avg}} = \epsilon_{\text{glass}} \frac{R_N - R}{R_N - R_S} + \epsilon_{\text{plasma}} \frac{R - R_S}{R_N - R_S} \quad (5.39)\]

In this way, the interface conditions are included without changing the matrix structure. For the boundary between glass and air a similar relation holds.

### 5.4.2 Symmetry Axis

As can be deduced from Figure 5.2, the south boundary is the symmetry axis. These points lie on the north-south grid, so $E_z$ is defined there. Since the solution should be rotationally symmetric, $\tilde{E}_z$ in the nodal point at the middle of the first cell north of the symmetry axis should be equal to the value at the nodal point of the first cell which would be south of the axis (i.e. mirror the grid on the symmetry axis). To find an expression for $E_z$ at the boundary the following Maxwell equation is used.

\[-k_0^{-1} r^{-1} \partial_r r \tilde{H}_\phi + \epsilon_r E_z = 0 \quad (5.40)\]

and with $\tilde{H}_\phi(0) = 0$, this gives

\[\epsilon_r E_z(0) = -k_0^{-1} \left[ -k_0^{-1} r^{-1} \partial_r r \tilde{H}_\phi(r) \right] \quad (5.42)\]

\[\epsilon_r E_z(0) = -k_0^{-1} \lim_{r \to 0} \left[ \partial_r \tilde{H}_\phi(0)r^{-1} \partial_r r^2 + \partial_r \tilde{H}_\phi(0)r^{-1} \partial_r r^3 + \ldots \right] \quad (5.43)\]

which leads to

\[\epsilon_r E_z(0) = -\frac{2}{k_0} \left[ \frac{\partial \tilde{H}_\phi}{\partial r}(0) \right] \approx -\frac{4 \tilde{H}_\phi(r_N)}{r_N} \quad (5.44)\]

where $r_N$ is the radius at the nodal point above the symmetry axis. This boundary condition is different from that of Janssen [2], where the nodal points and east-west grid coincide with the south boundary.
5.4.3 Boundary Conditions at Inlet and Outlet Sides

The chokes (which reside on the north boundary) confine the EM waves in the z-direction, how this is done is explained in the subsection for the chokes below. Therefore it does not really matter what the boundary condition on the east and west boundary is, and metal plates are assumed for both of these sides because of the simplicity. Since points on the east-west grid coincide with these boundaries, $E_r$-values are stored here as can be seen in Figure 5.2. Electric fields along a metal plate are compensated by fields created by electrons in the metal, Which means that in this case the electric field along the r-direction is 0. So, the boundary condition of both east and west walls is $E_r = 0$.

5.4.4 Boundary Conditions at Resonator Wall, Slit and Chokes

The northern boundary is the most complex one. This boundary corresponds to the inner wall of the resonator.

Since the north boundary contains points located on the north-south grid, $E_z$ is defined there. As the metal of the resonator is situated in the $\phi$-z plane, the E-fields along these directions is 0. So for the parts where the north boundary coincides with metal of the resonator, the $E$-field in the direction of the metal $E_z = 0$.

There are three regions which are different though. Two chokes and one slit are seen in Figure 5.2. The chokes are basically holes with a depth (in the r-direction) of a length equivalent to $\lambda/4$. The electric field component in the z-direction at the top of this hole is 0. Because of this, standing waves have a maximum amplitude of the electric field, exactly where the resonator wall would be situated if there were no choke. This means that waves travelling from the slit to the east and west wall encounter a sharp transition of the E-field (from 0 to the maximum value). The effect of this is that the waves are reflected back. Thus, the waves are “captured” between the two chokes. This can be compared to a metallic wall for EM waves, but with the difference that gas can flow through and out of the plasma region. Of course these chokes may not work perfectly, so a more extensive description than metal plates is used. This is description is given below.

The slit is a hole in the north wall which connects the resonator cavity with the regions that are contained inside the resonator (i.e. the glass, air and the plasma under investigation). The slit is the source of electromagnetic energy in the model. Waves can also reflect back from the simulated region towards the slit; this effect is not neglected.

Radial waveguide

To describe the boundary conditions of both the slit and the chokes, a description of the waves in a radial waveguide is needed. A standard solution for a radial waveguide, where the waves enter and exit through the outer wall, is derived by Marcuvicz [22]. With this wave description, boundary conditions are derived for the slit first and then for the chokes.

The EM-field is expressed in a superposition of forward and backward travelling modes. Since the modes are coming in or out from the slit at the maxi-
imum radius, it is more instructive to speak of imploding and exploding modes rather than incoming and reflected modes respectively. The amplitudes of the imploding (incoming) modes are designated with $A_n$, where $n$ is an index for the "order" of the mode. The amplitudes of exploding (reflected) modes are designated with $B_n$. Note that $B_n$ is not the magnetic flux density but an amplitude (which describes the reflected mode of both the electric as well as the magnetic field).

The general description of these waves in cylindrical coordinates is written as a function of a specific type of Bessel functions, the so called Hankel functions. These are complex functions which are constructed by using lower order Bessel functions [23]. They are solutions to the differential equations of type

$$x^2 \frac{d^2 w}{dx^2} + x \frac{dw}{dx} + (x^2 - f^2)w = 0$$

where $f$ is a constant and $x$ can be any complex variable. The Bessel functions of the third kind $H$ (Hankel functions) are written in terms of Bessel functions of the first kind $J$ and second kind $Y$ (which are both available in standard mathematical function libraries). Imploding waves are described in terms of

$$H^{(1)}_f(x) = J_f(x) + jY_f(x)$$

While exploding waves are written in terms of

$$H^{(2)}_f(x) = J_f(x) - jY_f(x)$$

The argument $x$ is in our case $\kappa_n r$, where

$$\kappa_n = \sqrt{k_0^2 - \frac{n\pi}{b}}$$

in which $b$ is the width of the slit. Note that $k_0$ is the wave number corresponding with a propagation in vacuum. Its index 0 refers to vacuum, and thus has nothing to do with the zero order $n = 0$.

The EM-fields are now described by a superposition of all imploding and exploding modes [22]

$$E_z(r, z) = \sum_{n=0}^{\infty} B_n e_{z,n}(z) H^{(2)}_0(\kappa_n r) + \sum_{n=0}^{\infty} A_n e_{z,n}(z) H^{(1)}_0(\kappa_n r)$$

$$E_r(r, z) = \sum_{n=0}^{\infty} B_n e_{r,n}(z) H^{(2)}_1(\kappa_n r) + \sum_{n=0}^{\infty} A_n e_{r,n}(z) H^{(1)}_1(\kappa_n r)$$

$$H_\phi(r, z) = \sum_{n=0}^{\infty} B_n h_{\phi,n}(z) H^{(2)}_1(\kappa_n r) + \sum_{n=0}^{\infty} A_n h_{\phi,n}(z) H^{(1)}_1(\kappa_n r)$$

The definitions of the normalised modal field distributions are

$$e_{z,n}(z) = \chi_n \cos \left( \frac{n\pi z}{b} \right) \frac{1}{b}$$

$$e_{r,n}(z) = \chi_n \sin \left( \frac{n\pi z}{b} \right) \frac{1}{b} \frac{\kappa_n}{\kappa_0}$$

$$h_{\phi,n}(z) = \chi_n \cos \left( \frac{n\pi z}{b} \right) \frac{1}{j\kappa_0 \sqrt{\mu_\epsilon_0}}$$

33
where $z$ is the distance along the slit (or choke), so $z = 0$ is the west-side of the slit (or choke) and $z = b$ is the east-side. Here, $\chi_n$ is defined as

$$\chi_n = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n \neq 0 \end{cases} \quad (5.55)$$

This $\chi_n$ is used in the normalised modal field distributions because of the normalisation, since

$$\int_0^\pi \sin^2(n\pi z/b)\,dz = \int_0^\pi \cos^2(n\pi z/b)\,dz = \begin{cases} b/2 & \text{if } n = 1, 2, \cdots \\ b & \text{if } n = 0 \end{cases} \quad (5.56)$$

so,

$$\int_0^\pi b^{-1} \chi_n \cos^2\left(\frac{n\pi z}{b}\right)\,dz = 1 \quad (5.57)$$

**Solution for chokes and slit**

To find the solution for the choke, assume that $R^{\text{choke}}$ is the radial position of the top of the choke. This is usually a distance of $\lambda/4$ north to the position where the inner resonator wall would be ($R^N$) if the choke were not present.

Because the top of the choke is metallic, the electric field along it $E_z(R^{\text{choke}}, z^{\text{choke}}) = 0$ at this position. Therefore, $A_n$ and $B_n$ can be related to each other by

$$A_n = -B_n \frac{H_0^{(2)}(\kappa_n R^{\text{choke}})}{H_0^{(1)}(\kappa_n R^{\text{choke}})} \quad (5.58)$$

To find the boundary condition for $E_z$ at the choke, this equation is substituted in Equation (5.49).

Note that higher-order modes $A_n$ for $n > 0$ are attenuated strongly in the radial waveguide. So only the fundamental mode propagates: $A_0 \neq 0$. Higher order reflected modes $B_n$ for $n > 0$ can not contribute to the (reflected) power flow either. The net power flow from the waveguide to the resonator is measured in the industrial setup. Since this power flow can also be found from the amplitude of the incoming fundamental mode $A_0$ and that of the reflected mode $B_0$, an iterative procedure is used to find the correct $A_0$. A guessed value is used for $A_0$. The entire EM-model is run and the amplitude $B_0$ of the reflected fundamental mode is calculated. Then, a new guessed value is used for $A_0$ depending on whether the value was too high or too low.

The power flow in radial direction at the slit is given by [24]

$$P_r = \frac{1}{2} \text{Re}(S_r) = \frac{1}{\mu_0 \pi \omega b^2 r} (|B_0|^2 - |A_0|^2) \quad (5.59)$$

Both of the boundary conditions for $E_z$ at the north boundary (Equation 5.49 combined with the above two equations for chokes and slit respectively) are now written in a form usable in the Matrix structure. Consider one of the northernmost cells, which borders the slit or a choke. Locations in this cell are designated with an $M$ for the nodal point, $N$ for the point on the north boundary and $S$ for the point located south of $M$ (on the north-south grid). See Figure 5.6 for a representation of this boundary configuration.

So for example $H^{(N)}_N$ is defined at the nodal point and $R^S$ is the radius at the south point of the cell. The value of $E^N_z$ is given by the boundary conditions
Figure 5.6: Schematical representation of the choke and its boundary condition for \( E_z \).

derived above. Since the following form is used as one of the discretised Maxwell equations in the matrix

\[
H^N_k + k_0^{-1} \partial_z E_r - k_0^{-1} \frac{E^N_E - E^S_E}{R^N - R^S} = 0
\]  

(5.60)

and the left half of the last term uses the value of \( E^N_z \) at the boundary, the boundary condition is rewritten to fit better in this form. Therefore \( E^N_z \) is multiplied by \( 1/(k_0(R^N - R^S)) \). The result of this is that at the choke

\[
\text{BC}_{\text{choke}} = \frac{1}{k_0} \frac{1}{R^N - R^S} \sum_{n=0}^{\infty} B_n \chi_n \cos \left( \frac{n \pi z}{b} \right) \left( H_0^{(2)}(\kappa_n r^N) - H_0^{(1)}(\kappa_n r^N) \frac{H_0^{(2)}(\kappa_n R)}{H_0^{(1)}(\kappa_n R)} \right)
\]

(5.61)

and at the slit

\[
\text{BC}_{\text{slit}} = \frac{1}{k_0} \frac{1}{R^N - R^S} \sum_{n=0}^{\infty} B_n \chi_n \cos \left( \frac{n \pi z}{b} \right) H_0^{(2)}(\kappa_n r^N) + \frac{1}{k_0} \frac{1}{R^N - R^S} \frac{1}{b} A_0 H_0^{(1)}(\kappa_0 r^N)
\]

(5.62)

Note that higher-order modes require a larger minimum number of points at the slit, this is equivalent to the Nyquist theorem. Model tests have shown that the zeroth-order mode is dominant. Neglecting \( n > 0 \) terms could simplify the above equations significantly and speed up calculation time (as it requires less points at these boundaries). The implemented EM module leaves the option of neglecting the higher-order modes to the user.

5.5 Matrix Structure

This section describes the structure of the final matrix that is used to solve all the electromagnetic equations at once for all fields at every point in the simulated volume.
The total equation is of type
\[ \tilde{M} \cdot \tilde{X} = \tilde{S} \]  \hspace{1cm} (5.63)

where \( \tilde{M} \) is the matrix which contains all three Maxwell equations. \( \tilde{X} \) contains all the field variables which need to be solved (and some boundary conditions) and \( \tilde{S} \) contains the source terms that are necessary for the slit. All these elements are described below.

First of all, \( \tilde{X} \), the vector of the fields stores the electric and magnetic fields in the following manner. The first elements of \( \tilde{X} \) are all the (normalised) \( \tilde{H}_\varphi \)-values at the nodal points. This starts with the cell on the southwest corner of the calculation volume. The next is one cell east of it and so on. At some point the south-east cell is reached. The next point will be one cell to the north, starting again at the west side. So increasing the position in the vector \( \tilde{X} \) corresponds to going from west to east along the nodal points in the grid, or if the easternmost cell is reached going one cell to the north and restarting west.

The first few rows of cells are physically in the plasma since the first row corresponds to the centre of the cylinder of the calculation volume. After a certain number of rows of plasma cells, the next row of cells is inside the glass. After another number of cells in the radial direction the next region is reached, the air shell. This goes on until the cells at the north wall are reached.

After the last \( \tilde{H}_\varphi \)-value (at the northeastern nodal point) the next elements in \( \tilde{X} \) will be those of \( \tilde{E}_r \). The points of \( \tilde{E}_r \) are stored in a similar way, starting at the southwest, going from west to east and when the east side is reached going one row up to the north. However this time these \( \tilde{E}_r \) points are not defined at the nodal points but on the east-west grid. Because a metallic boundary condition is used on the west and east wall, the \( \tilde{E}_r \) points there are equal to zero, so they do not need to be calculated, therefore these points are skipped in \( \tilde{X} \).

After all \( \tilde{E}_r \) values are stored, the next ones are those of \( \tilde{E}_z \). Again the rows are stored from west to east, starting at the southern row and moving north each time a row is completed.

After all points of the three fields are stored in this manner, one or more \( \tilde{B}_r \)'s (from Equation 5.49) are stored. These are the amplitudes of reflected waves which are used in the boundary condition at slit and chokes. Figure 5.7 shows an example of how the variables in a grid are stored in the vector \( \tilde{X} \). The indices in the figure correspond to the position of the variable in this vector.

The matrix elements of \( \tilde{M} \) contain the Maxwell equations for these fields stored in \( \tilde{X} \) and some boundary conditions. \( \tilde{M} \) is treated at the same time as the source term vector \( \tilde{S} \) since they are both part of the same equations. \( \tilde{M} \) represents a discretised version of the matrix in Equation (5.32) and also includes boundary conditions.

To better understand this matrix, it is divided in several blocks. Specifically in four by four parts, although several elements are equal to zero. These blocks are identified by two values. The first number is the row position in the matrix, while the second is the column position. For example matrix block M11 is the top left element, M12 will be a certain number of rows to the right of it, followed by M13 and M14. Below M11 is M21 and so on. These blocks have a certain width and height that is dependent on what function the block represents. To show this, the exact meaning of these blocks is described. The top three rows of
equation blocks represent the three Maxwell equations. The last row of equation blocks is an equation for boundary conditions at slit and choke.

To illustrate what the matrix elements look like, an example grid of 3 cells wide and 2 cells high is taken which can be seen in Figure 5.7. As mentioned before, the position of the variables in the vector $\vec{X}$ is denoted with an index in the figure. The first 6 positions in $\vec{X}$ are values of $H$, the next 4 positions represent $E$, values while at the last 6 positions $E_z$ values are stored (the ones at the north boundary are not stored).

![Figure 5.7: An example of the grid for which the matrix is written. The index is the position in $\vec{X}$.](image)

![Figure 5.8: The matrix which is solved to get a solution for $H$, $E$, $E_z$ and the $B$'s.](image)

See figure 5.8 for a representation of the matrix $\tilde{M}$. The top row of blocks
in $\tilde{M}$ form the following equation, which is evaluated at the nodal points.

$$M_{11} \cdot \tilde{H}_\phi + M_{12} \cdot E_r + M_{13} \cdot E_z + M_{14} \cdot B = S \tilde{1} \quad (5.64)$$

which corresponds to Equation (5.28)

$$1 \cdot \tilde{H}_\phi + k_0^{-1} \partial_z \cdot E_r - k_0^{-1} \partial_r \cdot E_z = 0 \quad (5.65)$$

So the block $M_{11}$ corresponds with the unity matrix.

$$M_{11} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.66)$$

Block $M_{12}$ is $k_0^{-1}$ times the numerical derivative in the $z$-direction. This means that it will be the value of $E_r$ at the east-side of the nodal point, minus the value of $E_r$ to the left of the nodal point divided by the length between these points. Since the $E_r$-values are already defined between nodal points, no interpolation is required for this. So block $M_{12}$ will be a factor of $k_0^{-1}$ times the inverse of the width of the control volumes times a diagonal element that looks like $-1 \cdot 1$. The value of $-1$ needs to be the part of $M_{12}$ that is multiplied by the value of $E_r$ to the west-side of the nodal point, and the value of $1$ corresponds with the factor in front of the $E_r$ to the east-side of the nodal point of the $H_\phi$ grid position that the Maxwell equation is solved at. At the easternmost and westernmost points there will be a special situation as the $E_r$ values are not even stored since they are zero anyways. For the example grid, the matrix block $M_{12}$ looks like this

$$M_{12} = \frac{1}{k_0 \Delta z} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad (5.67)$$

where $\Delta z$ is the width of a cell in the $z$ direction.

Matrix element $M_{13}$ is also a factor times the numerical derivative. In this case, the derivative in the $r$-direction and of $E_z$. So the $1$ and $-1$ need to correspond to the values of $E_z$ which lie on the control volume points north and south of the nodal point the equation is solved for. Then, it is divided by the length of the control volume in the radial direction to get the numerical derivative. Since the values of $E_z$ are stored by filling a row from west to east before going to the next row, points that lie above each other are the same number of positions apart in the vector $X$ as the number of points that represent the width of east-west of the $E_z$ grid (which is the north-south control volume points grid). For
the example grid

\[ M_{13} = \frac{1}{k_0 \Delta r} \begin{pmatrix}
1 & 0 & 0 & -1 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 
\end{pmatrix} \] (5.68)

where \( \Delta r \) is the width of a cell in the \( r \) direction. The minus sign before \( k_0^{-1} \) is incorporated in the matrix block. For the north boundary, no values are given in \( M_{13} \) as these are put into \( M_{14} \) instead. Matrix block \( M_{14} \) contains the terms of Equations 5.61 or 5.62 which are dependent on \( B_n \), while \( S1 \) contains the terms dependent on \( A_0 \) in case of the slit.

The second row of matrix blocks when multiplied by \( \vec{X} \) gives

\[ M_{21} \cdot \vec{H}_\phi + M_{22} \cdot \vec{E}_r = S2 \] (5.69)

which corresponds to Equation (5.29)

\[ k_0^{-1} \partial_z \cdot \vec{H}_\phi + \vec{\varepsilon}_r \cdot \vec{E}_r = 0 \] (5.70)

and is evaluated at the east-west grid points. As there are no boundary conditions, \( S2 = 0 \). Matrix element \( M_{21} \) is similar to \( M_{12} \), as it is also a derivative in the \( z \)-direction. In this case, derivation of \( \vec{H}_\phi \) instead of \( \vec{E}_r \). Also there are no points assumed to be equal to 0. In the example grid, this gives

\[ M_{21} = \frac{1}{k_0 \Delta z} \begin{pmatrix}
-1 & 1 & 0 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 
\end{pmatrix} \] (5.71)

Note the symmetry \( M_{21} = -M_{12}^T \).

