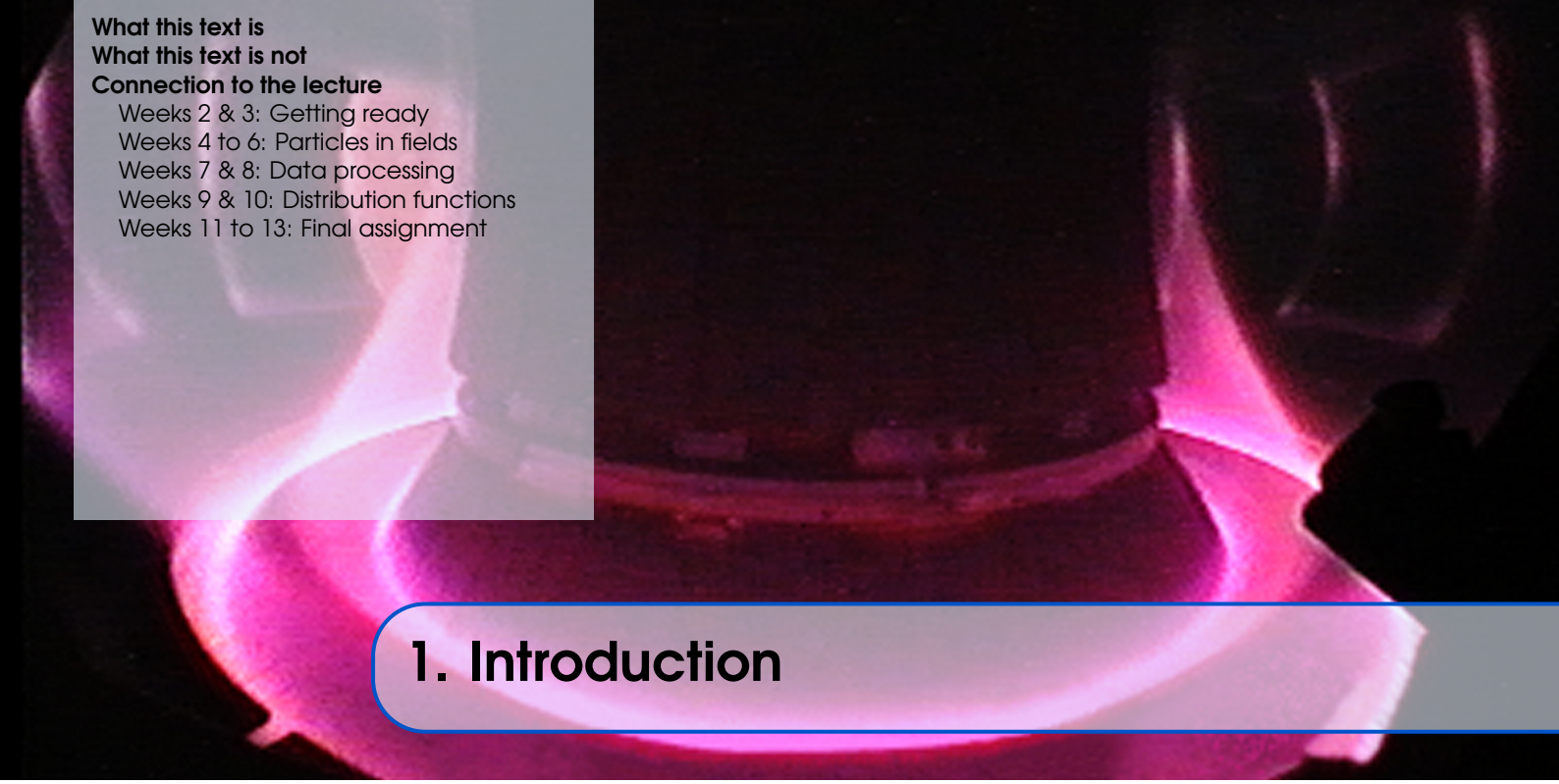
Interactive Introduction to Plasma Physics

Part II: Hands-on exercises

Adam Obrusník, Lenka Zajíková

October 2015 version

Contents



1. Introduction

1.1 What this text is

This is a **supplementary text** to the exercises to the lecture *F5170: Introduction to Plasma Physics*. The lecture itself contains many theoretical derivations and considerations and the aim of this text is **to mediate hands-on experience** to students who come in contact with plasma physics for the first time.

Throughout the text, you, as a student, will be presented to a number of **problems**, most of **which have to be solved with the help of computers**. You will, among other things, learn to solve systems of ordinary differential equations, to batch-process data or to numerically integrate data, which can not be described by an analytical function. You will find most of the skills and numerical techniques that you will learn throughout the semester useful also in other disciplines of physics.

The hands-on examples are designed to be as straightforward and illustrative as possible. It has been taken into account that most of the students may initially lack the programming skills, required for completing this course. For this reason, all the code is richly commented and documented. The idea is that **you will learn** the basics of Matlab programming **on these real-life commented examples**.

1.2 What this text is not

First and foremost, this text does not aim to be a comprehensive guide to Matlab programming and scripting. There are countless books and manuals on Matlab and its frameworks. The most relevant and accessible ones are listed in section 2.2.3. Be prepared that the text may occasionally be too concise, the language may be sloppy and the logic not obvious. That is because at this point, the text is not designed to work on its own but rather in combination with the exercises to *F5170: Introduction to Plasma Physics*.

This text is also not a reference that the user would be advised to cite in scientific publications or theses of any kind. It is always best to refer directly to the books or articles that this text is built on. Although obtaining the original references just to make sure a particular formula or figure is correct may seem tedious and pointless, it is a skill that you will certainly find useful in your future scientific work.

Finally, this text is not a text on numerical methods. Various differential equation solvers

are considered black boxes and their inner workings are not analyzed in detail. The authors avoid non-dimensionalization of all the equations that are solved because using meaningful physical units has the advantage that you will not only understand the phenomena qualitatively but you should also get an idea about the magnitudes of various quantities in plasma physics.

1.3 Connection to the lecture

The lecture spans across 13 weeks of a Fall semester at the Masaryk University. This text reflects the contents of the lecture and the idea is that you will progress by one chapter every two weeks. There is usually no exercise in the first week of the semester.

1.3.1 Weeks 2 & 3: Getting ready

Since the lecture has to have a bit of a head start, you will have whole two weeks to get the required software to work on your computer. Both Windows and Linux users will have to install the university-licensed **Matlab** software (see chapter 2). Students should also study the second part of this chapter to learn the absolute basics of Matlab code and logic.

1.3.2 Weeks 4 to 6: Particles in fields

In weeks 4 to 6, you should understand all the Matlab programs presented in chapter 3 and by the end of week 5, you should complete all the exercises in the respective chapter. The exercises focus on movement of particles in electric and magnetic fields and should help you to understand the nature of various particle drifts in plasmas.

1.3.3 Weeks 7 & 8: Data processing

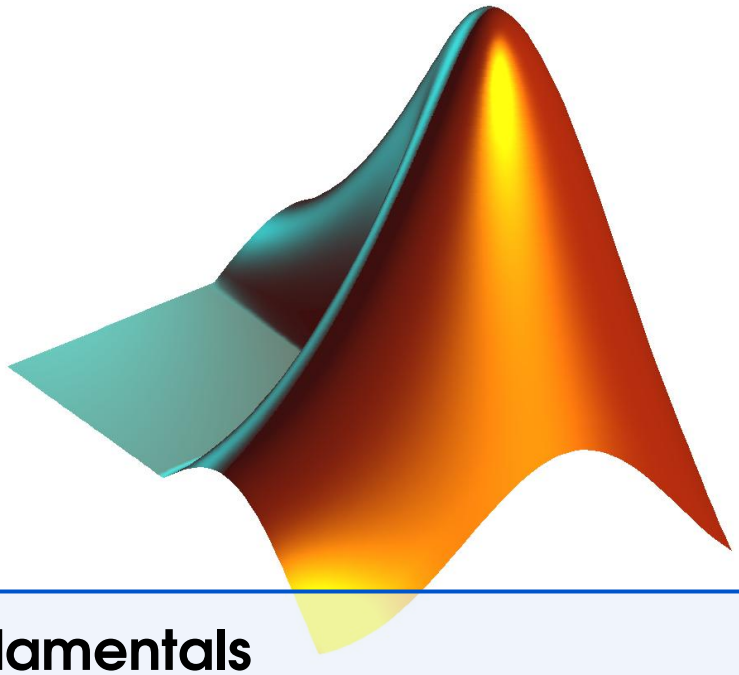
You will learn basics of batch data processing using examples from plasma physics. This includes interpolation and integration of discrete data in Matlab as well as data conversion. Although you will work with data relevant for plasma physics, you will find the skills obtained in this section useful in any other discipline of physics.

1.3.4 Weeks 9 & 10: Distribution functions

This short chapter should help you to understand distribution functions. You will be provided with programs that can visualize the time development of a distribution function. You will also analyze the shape of the Maxwellian distribution for various conditions.

1.3.5 Weeks 11 to 13: Final assignment

In the final assignment, you will utilize the knowledge obtained in all the previous sections. You will have to write or complete a program that solves particle balance in a laboratory argon plasma. You will also learn how the composition of the plasma changes with the pressure or the temperature of electrons and heavy species.



2. Matlab fundamentals

Matlab (originally Matrix Laboratory) is commercial software and a scripting language being developed by the company MathWorks. Unlike many traditional programming languages, Matlab allows the user to manipulate matrices and vectors in an intuitive way. It allows the users to define their own functions which can have virtually any input or output.

2.1 Installing Matlab at the Masaryk University

2.1.1 MUNI License

As of 2014, Masaryk University has the newest license of Matlab with a license pool of at least 250 licenses. This means, that 250 students or employees of MUNI can run Matlab simultaneously. In order to work properly, the MUNI license requires the user to be connected to the Internet and to MUNI's virtual private network (VPN). The license is limited to research and non-commercial usage.

2.1.2 Getting the installation files

The installation files for Matlab 2014b are available from the intranet of the Masaryk University. Please keep in mind that the Matlab installation file is currently approximately 7.5 GB large but inside Masaryk University's network, you will be able to download the installation files extremely fast. Starting from version 2013, Matlab is available only for 64bit computers and operating systems. If you want to use Matlab on a 32bit architecture, please ask the advisor.

Howto 2.1.1 — Downloading Matlab files. Follow these steps to get all the necessary files

1. Open inet.muni.cz in your web browser
2. Log in using your *university number* (UO) and your *primary password*
3. Go to *Software* → *Nabídka softwaru*
4. Locate Matlab in the software list and click *Získat*
5. After agreeing with the License agreement, download the following three to your computer
   - The `license.dat` file
   - The installation image, `R201xx_Windows.iso` for Windows machines or `R201xx_UNIX.iso` for Linux-based operating systems

- The *Autorizaní kód* – store it in a text file since the installer will ask you for it.

2.1.3 Installation

Having downloaded the ISO installation images, you have to mount them and run the setup. Mounting an ISO image means creating a virtual DVD drive, containing the files from the ISO. Alternatively, you could burn the ISO image onto a DVD and install it from there.

Howto 2.1.2 — Mounting the ISO on Windows. The most widely used software for mounting ISO files (i.e. creating a virtual DVD drive with the ISO files as a content) is probably Daemon Tools Lite.

Despite being the best software for mounting virtual drives, Daemon tools Lite forces a lot of unwanted software on the user. If you want to avoid an unwanted toolbar in your browser, go through the installation carefully, unchecking the corresponding option.

After the installation of Daemon Tools, mount the Matlab DVD:

1. Right-click the Daemon tools icon in your system tray
2. Go to *Virtual DVD* → *Device 0:* → *Mount Image*
3. Locate the ISO image and confirm
4. You will now see the Matlab DVD in *My Computer*

Howto 2.1.3 — Mounting the ISO on Linux. Many modern Linux distributions have built-in functionality for mounting ISO images. You can usually access this functionality by right-clicking the ISO file and choosing the *Mount* option.

If you cannot find a built-in mounting function in your distro, you can use AcetoneISO software, which is available in repositories of all major Linux distributions.

If you prefer a command line solution, you can type the following to the terminal, without having to install anything

```
mount -o exec R201xx_UNIX.iso /mnt/disk
```

with `/mnt/disk` being the directory where you want to mount the ISO

On Windows, the installer is started by the *setup.exe* file, while on Linux, you have to execute the *./install* binary. In both cases, the files are located in the top-lever directory of the installation DVD. When the installer asks you whether you want to *Activate Matlab*, click *Yes*.

On Linux, you may want to install Matlab as a super-user. Otherwise, a link in `/usr/local/bin` will not be created and you will not be able to run it simply by typing `matlab` to the command line.

The Matlab installation itself is straightforward and user-friendly so follow the installation guide. You will be asked to provide first the *Autorizaní kód*, then choose the installation location and then provide the license file. At one point, you will be asked about **installing toolboxes**. The toolboxes are additional function libraries, extending the functionality of Matlab. If you want to save some hard-drive space, you **do not have to install** toolboxes from these categories

- Test and measurement
- Computational finance

- Computational Biology
- Code generation

2.1.4 Running Matlab

Windows users will find a link to Matlab on their desktop and in their Start menu. On Linux, you can run Matlab by the `matlab` command (if you installed it under superuser privileges) or from its installation directory,

```
/path/to/Matlab/R20xx/bin/matlab
```

After starting Matlab, the main window will open. It should be identical on all operating systems.

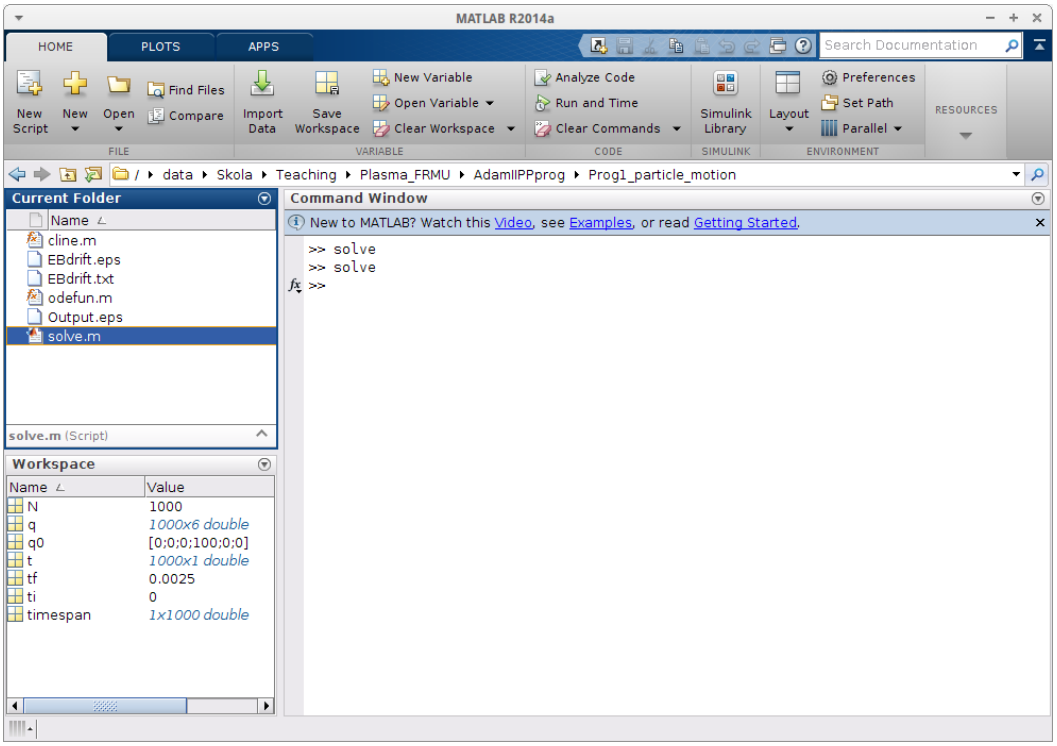


Figure 2.1: The main Matlab window

The window has three main sub-windows:

- **Current folder** shows you the contents of the folder you are in, especially all Matlab scripts that you can execute.
- **Workspace** lists all variables (numerical values or matrices) which are currently defined. By double-clicking a variable, its value will be displayed.
- **Command window** is the most important part. Here, you can define new variables, call functions or execute Matlab scripts. You will learn more about this in the next section.