The element \( M_{22} \) is equal to \( \vec{\varepsilon}_r \) at that position. So depending on whether the grid cell represents plasma, glass or air, this value can be different. The way the plasma affects the EM-waves is also via this term and \( M_{33} \). Note that to find \( \vec{\varepsilon}_r \) at the east-west grid, it has to be interpolated from values at the nodal points. If in the example grid the lower 3 cells are plasma and the upper 3 cells are glass we get

\[ M_{22} = \begin{pmatrix}
\vec{\varepsilon}_r(\text{plasma}) & 0 & 0 & 0 \\
0 & \vec{\varepsilon}_r(\text{plasma}) & 0 & 0 \\
0 & 0 & \vec{\varepsilon}_r(\text{glass}) & 0 \\
0 & 0 & 0 & \vec{\varepsilon}_r(\text{glass}) 
\end{pmatrix} \] (5.72)

The next row of matrix blocks multiplied by \( \vec{X} \) gives the following equation

\[ M_{31} \cdot \vec{H}_\phi + M_{33} \cdot \vec{E}_z = S3 \] (5.73)

which corresponds to Equation (5.30)

\[ k_0^{-1} r^{-1} \partial_z r \cdot \vec{H}_\phi + \vec{\varepsilon}_z \vec{E}_z = 0 \] (5.74)
and is evaluated at north-south grid points. Matrix element $M_{31}$ is rather complex compared to the other elements, because it corresponds to $-k_0^{-1}r^{-1} \delta_r$ applied on $\bar{H}_\phi$. So for one derivative, three different radii are needed.

\[
M_{31} = \frac{1}{k_0 r^{NS} \Delta r} \begin{pmatrix}
BC & 0 & 0 & 0 & 0 & 0 \\
0 & BC & 0 & 0 & 0 & 0 \\
0 & 0 & BC & 0 & 0 & 0 \\
r^{NP, south} & 0 & 0 & -r^{NP, north} & 0 & 0 \\
0 & r^{NP, south} & 0 & 0 & -r^{NP, north} & 0 \\
0 & 0 & r^{NP, south} & 0 & 0 & -r^{NP, north}
\end{pmatrix}
\]

(5.75)

here $r^{NP}$ corresponds to the radius at the nodal point and $r^{NS}$ with the radius at the north-south grid where the $\bar{H}_\phi$ and $E_z$ points respectively are defined that correspond to that column in the matrix. BC is the boundary condition at the symmetry axis as discussed in Equation (5.44). Again, $M_{33}$ are values of $\varepsilon_r$, interpolated from the nodal points to the north-south grid this time. Since some of these points lie on boundaries, the internal boundary condition as described is used here with $\varepsilon_r$ (average) calculated from the relative complex permittivity of glass and plasma. For the example grid the following holds

\[
M_{33} = \begin{pmatrix}
\varepsilon_r^{(\text{plasma})} & 0 & 0 & 0 & 0 & 0 \\
0 & \varepsilon_r^{(\text{plasma})} & 0 & 0 & 0 & 0 \\
0 & 0 & \varepsilon_r^{(\text{plasma})} & 0 & 0 & 0 \\
0 & 0 & 0 & \varepsilon_r^{(\text{avg})} & 0 & 0 \\
0 & 0 & 0 & 0 & \varepsilon_r^{(\text{avg})} & 0 \\
0 & 0 & 0 & 0 & 0 & \varepsilon_r^{(\text{avg})}
\end{pmatrix}
\]

(5.76)

$S_3$ is again 0, as no boundary conditions are used.

Finally the same Maxwell equation is solved for the top grid cells in the fourth row of matrix blocks. Multiplied by $\bar{X}$ it gives the following equation

\[
\begin{align*}
M_{41} \cdot \bar{H}_\phi + M_{44} \cdot E_z &= S_4 \\
\end{align*}
\]

(5.77)

Here the boundary conditions of Equations (5.61) and (5.62) are used. Apart from that, these equations are the same as the third row of matrix blocks. $f(A)$ is simply a function of $A_0$, this is the second term on the right-hand-side of Equation 5.62.

5.6 Power Absorption

Power is coupled into the electron energy balance as a source term. The absorbed power is calculated by use of the complex Poynting vector [24].

\[
\bar{S} = \frac{1}{2} \varepsilon_r \bar{E} \times \bar{H}^* \quad (5.78)
\]

To see the relation between the magnitude of the E-field components ($E_z$ and $E_\phi$) and the dissipated power in a cell, the radial component of the Poynting vector

\[
S_r = -\frac{1}{2} \varepsilon_r E_z H_\phi^* \quad (5.79)
\]
Taking the first of the set of Maxwell equations for the rotationally symmetric TM wave in cylindrical coordinates (Equation (5.28)) and the complex conjugate for the second (5.29)) and third one (5.30)), and multiplying these with $H_\phi^*$, $E_r$ and $E_z$, respectively, gives

\begin{align*}
(j\omega\mu_0 H_\phi + \partial_z E_r - \partial_r E_z)H_\phi^* &= 0 \\
(\partial_z H_\phi^* - j\omega E_r^*)E_r &= 0 \\
(-r^{-1}\partial_r r H_\phi^* - j\omega E_z^*)E_z &= 0
\end{align*}

Adding these equations and using

\begin{align*}
\partial_z(H(E_r H_\phi^*) &= (\partial_z E_r)H_\phi^* + (\partial_z H_\phi^*)E_r \\
and
r^{-1}(\partial_r r E_z H_\phi^*) &= (\partial_z E_r H_\phi^*) - (\partial_r E_z)H_\phi^* E_z
\end{align*}

results in

\begin{align*}
P_{\text{out}} + P_{\text{EM}} + P_{\text{diss}} &= 0
\end{align*}

where the outward electromagnetic power flow $P_{\text{out}}$ is introduced as

\begin{align*}
P_{\text{out}} &= \partial_z(E_r H_\phi^*) - r^{-1}\partial_r (r E_z H_\phi^*)
\end{align*}

and the electromagnetic energy stored in the volume $P_{\text{EM}}$ is introduced as

\begin{align*}
P_{\text{EM}} &= j\omega\mu_0|H_\phi|^2 - j\omega\epsilon_0\epsilon_r^{\text{real}}(|E_r|^2 + |E_z|^2)
\end{align*}

and the dissipated power $P_{\text{diss}}$

\begin{align*}
P_{\text{diss}} &= -\omega\epsilon_0\epsilon_r^{\text{imag}}(|E_r|^2 + |E_z|^2)
\end{align*}

is only dependent on the imaginary part of the complex relative permittivity.

### 5.6.1 Power Check

In this chapter it was shown, that the power is calculated in two ways. The first way is comparing the amplitudes of the incoming and reflected wave. The second way is to integrate the power calculated with the pointing vector over the entire grid. If the matrix is solved correctly it is expected these two powers are the same, aside from discretisation errors. In the program a check is made on every iteration of the EM module to see if this is indeed the case. To verify that both ways of evaluating the power indeed give the right quantities, a unit check is made. While this is trivial it is also important for the model that this is indeed correct, so it is briefly mentioned. The unit check for the power $P_1$ calculated on the entire plasma volume $V$ with the pointing vector gives

\begin{align*}
P_1 &\sim |E|^2 \epsilon_r^{\text{imag}} \cdot \epsilon_0 \cdot \omega \cdot V \\ &\sim \frac{|N|^2}{|C|^2} \cdot \frac{|C|}{|V||m|} \cdot [s]^{-1} \cdot [m]^3 \sim \frac{|N|^2|m|^3}{|J||m||s|} \sim \frac{|N||m|}{|s|}
\end{align*}
Since the amplitudes $A$ and $B$ are given as a length times an electric field their unit is $[m][N/C]$. The unit check for power $P_z$ is then given by

$$P_z \sim \frac{A^2 - B^2}{\mu_0 \omega b} \sim [m]^2 \frac{[N]^2}{[C]^2 \cdot [\text{J}] [\text{m}^{-1}] [\text{s}^{-1}] [\text{m}]} \sim \frac{[A]^2 [\text{J}] [\text{m}]}{[\text{[J]}^2 [\text{s}^{-1}][\text{m}]} \sim \frac{[N][m]}{[s]}$$

with $b$ as the width of the slit. These two dimensional checks show both powers are indeed of the correct units, whether they are also numerically the same is something that simulations show. Usually they agree within a factor of $10^{-10}$ with each other numerically.

### 5.7 Skin Depth

The so-called skin depth model is used as a simple approximation for the behaviour of EM waves in some plasmas. In the model the skin depth approximation is used to initialise the flow for the above introduced EM-model, as the skin depth model is many orders faster to calculate. This skin depth model is also used in a modified form to simulate a surfatron plasma. The skin depth is derived by using the wave equation. This equation is found by taking the curl of Faraday's law.

$$\nabla \times \nabla \times \vec{E} = -\nabla \times \frac{\partial \vec{B}}{\partial t}$$

(5.91)

A harmonic solution is taken where $\vec{B} \approx \vec{B}_0 \exp(\text{i} \omega t)$ and a similar exponential solution holds for the electric field and current density. All time derivatives then become simple multiplications with $\text{i} \omega$, so the time derivative can be moved in front of the divergence on the right hand side. When another Maxwell equation is then substituted and the entire equation is divided by its exponential term this results in

$$\nabla \times \nabla \times \vec{E} = -\text{i} \omega (\varepsilon_0 \text{i} \omega \vec{E} + \mu_0 \vec{J})$$

(5.92)

When Ohm's law is substituted too, the following equation is found

$$\nabla \times \nabla \times \vec{E} = \omega^2 \varepsilon_0 \vec{E} - \text{i} \omega \mu_0 \sigma \vec{E}$$

(5.93)

The first term on the right-hand-side is discarded and when assuming a simple flat wave this eventually leads to the approximation

$$\frac{E}{\delta^2} = \frac{1}{2} \omega \mu_0 \sigma E$$

(5.94)

which will give an estimate of the skin depth and the often used expression

$$\delta = \sqrt{\frac{2}{\omega \mu_0 \sigma}}$$

(5.95)

is found.
5.8 Grid Stretching

Sometimes, it is useful to have more grid points on places where large gradients are expected. For example, the skin depth model will give much better results when the radial spacing near the outer wall is much denser. The power input changes exponentially at the wall so a more detailed description can be generated if more points are available. But since taking a larger grid will make a simulation much slower it is a good thing to increase the density of points at the wall while leaving the density of points elsewhere much lower. So-called stretch functions can be made which accomplish this. The EM-module written in this research is implemented without making the assumptions of a uniform grid, so these stretch functions are also used in the created model.

Of course, these grid stretching techniques can also be used in the axial direction, to increase densities near the slit and other features with big gradients. Work has been done on this but in the final module this is not fully implemented yet. In Appendix G, more can be found about this subject.
Chapter 6

Simulations

6.1 Introduction

As stated before, the main focus of this graduation thesis was to create an EM coupling module (Section 5) that can be used in combination with the fluid (Section 4) and chemical (Section 3) modules to simulate a technological microwave plasma, especially one that is used in the manufacturing of optical glass fibres. This chapter will first discuss several simulations that were done in order to test our EM module in Section 6.2.

Simulations with a fixed relative complex permittivity are given to demonstrate the working of the chokes. The depth of the chokes is varied. It shows that the chokes are most effective when their depth is $\lambda/4$. Other simulations in Section 6.2 show how the EM module reacts to very high and very low (fixed) conductivities.

Secondly, this chapter will deal with simulations done with the Skin Depth model. To test parts of the Skin Depth model, a crude surfatron [4] EM plug-in for PLASIMO is created (see Appendix C.3). A surfatron launches EM waves along a plasma contained in a tube. In this case a plasma is created outside of the wave launcher, so many properties can be measured in such a plasma. An improved version of this model could be useful in the future to test different plasma compositions which can be compared with measurements. The plasma in a surfatron is more accessible than the one in the industrial setup at Draka so some parameters can be measured more easily. In the surfatron model, the user specifies a starting power and a fraction per meter that is absorbed in the axial direction. The cylindrical plasma region is discretised into slices. The model then calculates the power that is absorbed in each cylinder slice and uses a variant of the skin depth model to distribute this power radially within such a slice. Since the model is still quite crude and no comparison with experiments can be done, its results are not given here. It did provide some insight into the workings of the skin depth model and while implementing it, some bugs were fixed. Since the skin depth model is used to initialise the Draka simulations, it is compared with the results of a full simulation in Section 6.3.

To test some aspects of the chemistry and flow, a modified Poiseulle flow model with argon is created. The EM module used in this model and the flow are different from those in the Draka simulation. This model is used by others
as a basis for their simulations and has proven to be helpful finding some errors in the code for flowing systems. The results of this model are described in [11] and will not be given here, as this model was mainly constructed to see if the flow solver contained bugs which needed to be fixed for the simulation of the plasma at Draka. So thirdly, a complete simulation is done. This contains the flow, full EM module and argon chemistry. As mentioned before, this model is initialised with the skin-depth model.

6.2 EM Module Tests

To test the EM module, several artificial profiles for the permittivity were used. The real part of the relative permittivity is taken as 1 in all these simulations. The imaginary part is taken 0 on the outer radial part, and jumps to a fixed value for the inner radial part of the simulated volume. This fixed value can be changed by the user. Outside the simulated region, air is used which absorbs no EM power.

6.2.1 Chokes

The results for the ohmic dissipation can be seen in the Figure 6.1. One can see that the power is mostly dissipated in the middle between the 2 chokes. The slit position in this case is in the middle, the chokes are situated on the second grid cell from the east and west wall respectively. Note that simulations with more grid points, where it is possible to have the slit situated a little away from the middle, learned that slit position can have an impact on the shape of the ohmic power dissipation.

When varying the depth of the choke, this simulation (with the conductivity corresponding to an imaginary component of the complex relative permittivity of -600) shows the following changes.

The solution without chokes has more freedom in the z-direction. The effect of the chokes can be clearly seen, as the peaks positioned further to the east and west boundaries are lowered. The peaks in the middle are higher since the same amount of power is dissipated (400W).

A choke with a depth of less than \( \lambda/4 \) should in general be less effective. This effect is indeed clearly seen in Figure 6.1. A possible cause of the difference in intensity between the peaks immediately east and west of the slit, is that too few gridpoints were used in the z-direction. Since the slit and the chokes were positioned symmetrically in the z-direction, a symmetrical solution for the EM fields is also expected. Indeed, preliminary simulations with more gridpoints showed a more symmetrical power distribution.

6.2.2 Conductivity

In the next set of simulations that were done with fixed relative complex permittivities, the imaginary component of \( \varepsilon_r \) was varied. This component is related to the conductivity according to the description in Section 5.2.2.

With extremely large (negative) imaginary permittivity (and thus extremely large corresponding conductivity) as in Figure 6.2, the waves do not penetrate the surface, like in a metal. Peaks of the power dissipation on the surface with
Figure 6.1: Ohmic power dissipation at a fixed imaginary component of the relative permittivity of -600 on the inside of the volume. The dashed line shows the results with chokes and the whole line without chokes. The solution without chokes has more freedom in the z-direction. Chokes are positioned at the outer wall, 1 grid point away from the east and west walls. The slit is positioned in the middle of the outer wall. The effect of the chokes can be clearly seen, as the peaks positioned further to the east and west boundaries are lowered. The peaks in the middle are higher since the same amount of power is dissipated (400W).
a separation of about 6 cm (about half the wavelength of the microwaves) can be seen. That extend in the z-direction (along the length of the plasma column), these peaks are confined because of the chokes.

Figure 6.2: Ohmic power dissipation at a fixed imaginary components of the relative permittivity of -6000 on the inner half of the simulated volume. The waves extend all the way in the z-direction, and like in a metal do not penetrate the material but are confined on the surface (halfway of the simulated volume in the r-direction).

With a little lower conductivity, as in Figure 6.3 which resembles the situation in a plasma more, the wave is absorbed quite locally (and in approximation can be seen as the simpler Skin Depth model for one peak).

When the conductivity gets much smaller like in Figure 6.4 we see that the waves can penetrate much deeper into the material radially, at some point (Figure 6.5) they even reach the middle without being extinguished much.

6.3 Skin Depth Comparison

The ohmic dissipation of the skin depth model is compared to that of the complete Yee model (from the results of the model in the next section) in Figure 6.6. The agreement is good, so in the conductivity range of the plasma, the skin model is a valid approximation and can indeed be used to initialise the flow for the complete model.
Figure 6.3: Ohmic power dissipation at a fixed imaginary components of the relative permittivity of $-600$ on the inner half of the simulated volume. This situation resembles the hottest parts of the plasma (near the slit), and the dissipation of one of these peaks looks like what the skin depth model would give. Since the relative permittivity on the regions of the plasma with lower electron densities is usually much lower, only one peak is usually seen in a full plasma.
Figure 6.4: Ohmic power dissipation at a fixed imaginary components of the relative permittivity of -60 on the inner half of the simulated volume. Here the waves can easily penetrate the material.

Figure 6.5: Ohmic power dissipation at a fixed imaginary components of the relative permittivity of -6 on the inner half of the simulated volume. The waves can go all the way through to the middle, this situation resembles glass somewhat.
Figure 6.6: Comparison between skin model power input and complete Yee model. The dashed line is the complete EM simulation, while the full line is the skin depth model approximation.

6.4 PCVD Simulation

In this section, a simulation is executed which resembles that of Janssen [2] but with a few changes. In these simulations the full plasma area is used to simulate the plasma, the EM variables in the outer regions (for example glass and air) cannot be seen in the graphs. The model parameters are as follows. For the pressure a homogeneous second derivative Neumann condition (i.e. the derivative of the pressure in the z-direction at the inlet is 0) is chosen at the inlet, while the outlet pressure is fixed (Dirichlet) at 1000 Pa. For the inlet, the velocity distribution is chosen that is equivalent to a flow of 5 scc/s of Ar gas. For the outlet flow speed a homogeneous Neumann condition is taken (since the pressure is already defined there and the problem must not be overspecified). The heavy particle temperature at the inlet is taken as 500 K, while the electron temperature is 2000 K (these are both dirichlet conditions). The same dirichlet conditions are taken at the north wall for the temperatures. The power of the microwave generator is 400 W. The radius of the plasma area is 7.5 mm, the thickness of the glass is also taken as 7.5 mm. No air is used in this simulation (though it should be easy to add this).

In figure 6.7 we see that the pressure is almost constant over the length of the tube. There is a small bump at the spot where the plasma is at its highest temperatures. The pressure fall is used to propel the plasma through the tube. Since the pressure difference between the inlet and outlet is about 10 Pa, while in the simulations of Janssen [2] it is 65, a lower velocity in the z-direction is expected in our simulations. Since the pressure is almost constant, a quick
Figure 6.7: Pressure in the plasma region, a pressure fall of about 10 Pa is seen. In the simulations of Janssen, this pressure difference was 65 Pa.

estimate using the ideal gas law shows that temperatures scale with the velocity. Indeed a rough estimate shows that at the part where the temperature of heavy particles doubles, the speed is roughly double also.

The place where the ohmic dissipation is highest in Figure 6.8, also has the highest electron temperatures in the axial direction in Figure 6.9. An interesting radial temperature profile is created, since electrons are cooled by collisions with the wall, but the ohmic power dissipation, and thus the heating, is also highest near the wall. Since the electron temperature and ohmic power dissipation agree, the EM part of the simulation gives satisfactory results.

The temperature of the heavy particles can be seen in Figure 6.10. The values here are significantly lower than in Janssen [2]. A possible explanation might be higher losses at the wall in our model, since the dirichlet conditions on the north wall for the temperatures cool the plasma significantly.

The heavy particle velocity in Figure 6.11 is about a factor 5 lower than that found in the simulations of Janssen [2]. Since the pressure difference is also about 5 times lower in our case, and the pressure condition on the outlet is the same, the pressure condition in our model at the inlet might be different. Since this pressure is calculated from the velocity of particles given by the flow (in ssc/s) there might be something wrong in that part. Most notably the problem in this simulation may have been overspecified at the inlet. Since it was not possible to use relative densities at the inlet composition boundary condition, absolute values were used. As this determines the density, and the temperature is also given, the pressure is also determined at the inlet. In the flow solver one should either give a pressure or a velocity on a boundary, not both. By indirectly
Figure 6.8: Ohmic power dissipation, these values and its shape agree well with the results of Janssen [2].