2.2 Matlab fundamentals

This section briefly discusses the fundamentals of Matlab syntax and logic. It will not be very exhaustive because this text is built on the assumption that writing Matlab code does not have to be studied rigorously and can be understood on meaningful examples. At the end of this section, you should understand especially the following,

- What is a *variable* in Matlab and how to define one.
- How to access matrix and vector elements or select sub-matrices
- What is the difference between *script files and function files*.
- What is a *function* in Matlab and how to call it.

For some of you, the information in this section will be trivial but this text also remembers those, who never came in contact with any programming.

2.2.1 Variables and indexing

Working with variables in Matlab is as easy as in real life, you only have to realize that Matlab **performs numerical operations, not analytical**. What does this mean in practice? Try to define a variable simply by typing

```
v = 10
```

and let's assume it is the velocity of some object. You then define the object's mass in kilograms

```
m = 3
```

and you can calculate the object's kinetic energy as

```
Ek = 0.5*m*v^2
```

Now, when you try to re-define the velocity and look to the **Workspace** sub-window, you will see that the kinetic energy has not changed. That is because the variable `Ek` **only contains the numerical value**, the way in which this variable was calculated is not stored. You would have to re-run the corresponding command to get the updated value of `Ek`.

Working with matrix or vector variables is also simple in Matlab. Let us assume you want to rotate a vector in the 3D Cartesian space in the *xy*-plane. First, define the vector

```
r = [10, 20, 30]
```

and then define matrix of the rotation using `sin()` and `cos()` built-in functions

```
M = [cos(pi/3),-sin(pi/3),0; sin(pi/3),cos(pi/3),0; 0,0,1]
```

You can see that matrices are defined row-by-row. The elements in columns are separated by commas (,) and the rows are separated by semicolons (;). A vector is essentially just a $1 \times n$ or $n \times 1$ matrix. However, if you try to multiply `M` and `r` by typing

```
rrot = M*r
```

you will get an error saying that *inner dimensions do not agree*. That is because `r` is a row vector and you cannot multiply a 3×3 matrix with a 1×3 vector. In this case, the multiplication is performed by

```
rrot = M*r'
```

where the apostrophe denotes transposition of `r`.

R You noticed that upon defining a new variable, its value is displayed in the command window. To prevent that, end each command with a semicolon (;).

The standard operations that can be performed on scalars, vectors and matrices are addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (^). However, there

are also **element-wise operations**. For example, element-wise multiplication (`.*`) of two 1×3 vectors works like this

$$(1,4,5) .* (2,4,6) = (1*2, 4*4, 5*6) = (2,16,30). \tag{2.1}$$

Apart from element-wise multiplication, there is also element-wise division (`./`) and element-wise exponentiation (`.^`).

Finally, it is useful to know **how to access** individual **elements of matrices and vectors** or to create sub-matrices or sub-vectors. You can perform the following operations on a vector

```
a = [1,2,3,4,5,6,7];
b = a(4) % get fourth element of "a" = 4
b = a(1:3) % get subvector from el. 1 to 3 = [1,2,3]
b = a(4:end) % subvector from el. 4 to the end = [4,5,6,7]
```

With 2D matrices, you need to use two indices. Take in mind that the first index is the row and the second is the column

```
a = [1,2,3;4,5,6;7,8,9];
b = a(1,2) % get the element on 1x2 = 2
b = a(:,2) % get the second column = [2,5,8]
b = a(3,:) % get the third row = [7,8,9]
b = a(1:2,1:2) % get the submatrix = [1,2;4,5]
```

2.2.2 M-files

Defining variables and operating with them in the command window can save some time but that alone would make Matlab only a better pocket calculator. Typically, you will use Matlab or any similar scripting language for automation of repetitive or complicated tasks. There are two types of files which you can use for storing a sequence of commands, **script files** and **function files**. It may be a bit confusing that **both have the .m extension**, but that is the fact.

Script files

A script file contains a set of Matlab commands, one at each line, which are executed in a sequence. For example, if you measure a quantity over a long period of time and you want **to generate an up-to-date normalized plot** every day, it is convenient to use a script

Matlab script example

```
day =   [1, 2, 3, 4, 5, 7, 8, 10, 12, 15]; % days
count = [450, 363, 291, 220, 180, 150, 120, 100, 90, 80];
    % bacteria count
mc = max(count); % finds maximum value of count
norm_count = count/mc; % performs normalization

% Plots data on x-axis and count on y-axis
plot(day, norm_count, 'rx-');
xlabel('Days'); % sets x-axis title
ylabel('Normalized cell count [CFU]'); % sets y-axis title
title('Plasma sterilization experiment') % sets figure
   title

print -dpng -r72 'experiment.png'; % exports figure to png
```

On the first two lines, there are the measurement data. The `day` variable stores measurement days while the `count` variable stores values measured on each of these days. On lines 3 and 4, the data are normalized and from line 7 onwards, the data are plotted. Obviously, such a script file can save some time in our model situation.

The code above contains comments. You can and should use them to annotate any of your code so that you are able to recall what it does even after some time. The percent sign tells Matlab that everything which follows (until the end of line) is not a command and should be ignored.

If you want **to run a script file**, you simply change the **Current folder** to the folder where your script is and **type the script name** (without the extension .m) **into the command window**.

Exercise 2.1 Write a script file which will plot the following ozone density measurements in the logarithmic scale. Note that the `log()` function calculates the natural logarithm while `log10()` calculates the decimal logarithm.

position [mm]	0.0	0.1	0.2	0.3	0.5	0.6
O_3 density [m^{-3}]	$1.4 \cdot 10^{20}$	$1.0 \cdot 10^{20}$	$5.2 \cdot 10^{19}$	$1.6 \cdot 10^{19}$	$4.5 \cdot 10^{18}$	$9.1 \cdot 10^{17}$

■

Function files

M-files containing functions are a little different from the script files. First and foremost, a function is no longer a plain sequence of commands but it is an object which always has N inputs and M outputs, similar to functions in mathematics. To provide an example, a Matlab function that calculates the Larmor radius and cyclotron frequency of an electron could look like this

Matlab function example: gyro.m

```
function [r, omega] = gyro(B, vp)                    1
 q = -1.602e-19; % elementary charge [C]             2
 m = 9.109e-31; % mass [kg]                          3
                                                     4
 r = m*vp/(abs(q)*B); % Larmor radius                5
 omega = q*B/m; % Cyclotron frequency                6
end % end of function                                7
```

The behaviour of this function is defined on the first line. The variables `r` and `omega` are the output arguments and `B` and `vp` are input arguments. The **function name on the first line** (`gyro`), **has to be identical with the file name**.

This function can be executed either from the **Command window** or from another m-file. To test this, try running

```
[r0, w0] = gyro(1e-4, 1e3)                           1
```

from your **Command window**. Two new variables, `r0` and `w0` will appear in your **Workspace** with the correct values of Larmor radius and the cyclotron frequency.

Exercise 2.2 Write a function that will have electron density in m^{-3} and electron temperature in eV at the input and the Debye length and plasma frequency at the output. The necessary formulas can be found for instance in [Bit04]. ■

2.2.3 Where to go from here

In this section, only the absolute fundamentals of Matlab coding were explained. A more detailed introduction can be found at www.mathworks.com/help/matlab/getting-started-with-matlab.html and the authors of this text advise the students to read it.

Similar to all computer-related matters, keep in mind that Google is your friend. What you will find especially useful are links leading to stackexchange.com. Since Matlab is so widely used, you can be almost certain that someone else already got a similar problem resolved on this forum.

If you decide to use Matlab beyond this course, for your own research, MatlabCentral will be very useful, www.mathworks.com/matlabcentral. Most importantly, it contains detailed documentation of Matlab functions, most with usage examples. In addition, it is also a discussion board and a repository of examples and additional functions that you can download (e.g. functions for nicer plotting).

3. Motion of particles in E and B fields

3.1 Particles in constant E and B fields

The first problem that you will learn to implement is simple motion of a charged particle in electric and magnetic fields. This is a well known problem with well-known analytical solutions. This aim of this section is to demonstrate, how different are the approaches to solving a set of differential equations (DEs) on paper and in a computer program. The short program developed in this section is richly commented and explained and it is crucial that the reader understands what every line of code exactly does.

Movement of particles in complex, often non-uniform and time-dependent, electric and magnetic fields is often investigated in plasma physics as well as in other disciplines. Some of the most frequent applications include:

- **Mass spectrometers**, in which particles can be filtered by according to mass/charge ratio
- **Fusion reactors** which use magnetic fields for plasma confinement [**aa**]
- **Electron microscopes**, where the electron beam is guided by electric fields [FEI10]
- **Hall thrusters** used for propulsion in space (see figure 3.1)
- and many more...

3.1.1 Underlying equations

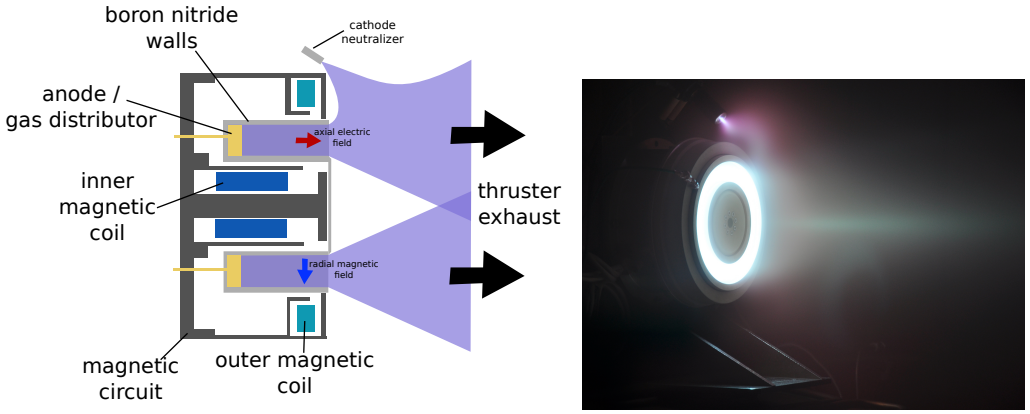
The code developed in this section will solve the equation of motion in three dimensions for a particle with a defined mass and charge. The force acting on the particle is the Lorentz force

$$\mathbf{F}_\mathrm{L} = q\left(\mathbf{E} + \mathbf{v} \times \mathbf{B}\right) = q\left(\mathbf{E} + \dot{\mathbf{r}} \times \mathbf{B}\right) \tag{3.1}$$

where q is the charge of the particle, $\mathbf{E}$ is the electric field, $\mathbf{v}$ is the velocity of the particle, $\mathbf{r}$ its position and $\mathbf{B}$ is the magnetic field. The equation of motion for the particle, therefore, takes the form

$$\ddot{\mathbf{r}} = \frac{q}{m}\left(\mathbf{E} + \dot{\mathbf{r}} \times \mathbf{B}\right) \tag{3.2}$$

with m being the mass of the particle. The motion equation in this form can be very useful for various general considerations and for finding analytical or approximative solutions



(a) A schematic by David Staack

(b) Hall thruster firing

Figure 3.1: Hall effect thrusters use electric and magnetic field to accelerate ions out from a plasma. As of 2014, Hall thrusters are routinely used with geostationary satellites as well as extraterrestrial missions. [GK08, page 15]

to the problem. However, if we want to solve this motion equation numerically, we will have to change our perspective a bit.

First and foremost, it has to be said that Matlab, as well as most numerical libraries, does not have functions which would allow you to solve second order ODEs directly. It is, however, very good with solving first-order ODEs as well as systems of first-order ODEs. With the use of the standard notation,

$$\mathbf{r} = (x, y, z), \tag{3.3}$$
$$\dot{\mathbf{r}} = (v_x, v_y, v_z), \tag{3.4}$$
$$\dot{\mathbf{B}} = (B_x, B_y, B_z), \tag{3.5}$$
$$\dot{\mathbf{E}} = (E_x, E_y, E_z) \tag{3.6}$$

equation (3.2) is equivalent to the following six ODEs.

$$\dot{x} = v_x, \tag{3.7}$$
$$\dot{y} = v_y, \tag{3.8}$$
$$\dot{z} = v_z, \tag{3.9}$$
$$\dot{v}_x = \frac{q}{m}\left(E_x + v_y B_z - v_z B_y\right), \tag{3.10}$$
$$\dot{v}_y = \frac{q}{m}\left(E_y + v_z B_x - v_x B_z\right), \tag{3.11}$$
$$\dot{v}_z = \frac{q}{m}\left(E_z + v_x B_y - v_y B_x\right). \tag{3.12}$$

Apparently, we transformed three second-order ordinary differential equations expressed by the vector equation (3.2) to a set of six first order differential equations.

R Transforming N differential equations of the second order to $2N$ differential equations of the first order is not uncommon in physics. Most readers have certainly encountered this in mechanics courses, when transferring between *Lagrangian* and *Hamiltonian* formalisms. In this sense, the formalism which is much more convenient for numerical analysis is the *Hamiltonian* formalism.

3.1.2 Solving the system of ODEs in Matlab

The Matlab function which is used for ODE solving is the `ode45` function. This function is capable of solving a set of first-order differential equations in the form

$$\dot{\mathbf{q}} = f(t, \mathbf{q}) \tag{3.13}$$

where t is the independent variable and $\mathbf{q} = \mathbf{q}(t)$ is a vector of dependent variables. It should now be apparent, why we reformulated the motion equation to the form described by equations (3.7) to (3.12). If you compare our reformulated equations to (3.13), you should see that in our case

$$\mathbf{q} = (x, y, z, v_x, v_y, v_z). \tag{3.14}$$

The function `ode45` is called in the following way

```
[t,q] = solver(@f,tspan,q0)
```

The first argument of the function `ode45` is the function $f(t, \mathbf{q})$ which evaluates the right-hand sides of the system of ODEs. The second argument is the interval on which the ODEs will be solved and the third argument are the initial conditions for $\mathbf{q}$.