Figure 6.9: Electron temperature. These values agree reasonably with the values found by Janssen [2].
Figure 6.10: Heavy particle temperature. This temperature is relatively low compared to the results in Janssen [2].

Figure 6.11: Velocity along the z-axis.
giving pressure, a conflicting boundary condition results. That the flow further downstream may be slightly different in other aspects can be explained by the many improvements to the flow model by K. Garloff. Another possible explanation is the much lower heavy particle temperature in our model. Because the pressure is more or less constant, a lower temperature means a different density and thus lower velocity.

Figure 6.12 and 6.13 show the density of the argon and argon ions. Of course the electron density is equal to the argon ion density in our case. Also the density of argon ions is related to the inverse of the density of argon.

![Argon density](image)

Figure 6.12: Argon density. The shape agrees well with that found by Janssen [2].

Finally the real part of the complex relative permittivity is shown in Figure 6.14 and the conductivity in Figure 6.15.
Figure 6.13: Argon+ density (and electron density).

Figure 6.14: Real part of electrical permittivity.
Figure 6.15: Electrical conductivity.
Chapter 7

Discussion

In this chapter, the results of the simulations and any knowledge gained from this study will be discussed. Also, future recommendations will be done.

The simulations with fixed permittivity show that the EM module reacts in a physically sensible way to artificial situations that range from materials that resemble a metal to a material with extremely low conductivity. The chokes also confine waves as expected, so this aspect of the EM module appears to work correctly as well.

The skin depth model gives comparable results to the full EM model in the permittivity range of our plasma. Thus it can be used as a means to initialise (and thus greatly speed up) the full model.

Furthermore, the EM parts like the ohmic dissipation and electron temperatures of the complete simulation gives similar results as Janssen [2], which is another indication that the EM module is implemented correctly. The stretching of the grid in the radial direction seems to work well also.

Note that the simulations in this report used a low number of grid points. To further improve the accuracy of simulations it is recommended that a higher number of grid points is used. This will slow down the model considerably though, so some optimisations to the model need to be done as well. In cases like simulations testing higher order modes and assymetry of the slit, more grid points are definitely needed.

Since the positions of the choke and slit edges are rounded to the nearest gridpoints, the accuracy is only correct with a very high number of gridpoints or when an exact double number of gridpoints is taken. The simulation is currently too slow with such a high number of points. The higher order modes of the choke and slit appeared not to be needed as they gave no notably different results as solutions with only the fundamental modes.

One aspect that needs to be looked at in the future is why the flow model and its interaction with the chemistry in our model gives different results than those in Janssen [2] (the EM part does give similar results). Most notably the pressure at the inlet, which follows from the gas flow, is too low to give the same flow speed as in Janssen [2]. So the boundary condition at the inlet needs to be changed. Preliminary simulations show a different flow, although it is still not the same as that of Janssen [2].

Sadly, it was not possible to compare the model to measured values save for very crude ones. In the industrial setup it is hard to measure plasma param-
eters, see the report by Deckers [26]. However, the values for electron density and temperature are in the same range as the simulated values. To get a better validation it is needed to create an experiment where the plasma can be measured more easily. A surfatron for example, would work better. The plasma in such a configuration is more accessible to diagnostics than a plasma situated inside the resonator of the industrial setup at Draka. If such an experiment is built, the model can be verified and refined further. When adding more complex chemistry, such a setup can be used to verify parts of such a model also.

One of the assumptions in the model, that the movement of the resonator is negligible, may have to be reviewed when deposition is built in. Heating of the wall can be an important effect and will be affected by this movement.
Bibliography

[1] Empedocles See Appendix H, Acragas, Greece


Appendix A

Plasimo Cookery Book
A.1 Introduction

Plasimo is a so called 'model factory'. A wide variety of plasma models can be constructed and executed with it. Since a plasma can be regarded as a fluid with special properties (it contains charged particles for example) it is also possible to employ Plasimo to construct models for problems in fluid dynamics. More general, since Plasimo uses transport equations and conservation laws to simulate a plasma, heat transport can be modeled as well.

It might be instructive to compare Plasimo with standard packages like Mathlab in which many models of physical systems can also be constructed. Plasimo models are different from those made with these packages in the sense that the coefficients in the partial differential equations (PDE) describing plasmas are strongly dependent on plasma conditions. Therefore special attention is needed for interaction procedures. Plasimo contains a vast library containing various matrix solvers one can use in these calculations.

A.1.1 Modularity

Plasimo has been built modularly. It contains a powerful parser for .gum input files. One of its features is automatic unit conversions. Plasimo can be easily extended by using plugins. The creation of plugins is described in another document, refer to plugin.txt in the doc directory.

A.2 Model syntax

A description of how one can generate a model is shown below. These models are either created with a simple text editor or from scratch with the aid of the graphical user interface of the wxPlasimo version. Refer to the examples tutorial on how to create specific models from scratch.

A model consists of several different nodes, an analogue would be directories and subdirectories in a hierarchical filestructure. Some nodes contain data which corresponds to files in this directories analogue.

If you are interested in which nodes can contain which other nodes (sub nodes) refer to the .tin files in the input/tin directory. For a complete overview read the plasimodoc.pdf file (after generating it with pdfTeX). wxPlasimo itself generates context sensitive menus derived from these .tin files.

Each model has to contain a root node, which describes what kind of model it is. This root node is called model. There are several model types, for example LTE, nonLTE and bulk.

To see which sub nodes are possible, right-click on the parent node, a context sensitive menu will appear. You can choose options like Add MainIterConfig, Add EM (with several options in a sub-menu) and such. Some of these nodes are required for each model, others are optional for some models.

A.2.1 Node description

A description of the most used nodes follows here.

The main (root) node in general is the model node. This main node can contain many different sub nodes depending on which main node you picked. For most models you will use an LTE or nonLTE model.
• Iteration configuration, this node describes the maximum number of iterations the model will make, and the desired residual accuracy at which you are content with your solution. Plasimo generates a plasma iteratively. A residual can be calculated by checking the conservation laws, this residual is then used in a subsequent iteration to improve on your solution.

• Discretizer type, currently only the Steadystate type is used here. In the future, time dependent models will use other types (CrankNicolson and ImplicitEuler).

• EM definition, here the source of EM energy is described which is generally used to heat the plasma. Several options exist here, choose one according to what you want to simulate. When one clicks on such a node many variables can be set. For example the microwave (MW) configuration has many options.

• Plasmaregion, this is one of the most important nodes in the model and is used to define the plasma. It contains many sub nodes. Some plasmas can contain multiple plasma regions. A more thorough description of these nodes follows.

Plasmaregion consists of the following sub nodes.

• Transport, this node contains many sub nodes itself that describe the diffusion, thermal conditions, specific heat, electrical conditions and viscosity of heavy particles in the plasma.

Several of these subnodes are variables which have to be updated on each iteration. To do this one has to add a calculator. For example in the Diffusion sub-node one can add a StdDiffusionCoeff calculator. Select the appropriate calculator for each of the variables according to what you want to simulate. Also refer to the tutorials and help window to see which calculators should be used for which variable.

• Grid, the grid in plasimo is always 2 dimensional. Plasimo uses a curvilinear orthogonal coordinate system. As long as your grid is a structured mesh and it suffices the above criteria it can have any form. Several standard geometries like cartesian, cylindrical and circular are already defined. But geometries like that of a lightbulb can also be constructed (as long as it has cylindrical symmetry).

• Boundary configuration, this node describes which pieces on each of the four boundaries are defined. You can add a label for each of the pieces for user convenience. Also the physical length of a piece can be set.

• Temperature, the initial function, and boundary conditions or other limitations the temperature has to comply to are set here. In nonLTE models there are two different nodes instead, TemperatureHP (for heavy particles) and TemperatureEL (electron temperature) which can have different restrictions.

• Mixture, this node contains a description of the chemical species, crosssections of collisions, particle relations and radiation. The species node contains a list of particle nodes, each describing physical properties of one
particle (ions, neutral particles, electrons, molecules, charged molecules and so on). The relations node contains a list of all reactions that can happen in the plasma between the species defined in the species node (for example the ionisation of one particle). Each of these reactions also has a rate defined in a subnode.

- Conservation rules, this node contains data about the filling temperature of the gas used to create the plasma in case of a closed system.

- Composition, this node contains rules the plasma has to comply to like charge neutrality, partial pressure and a temperature lookup table in case of LTE to make composition (Saha) calculations go faster.

- Flow, this node contains initial and boundary conditions of velocities for the axial, radial and azimuthal directions. Also start and boundary conditions of the pressure are defined here. These properties influence the flow in the plasma. Note the azimuthal component is currently not used as Plasimo is still 2 dimensional.

There are several other nodes in the plasmaregion like electron pressure. Or stream function, which one can use to validate the models conservation rules. This is also useful in your output, 'streamlines' will be created which correspond to lines of momentum transport.

A.2.2 LUT, boundary conditions and CRM

Several functions can be sped up with a LUT or lookup table, the option for these will appear in the context sensitive menu when one is available. A LUT is a table in which for different configurations a certain variable is calculated. This means that variable does not have to be recalculated every iteration, but merely an interpolation between these values has to be done. Using a LUT can speed up calculations significantly.

If you view a model file with a text editor you will see each node has a name and the data inside brackets { } define the content of it. Make sure there is a space between the bracket and the nodes name when creating a model with a text editor instead of the wxPlasimo interface. An underscore before the name of a section will disable it, this is very helpful when creating a model and comparing things.

Boundary conditions can be of a few different types. The entire boundary for one variable can be set to conditions like Dirichlet (set value), Neumann (first derivative), Second derivative and so on. These values can also be zero in which case the Homogeneous Dirichlet, Neumann, etc. conditions arise.

The boundary conditions can also contain a profile, for example if the temperature on the border of that boundary is lower then in the center. Last but not least plugins can be used to define the profile and behaviour of these boundary conditions (for example if they are dependent on the local pressure, or wall reactions are occurring).

The composition of the plasma can be optionally calculated by the use of a CRM, this CRM is fully integrated with the rest of the model. This CRM is used for example when a nonLTE model is defined (for LTE Saha can be used).
A.3 Running your model

When one has created a .gum file which contains all these nodes and the needed variables are defined in these nodes the model is complete.

Start up wxPlasimo and 'install' the model, if there are any errors in the model (missing nodes for example) the program will tell you exactly what is missing. Correct these mistakes accordingly.

Of course only the most simple of errors can be caught by the program. Make sure your model is physically correct. Check for any overspecification, which means that you left just enough freedom for the model to do its calculations. For example don’t specify the flow and a pressure difference between inlet and outlet, since that pressure difference (with viscosity and geometry) defines the flow. Also make sure elementary things like mass conservation are observed. So for example do not define both the mass inflow and the mass outflow, maybe this will converge to either a vacuum (if more mass exits the plasma then enters it) or a black hole (if more mass enters then exits) but probably such a model will not converge at all.

A.3.1 Diverging models

When all these things are done right, the model can be run. Because plasma situations are often not the most stable ones, make sure you adjust the relaxation factors (URF) accordingly if the model will not converge. Run a few simulations to see which parameters are causing the instabilities, then recheck if no boundary conditions are overspecified and play with the URF’s. URF’s are a kind of rubberband attached to a variable to make sure it does not oscillate too much between two consecutive iterations.

Some simulations might require a good approximation for the initial conditions, adjust those if the model takes a long time to converge or does not converge at all.

When playing with the parameters a nice feature of plasimo is that you can adjust the grid size, use this feature to speed up your calculations so you can check your adjustments quickly to see if you change was for the better or the worse.

As mentioned a possible source of divergence can be wrong starting conditions, since some solutions might be unstable already the solution might go to a wrong or diverging point if you have for example a completely wrong pressure starting condition. Check that these are more or less realistic, as even if your model converges it might take much longer to converge if you supply bad initial conditions. Therefore after running the model one time you might significantly speed up the convergence process of subsequent simulations by loading a file with the results of a previous simulation as starting conditions.

A.4 Expandability

One of the advanced features in plasimo is the creation of plugins. These allow you to, for example, add boundary conditions not similar to any available, add a totally new EM module or add calculators.
Writing plugins requires knowledge of C++ object oriented programming, yet it is very efficient to write plugins in plasimo. A very nice feature like automatic registering of the plugin makes this a powerful tool indeed. Writing plugins is described in chapter 3, [jan01], and practical experience are referred to.

Often users can create simple models without the need for plugins as the existing options are already very extensive.

A.5 Tutorials

The following example shows how to use PLASIMO to create models to simulate fluid flow.

Building a model from scratch is not easy with PLASIMO in the version I am currently using. First, start wxplasimo. Create a new model by clicking on File, New, and selecting Model Document. This creates a new model. Next, type the foil wing in the right panel:

Type ModelBulk

Note that this is case-sensitive. Right-click on the model node. Select the "Add other" option, and insert an unnamed node. Next, delete it, by selecting it, right-clicking it and clicking the "remove section" option. This process has created a root node. An other way of doing this is modifying an existing model. Possibly in future versions of PLASIMO this will become more elegant.

By right-clicking on the root node, a context sensitive menu appears. The next step is to use this menu to add a MainIterConfig node. This controls the iteration procedure. Here, the maximum number of iterations, the desired accuracy and the frequency of creating a log file can be selected. The number of iterations should at least be ten thousand, a tolerance of 1e-5 is sufficient for this example.

Next, create a discretizer node in the root node by selecting Add Other. This new node should be called Discretizer (case-sensitive!) In the right plane, type

Type SteadyState

Create a node in this Discretizer node, and remove it. This has created a SteadyState discretizer node.

Now, we can begin with the real model. Add a Plasmaregion node to the root node. Select a bulk plasma node. This node defines where the plasma exists. We are using Cartesian coordinates, so select a grid. Choose a default cartesian grid. Let's make a long, thin channel, 1 m long (set x1 to 1 m) and 0.01 m wide (set x2 to 0.01m). For both coordinates 30 grid points (n1 and n2) will suffice.

Now we add boundary configurations. Create a BoundaryConfig node in the Plasmaregion node. Now, in this node, 4 boundaries should be added: a north, south, east and west wall. In these boundaries, a label should be given to each piece of the boundary. In this case, the north wall is a rigid wall, so we append a line, in which we write that it is a wall.

PLASIMO does not use this information, it is merely for user convenience. Tell that the endpoint is 1 m from the beginning (PLASIMO uses units, these generally must be supplied). Do something similar for the other 3 sides.

The next step is adding a Flow(Simple) node. There are three velocities, v1 flows from west to east along the x1 axis and v2 flows from south to north along the x2 axis. v3 does not exist in this 2D problem, however, it should be specified.
The pressure node describes the pressure. When we look at the first node, there are three subnodes. IterConfig describes some numerical parameters. For this calculation, the underrelaxation factor should be 0.8 for all three velocities. Next, we see a function node. This contains a starting point for the calculation. The better this starting point matches the reality, the faster the calculation. A very poor choice may even cause the calculation to fail. 0 m/s is an acceptable guess for v1, v2 and v3.

The boundary conditions subnode is very important, because it defines the interaction between the plasma and the outside world. A good choice of boundary conditions is essential. For this problem, the north and south boundary are walls. This means, that all velocities are 0 on these walls. A formal term for this is an homogeneous Dirichlet condition. These should be used for all three velocities.

For v1, the derivative in the v1 direction is constant at the boundary. This is called an homogeneous Neumann condition. This means that the east and west boundary should have these conditions. In the other 2 directions, there is no flow at these boundaries, so here homogeneous Dirichlet conditions are appropriate.

The pressure is what drives this flow, so good modelling of this is of particular importance. Let's take \((0.1-0.05 \times x1/m) \times Pa\) (put this in the "function" subnode). Now, the pressure gradients at the walls should be zero, so add homogeneous Neumann nodes to the south and north walls. In the east and west directions, the pressure should be given. At the west wall, use a ConstDirichlet. Set the pressure to 0.1 Pa. At the east wall, the ConstDirichlet should be put to 0.05 Pa.

We then tell PLASIMO what kind of fluid we are modelling. Go to the flow(simple) node. We are using water here. It has a viscosity of 10^-3 Pa s. (Don't use the units here). In "The Add other entries" section, the following line is added to tell PLASIMO the initial density of water:

```
INIT..DENS 1e3 * kg/m^3
```

Finally, in the MainIterConfig, there is an Add Other Entries box. Here, init no should be used. This completes the model. Next, install the model. Click on the regions node, then on the DefaultCartesianGrid, and select the real node. Then, choose a few graphs that seem interesting to you. Pressure, v1 and v2 should yield some insight. Then, run it. This may take a while. If the solver claims it is finished after one step, click on step, and on run to start the iteration. Note the graphs are converging to the solution, the familiar Poiseuille profile.

Having done this, we note the excellent convergence. A different profile \((0.1-0.05 \times x1^2/m^2) \times Pa\) is worse. If we just use the average pressure of 0.075 Pa as an estimate, we will see very poor convergence. This is because we have done some really bad things. This starting value for the pressure is a very poor choice indeed. At first sight, it may seem okay because it represents the average value of the pressure. However, it creates steps in the pressure near the edges. This is particularly bad, because it creates a very large value of the derivatives near the edges. This demonstrates the importance of the starting condition.

Another point of importance is PLASIMO's inability to deal with turbulence. A key concept for turbulence is the Reynolds number \(Re\), that is defined as

\[ Re = \frac{\rho \times L \times v}{\nu} \]

with \(\rho\) the density, \(L\) the typical length scale, \(v\) the typical speed and \(\nu\) the viscosity. For our current problem, \(v=0.006\) m/s, \(L=0.01\) m, \(\rho=1000\) kg/m^-3.
and \( \nu = 0.001 \) Pas, which yields a Re of 6. Generally, PLASIMO fails somewhere between Re=1000 and Re=10000, when a good starting condition is supplied.

Saving your results.

In a real simulation, one obviously wants to save one's results. This is relatively easy in PLASIMO. The save functionality is located in MainIterConfig. The "output data path" is where one can specify the output. It is recommended that each model has its own directory. This way, all the files, and there will be quite a few, will be in the same place. You can also specify its extension and the save frequency. Ever 100 to 10000 iterations will do fine. PLASIMO can use the results of a previous calculation for the new one. The "Initialize from file" should be enabled, an the correct file specified.
Appendix B

Writing a Plugin
B.1 Introduction

One of the most powerful features of the Plasimo package is its modularity and the ease with which one can expand the capabilities of the program. Despite the fact that the package is very comprehensive there will always be simulations where one needs features not yet present in the Plasimo toolkit. This chapter will teach you how to create a so-called plugin to extend the capabilities of Plasimo.

We start with an example which will demonstrate the writing of a plugin. After that, some aspects are looked at for other types of plugins. Then a little overview is given so you can see quickly which files have to be created or changed. This will give you a start to write a plugin, but of course this short overview is not complete. When you want to write a plugin for yourself you also need to be familiar with the other Plasimo documentation (especially the automagically created .doxy files). Note this is a quite advanced feature of plasimo so it might be necessary to read more documentation before you can write a plugin.

B.2 Example 1: the Mach-1 Velocity Boundary Condition

We start with an example where a wall condition is created. This plugin was created to implement the mach 1 condition for the flow at the exhaust of a cylindrical plasma.

It calculates profile along the boundary you specify it for, of the velocity in the axis direction (in the model we use this plugin in). It is meant to be used for the outlet in a flowing plasma. Of course this boundary is not a physical wall since there is a flow through it. It is an imaginary boundary where several parameters are defined.

The file itself without all these comments in between can be found as
/pltranseqns/plugins/mach1cond.cpp
in your Plasimo source directory.

As you can see below, several header files need to be included. One can look at other plugin files to see which one are readily available.

/\*\* \file
  * Implements acoustic expansion velocity profile (M=1). Jan & Michiel.
  */
#include "plparser/node.h"

Data in Plasimo is stored in so called nodes, to access these nodes this header file is included. How these nodes are accessed is shown later on.

#include "plparser/log.h"
#include "plparser/fixme.h"
#include "plparser/cmplx.h"
#include "plmath/plasmath.h"

Several mathematics related and miscellaneous headers.
#include "plgeneric/provider.h"
This header file is needed to include the self registering features of Plasimo.

```plaintext
#include "plgeneric/accessor.h"
```

To access particle properties like density, diffusion of certain particles or the velocities in several directions, pressure, electrical conductivity and such (see the file itself) this needs to be included.

```plaintext
#include "plgrid/grid.h"
#include "plgrid/gridvar.h"
```

These are required for your plugin to work with grids in Plasimo.

```plaintext
#include "plgrid/vessel.h"
#include "plgrid/bndcond.h"
```

Of course because our plugin is a boundary condition, these grid related headers are needed.

```plaintext
#include "plgeneric/forcelnk.h"
FORCE_LINK_ME(mach1cond);
```

These lines makes sure that the current plugin is linked correctly when plasimo is compiled. It is mainly needed for cross platform compatibility (or rather, for fixing up Microsoft incompatibility with the world-wide standards) and for a statically linked version of Plasimo.

```plaintext
/** a quadratic profile with the maximum according to the Mach=1 condition. 
 * See the thesis of Janssen, eqn. 2.15. 
 */
class plMach1Cond : public plDirichletCond<REAL>
{

    Our plugin is written as a class, its a derived class from a standard dirichlet condition base class. Dirichlet means a constant value at the wall of the plasma, whereas Neumann defines a value for the derivative of a variable (note not to confuse a derivative of a class with a derivative to a value here ;). Our class has a real value as output, so a REAL template of the base Dirichlet class is used.

    public:
        /** After calling the base class' constructor, the power in the 
         * geometrical expression is read from the input node. 
         */
        plMach1Cond( plModelRegion *region, 
                     const plBoundaryPiece *piece, 
                     const plBoundaryCurve *bcurve, plNode& node) 
            : plDirichletCond<REAL>( region, piece, bcurve, node) 
        {
            m_power = node("Power");
            if ( node.firstL("Gamma") )
            {
                m_gamma=node("Gamma");
            } 
```

          73
else
{
    m_gamma=5.0/3.0;
}
}

The above lines is the constructor of our class, this constructor gets the data (pointers) from the standard Dirichlet condition class. To see which arguments this base class uses refer to plgrid/bndcond.h. Furthermore in the body of the constructor a member (of the plMach1Cond class) variable is set to the value of the node called Power. In the plasimo interface when our plugin is selected (make a boundary condition and use this plugin then click it to change its properties) this allows us to click on the Power node and change its value. As seen in the comment below, this 'Power' value determines the order of the polynome that defines the shape of the profile of our velocity. Likewise the protected member

\texttt{m\_gamma}

is set to 5/3 if no value is supplied or to the value you supply in the model for 'Gamma' \((Cp/Cv)\) in the interface.

/**
 * Implements the Mach=1 condition (sonic expansion). The maximum veloc
 * is multiplied with a geometry-dependent function. This function
 * is a polynome with the begin value set at the acoustic velocity
 * and the end value at 0. The order of the polynome is an input
 * parameter (power).
 */

virtual REAL F( unsigned int ndx) const

Here the standard boundary condition class apparently expects a member function \texttt{F} (look at plgrid/bndcond.h to see that this name has to be used) which is in our case dependent on the grid's index value. This index is a unique number for each point in the grid. Plasimo itself takes care of the iterations over the gridpoints if your plugin is defined correctly. So our function only has to generate a value of the speed on one point on the outlet boundary (where it intersects the flow symmetry axis, corresponding to the 'middle' of the outlet boundary in the physical setup).

unsigned ib,jb;
GetBndNdx(ndx, ib, jb);

Two variables, \texttt{ib} and \texttt{jb} are created. The function \texttt{GetBndNdx} is called to look up where in the grid the point is. \texttt{ib} and \texttt{jb} are indices of the point in the i and j direction. Whether this is the x and z direction is of course entirely dependent on how you defined the grid in your model. In any case, the location of our gridpoint is now known.

    // the boundary pressure:
    const REAL press = plAccessor::Pressure(Region()->Field()(ib,jb));
    const REAL dens = plAccessor::MassDens(Region()->Field()(ib,jb));

To calculate the value of our boundary condition we need the mass density and the pressure at this point. These values are accessed through the (guess what...) \texttt{plAccessor}, these apparently take a region as a parameter. This region is given at the i and j points we took earlier from the index value.
In this case the values we got were from a plNonParticleAccessor. If we want to know a particle dependent value, a so called plPerParticleAccessor is used. You can check the plgeneric/accessor.h to see which per particle and non particle accessors are available. Besides the grid index (thus position) a per particle accessor also needs to know the index of the particle species. For example in a model, the electron has a certain index and O5+ has a different one and so on. These indices are different in each model of course, depending on which species you defined. A plugin needs to be as general as possible so it can be re-used for different models.

```cpp
REAL umax = sqrt( m_gamma*press/dens );
```

The values of pressure and density we just got from the non particle accessors are used to calculate a value used in our expression. In this case the maximum value of the speed is calculated so we can generate a polynomial that defines the (relative) profile shape and later on we multiply with this umax value to get absolute values for our speed profile.

```cpp
const unsigned lndx = Piece()->NdxBegin();
const unsigned hndx = Piece()->NdxEnd();
const REAL mn = (*m-bcurve)[lndx];
const REAL mx = (*m-bcurve)[hndx];
const REAL coord = (*m-bcurve)[ndx];
```

To calculate the curve we need the coordinates of the indices on the wall we are specifying this boundary condition for. Therefore the beginning and end indices are saved first, we then use these indices to look up the position values of these points (in the direction along the wall we are creating the boundary condition for). We also look up the value of the current point. With these three coordinates we can now calculate how far along the wall our current grid point is located. We then use this to create the polynomial that we need (the profile we want to create).

This might look a bit complicated but it is needed to make the plugin work for differently defined grids and thus make the plugin usable for other types of models.

```cpp
const REAL relpos = 1.0 - pow( (coord-mn)/(mx-mn), m_power );
return umax*relpos;
```

The polynome is then stored in the relpos variable, which gives the relative position along the 'velocity axis'. Our polynome starts at a value of 0 for the speed on the outer rims (so where the plasma touches the real walls) and in the middle the speed is 1. We use a cylindrical plasma where the speed at the the point, where the symmetry axis intersects with the outlet wall, is taken as 1. And the speed on the outer walls (don't confuse this with the outlet 'wall'!) is taken as 0. Of course the plugin can be expanded to make it more general.

We now have the value for the speed at the current point relative to the center. We then multiply with the maximum velocity to get the velocity at our current grid point.
REAL m_power;
   
   // Cp/Cv
    REAL m_gamma;

Our member variable
m_power

which was used to store the value from the power node is protected, this is a variable we only need inside our plugin. Same goes for the member variable
m_gamma

};

REGISTER_PROVIDER( plBoundCond<REAL>, plMach1Cond, "Mach1Cond");

To make the plugin available for use this line is very important, it tells Plasimo that a wall condition calculator is available under the name Mach1Cond, and that it uses the implementation as found in the class plMach1Cond. Furthermore this plugin is used for generating a real valued boundary condition, so the real valued template is used of plBoundCond. So in summary this line self registers our plugin.

B.3 Finishing touches

The previous example showed how to write a plugin for a boundary condition and how to get values for density and pressure at one point in your grid. But this is far from the only applications of a plugin. In plasimo generally anything can be expanded by the use of plugins. This requires an understanding of the inner structure of Plasimo.

But once you know how this works, the power of the expandability of Plasimo through the use of self registering plugins becomes readily apparent.

This section will give a few notes on things that can be important when writing plugins. Of course this subject can only be skinned in this text, and again we refer to the code documentation generated with Doxygen\(^1\) once you are a little familiar with the plugins described in this text. Note that you should also make your plugin code suitable for Doxygen, refer to Doxygen documentation on how this is done. In short you need to use three slashes to comment out documentation comments, have your documentation before the line (or objects, members etc.) you want to comment on and you can use \TeX\ formatting to create an equation.

Also reading the source codes for other plugins can be very educational. These can be found in the source tree in the

pl*

directories in their plugin subdirectories. For example

plem/plugins

\(^1\)Doxygen is available at http://www.doxygen.org.
contains plugins that are related to the electromagnetic power input of the plasma. Likewise

plgeneric/plugins

contains generic Plasimo plugins, and

plgrid/plugins

is used for grid related plugins. plmath contains mathematics related plugins. plparser/plugins is related to the parser or data structure handling. plplot/plugins contains extensions related to the graphical output. plspecies/plugins has chemical and species related plugins. plscalarplugins contains code related to the transport equations. Finally plplugins contains some model type related plugins.

We illustrate a few of these things by the use of another example. In this case refer to plspecies/plugins/epssigmw.cpp but will not go as in depth as the previous example. In this plugin two calculator types are added. Therefore the plspecies/transcoef.h was included which relates to the transport coefficients. All other header files were already discussed (except plparse/consts.h which merely contains some physical constants which were needed in our calculation).

The plTransCalculator class templated as a real is used as a base class. Because we have to do two very similar things a base class is derived from it and this base class is then used to derive two classes we use as calculator types, but thats just an implementation detail.

Of note here is that at some point a per particle accessor is used to find for example the electron temperature at a certain point in your grid.

\[ elNdx = \text{Particles()}.\text{GetElectronNdx}(); \]

and then

\[ \text{const REAL Te = plAccessor::Temperature(point)[elNdx];} \]

or

\[ m = \text{Particles()[elNdx]->Mass();} \]

to find the mass of an electron.

Another thing of note is that units are used in the node:

\[ m_\omega = \text{node["Calculator"]("Frequency")["Hz"] \times 2.0 \times \text{Constant::pi};} \]

Another interesting aspect of this plugin is that Dolterate is a member function that is called when the plugin is used in Plasimo, the output value is passed on to the value pointer res, and a cross section matrix and the point are (const) inputs to this function.

Note that when this plugin was added several other files needed to be modified because one of the variables that was updated was not yet registered by Plasimo. So this is more complicated and in this case several things also need to be added in the plAccessor, nitemodel (and Itemodel), em .cpp and .h files and in the .tin files. Since this is probably the only case where such a construction was needed, we wont discuss this in more detail here. This kind of plugin knowledge is probably only needed for the developers of Plasimo and not for the end user.
One thing that is important to the end user is how to create a .tin file so your plugin gets a nice user interface (context sensitive menu and the form where you input data). Writing .tin files is pretty self explanatory when looking at existing files. For example to add something to the nite model interface we modified models.tin slightly and added something in the nite model section and added the calculators at the bottom near the other calculators.

```java
Calculator {
    Type MWConductivity
    Description String "Calculator for the electrical conductivity."
    Requires Type "Calculation type" readonly
    Requires Frequency "Field Frequency (set the same as in pofEM)" real Hz
}

ElectricalConductivity {
    Description String "Sigma calculation"
    Requires URF "Underrelaxation factor" real
    Requires Section Calculator
    Default {
        URF 1.0
    }
}
```

One important thing to know about this is that you have to put a space between the container (a sort of class) and the first bracket, the data you put in the required field 'frequency' is read out by your plugin from the node. For a more complete overview refer to the cookbook.txt file where we explain how to create a model (the framework of a general model is defined in these .tin files).

When creating your .tin files you also need to make an accompanying html document that describes your plugin. This way you create an interactive help function to the user of your plugin in Plasimo. Check the tin directory for examples on how this is done.

Plasimo manages underrelaxation factors (URF) itself so you dont need to include code for that in your own plugin. URF is a way of making a weiged average between the current iteration and the previous iteration, it can be used as a tool to make simulations more stable.

Also functions do iterations over the grid points automatically as long as you supply the correct member functions (Dolterate for example, or the F in the first example).

So in summary, if you need a plugin that is similar to an existing one you should copy the existing file to one with a different name. Change the class name and the creator and the registered name. Add your class to the makefile.am then add documentation and add a template to input/tin.

B.4 Summary

A short overview of Plasimo plugins is given in this section. It merely serves to remind you of what you need to do to make a plugin work in a brief fashion.

1. Check which includes are needed and what your base class is, check in what format the base class wants its 'input' and 'output'. In our case the
input was often a node and some other structures like a particle map or a curve or piece (in our case the exhaust 'wall'). Often you can just copy an existing file, modify its name, class name, register provider and members to create a new plugin.

2. Write your plugin. One thing of note is the use of nodes, which are used to read data input in your plugin. Another important thing is the plAccessor which is used to read values of parameters on the current iteration in your plasma. These accessors require spatial (grid) information. The particle accessors also require an index of your species. So for example to know a temperature (a per particle accessor) you need to supply information where (in the grid) and of which kind of particle you want to know it of. Also write comments in your plugin in a format suitable to be converted to html with Doxygen.

3. Add your plugin to the makefile.am to have it compiled.

4. Write a suitable .tin file to make sure your nodes are also usable in the graphical user interface. Refer to the cookbook.txt file for more information on this.

5. Add documentation in html referred to by the .tin files, so the user interface has a help function.

6. Only for making changes to the 'core plugins' of Plasimo (for some developers), you might need to know how to register your new variables in the plAccessor, the model files (to add your variable in the list of things that get updated every iteration) and any other files it might have impact on. With 'core plugins' I mean for example the nonLTE and LTE model definitions in the plplugins directory.
Appendix C

EM code

C.1 Draka EM Skin Plugin

#include "plconfig.h"
#include "pIgeneric/forceink.h"
#include "plem/base_em.h"
#include "plparser/log.h"
#include "plmath/consts.h"
#include "plmath/plasmath.h"
#include "plgrid/xyplotgvar.h"

FORCE_LINK_ME(draka..skin);

class plDrakaSkinData : public plBaseEMData
{
public:
    plDrakaSkinData( plModelRegion* reg, const plNode & emnode );
    virtual ~plDrakaSkinData();
    void CalculateFields();

private:
    // complete delta function, with DC value in low frequency, high 
    // sigma limit, and constant value in high frequency, low sigma limit. 
    REAL Delta(REAL sigma);
    // Power
    REAL m_P;
    // frequency
    REAL m_f;
    // volume
    REAL m_V;
    // angular frequency
    REAL m_omega;
    // attenuation factor after slit
    REAL m_alpha;
};
// Skin depth
REAL m_skin_depth;

// distance (cv) between slit and choke (assuming symmetric conf)
unsigned m_slitchoke;

// pos of slit (middle)
unsigned m_slitpos;

// pos of left choke
unsigned m_lchoke;

// pos of right choke
unsigned m_rchoke;

// function to use along z-direction
REAL gfunc(unsigned);

plDrakaSkinData::plDrakaSkinData( plModelRegion* reg, const plNode & emnode )
: pBaseEMData( reg, emnode)
{
    //error check: our module only functions on a cylindrical grid
    if (reg->Grid()->Type() != GridDefs::Cylindrical) {
        Log(4) << "DrakaSkin: wrong grid type. Need cylindrical.\n";
throw plException("DrakaSkin: need a cylindrical grid!");
    }

    // initializations
    m_skin_depth = 0;
    m_P = emnode("Power")["W"]; //power
    m_f = emnode("Frequency")["Hz"]; //frequency

    // dit staat er al 3 keer, das dus 2x overbodig
    const plGrid *g = ohm.Grid();
    const plFDGrid& npg = (*g)(plGrid::LocNP);
    m_slitpos = npg.n1()/2; // + 1;
    m_slitchoke = unsigned(emnode("Slitchoke"));
    m_lchoke = m_slitpos - m_slitchoke;
    m_rchoke = m_slitpos + m_slitchoke;

    m_omega = m_f * 2.0 * Constant::pi; //angular frequency

    // volume of the whole simulation area
    m_V = reg->Grid()->volume();
    Log(2)<< "Volume of the tube is " << m_V << " m3\n";
}

REAL plDrakaSkinData::gfunc(unsigned zpos)
{
    const plGrid *g = ohm.Grid();
    const plFDGrid& npg = (*g)(plGrid::LocNP);
    // const plFDGrid& cvg = (*g)(plGrid::LocCV);
    // unsigned slitpos = npg.n1()/2 + 1;
}
REAL dist = npg.c1().phys(zpos) - npg.c1().phys(m..slitpos);
REAL scaledz = 2 * dist /
    ( npg.c1().phys(m.rchoke) - npg.c1().phys(m..l choke) );
REAL gval = pow(fabs(scaledz)-1,4); // -pow(scaledz,4) + 1;

//#if 0
Log(2) << " gval=" << gval << "\n";
//#endif
return fabs ( gval);

void plDrakaSkinData::CalculateFields()
{
    //Convenient reference to the grid
    const plGrid *g = ohm.Grid();
    const plFDGrid& npg = (*g)(plGrid::LocNP);
    const plFDGrid& cvg = (*g)(plGrid::LocCV);

    //Ohmic dissipation (?)
    REAL POhm = 0.0;
    // attenuation of E field
    TBCI::Vector<REAL> att (npg.n2());
    REAL halfatt = 0.0;

    // Iterate from the left side of slit to the left choke, store the
    // power dissipated in the slices and scale up afterwards
    unsigned h,j;

    // Iterate from left choke to right choke for power calculation
    // the maximum is set at 1, and the entire power will have to
    // be scaled to match the input power afterwards.
    for ( h = m..l choke; h < m.rchoke; h++ )
    {
        REAL damp..sig = 0.0;
        REAL Power = m..P;

        // Iterate from second-topmost control volume (number n2-2, topmost control
        // volume is n2-1 )
        // Iterate from second-bottommost control volume (number 1, we stop counting
        // at 0)
        for ( j = (npg.n2()-2); j>=1; j-- )
        {
            const pos pos(h,j);
            // volume of this r-slice
            // r = npg.c2().phys(j);
            // \todo TODO: USE Grid::dx(...) HERE!
            const REAL dr = cvg.c2().phys(j) - cvg.c2().phys(j-1);
            // dV = 2.0 *Constant::pi * r * dr * dz;
            const REAL dV = g->volume(pos);
        }
// current attenuation based on previous one
if (j == npg.n2()-2) // at outside, start with 1
    att(j) = 1.0;
else // take previous value, and add second half step,
    // taken from the previous volume
    att(j) = att(j+1) * half_att;

// calculate new attenuation for half step
half_att = exp(-(0.5*dr) / Delta(sig(pos)));

// add first half step
att(j) *= half_att;

// volume averaged damping
damp_sig += sig(pos) * sqr(att(j)) * dV;

// TBCI::Vector<REAL> E_wall;
// E_wall.resize(npg.n2());
//Scaling with dissipated power: P= 1/2*sigma E^2
// but, there is a 2 in the definition of alpha
const REAL dz = cvg.c1().phys(h) - cvg.c1().phys(h-1);
const REAL PartInSlice = gfunc(h) * dz;
const REAL E_wall = sqrt((PartInSlice * Power)/damp_sig);

/// i for x2 direction: counter-intuitive - JvD
for (j = (npg.n2()-1); j>=1; j--) // npg.n2()-2 was het
{
    const Pos pos(h,j);
    //volume of this r-slice
    const REAL dV = g->volume(pos);

    const REAL E_field = E_wall * att(j);
    const REAL dissipation = 0.5 * sig(pos) * sqr(E_field);
    POhm += dissipation * dV;

    Log(2) << "diss=" << dissipation << " ,efield=" << E_field << "\n";
    m_Eimposed1(pos) = E_field;
    ohm(pos) = dissipation;
}

REAL P_scalefactor = m.P / POhm;
REAL E_scalefactor = sqrt(P_scalefactor);
POhm = 0.0;
for (h = 1; h < npg.n1()-1; h++)
{
    for (j = 0; j <= npg.n2()+1; j++)
    {
        const Pos pos(h,j);
}
m_Eimposedl(pos) *= E.scalefactor;
    ohm(pos) *= P.scalefactor;
    POhm += ohm(pos) * g->volume(pos);
}