The function `ode45` returns the solution in the form of a matrix with the following form

$$\mathtt{t} = \begin{pmatrix} t \\ 0.0 \\ 0.0001 \\ 0.0002 \\ 0.0003 \\ 0.0004 \\ 0.0005 \\ \dots \end{pmatrix} \qquad \mathtt{q} = \begin{pmatrix} x & y & z & v_x & v_y & v_z \\ 0 & 0 & 0 & 100.0 & 0 & 0 \\ -0.0000 & 0 & 99.91 & -3.95 & 0 & \\ 0.0005 & -0.0000 & 0 & 99.65 & -7.91 & 0 \\ 0.0007 & -0.0000 & 0 & 99.21 & -11.84 & 0 \\ 0.0010 & -0.0001 & 0 & 98.60 & -15.76 & 0 \\ 0.0012 & -0.0001 & 0 & 97.82 & -19.64 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix} \tag{3.15}$$

The column vector `q` contains the values of our independent variable (time) while the columns of the matrix `q` contain the values of our dependent variables (positions and velocities) at given times.

3.1.3 The Implementation

We are going to need two files to solve the motion equation. The first one, `odefun.m` will contain the function evaluating the r.h.s. of the system of differential equation while the file `solve.m` will contain the initial solution itself and plotting of the result. These two files have to be placed in the same directory and only the latter will be executed directly from Matlab. Let us first look at the file `odefun.m`.

Program 1: odefun.m

```
function dqdt = odefun(t,q),
        % This is the input function for the
        % ordinary differential equation solver ode45.
        % Input: independent variable t(time), 6-dim vector
          q
        % q = [x,y,z,v_x,v_y,v_z]
        % Output: time derivative of q, dqdt = dq/dt

```

```
qe = -1.602e-19;          % Elementary charge in Coulomb
m = 9.109e-31;  % Particle mass in kg

% Now, we define position and velocity variables
% using the vector of dependent variables, q.
% This would not be necessary but using
% "x" instead of q(1), etc.. makes the code
% below more readable
x = q(1);
y = q(2);
z = q(3);
vx = q(4);
vy = q(5);
vz = q(6);

Ex = 0;
Ey = 2e-6;
Ez = 0;
Bx = 0;
By = 0;
Bz = 1e-7;

% Now, we calculate the components of the 6-dim
% vector dq/dt.
dxdt = vx;      % dx/dt = v_x
dydt = vy;      % dy/dt = v_y
dzdt = vz;      % dz/dt = v_z
dvxdt = qe/m*(Ex + vy*Bz - vz*By); % dv_x/dt
dvydt = qe/m*(Ey + vz*Bx - vx*Bz); % dv_y/dt
dvzdt = qe/m*(Ez + vx*By - vy*Bx); % dv_z/dt
% Finally, we have to assign the calculated components
% to the vector variable dqdt, which is
% the output of the function
dqdt = [dxdt; dydt; dzdt; dvxdt; dvydt; dvzdt];

end % end of function
```

Although the source code is richly commented, let us now look closer at some of its important parts. Apparently, the file `odefun.m` is a Matlab function file. Line 1 tells the computer that this file contains a function called *odefun* which takes the independent variable t and the vector of dependent variables $\mathbf{q}$ as the input and it returns its derivative $\dot{\mathbf{q}}$ (written as `dqdt` in ASCII).

On lines 8 and 9, the elementary charge and particle mass are defined. The new set of variables which is defined on lines 16 to 22 is not necessary for the correct operation of the whole program but it is much more illustrative to use `x`, `y`, etc ... on lines 35 to 37, where the derivatives are expressed. On lines 23 to 28, the electric and magnetic fields are defined.

R Defining the constants `q0` and `m` inside the function is not a good practice but we do it here

to make the source code as clear as possible. Since the function `odefun()` will be called many times by the ODE solver, it would be better to define these constants as *global variables*.

Having defined the function, describing the right hand side of the system of ODEs (3.13), we can finally solve it and plot the results. The relevant commands are included in the file `solve.m`.

Program 1: solve.m

```
% This matlab script solves the equation of motion
% for a charged particle in presence of electric
% and magnetic fields.

% First, we need to define the initial conditions.
% Remember, q = [x,y,z,v_x,v_y,v_z]
q0 = [0;0;0;1e2;0;0];

% Now we define the time interval, on which
% we want to solve the ODE.
ti = 0;
tf = 2.5e-3;
N = 1000;
timespan = linspace(ti,tf,N);
% The function linspace(ti, tf, N) creates
% a linearly spaced vector beginning at "ti"
% ending at "tf" with "N" steps/components.

% And now for the solving itself
% The notation @odefun tells the program
% that the first argument is a function.
[t, q] = ode45(@odefun, timespan, q0);
% The result will be a vector t and a matrix q with N rows.

% The following commands display the calculated trajectory
mesh([q(:,1) q(:,1)], [q(:,2) q(:,2)], [q(:,3) q(:,3)], [t
   (:) t(:)],'EdgeColor', 'interp', 'FaceColor', 'none'); %
   plot
view(2) % sets the plot to top view
set(gca,'FontSize',16,'fontWeight','bold') % sets font size
xlabel('x [m]') % sets title of x-axis
ylabel('y [m]') % sets title of y-axis
zlabel('z [m]') % sets title of z-axis
title(['Particle trajectory, ti=', num2str(ti), ' s, tf=',
   num2str(tf), ' s']) % sets figure title

print -depsc 'Output.eps' % prints the figure to file
```

3.1.4 Exercises

Exercise 3.1 Copy the source code presented in the previous section to your computer or use m-files provided in the `Prog1_Particle_motion` directory.

- Run the program with pre-defined values you should get a curve similar to figure 3.2.
- What kind of drift is observed?
- What is the direction of the drift velocity for an electron and a positron?

■

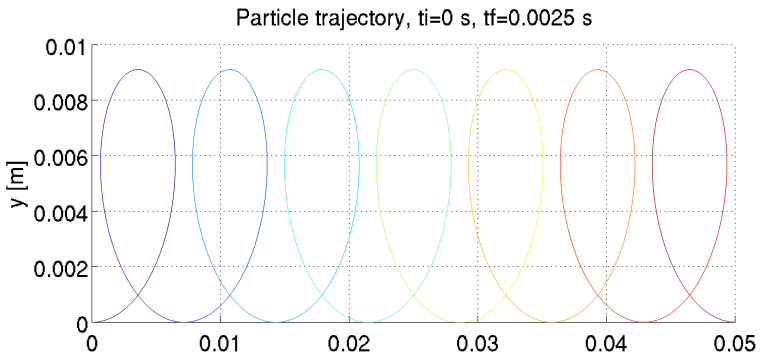


Figure 3.2: A trajectory of a particle in $\mathbf{E} = (0, 10^{-6}, 0)\,\mathrm{V/m}$, $\mathbf{B} = (0, 0, 10^{-7})\,\mathrm{T}$ and $\mathbf{q}(0) = (0, 0, 0, 100\,\mathrm{m/s}, 0, 0, 0)$

Exercise 3.2 Change the charge `qe` and the mass `m` of the particle to those of a proton.

- How many times do you have to increase the time scale in order to see the characteristic trajectory of the drift?
- Compare amplitude of the oscillations of electrons and protons and the magnitude of their drift velocities (read these data from the plots).

■

Exercise 3.3 Modify the Matlab program from exercise 3.1 in such a way that the motion equation is solved without electric field but with magnetic field quadratically growing in the direction of y

$$\mathbf{E} = (0, 0, 0) \tag{3.16}$$

$$\mathbf{B} = (0, 0, B_z) \tag{3.17}$$

$$B_z = B_0 \left(\frac{y}{y_0} \right)^2 \tag{3.18}$$

$$\mathbf{q}_0 = (0, 0, 0, \sqrt{2}v_0, \sqrt{2}v_0, 0) \tag{3.19}$$

In the equation above, y_0 is the characteristic distance that your particle travels. You can either find this empirically or you can estimate it from the theoretical velocity magnitude of your particle and the time scale `tf` of your simulation.

Do you observe any drift? For what values of B_0, y_0 and v_0 did you observe it? What is the direction of the drift velocity for an electron and for a positron? ■

Advanced exercise 3.1 Modify the program from this section so that the electric field

changes harmonically with time. We recommend using

$$E_x = E_0 \cdot \cos(\omega t), \tag{3.20}$$
$$E_y = 0, \tag{3.21}$$
$$E_z = 0, \tag{3.22}$$

and setting the initial velocity to

$$v_x = 0, \tag{3.23}$$
$$v_y = v_0, \tag{3.24}$$
$$v_z = 0. \tag{3.25}$$

Now, examine for several frequencies ω (for example 10^6, 10^7, 10^8 and 10^9 Hz) the motion of a proton and an electron.

- How do they react to the field?
- Compare how easily the electron and proton are influenced by the external field for various frequencies.

3.2 Van Allen radiation belt

The discovery of van Allen radiation belts is often seen as the first big discovery of the space age. Their existence was predicted by James van Allen and confirmed by the Explorer I satellite in 1958. The belts are a direct consequence of the Earth's magnetic field, which traps high-energy particles and prevents them from reaching the surface of the Earth. Since no shielding is ever perfect, there is always some flux of particles towards the surface of the Earth. These particles then ionize and excite the gas molecules in the lower layers of the atmosphere, which we observe as the aurora (pictured in the header of this chapter).

There are two stable van Allen belts, the first is located at 1 000 to 6 000 km above the surface while the second is located at 13 000 to 60 000 km [Bak12]. The third van Allen belt has been discovered very recently and it seems to appear and disappear over time [Sci13; Shp+13].

Since you should now have a very good understanding of charged particle motion in electric and magnetic fields, saying that the energetic particles are simply *trapped* in the magnetic field is perhaps too vague. In this section, we will modify the Program 1 we used before and use to calculate a trajectory of a high-energy proton in the Earth's magnetic field. In the process, we will learn what **global variables** are and how to use them in Matlab. You will also become familiar with several **advanced plotting commands**. This section is inspired by Lecture notes of S. Markidis [Mar13].

3.2.1 Underlying equations

The equations that will have to be solved and the method of their solution are identical to the previous section 3.1. The only principal difference is that the electric field will be zero and the magnetic field will be a strong function of position and will be approximated by the field of a magnetic dipole with constant magnetic dipole moment **m**.

R As mentioned above, we approximate the geomagnetic field by a field of a dipole. This is a very rough approximation which is valid only up to the distance of several R_e (Earth radii). Above that the magnetic field is strongly non-symmetrical due to its interactions with the solar wind. Most of the scientific activity in this field, both experimental and theoretical, is endorsed by NASA under the **Living with a star program** (lws.gsfc.nasa.gov).

Some very advanced theoretical models of the geomagnetic field and space weather are available at ccmc.gsfc.nasa.gov.

In the vector form, the magnetic field of a dipole is

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi}\left(\frac{3\mathbf{r}(\mathbf{m}\cdot\mathbf{r})}{r^5} - \frac{\mathbf{m}}{r^3}\right) \tag{3.26}$$

where $\mathbf{m}$ is the magnetic moment, μ_0 the vacuum permeability and

$$\mathbf{r} = (x, y, z), \tag{3.27}$$
$$r = |\mathbf{r}|\,. \tag{3.28}$$

Exercise 3.4 We will work in a Cartesian coordinate system with the origin at the center of the Earth and the z-axis identical with the Earth's axis. In this system, the magnetic moment of the planet is

$$\mathbf{m} = (0, 0, M). \tag{3.29}$$

- Express components of the magnetic field $\mathbf{B} = (B_x, B_y, B_z)$ in Cartesian coordinates.
- What is the value of M if the geomagnetic field at the equator is $3.12 \cdot 10^{-5}$ T?

■

3.2.2 Implementation

Just like in the previous case, we have to define the function which will return the right hand sides of our system of ODEs. The corresponding function m-file, `odefun.m` looks very similar to the previous cases.

Program 2: odefun.m

```
function dqdt = odefun(t,q),                                          1
        % This is the input function for the                          2
        % ordinary differential equation solver ode45.               3
        % Input: independent variable t(time), 6-dim vector          4
           q
        % q = [x,y,z,v_x,v_y,v_z]                                     5
        % Output: time derivative of q, dqdt = dq/dt                  6
                                                                      7
        % In solve.m, we define some global variables. Now,          8
           we need to tell
        % Matlab, that we will use "m" and "qe"                       9
        global m; global qe;                                          10
                                                                      11
        % Re-naming position and velocity variables                   12
        x = q(1);                                                     13
        y = q(2);                                                     14
        z = q(3);                                                     15
        vx = q(4);                                                    16
        vy = q(5);                                                    17
        vz = q(6);                                                    18
                                                                      19
```

```
% The electric field is zero here
Ex = 0;
Ey = 0;
Ez = 0;

% The magnetic field is now calculated using another
    function, bfield(r) which is a function of
   position. The function is stored in bfield.m
r0 = [q(1), q(2), q(3)];  % The position vector
B = bfield(r0);   % the function returns a vector B
% ... and we use this vector to define our component
   -wise B-field variables
Bx = B(1);
By = B(2);
Bz = B(3);

% From here on, it is~similar to Program 1
dxdt = vx;       % dx/dt = v_x
dydt = vy;       % dy/dt = v_y
dzdt = vz;       % dz/dt = v_z
dvxdt = qe/m*(Ex + vy*Bz - vz*By); % dv_x/dt
dvydt = qe/m*(Ey + vz*Bx - vx*Bz); % dv_y/dt
dvzdt = qe/m*(Ez + vx*By - vy*Bx); % dv_z/dt
dqdt = [dxdt; dydt; dzdt; dvxdt; dvydt; dvzdt];

end % end of function
```

The first difference is found on line 10. Instead of setting the variables `m` and `qe` to some particular value, we defined them as **global variables**. If you are not familiar with the concept of global and local variables, see the remark below.

A global variable is a variable that is available in all functions and programs where it is declared global. Normal variables (local) are not shared (inherited) between functions and programs.

Imagine that you want to use the variable `m` both in `solve.m` and `odefun.m`

- If you use **local variables**, you have to assign the value of the variable, i.e. `m = 1.627e-27;`, both in `odefun.m` and in `solve.m`. Therefore, you have to **write the numerical value into every file**.
- If you use **global variables**, you can set the value `m = 1.627e-27;` only in `solve.m` and you declare the variable as global by writing `global m;` both to `odefun.m` and in `solve.m`. Therefore, the **numerical value is written in one file only**.

The advantage of the second approach is apparent. If you want to change the mass `m`, you only have to re-write it once.

The second difference in `odefun.m` is around line 26. You can see that the magnetic field is not defined explicitly but it is obtained from an external function `bfield()`. The function is stored in the file `bfield.m`, it also uses some global variables (also defined in `solve.m`) and you have to complete it by writing correct expressions for B_x, B_y, B_z.

Program 2: bfield.m

```
function B = bfield(r0),
```

```
        % The function calculates the geomagnetic field B at
            the position r0. It works with Cartesian
           coordinates.
        % The function uses the following global variables:
        global Re; global q; global m; global mu0; global M;

        % If you did the exercises, the lines below should
           be clear
        x = r0(1);
        y = r0(2);
        z = r0(3);
        r = sqrt(x^2+y^2+z^2);

        % Complete the following based on the exercise above
        Bx = XXX;
        By = YYY;
        Bz = ZZZ;

        B = [Bx; By; Bz];
end
```

Finally, we need a script file, which includes all the necessary variables and calls the ode45 solver.