// debug stuff
Log(2) << "DrakaSkin: total cylinder power in = " << POhm
    << " W" << std::endl;
// fill boundaries, since the boundary volume (and thus the dissipation is zero)
// the dissipation density is irrelevant.
for (h = 1; h <= (npg.nl()-2); h++)
{
    m_Eimposed1(Pos(h,0)) = m_Eimposed1(Pos(h,1));
    m_Eimposed1(Pos(h,npg.n2()-1)) = m_Eimposed1(Pos(h,npg.n2()-2));
    ohm(Pos(h,0)) = ohm(Pos(h,1));
    ohm(Pos(h,npg.n2()-1)) = ohm(Pos(h,npg.n2()-2));
}
for (j = 0; j <= (npg.n2()-1); j++)
{
    m_Eimposed1(Pos(0,j)) = m_Eimposed1(Pos(1,j));
    m_Eimposed1(Pos(npg.nl()-1,j)) = m_Eimposed1(Pos(npg.nl()-2,j));
    ohm(Pos(0,j)) = ohm(Pos(1,j));
    ohm(Pos(npg.nl()-1,j)) = ohm(Pos(npg.nl()-2,j));
}

REAL plDrakaSkinData::Delta(REAL sigma)
{
    REAL k = m_omegafConstant::LightSpeed;
    REAL f = sqrt(1.0 + sqr(sigma/(m_omega*Constant::epsO)));
    // Error catch to prevent dividing by 0 if you have no absorption whatsoever.
    // Perhaps write a log?
    if (f <= 1.0) // very small sigma
    {
        Log(2) << "absorption too low !\n";
        return 23.0; // A random number
    }
    return 1.0 / (k*sqrt((f-1.0)/2.0));
}

class plDrakaSkin: public plBaseEMProxy<plDrakaSkinData>
{
    public:
        plDrakaSkin( plRegionManager* manager, const plNode & emnode )
            : plBaseEMProxy<plDrakaSkinData>( manager, emnode )
        {
            Log(2) << "In plDrakaSkin's constructor\n";
        }
        /** Calculates the fields by merely calling

virtual void DoIteration( REAL eps )
{
    Log(2) << "In plDrakaSkin::DoIteration\n";
    for ( unsigned ndx=0; ndx < m_emRegs.size(); ++ndx)
        m_emRegs[ndx]->CalculateFields();
}

protected:
virtual plDrakaSkinData *CreatePlasma( plModelRegion* reg,
    const plNode& node)
{
    return new plDrakaSkinData( reg, node);
}

REGISTER_PROVIDER( plBaseEM, plDrakaSkin, "DrakaSkin");

C.2 Draka EM Plugin

/** \ile em_draka.cpp
 * An EM plugin for calculating the field in a Draka plasma reactor.
 * \todo create input node [partly done – JvD]
 * BDMAT_ON will change the Matrix to BdMatrix. There are several
 * #if 0 #endif statements, you can play with different solvers there
 * lu.decomp seems to be the best in our case.
 * changed: Chokes now work
 * changed: Higher order modes implemented
 * changed: Grid fitting is implemented, fixed a bug in FindClosestPoint.
 * changed: A real plasma can now be loaded, some strange occurrences:
 *   1) Solution is very sensitive to changes in geometry (this may be
 *      physical though).
 *   2) Sim diverges so far, so need to play with it. Need to do
 *      some optimisations or this will take too long.
 * \author Michiel van den Donker
 * \date Mar 2003
 */

//\todo Document defines below
#define rregion 2
#define zregion 7
#define slit 3

//\todo defines below: niet vastzetten maar afh van geometrie
#define noslitpoints 1
#define nochokepoints 1
#define nochokepoints2 1

/*
dont switch this on, the MSVC comp wants 1.6 GB ram!
* But dont switch on, LU solver is SLOWER with BDMATs
* that have too many diagonals.
* Note that this must be defined before tbcLconfig.h
* gets included (possibly indirectly).
*/
#define BDMAT_ON 0

#include "plconfig.h"
#include "tbcLconfig.h"
#include "tbcLmatrix.h"
#include "tbcLvector.h"
#include "tbcLsolver.h"
#include "tbcLplasmath.h"
#include "plmathLconsts.h"
#include "plmathLcmplx.h"
#include "plmathLplasmath.h"
#include "plparserLnode.h"
#include "plparserLlog.h"
#include "plgenericLforcelnk.h"
#include "plgenericLrelax.h"
#include "plgridLgvinit.h"
#include "plgridLgridvar.h"
#include "plemLbase_em.h"

#include <algorithm>
#include <iostream>
#include <cmath>

I I I
\todo Remove this dependency: replace with plmath/hankel.h functions
#include "hankel.draka.h"

FORCE_LINK_ME(em.draka);

class plEMDrakaData : public plBaseEMData
{
    public:
    /** Initialise all the input variables we use.
     */
    plEMDrakaData( plModelRegion* reg, const plNode & emnode );
    /** The plasmas relative complex permittivity.
     */
    plGridVar<plComplex> epsnp;
    /** \todo Document me.
     */
    void CalculateFields();
    
    private:
    /** Complex root function for kappa
     * \bug Why is sqrt(-k'^2) taken to be -i*k instead of +i*k?
     */
plComplex kappafunc(double kappasqr) const;
/** Interpolates the relative complex permittivity to the
 * EW and NS grids.
 */
void epsInterpolate();
/** Calculate the power input according to the power balance.
 * The dissipated part Pdiss = -omega * epsilon_r.Img * |E|^2
 * We calculate the parts of Pdiss resulting from Ez and Er on
 * respectively the NS and EW grid, then interpolate both to
 * nodal point and add the terms.
 */
void PInterpolate();
/** Create the Maxwell Matrix, this is only done once. After
 * it is created the only elements that will change are
 * the ones with the complex permittivity on the diagonals
 * on the upper lefthand side of the matrix
 * and the elements with A in it (input amplitude of the
 * imploding wave).
 * Variables to help with iteration boundaries of M(i,j).
 * First we generate the matrix elements for the 1st Maxwell equation:
 * M11 * Hp + M12 * Er + M13 * Ez + M14 * B0 = 0
 * (or = cons*A at slit/chokes)
 * One can add higher order B's with M15 B1 and so on if needed.
 * We start at the upper left corner of M12 which is the
 * 0th row and the Hptot'th column.
 * Next we generate the matrix elements for the 2nd Maxwell equation:
 * M21 * Hp + M22 * Er = 0
 * Next we generate the matrix elements for the 3rd Maxwell equation:
 * M31 * Hp + M33 * Ez = 0
 * Next we generate the matrix elements for the boundary
 * conditions at the slit or chokes:
 * M41 * Hp + M44 * B = A
 */
void CreateMaxwellMatrix();
/** Fill the value of M at the supplied row and column.
 */
void FillM(int Mrow, int Mcol, const p!Complex& Mfillval);
/** M11 = 1.0 on all regions at nodal points since the
 * relative permeability (mu_r) is 1 everywhere.
 */
void FillM11(int koffset, int loffset);
/** M12 is a (numerical) derivative of Er in the z—direction divided
 * by k0. Since we do not use a Poisson grid the distance between
 * points in the z direction is the same no matter what the current
 * radial position is. Because of that we iterate in a different way,
 * we 'jump' across little matrix blocks, as we iterate first in the
 * radial direction instead of the axial direction to fill these.
 * This is done for speed reasons (the lookup of a length in the grid
 * and one division are done Nr[g] times less ). Note again that this
* does not work for deformed (poisson) grids.
*/
void FillM12(int koffset, int loffset);
/** Matrix element M13 is the derivative to r of Ez times the inverse
 * of k0.
 */
void FillM13(int koffset, int loffset);
/** Matrix element M14 is the boundary condition for Ez in slit
 * and chokes of the north wall.
 * This method also fills part of the source term vector relating
 * to the first Maxwell equations boundaries.
 */
void FillM14(int koffset, int loffset);
/** Matrix element M21 is 1/k0 times the derivative
 * in the z direction of the scaled Hp
 */
void FillM21(int koffset, int loffset);
/**
 * Document me
 */
void FillM22(int koffset, int loffset);
/**
 * Document me
 */
void FillM31(int koffset, int loffset);
/**
 * Document me
 */
void FillM33(int koffset, int loffset);
/**
 * Document me
 */
void FillM41(int koffset, int loffset);
/**
 * Document me
 */
void FillM44(int koffset, int loffset);
/** Change the eps values in the plasma region on M22 and M33.
 * These will be calculated on the nodal points and interpolated
 * to the NS and EW points by epsInterpolate().
 */
void AdjustMaxwellMatrix();
/** Fill a value in Mtotal if supplied a row and column index.
 */
void FillMtotal(int Mrow, int Mcol,
                const plComplex& Mfillval);
/** Write data from the solution vector to the plasimo variables.
 */
void WriteSolutionFields();
/** Write the fields to the standard error output.
 */
88
void DumpFields(std::ostream& os) const;

/** Calculate complex relative permittivity with complex conductivity
 * as \( \epsilon_r = 1 + \frac{\sigma}{j \omega \epsilon_0} \).
 * So if we know the real part of \( \epsilon \) and the real part of \( \sigma \),
 * we can calculate \( \epsilon = \epsilon_{\text{real}} - \sigma_{\text{real}} / (\omega \epsilon_0) \).
 */
void epsCalc();

/** Use closest point on each corner point to find the x1 coordinate
 * and calculate index of a point and thus dz[g].
 */
void MetricsToPoints(int Nzpl);

/** Find the closest gridpoint in the z (or x1) direction
 * A bubble sort (halving the region one searches in) is not
 * necessary, since this only happens in the initialisation stage.
 */
unsigned GetClosestPoint(const double) const;

// Data testing routine
void DumpMatrix(std::ostream& os) const;
void DumpMatrixPart(std::ostream& os,
const int, const int, const int, const int) const;
void DumpSourcePart(std::ostream& os,
const int, const int) const;
void WriteFile();

// Matrix type switch
#if BDMAT_QN
typedef TBCI::BdMatrix<plComplex> MatrixType;
#else
typedef TBCI::Matrix<plComplex> MatrixType;
#endif

// Hphi, Ez, Er en Bn in one vector, whoaah!
TBCI::Vector<plComplex> m_Solution;

// The initial Matrix, is copied to m_Mtotal before each iteration.
MatrixType m_M;

// The modified matrix for use in the current iteration.
MatrixType m_MtotaJ;

// Vector with the source terms divided by \( A_0 \) (mostly 0 except in slit).
TBCI::Vector<plComplex> m_Source;

// The modified source term with \( A_0 \) for use in the current iteration.
TBCI::Vector<plComplex> m_Stotal;

// Number of elements in the solution vector.
int Hptot, Ertot, Eztot, Btot, Msize;
int Hptot_plasma, Ertot_plasma, Eztot_plasma;
int Nztot, Nrtot;
int Nr[region];
double dz[region]; // Widths of the areas in z direction.
int Nz[region];
/** Total number of areas: index 0 = plasma, 1 = glass, 2 = air, etc.
  * \todo Could this more elegantly be a subregion-structure???
  * \todo Clarify comment. What is "etc.". Clearly define regions.
*/
int Nmodi[region];
int Nz_plasma, Nr_plasma;
double delta_r[region];
double R[region];
double Rchoke[7]; // of 7?
double m_Omega;
double m.Pin;

/// position and width of choke 1
double m_zchoke1, m_dchoke1;
/// position and width of choke 1
double m_zchoke2, m_dchoke2;
/// ???
double m_zslit, m_bslit;

double m_A0; // A0, input power coefficient
plRelax<double> m_relA0; // relaxation for A0
/// relative permittivity in regions outside plasma area.
/// \todo 4?? express in other defines!
plComplex eps_out[4];
double m_k0;
const plGrid* m_grid;

plGridVar<plComplex> epsew;
plGridVar<plComplex> epsns;

plGridVar<plComplex> Er_ew;
plGridVar<plComplex> Ez_ns;

plGridVar<double> P_ew;
plGridVar<double> P_ns;
plGridVar<double> P_ew_to_np;
plGridVar<double> P_ns_to_np;

};

plEMDrakaData::plEMDrakaData( plModelRegion *reg, const plNode & emnode )
  : plBaseEMData( reg, emnode),
    epsnp(reg, "Plasma complex permittivity", plGrid::LocNP),
    m_relA0(0.5),
    m_grid(reg->Grid()),
    epsew(reg, "Plasma complex permittivity EW", plGrid::LocEW),
    epsns(reg, "Plasma complex permittivity NS", plGrid::LocNS),
    Er_ew(reg,"Er on ew", plGrid::LocEW),
    90
Ez_ns(reg,"Ez on ns", plGrid::LocNS),
P_ew(reg,"Pdiss on ew", plGrid::LocEW),
P_ns(reg,"Pdiss on ns", plGrid::LocNS),
P_ew_to_np(reg,"Pdiss from Er on np", plGrid::LocNP),
P_ns_to_np(reg,"Pdiss from Ez on np", plGrid::LocNP)
{
    // \todo Make sure all variables here are initialised!
    // From the nodes also.

    Log(2) << "plEMDraka: Start of initialisation.\n";

    Nz_plasma = epsnp.n1() - 2;
    Nr_plasma = epsnp.n2() - 2;

    Hptot_plasma = Nz_plasma * Nr_plasma;
    Ertot_plasma = (Nz_plasma - 1) * Nr_plasma;
    Eztot_plasma = Nz_plasma * Nr_plasma;

    Ertot = Ertot_plasma;
    Eztot = Eztot_plasma;
    Hptot = Hptot_plasma;

    R[0] = Grid()->cd2().phys.max();
    R[1] = R[0] + 0.0075;

    Nr0 = Nr_plasma;
    for ( int g = 1; g < rregion; ++g)
    {
        // \todo Remove hardcoded numbers
        // \todo ADD MORE POINTS OUTSIDE PLASMA IF NEEDED
        Nr[g] = Nr_plasma; // = 1;
        // \todo dit mag niet, laat delta_R
        delta_r[g] = (R[g] - R[g-1]) * (1.0/Nr[g]);
    }

    Nrtot = Nr_plasma;
    for ( int g1 = 1; g1 < rregion; ++g1)
    {
        Nrtot += Nr[g1];
    }

    Nztot = Nz_plasma;

    Hptot = Nztot * Nrtot;
    Ertot = (Nztot - 1) * Nrtot;
    Eztot = Nztot * Nrtot;

    // \todo Move some more stuff to the input file.
    // m_Pin = 400;
    // m_Omega* = 2.46e9 * 2.0 * Constant::pi;

91
m.Pin = emnode("PowerIn")["W"];
m.Omega = emnode("Frequency")["Hz"] * 2.0 * Constant::pi;
m.kO = m.Omega * sqrt(Constant::muO * Constant::epsO);
Log(2) << " * k0 = " << m.kO << std::endl;

const double dresonator = Grid()->cdl().phys_max();
m.zchoke1 = 0.25 * dresonator;
m.zchoke2 = 0.75 * dresonator;
m.zslit = 0.5 * dresonator; // position of slit (not width!)

/// \todo Clarify comment below
/// 2 SLITPOINTS AND 2 CHOKE POINTS (per choke)
m.bslit = noslitpoints * dresonator/Nz.plasma;
m.dchokel = nochokepoints1 * dresonator/Nz.plasma;
m.dchoke2 = nochokepoints2 * dresonator/Nz.plasma;

/// \todo Do not hardcoded lambda. Express in frequency
 double lambda = 0.146; // wavelen of MW's
 Rchoke[5] = Rchoke[4];

/// \todo eps_out[>2] uninitialised ?
/// \todo Testing code is on by default?
 eps.out[1]=plComplex(1.0,0.0); // air FOR TESTING!
 eps.out[1]=plComplex(1.0,-0.01); // test "glas"
 eps.out[1]=plComplex(1.1,-0.0001); // glass
 eps.out[2]=plComplex(1.0,0.0); // air
 eps.out[3]=plComplex(1.0,0.0); // air
 eps.out[3]=plComplex(1.1,-0.0001); // glass

/// Start value for A0, assume no reflection.
m.A0 = sqrt(0.5 * Constant::muO * m.Omega * m.bslit * m.Pin);
m.re:A0.Store(m.A0);
for (int gz = 0; gz < zregion; ++gz)
{
    Nmodi[gz] = Nz[gz];
}
/// \todo Clarify comment
/ 2 chokes 1 modus

Msize = Hptot + Ertot + Eztot + Btot;

Log(2) << " * Calculating epsilon.\n";
epsCalc();

Log(2) << " * Interpolating epsilon.\n";
epsInterpolate();

Log(2) << " * Resizing matrix.\n";
m_Source.resize(Msize);
m_Solution.resize(Msize);
m_Stotal.resize (m_Source);

m_Mtotal.resize( Msize, Msize);
m_J.resize( Msize, Msize );

Log(2) << " * Creating " << Msize << "x" << Msize << " Maxwell matrix.\n";
CreateMaxwellMatrix();
Log(2) << "plEMDraka: initialisation finished.\n";

void plEMDrakaData::epsCalc()
{
    const double sig_to_epsJm = -1.0/(m_Omega*Constant::eps0);
    for (plFDGrid::positer piter = epsnp.fdgrid ().begin();
        piter != epsnp.fdgrid ().end(); ++piter)
    {
        const double eps_Re = epsr(*piter);
        const double eps_Im = sig.To_eps.Im * sig(*piter);
        epsnp(*piter) = plComplex( eps_Re, eps_Im);
    }
}

// deze werken dus niet helemaal goed!
inline void plEMDrakaData::epsInterpolate()
{
    CopyAndInterpolate( epsnp, epsew );
    CopyAndInterpolate( epsnp, epsns );
}

void plEMDrakaData::Plnterpolate()
{
    const double eps0.omega_2=0.5*Constant::eps0*m_Omega;
    double Ptest = 0.0;

    // First we calculate the dissipated power resulting from Er on EW grid.

for (plFDGrid::positer p0 = P_ew.fdgrid().begin();
p0 != P_ew.fdgrid().end(); ++p0)
{
    P_ew(*p0) = -norm(Er_ew(*p0))*epsE_omega0_2;
Ptest += P_ew(*p0) * P_ew(Grid())->volume(*p0);
}
// calculate the dissipated power resulting from Ez on the NS grid.
for (plFDGrid::positer p1 = P_ns.fdgrid().begin();
p1 != P_ns.fdgrid().end(); ++p1)
{
    P_ns(*p1) = -norm(Ez_ns(*p1))*epsN_omega0_2;
Ptest += P_ns(*p1) * P_ns(Grid())->volume(*p1);
}
Log(2) << "Ptest (NS/EW vol. P integration) = " << Ptest << std::endl;

// interpolate the dissipated power resulting from Er to the NP grid.
for (plFDGrid::positer p2 = P_ew.begin(); p2 != P_ew.end(); ++p2)
{
    P_ew_to_np(*p2) = plGridOps::ewToNodal(P_ew_to_np.fdgrid(),
                                            P_ew, *p2);
}
// interpolate the dissipated power resulting from Ez to the NP grid.
for (plFDGrid::positer p3 = P_ns.begin(); p3 != P_ns.end(); ++p3)
{
    P_ns_to_np(*p3) = plGridOps::nsToNodal(P_ns_to_np.fdgrid(),
                                            P_ns, *p3);
}

/// \todo for some reason this does not work, so done in another way.
// CopyAndInterpolate( P_ew, P_ew_to_np );
// CopyAndInterpolate( P_ns, P_ns_to_np );

// add terms to find the total dissipated power on a nodal point.
double Ptot=0.0;
for (plFDGrid::positer p4 = ohm.begin(); p4 != ohm.end(); ++p4)
{
    const double P_Ez = P_ns_to_np(*p4);
    const double P_Er = P_ew_to_np(*p4);
    ohm(*p4) = P_Er + P_Ez;
Ptot += ohm(*p4)*m_grid->volume(*p4);
}
// Total power dissipated in plasma (calculated in 2 ways to check).

// Calculate sum(B)
/// \skip left chokes modi to find slit B0
const plComplex Bsum = m_Solution(Hp0tot+Ertot+Eztot+Nmodi[slit-2]);
double PregionTot = 0.0;
int Nr_prev = 0;
for (int g = 1; g < rregion; ++g)
{
    double EsqrV = 0.0;
    // \todo Nr[0] goed gedef? anders moet dit naar achteren Nr[g];
    Nr_prev += Nr[g-1];
    for (int nr = 0; nr < Nr[g]; ++nr)
        for (int nz = 0; nz < Nztot; ++nz)
        {
            const double Rbottom = R[g-1] + nr*delta.r[g];
            const double Rtop = Rbottom + delta.r[g];
            const double dz_vol = m_grid->dx(plGrid::LocNP, plGrid::EastWest, Pos(1, Nrtot);
            const double vol = dz_vol * Constant::pi * (Rtop * Rtop - Rbottom * Rbottom);

            // this index runs over all CV’s (NP without 0-volumes)

            // E and W E-field
            const int NR=Hptot + (nr+Nr_prev)*(Nztot-1) + nz;
            plComplex Er.W;
            if (nz==0)
                Er.W = 0.0;
            else
                Er.W = m_Solution(NR-1);
            plComplex Er.E;
            if (nz==Nztot-1)
                Er.E = 0.0;
            else
                Er.E = m_Solution(NR);

            // N and S E-field
            const int NZ = Hptot + Ertot + (nr+Nr_prev)*Nztot + nz;
            const plComplex Ez.S = m_Solution(NZ);
            const plComplex Ez.N;
            if (nr == Nr[g]-1 && g == rregion-1)
            {
                // \todo NO HIGHER ORDER MODES? I ONLY SEE KO!
                const double kr = m.k0 * R[g];
                Ez.N = -m.bslit * (Bsum * Hankel2(0,kr)
                                 + m.A0 * Hankel1(0,kr));
            }
            else
            {
                Ez.N = m_Solution(NZ + Nztot);
            }
        } /* end inner loop */
    double Er2.avg = 0.5*(std::norm(Er.W)+std::norm(Er.E));
    double Ez2.avg = 0.5*(std::norm(Ez.N)+std::norm(Ez.S));
    EsqrV += (Er2.avg+Ez2.avg)*vol;
} /* end for */


```cpp
double Preg=-EsqrV*0.5*eps_out[g].imag()*Constant::eps0*m_Omega;
Log(2) << "P absorbed by region " << g << " is: " << Preg << std::endl;
PregionTot+=Preg;
Log(2) << "P absorbed by outer regions PregionTot = " << PregionTot << std::endl;

// These are NOT visualised.
Log(2) << "Ptot (NP vol. P integration) = " << Ptot << std::endl;

// And from the Poynting vector at the slit boundary.

double Bsquared = std::norm(Bsum);

// Power check
const double Pcheck = 2.0*(m_A0*m_A0-Bsquared)
                        / (Constant::mu0*m_Omega*m_bslit);
const double Pratio=(Ptot+PregionTot)/Pcheck;
Log(2) << "Power check from amplitudes A = " << m_A0
       << " and |B| = " << sqrt(Bsquared) << std::endl;
Log(2) << "gives Pcheck = " << Pcheck
       << " So Pcheck/Pin = " << Pcheck/m_Pin << std::endl;
Log(2) << "(Ptot+PregionTot)/Pcheck = " << Pratio << std::endl;
Log(2) << "Gives a deviation of " << 100*(1.0-Pratio) << "%\n";

// Adjust A0 for the next iteration, with relaxation.
m_relAO.Store(m_A0);

if (Pcal == 0.0 || Pcal < 0.0)
{
    /// \todo Add comment about the consequences of Bsquared==0
    /// div0 error!
    if (Bsquared == 0.0)
    {
        Bsquared = 1.0;
    }
    /// \bug This algorithm does not prevent A0sqr<0 if Pcal<0!
    const double A2=0.5*Constant::mu0*m_Omega*m_bslit*Pcal+Bsquared;
m_A0 = sqrt(A2);
    if (Pcal < 0.0)
        Log(3) << "Negative power!\n";
    else
        Log(3) << "div 0 protection for A activated!\n";
}
```
else
{
    m_A0 *= sqrt(m_Pin/Pcal);
}

m_relAO.Apply(m_A0);

Log(2) << "new A0 = " << m_A0 << std::endl;

unsigned plEMDrakaData::GetClosestPoint(const double z_value) const
{
    #ifndef MSCVRT
using std::fabs;
    #endif
    const plFDGrid & fdgrdew = (*m_grid)(plGrid::LocEW);

    unsigned z_index = 0;
    double old_res = z_value;

    for (unsigned i = 1; i < (unsigned)Nz_plasma; ++i)
    {
        const double res = z_value - fdgrdew.x(Pos(i,1));
        if (fabs(res) < fabs(old_res))
        {
            old_res = res;
            z_index = i;
        }
    }

    return z_index;
}

void plEMDrakaData::MetricsToPoints(int Nzpl)
{
    int z_pos[7+1];
    z_pos[0] = 0;
    z_pos[1] = GetClosestPoint( m_zchoke1 - 0.5 * m_dchoke1 );
    z_pos[2] = GetClosestPoint( m_zchoke1 + 0.5 * m_dchoke1 );
    z_pos[3] = GetClosestPoint( m_zslit  - 0.5 * m_bslit  );
    z_pos[4] = GetClosestPoint( m_zslit  + 0.5 * m_bslit  );
    z_pos[5] = GetClosestPoint( m_zchoke2 - 0.5 * m_dchoke2 );
    z_pos[6] = GetClosestPoint( m_zchoke2 + 0.5 * m_dchoke2 );
    //\ todo Check value of z_pos[7], get rid of comment
    z_pos[7] = Nzpl;// -2; // epsnp.n1(); // diganiegoe check -1

    // ew grid afst.
    for (int zind = 0; zind < 7; ++zind)
    {
        const plFDGrid & fdgrdew = (*m_grid)(plGrid::LocEW);
    }
Nz[zind] = z_pos[zind+1] - z_pos[zind];  
dz[zind] = fdgrdew.x(Pos(z_pos[zind+1],1)) -  
fdgrdew.x(Pos(z_pos[zind],1));  
if (Nz[zind] == 0 || dz[zind] == 0.0)  
{  
    Log(3) << "Z-region " << zind << " is of " << dz[zind]  
    << " length, stretch grid or change #x1 points!\n";  
}  
Log(2) << " * " << Nz[zind] << " points of length " << dz[zind]  
<< " between " << z_pos[zind] << " and "  
<< z_pos[zind+1] << std::endl;  

m_bslit = dz[slit];  
m_dchoke1 = dz[slit-2];  
m_dchoke2 = dz[slit+2];  

/// 	odo Elaborate on comment below  
/// these are not used anywhere so no need to calculate them too  
/// m_zchoke1 =  
/// m_zchoke2 =  
/// m_zslit =

inline void plEMDrakaData::FillM(int Mrow, int Mcol,  
const plComplex& Mfillval)  
{  
#if BDMAT_ON  
    m_M.autoinsert(Mfillval,Mrow,Mcol);  
#else  
    m_M(Mrow,Mcol) = Mfillval;  
#endif  
}  

inline void plEMDrakaData::FillMtotal(int Mrow, int Mcol,  
const plComplex& Mfillval)  
{  
#if BDMAT_ON  
    m_Mtotal.autoinsert(Mfillval,Mrow,Mcol);  
#else  
    m_Mtotal(Mrow,Mcol) = Mfillval;  
#endif  
}  

void plEMDrakaData::FillM11(int koffset, int loffset)  
{  
    /// 	odo Elaborate on comment below this line  
    /// normering maakt er een min van!  
    for (int i = 0; i < Hptot; ++i)  
    {  
        
}  
}
// First the elements in the plasma region are added.

// First the derivatives on the west boundaries (since Er = 0 there)
// are generated.
for (int g = 0; g < rregion; ++g)
{
    int i = 0, j;
    double dz = grid->dx(plGrid::LocNP, plGrid::EastWest, Pos(i+1, 0));
    double C0 = 1.0/(dz * m.k0);
    for (j = 0; j < Nrtot; ++j)
    {
        const int row_offset = Nztot * j;
        const int column_offset = (Nztot-1) * j;
        const int k = i + koffset + row_offset;
        const int l = i + loffset + column_offset;
        FillM(k, l, CO);
        FillM(k, l-1, -CO);
    }
}

// Now the inner region where we use the normal derivatives.
for (i = 1; i < Nztot-1; ++i)
{
    dz = grid->dx(plGrid::LocNP, plGrid::EastWest, Pos(i+1, 0));
    C0 = 1.0/(dz * m.k0);
    for (j = 0; j < Nrtot; ++j)
    {
        const int row_offset = Nztot * j;
        const int column_offset = (Nztot-1) * j;
        const int k = i + koffset + row_offset;
        const int l = i + loffset + column_offset;
        FillM(k, l, C0);
        FillM(k, l-1, -C0);
    }
}

// And the east boundary, where we take Er = 0 again.
i = Nztot-1;
dz = grid->dx(plGrid::LocNP, plGrid::EastWest, Pos(i+1, 0));
C0 = 1.0/(dz * m.k0);
for (j = 0; j < Nrtot; ++j)
{
    const int row_offset = Nztot * j;
    const int column_offset = (Nztot-1) * j;
    const int k = i + koffset + row_offset;
    FillM(k, i, C0);
}
const int l = i + loffset + column_offset;
FillM(k, l-1, -C0);
}

void plEMDrakaData::FillM13(int koffset, int loffset)
{
    int Ngtot = 0;
    const int lmax = loffset + Eztot;

    for (int g = 0; g < rregion; ++g)
    {
        for (int j = 0; j < Nr[g]; ++j)
        {
            // todo klopt dit?
            const double dr = (g == 0)
                ? m_grid->dx(plGrid::LocNP, plGrid::NorthSouth, Pos(O, j+1))
                : delta_r[g];
            const double C0 = 1.0 / (m.kO*dr);
            for (int i = 0; i < Nztot; ++i)
            {
                // CSE should optimize this
                const int k = i + koffset + j*Nztot + Ngtot;
                int l = i + loffset + j*Nztot + Ngtot;
                FillM(k, l, C0);
                // To get the derivative in the r direction one
                // needs to skip Nz points on the same line.
                l += Nztot;
                if (!pl likely (1 < lmax))
                {
                    FillM(k, l, -C0);
                }
            }
        }
    }
    Ngtot += Nr[g] * Nztot;
}

void plEMDrakaData::FillM14(int koffset, int loffset)
{
    int row_offset = Hptot - Nztot + Nz[0];
    int column_offset = 0;

    for (int g = 1; g < zregion; g += 2)
    {
        const plFDGrid & fdgrdew = (*m_grid)(plGrid::LocEW);
        const double zleft = fdgrdew.x(Pos(row_offset-Hptot+Nztot, 1));
        for (int i = 0; i < Nz[g]; ++i)
        {
            // CSE should optimize this
            const int k = i + koffset + j*Nztot + Ngtot;
            int l = i + loffset + j*Nztot + Ngtot;
            FillM(k, l, C0);
            // To get the derivative in the r direction one
            // needs to skip Nz points on the same line.
            l += Nztot;
            if (!pl likely (1 < lmax))
            {
                FillM(k, l, -C0);
            }
        }
    }
    Ngtot += Nz[g] * Nztot;
}
const plFDGrid & fgdrdnp = (*m_grid)(plGrid::LocNP);
double C0 = 1.0/(delta_r[region-1] * dz[g] * m_k0);
// z en zleft ff checken.
const double z = fgdrdnp.x(Pos(row_offset-Hptot+Nztot+i+1, 1)) - zleft;
for (int j = 0; j < Nmodi[g]; ++j)
{
    // Nmodi = Nz[g]
    const double kappapart=j*Constant::pildz[g];
    const double kappasqr=m_k0*m_k0-kappapart*kappapart;
    // make a 'complex root function'
    const plComplex kappa = kappafunc(kappasqr);
    Log(2) << "kappa\^2 = " << kappasqr
        << "", kappa = " << kappa << std::endl;
    const plComplex krn = kappa*R[region-1];
    const plComplex H02 = Hankel2(0,krn);
    const plComplex H1 = Hankel1(0,krn);

    plComplex C1;
    if (g == slit)
    {
        C1 = 0;
    }
    else
    {
        const plComplex kR = kappa*Rchoke[g];
        const plComplex H2 = Hankel2(0,kR);
        const plComplex H3 = Hankel1(0,kR);
        C1 = (H1 / H3) * H2;
        // C1 = 0; // bij hogere orde
    }
    const plComplex BCondVal =
        (C0 * cos(j*Constant::pi*zfdz[g]) * (H02 - C1));
    FillM( koffset+i+row_offset,
           loffset+column_offset+j,
           BCondVal);
    if (j == 0)
    {
        if (g == slit)
        {
            // m_Source still has to be
            // multiplicated with A0
            const plComplex slitval=-C0*H1;
            m_Source(koffset+i+row_offset) = slitval;
            // \todo chk if indices on slit
        }
        // \todo correct? this is X (big chi).
        C0 *= 2;
    }
}

101
column_offset += Nmodi[g];
row_offset += Nz[g+1] + Nz[g];
}
// hankelfunkties opnieuw bekijken voor de zekerheid.
}

void plEMDrakaData::FillM21(int koffset, int loffset)
{
    for (int i = 0; i < Nztot-1; ++i)
    {
        // We assume a non poisson stretched grid.
        const double dz = m_grid->dx(plGrid::LocEW, plGrid::EastWest,
                                    Pos(i+1,1));
        const double C0 = 1.0/(m_k0*dz);
        for (int j = 0; j < Nrtot; ++j)
        {
            const int k = i + koffset + j*(Nztot - 1);
            const int l = i + loffset + j*(Nztot);
            FillM(k, l, C0);
            FillM(k, l+1, -C0);
        }
    }
}

void plEMDrakaData::FillM22(int koffset, int loffset)
{
    // Fill in values for permittivity in plasma region is
    // not done in construction.

    // Fill permittivity values for areas outside of plasma in M11.
    int i2 = Ertot_plasma;
    // Start at region 1, just outside plasma.
    for (int g = 1; g < rregion; ++g)
    {
        // Number of Hp points of current region g.
        const int imax = Nr[g] * Nztot;
        for (int i = 0; i < imax; ++i)
        {
            // Set the coefficient to the permittivity that is
            // valid for this region.
            const int k = i + i2 + koffset;
            const int l = i + i2 + loffset;
            FillM(k, l, eps_out[g]);
        }
        i2 += imax; // Start where previous region ended.
    }
}

void plEMDrakaData::FillM31(int koffset, int loffset)
\{ 
const plFDGrid & fdgrdnp = (*m_grid)(plGrid::LocNP);
const plFDGrid & fdgrdns = (*m_grid)(plGrid::LocNS);

// Plasma area.

// We start with the equations for Ez on the south wall, a special
// boundary condition is used here.

double dr = fdgrdnp.y(Pos(1,1));
double C0 = 2.0 / (m.k0 * dr); // niet negatief door normering!!
double C1, C2;
int i; // MSVC announce
for (i = 0; i < Nztot; ++i) {
    FillM(i+koffset, i+loffset, C0);
}

// Now the internal points are taken inside the plasma, the points
// at the north wall are not inside this iteration (they will be treated
// seperately later on).

for (int j = 1; j < Nr[0]; ++j) {
    dr = m_grid->dx(plGrid::LocNS, plGrid::NorthSouth, Pos(1, j));
    C0 = 1.0 / (m.k0*dr);
    C1 = C0 * fdgrdnp.y(Pos(1,j+1)) / fdgrdns.y(Pos(1,j));
    C2 = -C0 * fdgrdnp.y(Pos(1,j)) / fdgrdns.y(Pos(1,j));
    for (i = 0; i < Nztot; ++i) {
        const int l = loffset + i + j * Nztot;
        const int k = koffset + 1;
        FillM(k, 1, C1); // not negative, norming!
        FillM(k, 1-Nztot, C2); // negative, norming!
    }
}

// Now the boundaries between plasma and other regions are done.

dr = (0.5 * delta_r[1] + R[0] - fdgrdnp.y(Pos(1,Nr[0]))) ;
C0 = 1.0 / (m.k0 * dr * R[0]);
C1 = C0 * ( R[0] + 0.5 * delta_r[1] ); // not / !!!
C2 = -C0 * ( fdgrdnp.y(Pos(1,Nr[0])) );
for (i = 0; i < Nztot; ++i) {
    const int l = loffset + Nr[0] * Nztot + i;
    const int k = koffset + 1;
    FillM(k, 1, C1);
    FillM(k, 1-Nztot, C2);
}

int l = loffset + Nr[0] * Nztot + Nztot - 1;
// And finally all the other regions (in r-direction).
for (int g = 1; g < rregion; ++g) 
\}
{ for (int j = 1; j < Nr[g]; ++j)
{
    C0 = 1.0 / ( (R[g-1] + j*delta_r[g]) * m_k * delta_r[g] );
    C1 = ( R[g-1] + (j+0.5)*delta_r[g] ) * C0;
    C2 = - ( R[g-1] + (j-0.5)*delta_r[g] ) * C0;
    for (i = 0; i < Nztot; ++i)
    {
        const int k = koffset + ++l;
        FillM(k, 1, Cl);
        FillM(k, 1-Nztot, C2);
    }
}
}

void pLEMDrakaData::FillM33(int koffset, int loffset)
{
    // Fill permittivity values for areas outside of plasma in M33.
    // The plasma area and the boundary between plasma and first
    // area are NOT filled here, but are in the update routine.
    int i2 = Eztot_plasma;

    // Fill internal values in regions.
    for (int g = 1; g < rregion; ++g)
    {
        // Number of Ez points of current region g.
        //\todo Elaborate on comment below
        // als (Nr[g]-1) bij i=0 ipv Nztot
        const int imax = Nr[g] * Nztot;
        for (int i = Nztot; i < imax; ++i)
        {
            // Set the coefficient to the permittivity that is
// valid for this region.
const int k = i + i2 + koffset;
const int l = i + i2 + loffset;
FillM(k, l, eps..out[g]);
}

// Start where previous region ended:
i2 += Nr[g] * Nztot;