Program 2: solve.m

```
% This matlab script solves the equation of motion
% for a charged particle in Earth's magnetic field.

% To make the program run faster, we will define all our
% constants as global variables.
global Re; global m; global qe; global mu0; global M;
Re = 6378137;     % Earth radius in meters
m = 1.627e-27;    % Particle mass in kg
qe = 1.602e-19;   % particle charge in Coulomb
esc = 1;          % Earth scale - for plotting
c = 299792458;    % Speed of light in m/s
mu0 = 4*pi*1e-7; % Vacuum permeability in V*s/(A*m)
M = 7.94e22;      % Earth's magnetic moment in H/m

% Initial conditions for position
% Defined in spherical coordinates for convenience ...
r0 = 3*Re;
phi0 = 0;
theta0 = pi/2;
% ... and transformed to Cartesian.
x0 = r0*sin(theta0)*cos(phi0)
y0 = r0*sin(theta0)*sin(phi0)
z0 = r0*cos(theta0)

% Initial conditions for velocity
```

```
% defined using particle energy and two angles
Ek_eV = 5e7;                % energy in eV
Ek = Ek_eV*1.602e-19;       % energy in Joule
v_r0 = c*(1+m*c^2/Ek)^-0.5 % relativistic velocity magnitude
v_phi0 = 0;                 % azimuthal angle in rad
v_theta0 = pi/4;            % polar angle in rad

% Coordinate transformation for velocity.
vx0 = v_r0*sin(v_theta0)*cos(v_phi0)
vy0 = v_r0*sin(v_theta0)*sin(v_phi0)
vz0 = v_r0*cos(v_theta0)

% Defining the vector of initial conditions
q0 = [x0;y0;z0;vx0;vy0;vz0];

% Defining the time interval
ti = 0;          % initial time in seconds
tf = 20;         % final time in seconds
N = 10000;       % number of steps
timespan = linspace(ti,tf,N);

% Solving the ODE.
% Please note that odefun(t,q) is different from Program 1
[t, q] = ode45(@odefun, timespan, q0);


% Having solved the equation, the data has to be plotted.
% It is not necessary to understand what each of the
   commands
% below does.

h = figure;      % Creates a new figure

% The following set of commands plots the magnetic lines of
   force. [Adapted from MagLForce script by  A. Abokhodair]
n = 4;
d2r = pi/180; r2d=1/d2r;
tht=d2r*(0:5:360)';
phi=d2r*(0:ceil(180/n):180);
hh=phi*r2d;
A=r0;
r=A*sin(tht).^2;
rho=r.*sin(tht);
x=rho*cos(phi); y=rho*sin(phi);
[nR,nC]=size(x);
u=ones(1,nC); z=r.*cos(tht)*u;
plot3(x/Re,y/Re,z/Re,'r','LineWidth',1.0);

hold on; % This tells Matlab that whatever you are going to
```

```
   plot next will be included in the same figure!!!

% Plotting an ellipsoid, representing the Earth.
[u,v,w]=sphere(30);
surf(esc*u, esc*v, esc*0.9*w);
colormap('default');
camlight right;
lighting phong;
axis equal % This forces the axes to have the same scale!

% Plotting the particle trajectory and add labels to axes.
plot3(q(:,1)/Re, q(:,2)/Re, q(:,3)/Re,'LineWidth',1.0)
set(gca,'FontSize',18) % sets font size
xlabel('x [R_e]') % sets title of x-axis
ylabel('y [R_e]') % sets title of y-axis
zlabel('z [R_e]') % sets title of z-axis
title(['Proton trajectory, E = 50 MeV, t_i=', num2str(ti), '
    s, t_f=', num2str(tf), ' s']) % sets figure title

print -dpng -r200 'ProtonMagField.png' % prints the figure
   to file using the png format with 200 dpi resolution.
```

The `solve.m` file is quite similar to the one we used in the previous one. On lines 6 to 13, global variables are defined. It is best practice that **global variables declared** global and assigned their values **at the beginning of the first file that is executed**. On lines 15 to 39, the initial conditions are defined. To make the them more intuitive, the initial conditions are defined in spherical coordinates and then transformed to Cartesian ones. The differential equation is solved on line 49 and from line 57 onwards, the plotting commands are listed. Please note the usage of the `hold on` command, which tells Matlab that all the next plots will be added to the same figure. Also note that **all data that is plotted is scaled** by the Earth's radius `Re`.

If you completed the expression for the magnetic field correctly, running the program with the pre-defined values should give a plot similar to figure 3.3.

Exercise 3.5 Analyze the motion of the high-energy proton in the geomagnetic field. What are the three components of the motion? ■

Exercise 3.6 Even though our model is quite simple, it can be helpful in understanding the basic structures of the radiation belts. Try changing the initial position of the proton.

- What is the maximum initial distance `r0`, for which the proton still has a stable trajectory?
- What is the minimum initial distance, for which the proton does not hit the surface of the Earth?

■

Advanced exercise 3.2 Replace the proton in the previous model with an electron and try to find the initial conditions (especially the distance from the Earth), for which the electron will have stable trajectory. Think before you code, will electrons require higher or lower magnetic field to be confined? How does the drift of an electron differ from that of a proton? ■

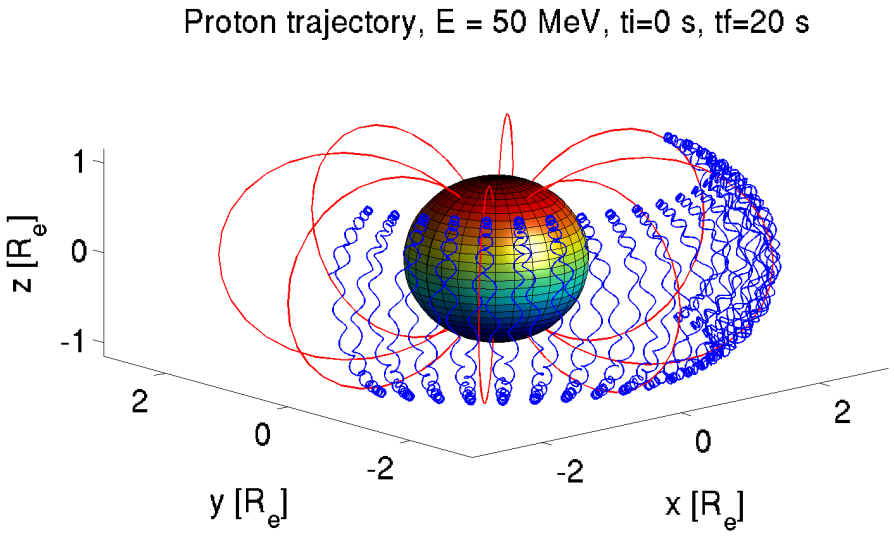


Figure 3.3: Motion of a high-energy proton in the geomagnetic field

4. Data processing

4.1 Introduction

The Matlab skills that you will learn in this chapter will be useful also outside plasma physics. In particular, the chapter focuses on data processing, including interpolation and integration of discrete data. Furthermore, you will learn to use so-called *loops* and *conditions* in order to batch-process large sets of data.

4.2 Loops and conditions in Matlab

Let us first look at two very important elements of Matlab programming and programming in general, *loops* and *conditions*. If you have previous experience with programming, this section will probably seem quite trivial but if you do not, you will learn skills that you will almost certainly find useful in your future career of a scientist.

Loops are structures that allow you to repeat a certain sequence of instruction multiple times. Such repetitive tasks in physics may include

- plotting multiple data files in the same format,
- converting large amounts of data,
- solving a differential equation for multiple sets of initial/boundary conditions.
- . . .

There are two types of loops in Matlab, the *for* loop and the *while* loop. The former executes the specified sequence of commands N times (e.g. “integrate the data in first 50 files”) while the latter keeps executing the sequence as long as some expression is true (e.g. “keep integrating the data until you reach an integral larger than 10”). However, the range of applications of the *while* loop is very narrow and it is quite easy to make it endless. For this reason, we will only focus on the *for* loop, which is completely sufficient for purposes of most physicists.

Conditions are used when the sequence of commands that you want to execute depends on whether some statement is true or false. They are specially useful in combination with loops. An example you can think about is automated plotting; you can create a program that will plot the data in logarithmic scale if the difference between the lowest and the highest values is larger than two orders of magnitude.

4.2.1 For loop

As already mentioned, the *for* loops makes it possible to perform a certain operation N times. The loop is used as follows.

```
for j=1:10,
  % your commands
end
```

The loop above will run ten times. The variable `j` is called the *loop variable* and you can use it inside the loop. Let us now write a program that actually does something at least a little useful. In particular, we will create plots of $y = x^n$ with $n \in \langle 1,6 \rangle$ and $x \in \langle 0,1 \rangle$.

```
figure;  % creates figure
colors = ['r', 'g', 'b', 'm', 'k', 'y', 'c', 'r']; % this is
    a vector of strings (texts)
for j=1:8,   % will run for 8 values of j
  x = linspace(0,1,50); % we saw this before, creates a
     linearly spaced vector from 0 to 1 with 50 values.
  color = colors(j); % selects j-th element from colors
  plot(x, x.^j, color);
  hold on;   % we want everything in the same figure
end
```

Exercise 4.1 Modify the program above so that it plots

$$y = \frac{\sin(nx)}{x} \qquad \text{for } x \in \langle 0, 2\pi \rangle \text{ and } n \in \langle 1,5 \rangle \tag{4.1}$$

Do not forget that `x` is a vector and you have to use element-wise operations. ■

In plasma physics as well as in other disciplines of physics, one often has to work with data obtained from literature and often it happens that the data have a different unit than you need. The next exercise program serves as a batch-converter of collisional cross-sections which have been specified in the unit of cm^2 to the unit of m^2. In order to write the program, we are going to need two more useful functions, `load()` and `dlmwrite()`.

The **load** function is very intuitive, it loads data into a matlab variable from a file. The input data file can have several formats (tab-separated data, comma-separated data, etc...) and it is loaded to a variable by the command

```
matrix = load('datafile.dat');
```

On the other hand, the function for saving data to an ASCII file is equally straightforward.

```
dlmwrite('new_datafile.dat', matrix, '\t');
```

As you can see, it takes three parameters, namely the output file, the variable that is saved to the file and the delimiter delimiting columns in the output file. Typically used delimiters are `','` (comma), `';'` (semicolon) and `'\t'` (tab).

In the next exercise, you also need to know how to work with *strings* in Matlab. *String* is merely a programming term for a text. Since *string* in Matlab is basically a vector of letters, you can use the same syntax with the exception that strings have to be enclosed in quotation marks (`'`).

Let us demonstrate how to join strings on a simple example. The following script will open and plot the content of files `csk1.dat` to `csk3.dat` (these files can be found in the `Prog3_cross_section_convert` directory).

```
colors = ['r', 'g', 'b', 'm', 'k', 'y', 'c', 'r'];
figure;
for j=1:3,
  color = colors(j);
  matrix = load(['csk', num2str(j), '.dat']);
  xdata = matrix(:,1); % first column
  ydata = matrix(:,2); % second column
  plot(xdata, ydata, color);
  hold on;
end
xlim([0,1e6]); % set x-axis limits
```

The data you just plotted are not random curves, they are cross sections of collisions of electrons with argon atoms. The x-axis is the electron temperature in Kelvin while the y-axis is the collisional cross-section in m^2. In particular, the `csk1.dat` file contains the data for an elastic collision

$$e + Ar \rightarrow e + Ar, \tag{4.2}$$

`csk2.dat` the data for argon excitation

$$e + Ar \rightarrow e + Ar^*, \tag{4.3}$$

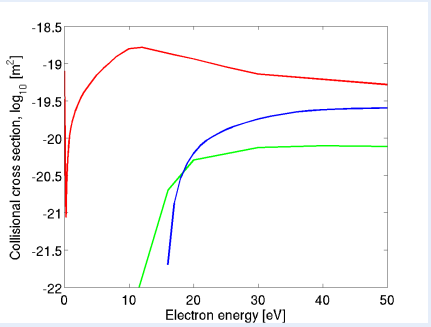
and `csk3.dat` the data for ionization

$$e + Ar \rightarrow 2e + Ar^+. \tag{4.4}$$

Exercise 4.2 The program above produces a plot, which is not very intuitive mainly because the magnitudes of the plots are very different. Modify the program so that

- the electron energy is expressed in the unit of eV,
- the y-axis scale is logarithmic.

The resulting plot should look comparable to this one



with the red line corresponding to the elastic collision, the green line to excitation and the blue line to ionization.

- Look at the picture, what are the excitation and ionization thresholds?

■

Exercise 4.3 Modify the plotting program so that it converts the x-data from Kelvin to electronvolt and saves each cross-section to tab-delimited `csevN.dat` with `N` being the corresponding file number. ■

4.2.2 If . . . elif . . . else conditioning

In the previous section, you learned about *for* loops which allow you to perform the same task repetitively. You can further step up your programs by using *conditions*. By including conditions in your programs, you can a certain set of commands if a statement is true and an another set of commands if it is false.

Let's assume that you are developing a very complicated program, in which you often need to find inverse matrices. As you know, the inverse matrix A^{-1} to matrix A is defined as

$$A^{-1}A = I \tag{4.5}$$

where I is the identity matrix.

Exercise 4.4 The Matlab function which allows you to calculate the inverse matrix is called `inv()` and its only argument is the matrix that you want to invert.

- Define a 3x3 singular matrix A (determinant is zero) and try to invert it using `inv()`. What does the resulting matrix look like?

■

As you saw in the exercise above, trying to calculate inverse matrix to a singular matrix does not lead to any useful result. It would be much more practical to define a custom function which will perform the inversion **only if the input matrix is not singular**. If the matrix is singular, your function will display an error and stop your program. The function we just described could look something like this

```
function inverted = myinv(matrix),                                    1
        rows = length(matrix(:,1)); % calculates the length          2
           of the first column, i.e. the number of rows
        cols = length(matrix(1,:)); % calculates the length          3
           of the first row, i.e. the number of columns
                                                                      4
        if(rows ~= cols), % if rows DOES NOT EQUAL cols               5
                disp('The supplied matrix is not a square            6
                   matrix')
                return % terminates the script                        7
        elseif(det(matrix) == 0); % if determinant is zero           8
                disp('The matrix is singular.')                       9
                return % terminates the script                        10
        else, % if both the conditions above are false                11
                inverted = inv(matrix); % performs inversion         12
                    and returns the inverted matrix
        end                                                           13
end                                                                   14
```

The syntax of the `if...elseif...else` conditioning should now be evident from the code above. You can see that the statement that needs to be evaluated is written in round brackets after `if` or `elseif`. Matlab tests the conditions from up to bottom, i.e.

1. The shape of the matrix is tested first. If not square, the error is displayed.

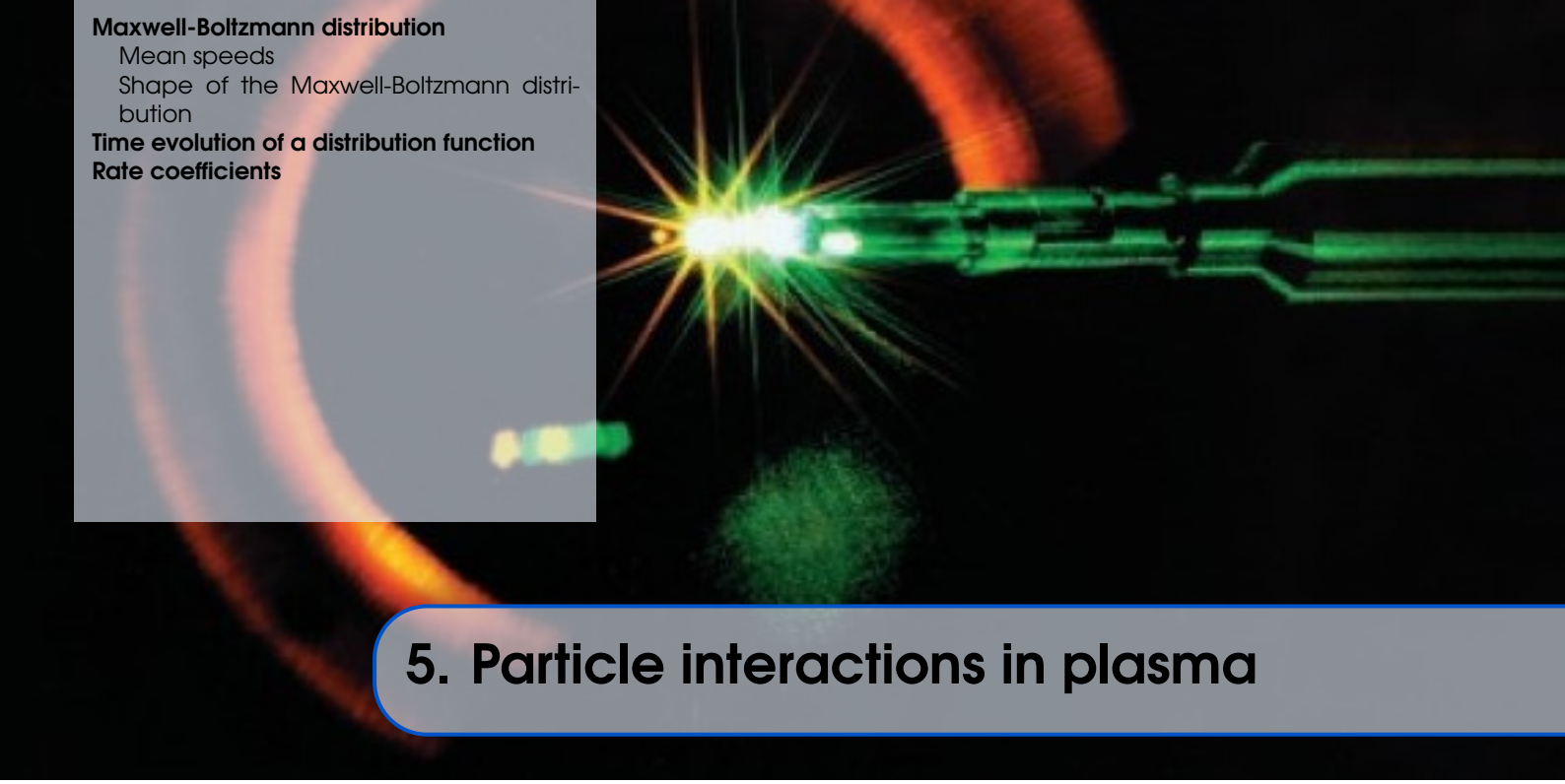
2. If the matrix is square, the determinant is tested. If zero, the error is displayed
3. **Only if the matrix is square and non singular**, the inversion is performed.

Exercise 4.5 Verify that the function `myint()` behaves correctly, try entering several non-square matrices and several singular matrices. Add another `elseif` statement which will display a warning if your matrix rank is larger than 10. Test if it works. ■

Advanced exercise 4.1 Look into the directory `Prog4_cross_section_convert_if` which contains three files with collisional cross-sections. Two of these files include electron temperature in eV while one of them includes electron energy in Kelvin.

- Modify the program that we used in exercise 4.2 using a reasonable `if...else` condition so that it decides which files should be converted and which not.
- A comparable simple piece of code could be very useful when compiling a collisional-radiative plasma model with thousands of similar files that need to be pre-processed. However, what are the limitations of your program?

■



5. Particle interactions in plasma

This chapter illustrates the differences between equilibrium and non-equilibrium distribution functions and shows the impact of the shape of the distribution function on macroscopic variables. In this chapter, you will learn basics of **numerical integration** and **interpolation** in Matlab.

5.1 Maxwell-Boltzmann distribution

5.1.1 Mean speeds

In the lecture, you learned that under equilibrium conditions, the velocity of particles is governed by the **Maxwell-Boltzmann** distribution. In particular, the distribution function for the velocity magnitude $F(v)$ is

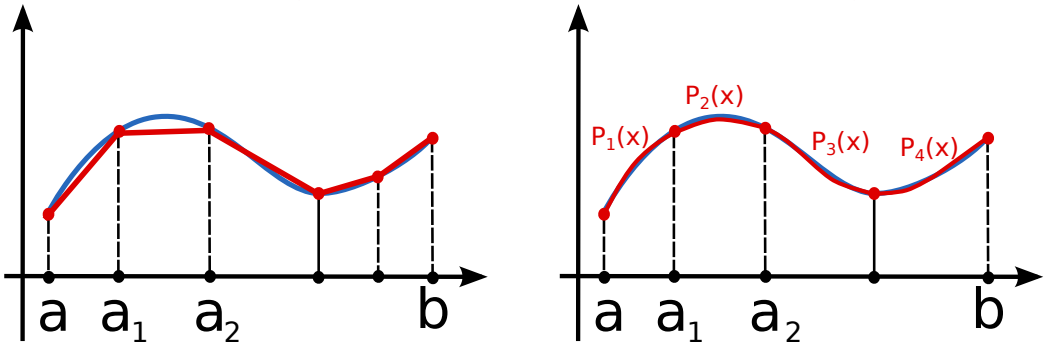
$$F(v) = 4\pi n v^2 \left(\frac{m}{2\pi kT}\right)^{3/2} \exp\left(\frac{-mv^2}{2kT}\right). \tag{5.1}$$

Distribution functions, especially those of electrons, can tell you a lot about your plasma, if you know how to read and interpret them. In particular, it is very important to understand what influence do distribution functions of different shapes have on macroscopic variables that everyone can intuitively understand. The macroscopic variables are very often given by quite simple integrals of distribution functions. Therefore, this section will teach you how to numerically integrate functions in Matlab.

R Some of the integrals that we will evaluate numerically in this section also have an analytical solution. However, in most laboratory plasmas, the distribution functions of some particles (especially electrons) can not be described by an analytical expression. Hence we will prefer numerical integration over analytical. Again, knowing how to integrate data numerically will be useful for you even in other disciplines of physics.

There are several numerical techniques for numerical integration, the simplest one is the **trapezoidal rule** that you may remember from mathematical analysis lectures. In the trapezoidal rule, the function is approximated by trapezoids and the total surface area of the trapezoids gives you the estimate of the integral of the function between points a and b. The trapezoidal rule is schematically illustrated in figure 5.1(a). A more advanced method is the **Simpson's rule**

which approximates the function $f(x)$ to be integrated by several consecutive polynomials $P_i(x)$ (see figure 5.1(b)). Therefore, the Simpson's rule can achieve better accuracy, especially with fast-varying functions.



(a) Trapezoidal rule (b) Simpson's rule

Figure 5.1: An illustration of the trapezoidal rule for numerical integration (left) and the Simpson rule (right). In the Simpson rule. the upper side of the "trapezoids" is not linear but rather given by a polynomial, which can be integrated analytically.

Figure 5.1(a) also suggests, that you can make the trapezoidal rule arbitrarily accurate by decreasing the size of the trapezoids. Therefore, **the trapezoidal rule will be sufficient for our purposes**. The trapezoidal rule integration algorithm is implemented in Matlab under the name `trapz()`. The function takes two vectors as arguments. The first contains the x-data while the second contains f(x).

```
x = linspace(0, pi, 100);
fx = sin(x.^2)./log(x);
I = trapz(x, fx);
```

You do not have to specify the lower and upper bounds of the definite integral because these are given by the minimum and maximum values of the vector `x`. In other words, the two lines above calculate the following definite integral

$$I = \int_{x_{\min}}^{x_{\max}} \frac{\sin(x^2)}{x}\,\mathrm{d}x = \int_0^{\pi} \frac{\sin(x^2)}{x}\,\mathrm{d}x \tag{5.2}$$

In the following exercise, you will learn to use the `trapz()` function and evaluate its accuracy on something that can be verified analytically.

Exercise 5.1 Program 5 below is a template for plotting the Maxwell-Boltzmann distribution and for calculating and plotting the mean speeds of the distribution function. Complete the program by

- filling in the expression for $F(v)$ on line 7 (speed distribution function in 3D),
- calculating the mean speed on line 12 and mean quadratic speed on line 13 (both using the trapezoidal rule),
- providing analytical expressions for mean speed, mean quadratic speed and most probable speed on lines 16 to 18.

Once you complete the program, answer the following questions.

- Do the numerical results agree with the analytical ones?

- Look at the plot. Which of the three speeds is the lowest, which is the highest? Does their order change when you increase/decrease the temperature?
- Try changing setting the number of steps in `linspace()` to 10, 50, 100, 500. What can you say about the agreement of numerical and analytical results?

■

Program 5: Maxwell-Boltzmann distribution and speed

```
m0 = 1.6605e-27; % a.m.u. in kilograms
m = 40*m0;     % particle mass in kilograms
kB = 1.380e-23;  % Boltzmann constant in m^2 kg s^-2 K^-1
T = 1000;        % temperature in Kelvin

v = linspace(0,4000,500); % x data
Fv = ...; % distribution function
plot(v, Fv);
hold on;

% numerical expressions
v_mean = ...; % mean speed
v_sq_mean = ...; % mean quadratic speed

% analytical expressions
v_mean_an = ...; % mean speed
v_sq_mean_an = ...; % mean quadratic speed
v_mp_an = ...; % most probable speed

lineht = max(Fv)*1.1;   % line height, only for plotting
plot([v_mean, v_mean], [0, lineht], 'r');       % mean speed
    (red)
plot([v_sq_mean, v_sq_mean], [0, lineht], 'g'); % mean
   quadratic speed (green)
plot([v_mp_an, v_mp_an], [0, lineht], 'm');     % most
   probable speed (pink)
hold off
```

Advanced exercise 5.1 In the program above, we calculated the mean velocity and the mean quadratic velocity numerically while the most probable velocity was only calculated analytically. To calculate it numerically, you would have to find a maximum of our distribution function (which is given by vectors `v` and `Fv` in our program). Use the internet to figure out a way to do it. ■

As you know from the lecture, the distribution function is normalized so that

$$\int_0^\infty F(v)\mathrm{d}v = n. \tag{5.3}$$

Therefore, when you integrate the distribution function from some velocity v_0 to ∞, you obtain the number of particles which have velocity greater or equal to v_0. If you integrate $F(v)$ from v_0 to v_1, you obtain the number of particles with velocities in the interval $\langle v_0, v_1 \rangle$. With this in mind, try to complete the exercise 5.2.

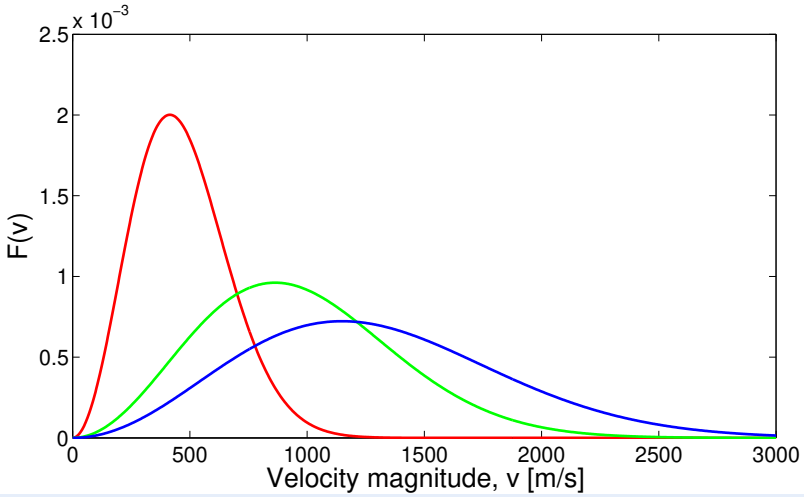
Exercise 5.2 The mass of nitrogen molecules is 28 a.m.u. and their number density at ambient conditions is approximately $1.7 \cdot 10^{25}$ m^{-3}. How many nitrogen molecules in your room are faster than 50, 500, 1000, 2500, 5000 and 10000 m/s? ■

5.1.2 Shape of the Maxwell-Boltzmann distribution

You have probably noticed that the equilibrium **Maxwell-Boltzmann distribution function is given only by two macroscopic parameters**, in particular the mass of a particle m and the temperature T. The exercise below does not require any programming but if you do not know the answer right away, you will find the program in the previous section very helpful.

Exercise 5.3 The following figure shows three equilibrium distributions of speed with unknown parameters m and T.

- If the particle mass m is constant, which of the distribution functions will corresponds to lowest temperature and which to highest?
- If, on the other hand, the temperature T is constant, which of the distribution functions corresponds to lowest and highest particle mass?



■

5.2 Time evolution of a distribution function

In the previous section, we thoroughly analyzed the key properties of the Maxwell-Boltzmann distribution. However, the Maxwell-Boltzmann distribution is only achieved in equilibrium and prior to achieving equilibrium, the distribution function can be almost arbitrary. In addition, we only took into account homogeneous and isotropic distribution functions in the previous section (i.e. depends only on velocity magnitude). In this section, we will still assume that the space is homogeneous (i.e. the distribution function does not depend on the coordinate $\mathbf{r}$) but now, the distribution function can also be anisotropic (i.e. can depend on the velocity vector $\mathbf{v}$, not only on its magnitude).

The program that you will use for analyzing the time evolution of the distribution function is not very complicated. It is based on the **simplest non-equilibrium formulation of the Boltzmann kinetic equation**. In this approximation, we assume that the external force acting on the particles is zero,

$$\mathbf{F} = 0, \tag{5.4}$$

and the collision term is expressed in the Krook form

$$\left(\frac{\partial f}{\partial t}\right)_{\text{coll}} = -\frac{f - f_0}{\tau} = -\nu_{\text{m}} \cdot (f - f_0) \tag{5.5}$$

where f_0 is the equilibrium distribution function, τ is the relaxation time and ν_{m} is the collision frequency. The whole Boltzmann equation, therefore, reduces to

$$\frac{\partial f(\mathbf{v},t)}{\partial t} = -\nu_{\text{m}} \cdot (f(\mathbf{v},t) - f_0(\mathbf{v})) \tag{5.6}$$

which has an analytical solution of

$$f(\mathbf{v},t) = f_0(\mathbf{v}) + [f(\mathbf{v},0) - f_0(\mathbf{v})]\, e^{-\nu_{\text{m}} t}. \tag{5.7}$$

Therefore, the program actually does not solve the differential equation but only plots the solution in an intuitive way. The source code with extensive comments is listed below. The following lines are crucial for you when using the program

- **Line 5** sets the time interval, on which the solution is plotted
- **Line 6** sets the velocity interval, similar to the previous program
- **Lines 11 to 13** set the initial distribution function.
- **Line 32** changes how the plots scale. If it is commented out, the plots will have variable scale, if you uncomment it, the scale will be constant.