// Fill the boundaries between 2 regions, except between the
// plasma and the first region.
// \todo CHECK of deze indices ook goed zijn.
if ( rregion > 1 ) {
i2 = Eztot_plasma;
for ( int g = 1; g < rregion-1; ++g ) {
    // Q = (R-Rs)/(Rn-Rs) = delta_rS / (delta_rN + delta_rS)
    // Q = 0.5*delta_r[g]/(0.5*delta_r[g]+0.5*delta_r[g+1])
    const double Q = delta_r[g]/(delta_r[g]+delta_r[g+1]);
    const p!Complex eps..average = Q * eps..out[g]
    + (1.0-Q) * eps..out[g+1];
i2 += Nztot * Nr[g];
for ( int j = 0; j < Nztot; ++j ) {
    FillM(j+i2+koffset, j+i2+loffset, eps..average);
}
}

void plEMDrakaData::FillM41(int koffset, int loffset )
{
    // This term is for the chokes and slit.
    int k=0;
    int l=Hptot - Nztot + Nz[0];
    const double C0 = (1.0/(m.k0*R[rregion-1]))-(2.0/(m.k0*delta_r[rregion-1]));
    for ( int g = 1; g < rregion; g += 2 ) {
        for ( int j = 0; j < Nz[g]; ++j ) {
            // k + koffset te groot?
            FillM(k+koffset, l+j+loffset, C0);
k++;
        }
        l += Nz[g] + Nz[g+1];
    }
}

void plEMDrakaData::FillM44(int koffset, int loffset )
{
int k = 0, k2 = 0, l = 0;
const plFDGrid & fdgrdew = (*m_grid)(plGrid::LocEW);
const plFDGrid & fdgrdnp = (*m_grid)(plGrid::LocNP);

for (int g = 1; g < zregion; g += 2)
{
    k2 += Nz[g-1];
    const double invdz = 1.0/dz[g]; // dz mag niet 0 zijn!
    for (int n = 0; n < Nmodi[g]; ++n)
    {
        // Nmodi = Nz[g]
        const double C5 = n*Constant::pi * invdz;
        // kappa kan ook complex zijn
        const plComplex kappa = kappafunc(m.k0*m.k0 - C5*C5);
        Log(2) << "kappa'2 = " << (m.k0*m.k0 - C5*C5) << std::endl;
        const plComplex arg1 = kappa*R[region-1];
        plComplex C3;
        if (g == slit)
        {
            C3 = 0;
        }
        else
        {
            const plComplex arg2 = kappa*Rchoke[g];
            C3 = Hankel2(0,arg2)/Hankel1(0,arg2);
        }
        plComplex H1.0 = Hankel1(0, arg1);
        plComplex H1.1 = Hankel1(1, arg1);
        plComplex H2.0 = Hankel2(0, arg1);
        plComplex H2.1 = Hankel2(1, arg1);
        // plComplex C0 = H2.0 - C3*H1.0;
        // plComplex C1 = H2.1 - C3*H1.1;
        plComplex C2=2.0*kappa*invdz/(delta_r[region-1]*m.k0*m.k0);
        plComplex C4=eps_out[region-1]*invdz;
        Log(2) << std::endl;
        if (n > 0)
        {
            C2 *= 2.0;
            C4 *= 2.0;
        }
    }
    const double zleft = fdgrdew.x(Pos(k2,1));
    for (int j = 0; j < Nz[g]; ++j)
    {
        const double z = fdgrdnp.x(Pos(k2+j+1,1));
    }
}
void plEMDrakaData::CreateMaxwellMatrix()
{
    FillM11(0, 0);
    FillM12(0, Hptot);
    FillM13(0, Hptot + Ertot);
    FillM14(0, Hptot + Ertot + Eztot);

    FillM21(Hptot, 0);
    FillM22(Hptot, Hptot);

    FillM31(Hptot + Ertot, 0);
    FillM33(Hptot + Ertot, Hptot + Ertot);

    FillM41(Hptot + Ertot + Eztot, 0);
    FillM44(Hptot + Ertot + Eztot, Hptot + Ertot + Eztot);
}

void plEMDrakaData::AdjustMaxwellMatrix()
{
    int zpos, rpos;
    // fill epsilon values for Er (EW Grid) in M22
    for (rpos = 0; rpos < Nr_plasrna; ++rpos)
        for (zpos = 0; zpos < Nz_plasrna-1; ++zpos)
        {
            const int offset1 = Hptot + zpos + rpos * (Nz_plasrna-1);
            FillMtotal(offset1, offset1, epsew(Pos(zpos+1,rpos+1)));
        }
// fill epsilon values for Ez (NS Grid) except north boundary in M33
for (rpos = 0; rpos < Nr_plasma; ++rpos)
for (zpos = 0; zpos < Nz_plasma; ++zpos)
{
    const int offset2 = Hptot + Ertot + zpos + rpos * Nz_plasma;
    FillMtotal(offset2, offset2, epsns(Pos(zpos+1, rpos)));
}

// fill epsilon values for Ez (NS Grid) on the north boundary in M33
const plFDGrid & fdgrdnp = (*m_grid)(plGrid::LocNP);

// delta_rS is the distance between the northernmost nodal point
// (besides the 0-volume of course) and the north edge of the plasma
const double delta_rS = R[0] - fdgrdnp.y(Pos(1, Nr[0]));

// Q = (R-Rs)/(Rn-Rs) = delta_rS/(delta_rS + delta_Rn)
// 1-Q = (Rn-R)/ (Rn-Rs)
// eps_avg = Q * eps_plasma + (1-Q) * eps_glass
const double Q = delta_rS / (delta_rS + 0.5 * delta_r[1]);

for (zpos = 0; zpos < Nz_plasma; ++zpos)
{
    const int offset3 = Hptot + Ertot + zpos + Nr_plasma * Nz_plasma;
    const plComplex eps_average = Q * epsnp(Pos(zpos+1, Nr[0]))
    + (1.0-Q) * eps_out[1];
    FillMtotal(offset3, offset3, eps_average);
}

// Updating source terms with changed A0.
for (int i = 0; i < Msize; ++i)
{
    m_Stotal(i) = m_A0 * m_Source(i);
}

///\todo Why take the negative imaginary number for negative arguments?
inline plComplex plEMDrakaData::kappafunc(double kappasqr) const
{
    return (kappasqr < 0.0)
        ? plComplex(0, -sqrt(-kappasqr))
        : sqrt(kappasqr);
}

void plEMDrakaData::CalculateFields()
{
    // Calculate complex relative permittivity on NP grid
    // from v.eh, omega and omega.pl.
    epsCalc();

    // Interpolate permittivity from NP grid to EW and NS grids.
epsInterpolate();

// Copy all other Matrix elements from initially constructed m.M.
m_Mtotal = m_M;
m_Stotal = m_Source;

// Change eps in Matrix elements M22 and M33.
AdjustMaxwellMatrix();

cerr << "Dumping matrices.\n";
DumpMatrix(std::cerr);
WriteFile();

// Solve Matrix equation
m_Solution = TBCI::lu_solve(m_Mtotal, m_Stotal);

// Write Er, Ez, Hphi to plGridVar
WriteSolutionFields();
// DumpFields(std::cerr);

// Interpolate Power from NS and EW grid and add them on NP grid.
// Write power to NP Grid and check validity of total power balance,
// change A accordingly for next iteration.
Plnterpolate();

void plEMDrakaData::WriteSolutionFields()
{
    int ix, iy;
    int offset = Hptot;
    int counter = 0;
    // Fill Er_ew values, internally.
    for (iy = 1; iy <= Nr_plasma; ++iy)
        for (ix = 1; ix < Nz_plasma; ++ix)
        {
            Er_ew(Pos(ix,iy)) = m_Solution(offset + counter);
            counter++;
        }
    // Fill Er_ew values on east and west side, including corners.
    for (iy = 0; iy <= Nr_plasma+1; ++iy)
    {
        Er_ew(Pos(0, iy)) = 0.0;
        Er_ew(Pos(Nz_plasma, iy)) = 0.0;
    }
    // Fill Er_ew values on south and north boundary (except corners).
    // Set to same value as the point next to it, since volume = 0 anyways.
    for (ix = 1; ix < Nz_plasma; ++ix)
    {
        Er_ew(Pos(ix, 0)) = Er_ew(Pos(ix, 1));
        Er_ew(Pos(ix, Nr_plasma+1)) = Er_ew(Pos(ix, Nr_plasma));
    }
}
offset = Hptot + Ertot;
counter = 0;
// Fill Ez.ns values, internally
for (iy = 0; iy <= Nr_plasma; ++iy)
for (ix = 1; ix <= Nz_plasma; ++ix)
{
    Ez.ns(Pos(ix,iy)) = m_Solution(offset + counter);
    counter++;
}

// Fill Ez.ns values on east and west side
// Set to same value as the point next to it, since volume = 0 anyways.
for (iy = 0; iy <= Nr_plasma; ++iy)
{
    Ez.ns(Pos(0, iy)) = Ez.ns(Pos(1, iy));
    Ez.ns(Pos(Nz_plasma + 1, iy)) = Ez.ns(Pos(Nz_plasma, iy));
}

void plEMDrakaData::DumpFields(std::ostream& os) const
{
    int i, j, solutionindex;
    // Check for zero-rows, unconvranging solution then.
    for (i = 0; i < Msize; ++i)
    {
        double tempadd = 0.0;
        for (j = 0; j < Msize; ++j)
        {
            tempadd += std::norm(m_Mtotal(i,j));
        }
        if ( tempadd == 0.0 )
            os << "ZERO ROW AT " << i << endl;
    }

    // Check for solutions, variables are non zero then.
    os << "Solution not zero at: \n";
    for (i = 0; i < Msize; ++i)
    {
        const double tempvar = std::norm(m_Solution(i));
        if ( tempvar > 0.0 )
            os << i << ", ";
    }
    os << std::endl;

    os << "\nH_phi \n";
    for (j = Nrtot-1; j >= 0; --j)
    {
        for (i = 0; i < Nztot; ++i)
            os << std::endl;
    }
{  
solutionindex = i + j*Nztot;
  os << m.Solution(solutionindex);
}
os << std::endl;
}
os << "\nE.r\n";

for (j = Nrtot-1; j >= 0; --j)
{
  for (i = 0; i < Nztot; ++i)
  {
    solutionindex = Hptot + i + j*(Nztot-1);
    os << m.Solution(solutionindex);
  }
os << std::endl;
}
os << "\nE..z\n";

for (j = Nrtot-1; j >= 0; --j)
{
  for (i = 0; i < Nztot; ++i)
  {
    solutionindex = Hptot + Ertot + i + j*Nztot;
    os << m.Solution(solutionindex);
  }
os << std::endl;
}

/// \todo Add functionality of remove empty for loop
for (i = 0; i <= Nztot; ++i)
{
  // if g == slit dan raar (save in ander deel?)
  // if g == choke ook raar
  // if g == wand, dan 0
}
os << "\nA and B dump\n";

// skip left chokes modi to find slit B0
const pComplex Bsum = m.Solution(Hptot+Ertot+Eztot+Nmodi[slit-2]);
const double BdivA = std::abs(Bsum)/m.A0;
os << "A = " << m.A0 << std::endl;
os << "|B|/A = " << BdivA << " (should be <= 1.0)\n";
}

void plEMDrakaData::DumpMatrixPart(std::ostream& os,
const int ileft ,
const int jleft ,
const int nrows,
const int ncolumns) const
{
    for (int i = 0; i < nrows; ++i)
    {
        for (int j = 0; j < ncolumns; ++j)
        {
            os << m_Mtotal(i+ileft, j+jleft);
        }
        os << std::endl;
    }
}

void plEMDrakaData::DumpSourcePart(std::ostream& os,
    const int ileft,
    const int nrows) const
{
    // Dumped horizontally to give more space
    for (int i = 0; i < nrows; ++i)
    {
        os << m_Source(i+ileft);
    }
    os << std::endl;
}

void plEMDrakaData::DumpMatrix(std::ostream& os) const
{
    os << "Starting DumpMatrix\n";
    os << "Dumping M11\n";
    DumpMatrixPart(os, 0, 0, Hptot, Hptot);
    os << "\nDumping M12\n";
    DumpMatrixPart(os, 0, Hptot, Hptot, Ertot);
    os << "\nDumping M13\n";
    DumpMatrixPart(os, 0, Hptot+Ertot, Hptot, Eztot);
    os << "\nDumping M14\n";
    DumpMatrixPart(os, 0, Hptot+Ertot+Eztot, Hptot, Btot);
    os << "\nDumping source terms for S1^T\n";
    DumpSourcePart(os, Hptot);
    os << "\nDumping M21\n";
    DumpMatrixPart(os, Hptot, 0, Ertot, Hptot);
    os << "\nDumping M22\n";
    DumpMatrixPart(os, Hptot, Hptot, Ertot, Ertot);
    os << "\nDumping M23 (should be 0)\n";
    DumpMatrixPart(os, Hptot, Hptot+Ertot, Ertot, Eztot);
    os << "\nDumping M24 (should be 0)\n";
    DumpMatrixPart(os, Hptot, Hptot+Ertot+Eztot, Ertot, Btot);
    os << "\nDumping source terms for S2^T\n";
    DumpSourcePart(os, Hptot, Ertot);
    os << "\nDumping M31\n";
}
DumpMatrixPart (os, Hptot+Ertot, 0, Eztot, Hptot);
    os << "\nDumping M32 (should be 0)\n";
DumpMatrixPart (os, Hptot+Ertot, Hptot, Eztot, Ertot);
    os << "\nDumping M33\n";
DumpMatrixPart (os, Hptot+Ertot, Hptot+Ertot, Eztot, Eztot);
    os << "\nDumping M34 (should be 0)\n";
DumpMatrixPart (os, Hptot+Ertot, Hptot+Ertot+Eztot, Eztot, Btot);
    os << "\nDumping source terms for S3'T\n";
DumpSourcePart (os, Hptot+Ertot, Eztot);

    os << "\nDumping M41\n";
DumpMatrixPart (os, Hptot+Ertot+Eztot, 0, Btot, Hptot);
    os << "\nDumping M42 (should be 0)\n";
DumpMatrixPart (os, Hptot+Ertot+Eztot, Hptot, Btot, Ertot);
    os << "\nDumping M43 (should be 0)\n";
DumpMatrixPart (os, Hptot+Ertot+Eztot, Hptot+Ertot+Eztot, Btot, Eztot);
    os << "\nDumping M44\n";
DumpMatrixPart (os, Hptot+Ertot+Eztot, Hptot+Ertot+Eztot+Eztot, Btot, Btot);
    os << "\nDumping source terms for S4'T\n";
DumpSourcePart (os, Hptot+Ertot+Eztot, Btot);

    os << "\nEnd of DumpMatrix\n";
}

void plEMDrakaData::WriteFile()
{
    std :: ofstream mat_file ("dump matrix. txt");
    mat_file << m_Mtotal;

    std :: ofstream src_file ("dumpsource. txt");
    src_file << m_Stotal;
}

class plEMDraka : public plBaseEMProxy<plEMDrakaData>
{
public:
    plEMDraka( plRegionManager *manager, const plNode & emnode )
    : plBaseEMProxy<plEMDrakaData>( manager, emnode )
    {
        Log(2) << "In plEMDraka's constructor" << std::endl;

        // first iteration the EM has to be calculated!
        em_drakacounter = 10;
    }

    virtual void DoIteration(double eps)
    {
        if ( m_emRegs.size() == 0)
        {
            throw plException("The Draka EM module requires "
"a plasma region.");
        }
    }
}
if (em.drakacounter >= 10)
{
    m_emRegs[0]->CalculateFields();
    em.drakacounter = 0;
}
else ++em.drakacounter;
}

protected:
virtual plEMDrakaData* CreatePlasma(plModelRegion* region,
const plNode& emnode)
{
// we have one already?
if (m_emRegs.size() != 0)
{
    throw plException("The Draka EM module supports only
" one plasma region.");
}
return new plEMDrakaData( region, emnode);
}

private:
int em.drakacounter;
};

REGISTER_PROVIDER( plBaseEM, plEMDraka, "EMDraka");

C.3 Surfatron EM Plugin

#include "plgeneric/forcelnk.h"
#include "plem/base-em.h"
#include "plparser/log.h"
#include "plmathlconsts.h"
#include "plmath/plasmath.h"
#include "plgrid/xyplotgvar.h"

FORCE_LINK_ME(surfak.skin);

class plSurfaSkinData : public plBaseEMData
{
public:
    plSurfaSkinData( plModelRegion* reg, const plNode & emnode )
    : plBaseEMData( reg, emnode)
    {
        m_skin_depth = 0;
        if (reg->Grid()->coord_type != cylindrical) {
            Log(4) << "SurfaSkin: wrong grid type. Need cylindrical." << end;
            throw plException("SurfaSkin: need a cylindrical grid!");
        }
        m_P = emnode("Power")["W"];
        m_f = emnode("Frequency")["Hz"];
m_alpha = emnode("Alpha")['"']; // percentage of power flow that is coupled in m_omega = m_f * 2.0 * Constant::pi;
const plGrid *g = reg->Grid();

// volume
m_V = Constant::pi * g->m_np.x1(g->n1()-1) *
        (sqrt(g->m_np.x2(g->n1()-1)) - sqrt(g->m_np.x2(0)));
}

plSurfaSkinData() {
}

void CalculateFields() {
    const plGrid *g = ohm.Grid();
    REAL damp...sig = 0.0;
    REAL halLatt = 0.0;
    REAL POhm = 0.0;
    int i, j;
    // attenuation of E field
    TBCI::Vector<REAL> att;
    att. resize (g->n2());

    for (h = 1; h < g->n1(); h++) {
        REAL dz = g->m_cv.x1(i) - g->m_cv.x1(i-1);
        REAL Power;
        if (m_p > POhm) Power = (m_p - POhm) else Power = 0.0; // ; na )?

        for (i = g->n2(); i>0; --i) {
            REAL r = g->m_np.x2(i);
            REAL dr = g->m_cv.x2(i) - g->m_cv.x2(i-1);

            // volume of this r-shell
            REAL dV = 2.0 * Constant::pi * r * dr * dz;
            // current attenuation based on previous one
            if (i == (int)g->n1()-1) // at outside, start with 1
                att(i) = 1.0;
            else // take previous value, and add second half step
                att(i) = att(i+1) * halLatt;

            // calculate new attenuation for half step
            REAL local.delta;
            local.delta = Delta(sig(h,i));
            half.att = exp(-(0.5*dr)/local.delta);

            // add first half step
            att(i) *= half.att;

            // volume averaged damping
            damp...sig += sig(h,i) * sqrt(att(i)) * dV;
        }
    }
}

} // TBCI::Vector<REAL> E.wall;
E_wall.resize(g->n2());
REAL E_wall = sqrt( 2*(m_.alpha * Power)/damp_sig);

for (i = g->n2()-1; i>=0; --i) {
  REAL r = g->m_n.p.x2(i);
  REAL E_field = E_wall * att(i);
  REAL dissipation = 0.5 * sig(h,i) * sqrt(E_field);
  REAL dr = g->m_cv.x1(i) - g->m_cv.x1(i-1);
  REAL dV = 2.0 * Constant::pi * r * dr * dz;
  POhm += dissipation * dV;
  m_Eimposed1(h,i) = E_field;
  ohm(h,i) = dissipation;
}

Log(2) << "SurfaSkin: total cylinder power in = " << POhm << " W" << std::endl;
}

private:
// complete delta function, with DC value in low frequency, high
// sigma limit, and constant value in high frequency, low sigma limit.
REAL Delta(REAL sigma) {
  REAL k = m_.omega/Constant::LightSpeed;
  REAL f = sqrt(1.0+sqr(sigma/(m_.omega*Constant::eps0)));
  if (f <= 1.0) // very small sigma
    return 1e6; // 1000 km is enough
  return 1.0/(k*sqrt((f-1.0)/2.0));
}

REAL m.P;
REAL m.f;
REAL m.V;
REAL m.omega;
REAL m.alpha;

};

class plSurfaSkin: public plBaseEMProxy<plEMSkinData>
{
public:
  plSurfaSkin( plRegionManager* manager, const plNode & emnode )
    : plBaseEMProxy<plEMSkinData>( manager, emnode )
  {
    Log(2) << "In plEMSkin's constructor" << endl;
  }

  // Calculates the fields by merely calling
  * plEMUniformACData::CalculateFields() for each region.
  * (Note that these regions are not coupled electromagnetically.)
  */
  virtual void DoIteration( REAL eps )
  {
    Log(2) << "In plEMSkin::DoIteration" << endl;
 116
for ( unsigned ndx=0; ndx < m_emRegs.size(); ++ndx)
{
    m_emRegs[ndx]->CalculateFields();
}

REGISTER_PROVIDER( plBaseEM, plSurfaSkin, "SurfaSkin");
Appendix D

Conductivity Code

/** \file
 * Implements the calculators for epsr, the real part of the relative
 * complex permittivity and sigma, the relative complex permittivity for
 * a microwave plasma like that of the POF plasma.
 * \author Michiel van den Donker, Jan van Dijk.
 * \date October 2001
 */

#include "plconfig.h"
#include "plmath/plasmath.h"
#include "plparser/node.h"
#include "plspecies/transcoef.h"
#include "plmath/consts.h"
#include "plgeneric/accessor.h"
#include "plgeneric/forceink.h"

FORCE.LINK.ME(epsrsigmw);

/** This class is as a base class for calculators for the relative permittivity
 * and conductivity in high-frequency plasmas. The conductivity has a relation
 * with the imaginary part of the permittivity according to equations on
 * page 138 of the Thesis of G.M.Janssen.
 */
class plBasePermittivity : public plTransCalculator<REAL>
{
 private:
  /** The angular frequency of the MW. Note that this value
   * should be equal to the actual EM field frequency.
   */
  REAL m_omega;

 protected:
  /// returns the angular field frequency.
  REAL Omega() const { return m_omega; }
  /** Calculates the energy-averaged cross section for e-h
   */

  // Other code...