Program 6: Plotting time evolution of a distribution function

```
m0 = 1.6605e-27; % a.m.u. in kilograms                                    1
m = 40*m0;     % particle mass in kilograms                               2
kB = 1.380e-23;  % Boltzmann constant in m^2 kg s^-2 K^-1                 3
Vm = 5e8;     % collision frequency, Hz                                   4
time = linspace(0, 1e-8, 100); % time interval                            5
v = linspace(-2000,2000,200); % velocity interval                         6
                                                                          7
% initial distribution function, f_0                                      8
sigma = 10; % Initial width                                               9
v0 = 500; % initial velocity                                              10
fx_init = 1/(sqrt(2*pi)*sigma)*exp(-(v-v0).^2/(2*sigma^2));               11
fy_init = v*0;                                                            12
fz_init = v*0;                                                            13
                                                                          14
% Calculating the equilibrium temperature, the following                  15
   follows from energy conservation.
v_sq = sqrt(trapz(v, (fx_init+fy_init+fz_init).*v.^2));                   16
Teq = 1/3*m*v_sq^2/kB;                                                    17
                                                                          18
% equilibrium distribution function                                       19
fx_eq = sqrt(m/(2*pi*kB*Teq)) * exp(-m*v.^2/(2*kB*Teq));                  20
fy_eq = sqrt(m/(2*pi*kB*Teq)) * exp(-m*v.^2/(2*kB*Teq));                  21
fz_eq = sqrt(m/(2*pi*kB*Teq)) * exp(-m*v.^2/(2*kB*Teq));                  22
                                                                          23
fx = fx_init;                                                             24
fy = fy_init;                                                             25
fz = fz_init;                                                             26
```

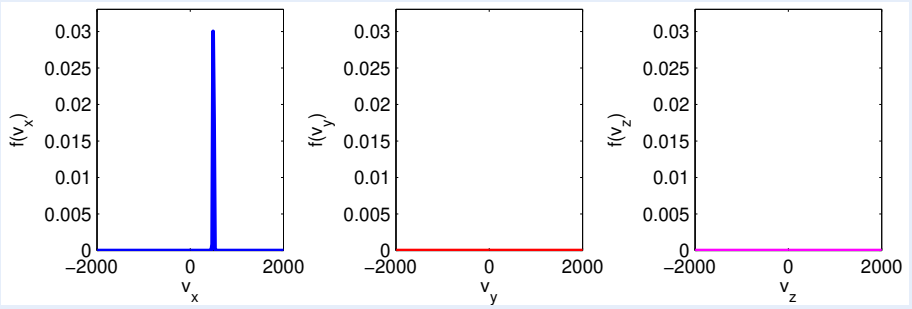
```
hFig = figure; % creating the figure, setting dimensions
set(hFig, 'Position', [100 300 1000 300]);
for j=1:length(time), % for loop making the time steps
    absmax = max([fx, fy, fz])*1.1; % variable y-axis scale
    % absmax = max([max(fx_init),max(fy_init),max(fz_init)])
       ; % uncomment this line for fixed y-axis scale
    subplot(1,3,1); % first subplot - x
    plot(v, fx, 'b', 'LineWidth', 2);
    ylabel('f(v_x)');
    xlabel('v_x');
    ylim([0, absmax]);
    whitebg('white')
    subplot(1,3,2); % second subplot - y
    plot(v, fy, 'r', 'LineWidth', 2);
    ylabel('f(v_y)');
    xlabel('v_y');
    ylim([0, absmax])
    subplot(1,3,3); % third subplot - z
    plot(v, fz, 'm', 'LineWidth', 2);
    ylabel('f(v_z)');
    xlabel('v_z');
    ylim([0, absmax]);
    % setting figure title - time
    ax=axes('Units','Normal','Position',[.075 .075 .85 .85],
       'Visible','off');
    set(get(ax,'Title'),'Visible','on')
    title(['t = ', num2str(time(j)), ' s']);
    % calculating new values of fx, fy, fz
    fx = fx_eq + (fx_init - fx_eq)*exp(-time(j)*Vm);
    fy = fy_eq + (fy_init - fy_eq)*exp(-time(j)*Vm);
    fz = fz_eq + (fy_init - fz_eq)*exp(-time(j)*Vm);
    M(j) = getframe(gcf); % appends a frame to variable M
end
movie2avi(M, 'out.avi', 'fps', 7); % saves the movie
```

Exercise 5.4 Run the program above, watch the output animation and answer the following questions

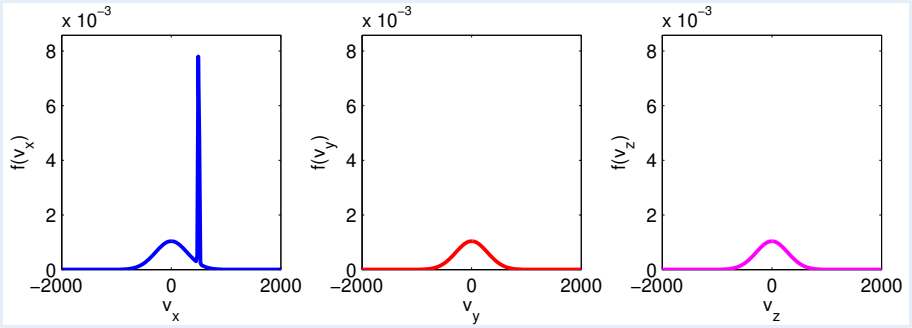
- What is the physical meaning of the initial condition for the distribution function?
- What kind of particles could the distribution functions describe?
- The collision frequency is $5 \cdot 10^8$ Hz, which is a reasonable value. What is the time necessary for reaching the equilibrium?
- Try increasing and decreasing the collision frequency in Program 6. What happens with the time necessary for reaching equilibrium and why?

Here is an example of what the output should look like at various times. Please note that the y-axis scale is changing.

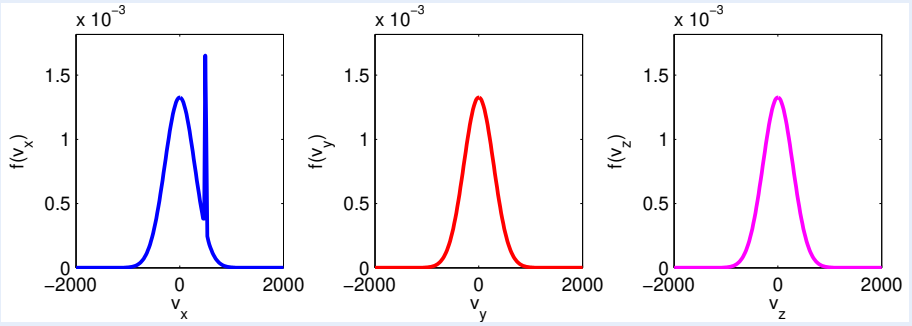
$t = 0$ ns



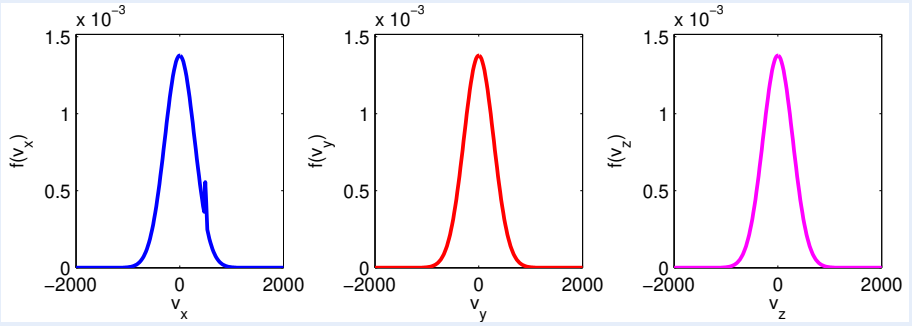
$t = 3$ ns



$t = 6.5$ ns



$t = 10$ ns



■

5.3 Rate coefficients

Finally, let us demonstrate how different distribution functions can influence macroscopic properties, such as the rate coefficients of plasma-chemical reactions. The rate coefficient of a reaction is calculated using the collisional cross section σ_r as

$$k_r = \langle \sigma_r(v)v \rangle = \int_0^\infty \sigma_r(v)vF(v)\mathrm{d}v. \tag{5.8}$$

As you can see, we will again be working only with the distribution function for speed because it is reasonable to assume that the cross-section of a particular reaction does not depend on the whole velocity vector but only on its magnitude.

When calculating the integral (5.8), there is one technical issue that has to be overcome, the interpolation of the collisional cross-section $\sigma_r(v)$. **Cross-sections of most reactions can not be expressed analytically and are available only as tabulated data**. The interpolation is necessary because you need to compute the element-wise product $\sigma_r(v)vF(v)$ and $\sigma_r(v)$ and $F(v)$ do, generally, have different data sampling.

In Matlab, you can interpolate 1D data using the `interp1()` function. Let's say that you have a function given by vectors `xdata` (sample points) and `ydata` (values) and you want to obtain function values with much finer sampling. The syntax would be as follows

```
xdata = [1, 5, 10, 15, 20];
ydata = [1, 35, 110, 235, 410];

xinterp = linspace(8,17,0.5);
yinterp = interp1(xdata, ydata, xinterp, 'pchip', 0);
```

You can see that the `interp1()` function has five parameters, the vector of sample points (`xdata`), the vector of function values (`ydata`), the vector of new sample points (`xinterp`) and the interpolation method. The fifth parameter (zero) means that no extrapolation is performed and **for all points outside the interval given by `xdata`, the function value is zero.**

With `interp1()`, you can choose from several interpolation methods. The most popular choices are `'nearest'` which interpolates the data based on the nearest neighbor, `'linear'` which performs linear interpolation, `'pchip'` which performs cubic interpolation and `'spline'` which interpolates the data using piecewise-continuous polynomials.

Let us now examine, what will happen if we try to calculate the rate coefficient for electron impact ionization of argon

$$\mathrm{e} + \mathrm{Ar} \rightarrow 2\mathrm{e} + \mathrm{Ar}^+. \tag{5.9}$$

The calculation will be performed for two distribution functions, the Maxwell-Stefan distribution function and for a nearly mono-energetic beam **with the same mean speed**. The program below should not contain anything we have not encountered before. Its output are two numbers, the rate coefficient for Maxwell-Boltzmann distribution and the rate coefficient for the mono-energetic beam.

"Program 7: Electron impact ionization"

```
m = 9.109e-31; % electron mass
kB = 1.380e-23;  % Boltzmann constant in m^2 kg s^-2 K^-1
TeV = 15; % electron temperature in eV
T = TeV*11604; % electron temperature in Kelvin
```

```
5
% loading the cross-section                                          6
sigdata = load('sigmaion.dat');                                      7
xsig = sigdata(:,1);                                                 8
ysig = sigdata(:,2);                                                 9
                                                                     10
v = linspace(0,2e7,1e6); % sampling for speed                        11
sigma = interp1(xsig, ysig, v, 'pchip', 0); % interpolating          12
   the cross-section
                                                                     13
% Maxwell-Boltzmann distribution                                     14
Fv = 4*pi*v.^2.*(m/(2*pi*kB*T))^1.5.*exp(-m*v.^2/(2*kB*T));          15
   % distribution function
kr_MB = trapz(v, v.*Fv.*sigma)                                       16
vmean = trapz(v, Fv.*v);                                             17
                                                                     18
% Distribution function of a nearly mono-energetic beam             19
v0 = vmean;                                                          20
width = vmean/100;                                                   21
Fv2 = 1/(sqrt(2*pi)* width) * exp(-(v-v0).^2/(2*width^2)); %         22
    distribution function
kr_beam = trapz(v, v .* Fv2 .* sigma)                                23
```

Exercise 5.5 Run the program and compare the two rate coefficients. You will notice that they are not the same, although the mean velocity is the same in both cases. Provide an explanation. ■

Exercise 5.6 Modify the program so that it plots $\sigma_r(v)$ and answer the following questions.

- What is the ionization threshold in electronvolts?
- Where does the cross-section reach the maximum value.

■

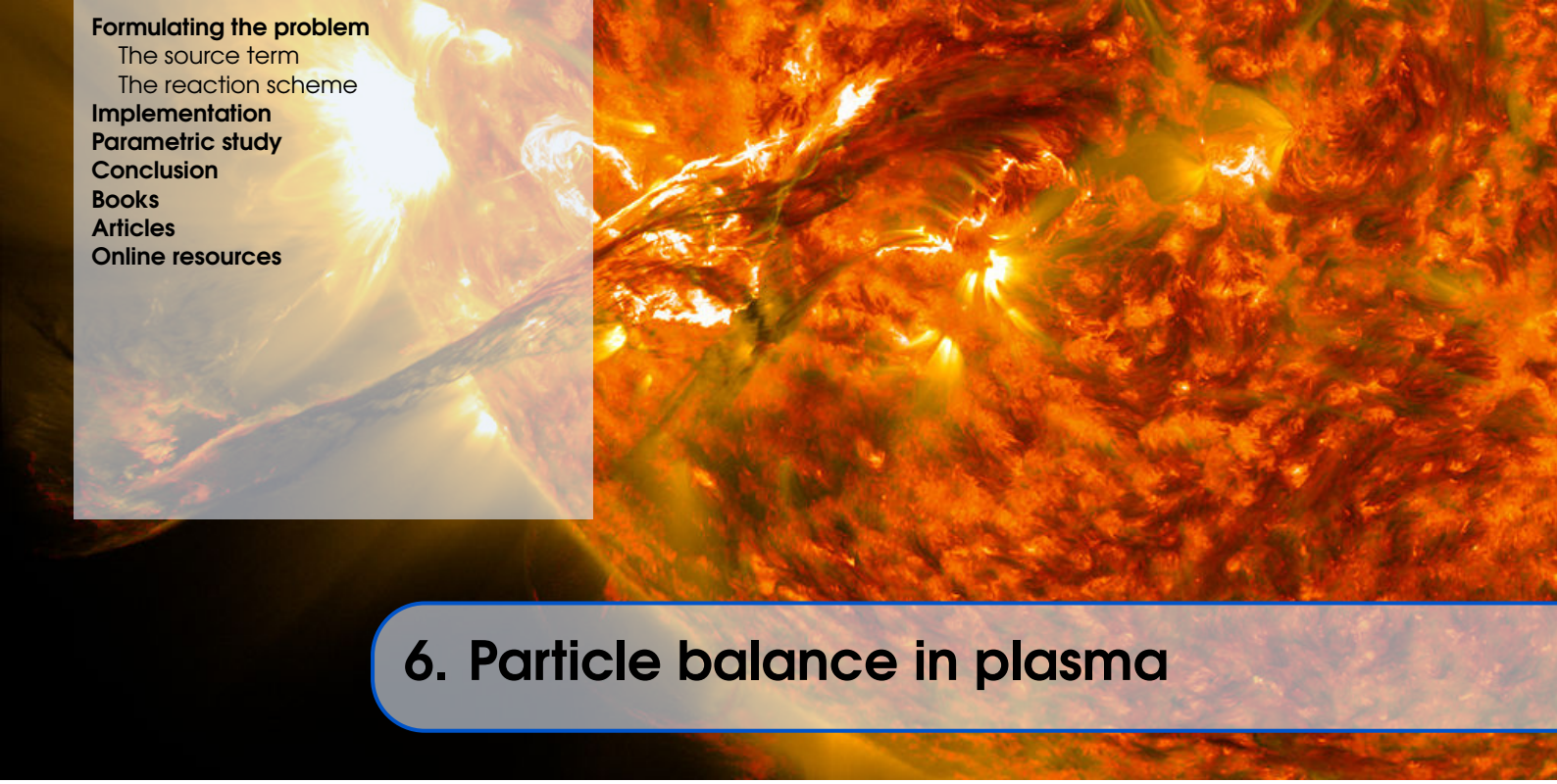
Exercise 5.7 Run the program for several values of electron temperature (defined in electronvolt on line 3).

- What happens with the rate coefficients with increasing electron temperature?
- Is there electron temperature for which the rate coefficient for the nearly mono-energetic beam exceeds the Maxwell-Boltzmann coefficient? Provide an explanation why this is/is not possible.

■

Advanced exercise 5.2 Rewrite Program 7 so that it uses electron energy rather than electron velocity for the calculations. Using electron energy is much more common in plasma physics. ■

R There are several publicly available databases, from which you can obtain collisional cross-sections. The most comprehensive one is probably LxCat, available at lxcat.net. Another quite extensive database is the ALADDIN database, available at www-amdis.iaea.org/ALADDIN/.



6. Particle balance in plasma

As you already know, **plasmas consist of many types of particles** (species). Apart from neutral ground-state atoms and molecules, there can be various excited species (as discussed in *Interactive Introduction to Plasma Physics: part I*), positive or negative ionic species and, of course, the electrons by which the plasma is sustained. Furthermore, each species can generally have different temperature.

Knowing the temperature of individual species and their concentration is very important for the applications of plasmas. For example, in biological applications of laboratory plasmas, it is crucial to maintain high electron temperature while keeping the temperature of heavy particles low.

6.1 Formulating the problem

In this last chapter of this text, you will implement a program that calculates the concentrations of individual particle species in atmospheric-pressure argon plasma for a given electron temperature T_e and gas temperature T_g. The model will be again implemented in Matlab and it will be **zero-dimensional**. Practically, this means that the number densities of each particle species are only a function of time, not position,

$$n_\alpha = n_\alpha(t). \tag{6.1}$$

6.1.1 The source term

Under the zero-dimensional approximation that we made, the continuity equation for species α, simplifies to an ordinary differential equation

$$\frac{\partial n_\alpha}{\partial t} = S_\alpha. \tag{6.2}$$

This equation looks relatively simple but it has to be pointed out that the source term S_α is a complicated function of temperatures (which are constant in our model) and number densities of other particle species.

Generally speaking, the source term for particle species α is a sum of contributions from all reactions that include the species α,

$$S_\alpha = \sum_r S_{\alpha,r}. \tag{6.3}$$

The contribution $S_{\alpha,r}$ represents how many particles of species α are produced or consumed in reaction r per unit time. The source term depends on the order of the reaction. In our model, we will consider two orders of reactions, **two-body reactions** and **three-body reactions**. As the names suggest, two-body reactions are reactions between two particles and three body reactions are reactions between three particles at the same time. However, keep in mind that the order of the reaction tells you only **how many reactants enter the reaction**, it does not say anything about the number of products. An example of a two body reaction is the following electron-impact ionization of argon (later, we will designate this reaction R4).

$$\mathrm{e} + \mathrm{Ar}^* \longrightarrow 2\mathrm{e} + \mathrm{Ar}^+ \qquad \text{(R4)} \tag{6.4}$$

In this reaction, an electron collides with an excited argon atom and provides the energy necessary for ionization. The reaction will, contribute to source terms of all the reactants and products, in particular

$$S_{\mathrm{e,R4}} = +k_1(T_\mathrm{e}, T_\mathrm{g}) \cdot n_\mathrm{e} \cdot n_{\mathrm{Ar}^*}, \tag{6.5}$$
$$S_{\mathrm{Ar}^+,\mathrm{R4}} = +k_1(T_\mathrm{e}, T_\mathrm{g}) \cdot n_\mathrm{e} \cdot n_{\mathrm{Ar}^*}, \tag{6.6}$$
$$S_{\mathrm{Ar}^*,\mathrm{R4}} = -k_1(T_\mathrm{e}, T_\mathrm{g}) \cdot n_\mathrm{e} \cdot n_{\mathrm{Ar}^*}. \tag{6.7}$$

You can see, that the source term contribution contains the rate coefficient, which can depend on the temperatures of electrons and heavy particles. Our model is a **two-temperature model**, in which the electron temperature is different but all the other types of particles (ground-state, ions, excited species) have the same temperature T_g. You can also see that the contributions differ by the $\pm$ sign, which is also quite straightforward. Since reaction (R4) **produces one new electron and one new ion**, the sign of the corresponding source terms is positive. On the other hand, the reaction **consumes one excited argon atom**, therefore the corresponding source term contribution has a negative sign.

An **example of a three-body reaction** is for example the formation of argon molecular ion. You may find the existence of this particle quite surprising, given that argon is an inert gas, but in high-pressure plasmas, they play quite an important role. The reaction leading to the formation of this ion is

$$\mathrm{Ar}^+ + 2\mathrm{Ar} \longrightarrow \mathrm{Ar}_2^+ + \mathrm{Ar}. \qquad \text{(R8)} \tag{6.8}$$

Analogical to the two-body reactions, the three-body reaction contributes to the source terms of individual species the following way

$$S_{\mathrm{Ar}_2^+,\mathrm{R8}} = +k_8(T_\mathrm{g}) \cdot n_\mathrm{Ar} \cdot n_\mathrm{Ar} \cdot n_{\mathrm{Ar}^+}, \tag{6.9}$$
$$S_{\mathrm{Ar}^+,\mathrm{R8}} = +k_8(T_\mathrm{g}) \cdot n_\mathrm{Ar} \cdot n_\mathrm{Ar} \cdot n_{\mathrm{Ar}^+}, \tag{6.10}$$
$$S_{\mathrm{Ar,R8}} = -k_8(T_\mathrm{g}) \cdot n_\mathrm{Ar} \cdot n_\mathrm{Ar} \cdot n_{\mathrm{Ar}^+}. \tag{6.11}$$

This time, the rate coefficient depends only on the temperature of heavy species T_g because no electrons enter the reaction. In addition, the rate coefficient will have a different unit (see exercise below).

Exercise 6.1 In our formulation, the source term always has the unit of $1/(\mathrm{m}^3 \cdot \mathrm{s})$, what is the physical interpretation? What is the unit of the rate coefficient k_r for two-body and three-body reactions? ■

Table 6.1: The list of all species included in the argon plasma model

Ar	**ground-state argon atom** (concentration obtained from the state equation)
Ar^*	**excited argon atom** (groups resonant and metastable 4s states of argon)
Ar^+	**argon atomic ion**
Ar_2^+	**argon molecular ion**
e	**electron**

6.1.2 The reaction scheme

The reaction scheme implemented in this work is adapted from an article by Baeva et. al. which focuses on numerical simulations of an atmospheric-pressure microwave plasma jet operating in argon [Bae+12]. The model includes the particle species listed in table 6.1. It should be pointed out that the continuity equation (6.2) is solved only for four species, Ar^*, Ar^+, Ar_2^+ and e. The number density of ground-state argon can be determined from the state equation for the ideal gas, which works very good for monoatomic gases at high pressures. If the argon gas was not ionized, the number density of argon would simply be

$$n_{\mathrm{Ar}} = \frac{p}{k_{\mathrm{B}} T_{\mathrm{g}}} \tag{6.12}$$

where p is the pressure, k_{B} the Boltzmann constant and T_{g} the temperature of heavy particles. However, the equation above is not valid if a part of the argon gas is ionized or excited.

Exercise 6.2 What is the actual number density of ground-state argon in plasma if the number density of excited atoms is n_{Ar^*}, the number density of atomic ions is n_{Ar^+} and the number density of molecular ions is $n_{\mathrm{Ar}_2^+}$? ■

The particle species listed in table 6.1 above can undergo 11 reactions in total. The complete set of these reactions is listed in table 6.2. In this reaction scheme, the authors assumed that the energy distribution functions of electrons and heavy particles are both Maxwellian. This, combined with advanced fitting, allowed them to express the rate coefficients analytically.

Table 6.2: The reaction scheme for high-pressure argon plasma (adapted from [Bae+12]). The electron temperature is always in eV while the gas temperature is in K.

	reaction	**rate coefficient**	**unit**
(R1)	$\mathrm{e} + \mathrm{Ar} \longrightarrow \mathrm{e} + \mathrm{Ar}^*$	$4.9 \cdot 10^{-15} \cdot \sqrt{T_{\mathrm{e}}} \cdot \exp(-11.65/T_{\mathrm{e}})$	m^3/s
(R2)	$\mathrm{e} + \mathrm{Ar}^* \longrightarrow \mathrm{e} + \mathrm{Ar}$	$4.8 \cdot 10^{-16} \cdot \sqrt{T_{\mathrm{e}}}$	m^3/s
(R3)	$\mathrm{e} + \mathrm{Ar} \longrightarrow 2\mathrm{e} + \mathrm{Ar}^+$	$1.27 \cdot 10^{-14} \cdot \sqrt{T_{\mathrm{e}}} \cdot \exp(-15.76/T_{\mathrm{e}})$	m^3/s
(R4)	$\mathrm{e} + \mathrm{Ar}^* \longrightarrow 2\mathrm{e} + \mathrm{Ar}^+$	$1.37 \cdot 10^{-13} \cdot \sqrt{T_{\mathrm{e}}} \cdot \exp(-4.11/T_{\mathrm{e}})$	m^3/s
(R5)	$\mathrm{e} + \mathrm{e} + \mathrm{Ar}^+ \longrightarrow \mathrm{e} + \mathrm{Ar}$	$8.75 \cdot 10^{-39} \cdot T_{\mathrm{e}}^{-4.5}$	m^6/s
(R6)	$\mathrm{e} + \mathrm{Ar}_2^+ \longrightarrow \mathrm{Ar} + \mathrm{Ar}^*$	$1.04 \cdot 10^{-12} \cdot (T_{\mathrm{e}}/0.026)^{-0.67} \cdot \frac{1-\exp(-418/T_{\mathrm{g}})}{1-0.31\exp(-418/T_{\mathrm{g}})}$	m^3/s
(R7)	$\mathrm{e} + \mathrm{Ar}_2^+ \longrightarrow \mathrm{e} + \mathrm{Ar} + \mathrm{Ar}^+$	$1.11 \cdot 10^{-12} \cdot \exp\left(-2.94 - 3\frac{T_{\mathrm{g}}-300}{T_{\mathrm{e}} \cdot 11604}\right)$	m^3/s
(R8)	$\mathrm{Ar}^+ + 2\mathrm{Ar} \longrightarrow \mathrm{Ar} + \mathrm{Ar}_2^+$	$2.25 \cdot 10^{-43} (T_{\mathrm{g}}/300)^{-0.4}$	m^6/s
(R9)	$\mathrm{Ar} + \mathrm{Ar}_2^+ \longrightarrow \mathrm{Ar}^+ + 2\mathrm{Ar}$	$0.522 \cdot 10^{-15} T_{\mathrm{g}}^{-1} \exp(-15\,131/T_{\mathrm{g}})$	m^6/s
(R10)	$\mathrm{Ar}^* + \mathrm{Ar}^* \longrightarrow \mathrm{e} + \mathrm{Ar}^+ + \mathrm{Ar}$	$6.2 \cdot 10^{-16}$	m^3/s
(R11)	$\mathrm{Ar}^* + \mathrm{Ar} \longrightarrow \mathrm{Ar} + \mathrm{Ar}$	$3.0 \cdot 10^{-21}$	m^3/s

Exercise 6.3 Reactions (R1), (R3) and (R4) in table 6.2 all include an exponential part $\exp(-C/T_e)$ which is zero at low electron temperatures and approaches one at higher electron temperatures.

- What do you think is the unit and the physical meaning of the constants 11.65, 15.76 and 4.11?
- Looking at the rate coefficients of these three reactions, which one will probably be the most important ionization channel?

■

Exercise 6.4 Express the source terms of the following four species: e, Ar^*, Ar^+, Ar_2^+ using rate coefficients k_1 to k_{11} and the species' densities $n_{Ar}, n_e, n_{Ar^*}, n_{Ar^+}, n_{Ar_2^+}$. ■

As you can see, even for the relatively simple argon atom, the number of possible reactions is quite high. The number of reactions taking place in plasmas rises dramatically with the complexity of the gas in which plasma is ignited. With molecular gases, various rotational and vibrational excitations have to be taken into account, as well as dissociation and re-association of the molecule. Just to get the idea of the complexity, you would need **several dozen reactions** to provide a minimum description **of a plasma in an O_2/N_2 mixture**. In order to describe plasmas **in argon with ambient air** (including humidity and other admixtures), **the number of reactions rises to more than 4000** [GB13].

6.2 Implementation

Now we know everything in order to solve the system of ordinary differential equations in the following form

$$\frac{\partial}{\partial t}\begin{pmatrix} n_e \\ n_{Ar^*} \\ n_{Ar^+} \\ n_{Ar_2^+} \end{pmatrix} = \begin{pmatrix} S_e \\ S_{Ar^*} \\ S_{Ar^+} \\ S_{Ar_2^+} \end{pmatrix} \tag{6.13}$$

We have solved a system of mathematically similar ordinary differential equations (ODEs) in chapter 3. Therefore, if you are unsure about anything regarding the code, try looking back at the exercises where we dealt with motion of particles in E and B fields.