  // More code...

  // Further code...

  // Additional code...

  // Conclusion...

  // Final notes...

  // End of document...

};

118
REAL CalcEHFrequency(const plCrossSectionMatrix & cs,
                      const plConstValueRef<REAL> & point);

REAL PlasmaFrequency(REAL n_e);

public:

REAL plBasePermittivity( const plNode& node, const plParticleMap& pmap,
                         const plBaseRelationMap& rmap);

CALCULATE THE ANGULAR FREQUENCY \( m_\omega \) FROM THE FREQUENCY OF THE
MICROWAVE. \( m_\omega = \text{frequency} \times 2 \pi \)

plBasePermittivity::plBasePermittivity( const plNode& node,
                                       const plParticleMap& pmap,
                                       const plBaseRelationMap& rmap)

: plTransCalculator<REAL> (node, pmap, rmap)

    \{ 
        m_\omega = node("Frequency")["Hz"] * 2.0 * Constant::pi;
    
    \}

REAL plBasePermittivity::CalcEHFrequency(const plCrossSectionMatrix & cs,
                                         const plConstValueRef<REAL> & point)

\{ 

REAL n_sig = 0.0;
unsigned elNdx = Particles().GetElectronNdx();
for (unsigned i=0; i<Particles().size(); ++i)
\{ 
    // skip e-e interaction
    if (i == elNdx)
        \{ 
            continue;
        \}
const REAL n_h = plAccessor::Density(point)[i];
n_sig += n_h*cs(elNdx, i);
\}

momentum transfer.
*/

REAL CalcEHFrequency(const plCrossSectionMatrix & cs,
                      const plConstValueRef<REAL> & point);

/** Calculates the plasma frequency as a function of the electron
 * density \( n_e \)
 */

REAL PlasmaFrequency(REAL n_e);

public:

/** Calls the base class constructor, then initialises the
 * member omega on the basis of \( \text{a node.} \)
 */

plBasePermittivity( const plNode& node, const plParticleMap& pmap,
                    const plBaseRelationMap& rmap);

/** The sum \( n_{\text{sig}} \) of heavy particle density times the cross section is
 calculated by iterating over all particles except the electron itself.
 Then the velocity \( \text{vel} = \text{sqrt} \{ \text{8 k_B T_e} \} \{ \pi m_e \} \) \( \text{is}
 calculated.
 Now \( n_{\text{sig}} \text{vel} \) gives the frequency of electron-heavy particle
 collisions .
 */

REAL plBasePermittivity::CalcEHFrequency(const plCrossSectionMatrix & cs,
                                         const plConstValueRef<REAL> & point)

\{ 

REAL n_sig = 0.0;
unsigned elNdx = Particles().GetElectronNdx();
for (unsigned i=0; i<Particles().size(); ++i)
\{ 
    // skip e-e interaction
    if (i == elNdx)
        \{ 
            continue;
        \}
const REAL n_h = plAccessor::Density(point)[i];
n_sig += n_h*cs(elNdx, i);
\}
const REAL Te = plAccessor::Temperature(point)[elNdx];
// calculate the electron velocity
const REAL vel = sqrt((8.0 * Constant::Boltzmann * Te) /
( Constant::pi * Particles()[elNdx]->Mass()));
// return the frequency of e-h collisions, <v> * <n_.h * crosssection>
return vel * n_.sig;

/** Plasma frequency is calculated in this member by
 * \( \omega_{pl} = \frac{\sqrt{n_e \cdot \text{e}^2}}{\epsilon_0 m_e} \)
 */
REAL plBasePermittivity::PlasmaFrequency(REAL n_e)
{
    return sqrt(n_e * sqr(Constant::ElementaryCharge) /
( Constant::ElectronMass * Constant::epsO));
}

/** This class implements the permittivity calculation using
 * the base class for \( \nu_{eh} \) and plasma frequency
 */
class plRelativePermittivity : public plBasePermittivity
{
public:
    // Calculate the real part of the relative complex permittivity
    virtual void DoCalculate( REAL & res,
        const plCrossSectionMatrix & cs,
        const plConstValueRef<REAL> & point );
    plRelativePermittivity ( const plNode & node,
        const plParticleMap& pmap,
        const plBaseRelationMap& rmap)
        : plBasePermittivity (node, pmap, rmap){}
};

/** The real part of the relative permittivity is calculated by
 * \( perm_{real} = 1 - \frac{\text{sqr}(\omega_{pl})}{\text{sqr}(\omega) + \text{sqr}(\nu_{eh})} \)
 */
void plRelativePermittivity::DoCalculate( REAL & res,
    const plCrossSectionMatrix & cs,
    const plConstValueRef<REAL> & point )
{
    const REAL nu_eh=CalcEHFrequency(cs, point);
    const REAL n.e=plAccessor::Density(point)[Particles().GetElectronNdx()];
    const REAL omega_pl = PlasmaFrequency(n.e);
    res = 1.0 - sqr(omega_pl) / (sqr(Omega()) + sqr(nu_eh));
    if (res < 0.0 ) {
        Log(0) << "Real part of Relative Permittivity was < 0,\n" << "Plasma freq. = " << omega_pl
        << " Resonator freq. = " << Omega()
        << " nu_eh = " << nu_eh << "\n";
    // res = 0.0;
REGISTER_PROVIDER(plRealValTransCalculator, plRelativePermittivity, "MWPermittivity");

/** Calculates the electrical conductivity (sigma) which can be used
 to calculate the imaginary part of the complex relative
 permittivity. For the approximation we used here see formula B.45
 on page 112 in Anhang B of the Diplomarbeit by K. Garloff.
 This approximation is valid if omega_p is small compared to the
 other frequencies. Also see page 133 of the thesis
 of G.M. Janssen.
 */
class plElectricalConductivity : public plBasePermittivity
{
    public:
    virtual void DoCalculate(REAL & res,
                const plCrossSectionMatrix & cs,
                const plConstValueRef<REAL> & point);

    plElectricalConductivity (const plNode & node,
                const plParticleMap & pmap,
                const plBaseRelationMap & rmap)
      : plBasePermittivity (node, pmap, rmap) {}
};

/** The electrical conductivity (sigma in the model, not to be confused with
 a cross section) is calculated by
 \( \sigma = \epsilon_0 \frac { \sqrt{\omega - pl} \ \nu_{eh} } { \sqrt{\omega} + \sqrt{\nu_{eh}} } \) */
void plElectricalConductivity::DoCalculate(REAL & res,
                const plCrossSectionMatrix & cs,
                const plConstValueRef<REAL> & point)
{
    const REAL nu_eh = CalcEHFrequency(cs, point);
    const REAL n_e = plAccessor::Density(point)[Particles().GetElectronNdx()];
    const REAL omega_pl = PlasmaFrequency(n_e);
    res = Constant::eps0 * sqrt(omega_pl) * nu_eh / (sqrt(omega) + sqrt(nu_eh));
}

REGISTER_PROVIDER(plRealValTransCalculator, plElectricalConductivity, "MWConductivity");

121
Appendix E

Cascaded Arc Mach 1 Boundary Condition Plugin

/** \file
 * Implements acoustic expansion velocity profile (M=0). Jan & Michiel.
 * Generalized for the not-so-uncommon case of a Mach number that is not
 * exactly one.
 */
#include "plparser/node.h"
#include "plparser/log.h"
#include "plmath/complex.h"
#include "plmath/plasmath.h"
#include "plgeneric/provider.h"
#include "plgeneric/accessor.h"
#include "plgrid/grid.h"
#include "plgrid/gridvar.h"
#include "plgrid/bndcond.h"
#include "plgrid/vessel.h"
#include "plgeneric/forceink.h"

FORCE_LINK_ME(mach1cond);

/** a quadratic profile with the maximum according to the Mach=1 condition.
 * See the thesis of Janssen, eqn. 2.15.
 */
class plMach1Cond : public plDirichletCond<REAL>
{
public:
    /** After calling the base class' constructor, the power in the
     * geometrical expression is read from the input node. If a value
     * for Gamma is specified, it is copied into the m_gamma member,
     * otherwise the default value of Cp/Cv=5/3 is used.
     */
    plMach1Cond( plModelRegion *region,
                      const plBoundaryPiece *piece,
const plBoundaryCurve *bcurve, const plNode& node) :
plDirichletCond<REAL>( region, piece, bcurve, node)
{
    \	u-todo FIXME: It would be better using the calculated
    \	Gamma, which is available since short.
    
m_power = node("Power");
    if ( node.firstL("Gamma") )
        m_gamma = node("Gamma");
    else
        m_gamma = 5.0/3.0; m_power = node("Power");
    if ( node.firstL("MachNumber") )
        m_MachNumber = node("MachNumber");
    else
        m_MachNumber = 1;
}

/**
  * Implements the Mach=1 condition (sonic expansion).
  * The maximum velocity
  * is multiplied with a geometry-dependent function. This function
  * is a polynomial with the begin value set at the acoustic velocity
  * and the end value at 0. The order of the polynomial is an input
  * parameter (power).
  */
virtual REAL F( unsigned int ndx) const
{
    Pos ipos = GetBndNdx(ndx);
    // the boundary pressure:
    const REAL press = plAccessor::Pressure( Region() -> Field(plGrid::LocNP)(ipos));
    const REAL dens = plAccessor::MassDens( Region() -> Field(plGrid::LocNP)(ipos));
    REAL umax = m_MachNumber*sqrt( m_gamma*press/dens );
    const unsigned indx = Piece() -> NdxBegin();
    const unsigned hndx = Piece() -> NdxEnd();
    const REAL mn = (*m_bcurve)[indx];
    const REAL mx = (*m_bcurve)[hndx];
    const REAL coord = (*m_bcurve)[ndx];
    const REAL relpos = 1.0 - pow( (coord-mn)/(mx-mn), m_power );
    return umax*relpos;
}

protected:
    // the exponent in the velocity profile (see Thesis Janssen, table 4.1)
    REAL m_power;
    // Cp/Cv
    REAL m_gamma;
    // The desired Mach number
    REAL m_MachNumber;
};

REGISTER_PROVIDER( plBoundCond<REAL>, plMach1Cond, "Mach1Cond");

123
Appendix F

Draka Inlet Plugin

/** \file velprof.cpp
 * Implements velocity profile with fixed mass flux,
 * according a power law (parabolic or higher).
 * Needed for flow for POF simulation. By Michiel.
 * Some changes by KG.
 */
#include "plconfig.h"
#include "plparser/node.h"
#include "plparser/log.h"
#include "plmath/cmplx.h"
#include "plmath/plasmath.h"
#include "plgeneric/provider.h"
#include "plgeneric/accessor.h"
#include "plgrid/grid.h"
#include "plgrid/gridvar.h"
#include "plgrid/bndcond.h"
#include "plgrid/vessel.h"
#include "plgeneric/forceink.h"

FORCE_LINK_ME(velprof);

/** a quadratic or higher power velocity profile according to the flow parameters
 * See the thesis of Janssen, page 28.
 */
class plMassFluxPowerLawVelocityProfile : public plDirichletCond<REAL> {

protected:
    /// the Flow in kg/s given by the user (may be specified in sccs*mass)
    REAL m_MFlow;
    /// the exponent in the velocity profile (see Thesis Janssen, table 4.1)
    REAL m_Power;

protected:
    /** cache, only updated once per iter by Prepare() */
REAL m_umax;
REAL m_center, m_max;

/** profile function: The velocity profile; parabolic or higher
*  \$1 - \left(\frac{r-r_{\text{cent}}}{r_{\text{max}}-r_{\text{cent}}}\right)^{\text{Power}}\$
*  Does only make sense in cylindrical config with this piece
*  starting at the symmetry axis.
*  - For a flow between concentric cylinders (annulus), see
*  Bird/Stewart/Lightfoot: Transport phenomena, 2.4
*  (parabolic still may be an acceptable approximation)
*  - For a flow that enters in axial direction, things
*  should be investigated carefully. If the flow comes from
*  a circular tube, things should be fine.
*/
REAL profile (const REAL radius) const
{
    return (1.0 - pow(fabs((radius - m_center) / (m_max - m_center)), m_Power));
}

public:
/** After calling the base class ' constructor, the flow
*  and power is read from the input node.
*/
plMassFluxPowerLawVelocityProfile( plModelRegion *region,
const plBoundaryPiece *piece,
const plBoundaryCurve *bcurve, const plNode& node)
: plDirichletCond<REAL>( region, piece, bcurve, node),
  m_umax(0)
{
    m_MFlow = node("Flow")["kg/s"];  
    m_Power = node("Power");

    const unsigned lndx = Piece() -> NdxBegin();
    const unsigned hndx = Piece() -> NdxEnd();

    REAL min = (*m_bcurve)[lndx];
    m_max = (*m_bcurve)[hndx];
    /** Try to be clever:
     *  - If the min index is 0, than we are probably on axis
     *  and the max. flow is here
     *  - If not, the max should be in the center between the walls
     *  - User may override it by specifying Center coordinate.
     */
    if (node.firstL("Center") )
        m_center = node("Center")["m"];
    else
        m_center = (lndx == 0? 0.0: (min + m_max)/2.0);
    Log(1) << "VelocityProfile Center @ " << m_center << " m\n";
}
/** We want to do the integration just once per iteration,
virtual void Prepare()
{
    // massflow = atomic_mass * flow (scs/s) * 1 atm * cm^2 / (k_B * T_0)
    const unsigned indx = Piece()->NdxBegin();
    const unsigned hndx = Piece()->NdxEnd();
    m_max = (*m_bcurve)[hndx];
    // REAL min = (*m_bcurve)[indx];

    // \int_{r_{min}}^{r_{max}} u_{tot} = \int_{r_{min}}^{r_{max}} \frac{1}{2\pi} u_{max} \mbox{profile}(r) \, dr
    REAL u_tot = 0;
    REAL surface = 0;
    for (unsigned k = indx; k < hndx; k++)
    {
        REAL r = (*m_bcurve)[k];
        REAL rho = plAccessor::MassDens(Region())->Field(plGrid::LocNP)(Ge
        u_tot += BoundSurf() * rho * profile(r);
        surface += BoundSurf();
    }
    m_umax = m_MFlow / u_tot;
    Log(O) << "Debug: inflow: Surf: " << surface
    << ", calculated u_max: " << m_umax << std::endl;
}

/** Implements the Velocity profile according to the flow in kg/s
 * may be specified in scs*molecmass by the user (thx to unit conversions ...)
 */
virtual REAL F( unsigned int ndx) const
{
    const REAL rad = (*m_bcurve)[ndx];
    return m_umax * profile(rad);
}

REGISTER_PROVIDER(plBoundCond<REAL>, plMassFluxPowerLawVelocityProfile, "MassFlu
Appendix G

Multi Point Grid Stretch Code

/**
 * Calculates the coefficients for an n-points polynomial stretch function.
 * x and s are the vectors defining the points where you specify a derivative at,
 * and a is your output coefficient vector. Sep 02, MD.
 *
 * We have 2m+2 equations:
 * (1) \( f(0) = 0 \)
 * (2) \( f(1) = 1 \)
 * (3) \( f'(x_0) = s_0 \)
 * (4) \( f''(x_0) = 0 \)
 * ...
 * (2m+1) \( f'(x_m) = s_m \)
 * (2m+2) \( f''(x_m) = 0 \)
 *
 * For the reasoning behind equations (4) and (2m+2) see comments to
 * onepnt.poly_stretch::calc_coeffs()
 *
 * Here, we use Gauss-Jordan elimination, using TBCI matrices.
 * Matrix equation is of the form \( M \cdot a = b \) where
 * \( M \) is a matrix correlating the vector \( a \) (your polynome coefficients) and
 * \( b \) (the inhomogenous term). \( M \) contains the (constant) terms with
 * all the \( x \cdot m \)'s and powers thereof (\( x \cdot m \) are NOT your variables, \( a \cdot n \) are!).
 *
 * \( m \) is the number of points you specify a derivative at,
 * \( n \) is the order of your polynome.
 * \( n = 2m+2 \) because there is 1 equation for the start and one for the end
 * of your stretch function, and 2 for each specified point density.
 **/

void npt_poly_stretch::calc_coeffs (const int m, const double const &x, const double const &s, double &a)
{
    int i;
int n = 2*m + 2;

// Of course M is a square matrix, it contains x.m's and powers thereof.
TBCI::Matrix<double> M (2*m,n-2);

// In b the first 2 equations are not stored, since those are not
// solved in the matrix but separately, therefore b starts at the
// equation for a[2] instead of a[0].
TBCI::Matrix<double> b (n-2,1);

// Fill the matrices with 0 (not really needed).
M.fill (0.0);
b.fill (0.0);

// The zeroeth coefficient suffices to the equation f(0) = 0, because only
// the constant term in the polynomial remains we can set this term to 0.
a[0] = 0.0;

// Fill the matrix to solve the coefficients a[2]...a[n]
// s.index is the index of the point that you specify a derivative for.
// For example in a two point stretch function you have s[0] specified on x[0]
// and s[1] on x[1], and s.index would iterate from 0 to 1.
// Of course x[n] is not exactly known until the polynome is known, since you
// specify y[n], but the iteration to find x[n] is done elsewhere.

for ( int s.index=0; s.index<m; s.index++)
{
    // a.index is the index which gives the coefficient of a, so for example
    // at a[2] (the coefficient for x'2 in your to-solve polynomial) a.index is 2.
    for ( int a.index=0; a.index<n-2; a.index++)
    {
        // Sets the matrix coefficients for f'(x[s.index]) = s[s.index]
        M(s.index, a.index) = float(a.index+2) * pow(x[s.index],a.index+1) - 1.1;

        // Sets the matrix coefficients for f''(x[s.index]) = 0
        // These equations are under the one for the first derivative
        M(s.index+m,a.index) = float((a.index+1)*(a.index+2)) * pow(x[s.index],a.index+1);
    }

    // Sets the constant part for f'(x[s.index]) = s[s.index], the 1.0 term
    // Results from the equation for a[1] which is not inside the matrix.
b(s.index,0) = s[s.index] - 1.0;

    // Sets the constant part for f''(x[s.index]) = 0
    // Was already 0 at init but it can never hurt to do it again.
b(s.index+m,0) = 0.0;
}

128
// The restriction that f(1) = 1 gives an equation of the form \[ \sum_{n=1}^{n} a[n] = 1 \]
// This gives us the equation for a[1] expressed in all other a[n]'s
// This equation was already used in for a[1] with the derivation of the
// matrix M we use to solve a[2]...a[n].

// Use Gauss-Jordan to solve this matrix.
char errcode = gaussj(M, b);

double suma = 0.0;
// copy the coefficients from b to a[2]...a[n]
for (i=2; i<n; i++)
{
    a[i] = b(i-2,0);
    suma += a[i];
}

// now calculate a[1] from all the other coefficients.
a[1] = 1.0 - suma;
Empedocles lived in Acragas, which is now known as Agrigento, Sicily. He was not only a natural philosopher, but also poet, seer, physician, social reformer, a person of great enthusiasm, known as a charlatan by some, and as a hero by others. He was the author of a physical-cosmological poem titled On Nature, now known only through fragments of quotations from later writers. It is perhaps through the influence of Xenophanes, who had settled in Sicily, that Empedocles can be placed in the Miletian tradition of natural philosophy that taught that principles in the form of matter were the only principles of all things. Xenophanes had postulated that everything was composed of two elements water and earth. Empedocles extend this postulate to four basic elements: earth, water, air, and fire. Different mixtures of these four elements produced the materials of our common experience.