Since this is the final chapter of this text, you will only be provided with a template of the whole program and you have to fill in the missing pieces. The main file containing the solution looks like this:

"Program 8: solve.m"

```
% setting global variables
global p;
global kB;
global Tg;
global Te;
p = 1e5;          % pressure in Pa
kB = 1.38e-23;    % Boltzmann constant in m^2 kg s^-2 K^-1
Tg = 400;         % Gas temperature in Kelvin
Te = 2;           % Electron temperature in eV

tsteps = 1000; % number of time steps
```

```
tspan = logspace(-11, -6, tsteps); % this time, we do not use linear spacing for the time but rather logarithmic
% Initial number densities in m^-3
nArs_init = 1e12; % Ar*
nArp_init = 1e12; % Ar+
nAr2p_init = 1e12; % Ar_2+
ne_init = nArp_init+nAr2p_init; % electrons, follows from global neutrality
% The vector of initial densities
initial = [nArs_init, nArp_init, nAr2p_init, ne_init];

% Solving the system of ODEs
[t, sol] = ode45('odefun', tspan, initial);

% Plotting the data
close all
figure;
hold on;
nAr = p./(kB*Tg)-sol(:,2)-0.5*sol(:,3)-sol(:,1);
plot(log10(t), log10(nAr), 'c');
plot(log10(t), log10(sol(:,1)), 'r');
plot(log10(t), log10(sol(:,2)), 'b');
plot(log10(t), log10(sol(:,3)), 'm');
plot(log10(t), log10(sol(:,4)), 'k');
ylim([12, 26])
legend('Ar', 'Ar^*', 'Ar^+', 'Ar_2^+', 'electrons', 'Location', 'northwest')

xlabel('Time, log_{10} [s]');
ylabel('Number density, log_{10} [m^{-3}]');
title(['p=', num2str(p), ' Pa, T_g=', num2str(Tg), ' K, T_e=', num2str(Te), 'eV']);
set(gca,'fontsize', 16);
```

The program above should not surprise you now. The biggest difference with our previous programs solving ordinary differential equations lies on line 12. In particular, we do not use a linearly-spaced vector for time and we use a logarithmically spaced vector instead. This is because the processes in plasma are often logarithmic with respect to time.

Similar to chapter 3, the `solve.m` program uses the `ode45()` solved to calculate the time evolution of our system. The function evaluating the right-hand side of our system of equations (6.13) is called `odefun()` and is stored in a Matlab function file with a similar name.

"Program 8: odefun.m"

```
function dqdt = odefun(t, q),
    % We will use global variables defined in solve.m
    global p;
    global kB;
    global Tg;
    global Te;
    % the vector q contains the densities of Ar*, Ar+, Ar_2+
```

```
        and electrons
    nArs = q(1);
    nArp = q(2);
    nAr2p = q(3);
    ne = q(4);
    % the density of argon is calculated using the state
       equation and the densities of other heavy species
    nAr = p./(kB*Tg)-nArs-0.5*nAr2p-nArp;
    % The rate coefficients are all stored in separate
       function files
    k1 = f_k1(Te, Tg);
    k2 = f_k2(Te, Tg);
    k3 = f_k3(Te, Tg);
    k4 = f_k4(Te, Tg);
    k5 = f_k5(Te, Tg);
    k6 = f_k6(Te, Tg);
    k7 = f_k7(Te, Tg);
    k8 = f_k8(Te, Tg);
    k9 = f_k9(Te, Tg);
    k10 = f_k10(Te, Tg);
    k11 = f_k11(Te, Tg);
    % Now, it is necessary to define the source terms
    SArs = +k1*ne*nAr ...
        -k2*ne*nArs ...
        -k4*ne*nArs ...
        +k6*ne*nAr2p ...
        -2*k10*nArs*nArs ...
        +k11*nArs*nAr;
    SArp = ...;
    SAr2p = ... ;
    Se = ... ;
    % and finally, the derivative of the input vector is
       returned
    dqdt = [SArs; SArp; SAr2p; Se];
end
```

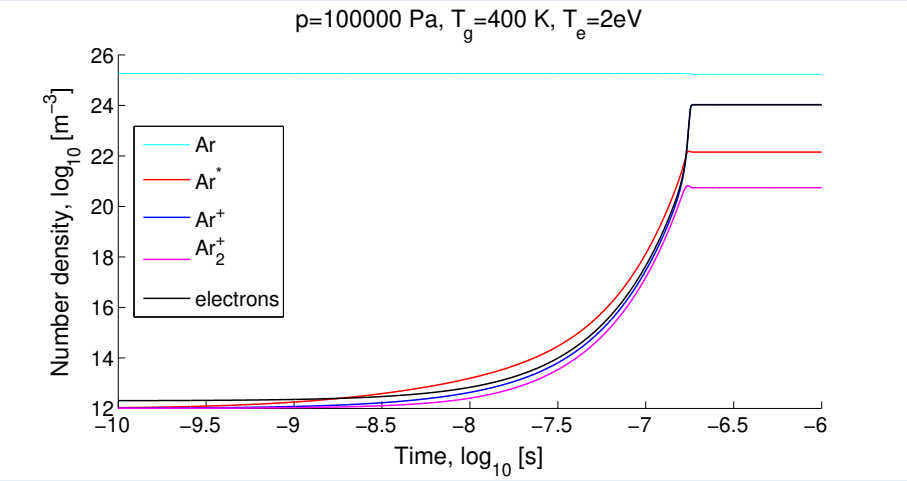
In the `odefun()` function, the source terms for individual particle species have to be calculated. The source term for Ar^* (variable `SArs`) has already been pre filled but the source terms for Ar^+ (variable `SArp`), Ar_2^+ (variable `SAr2p`) and electrons (variable `Se`) **have to be completed based on exercise 6.4**. Furthermore, the **rate coefficients** `k1` to `k11` **are defined using eleven external functions** `f_k1()` to `f_k11()` **which take the gas temperature** T_g **in Kelvin and electron temperature** T_e **in eV as input**. Of course, not all the rate coefficients will depend on both T_g and T_e but for consistency, both these temperatures are passed to the functions. You will have to create these 11 simple functions based on the reactions and rate coefficients in table 6.2.

Exercise 6.5 Complete the program, the template for which was provided above. In particular, you will have to

1. Write the expressions for particle source terms in file `odefun.m`.
2. Create 11 functions, `f_k1.m` to `f_k11.m`. Each of these files will calculate the rate

coefficient of the corresponding reaction. **Please note that the functions must be able to work with vector arguments, i.e. you have to use element-wise operations with T_g and T_e.**

3. Run the code with pre-defined values of pressure p, T_g and T_e. Check if the output looks similar to the figure below.



Please take into consideration that this is the first time this text has been used in class and the authors may have also made a mistake in the source terms. If your output looks **a little different** and you are sure that your source terms are correct, do not worry.

■

Now, when your program works, you can try changing various parameters in order to get a qualitative and quantitative idea how the argon plasma works.

Exercise 6.6 Try increasing and decreasing the electron temperature and answer the following questions.

- How does the equilibrium electron density change and why?
- How does the ignition time change?
- What is the dominant ion in the ignition phase and what is the dominant ion when the plasma stabilizes? Does the dominance of the two ions change with electron temperature?

■

Exercise 6.7 Run the program for your chosen value of electron temperature and for pressures of 10^3, 10^4, 10^5 and 10^6 Pa and answer the following questions.

- How does the equilibrium electron density change and why?
- How does the ignition time change?

■

6.3 Parametric study

Programs which are in principle similar to the program that we developed in this section are often used in plasma physics for examining plasma properties at various conditions, although the number of reactions taken into account is typically much higher (several thousand). It is often desirable to know how the plasma composition changes at various electron temperatures, neutral gas temperatures or pressures.

Advanced exercise 6.1 Modify the program in the previous section so that it solves the system of equations for several values of T_e/p and plots the steady-state number densities as functions of T_e/p. You can do this by adding a for loop.

What makes this exercise difficult is the fact that the ignition time changes quite quickly with electron temperature and pressure. **Therefore, the time interval has to be updated in each iteration** according to the current value of T_e and p, otherwise the solution will take very long. ■

R When solving similar sets of equations for a large number of conditions, advanced numerical techniques have to be employed to achieve reasonable computation times. Sometimes, the equations are solved in a logarithmic form, i.e. not for the number densities of particles but for their logarithms,

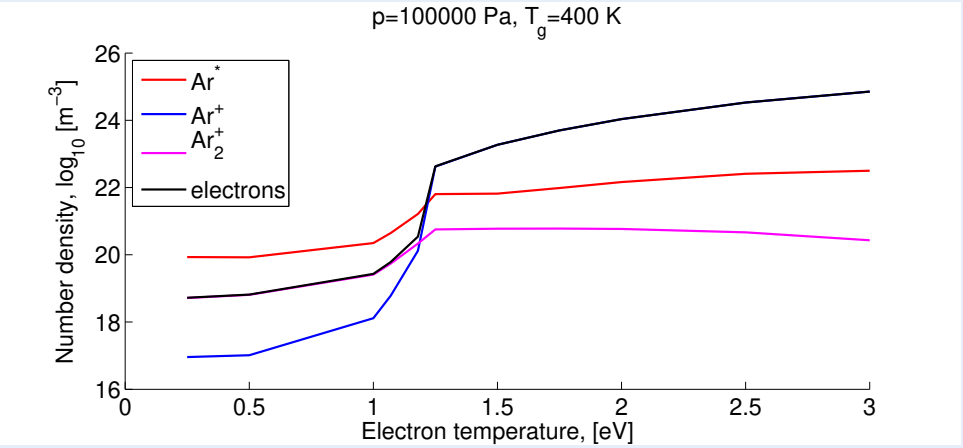
$$l_j = \ln n_j. \tag{6.14}$$

This trick can make the convergence much faster because the values of n_j vary by many orders of magnitude during the solution while values of l_j vary only within an order of magnitude.

The exercise above may be quite difficult and unnecessarily time-consuming for students who do not intend to use Matlab on regular basis. However, even if you didn't implement the program in the advanced exercise above, try to complete the following one.

Exercise 6.8 As already mentioned, the composition of plasma often changes dramatically with electron temperature. The figure below shows the number densities of various particle species as a function of electron temperature at the constant pressure of $p = 10^5$ Pa and gas temperature $T_g = 400$ K. Think about the following questions and answer them.

- If you wanted to use the plasma as a light source, what temperature would you use and why?
- What is a simple explanation for the decrease in Ar_2^+ with increasing T_e?



6.4 Conclusion

This brings us to the end of the study material. If you have come thus far and completed all the exercises successfully, accept our congratulations. The authors sincerely hope that the exercises and demonstrations in this study text helped you to deepen the understanding of the complex processes taking part in plasmas. We also hope that the Matlab skills that you learned here will be useful to you, no matter what discipline of physics you end up in.

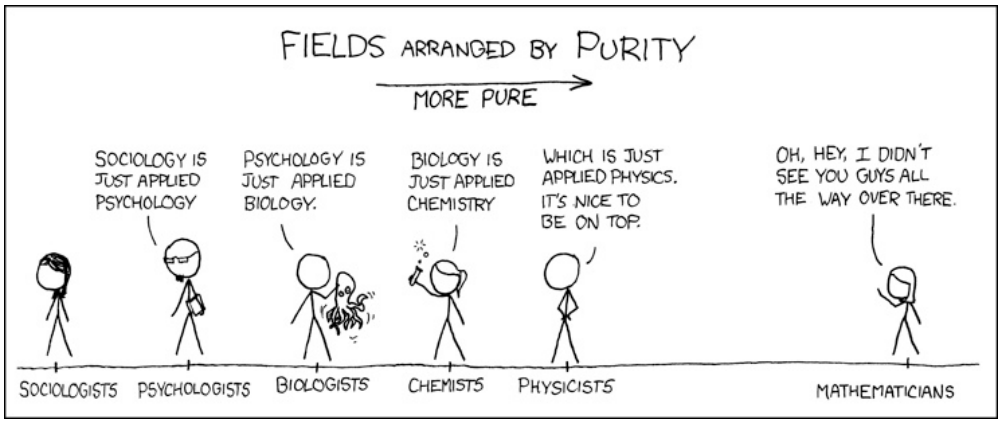
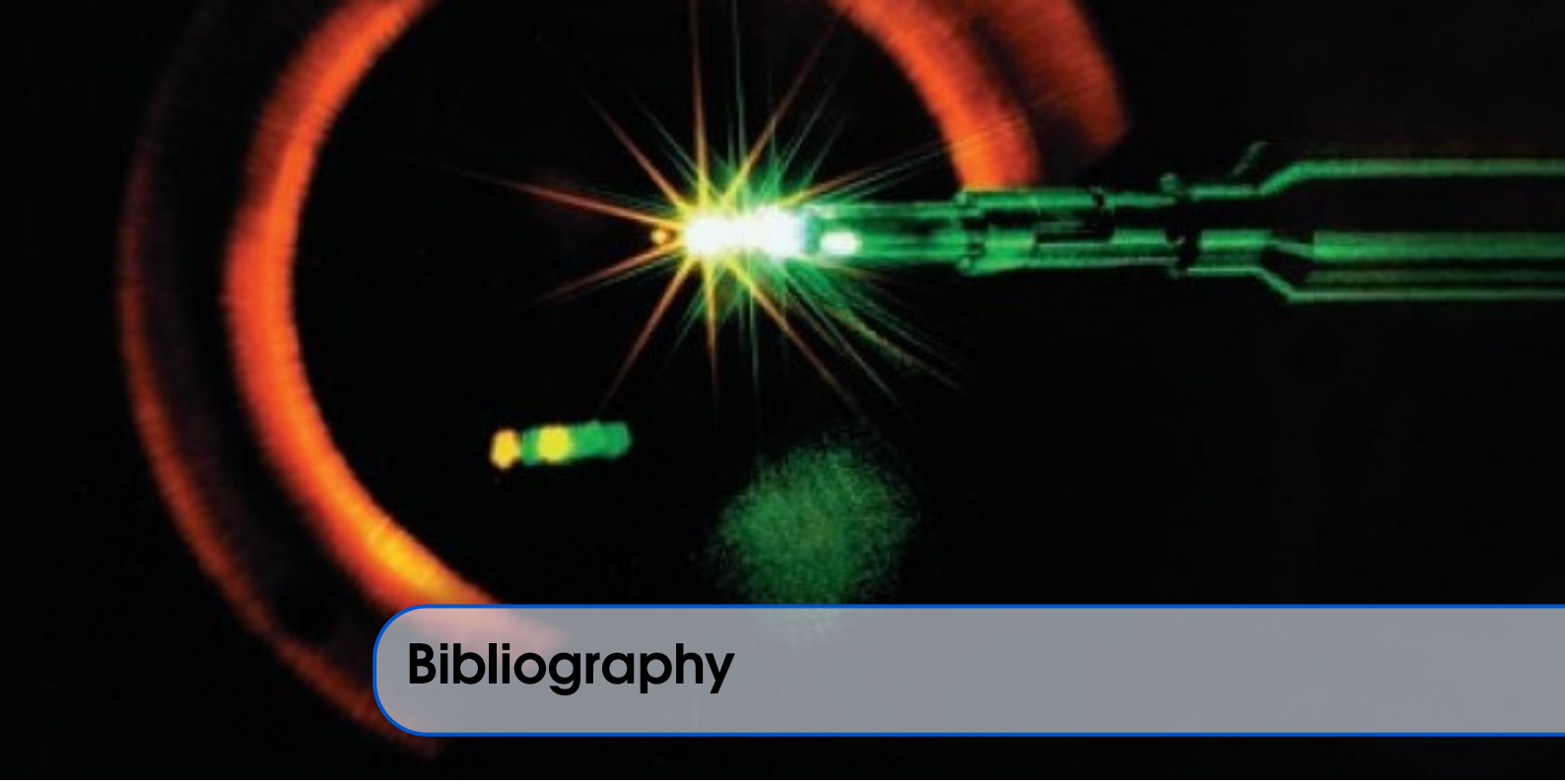


Figure 6.1: Downloaded from xkcd.org



Bibliography

Books

[Bit04] J. A. Bittencourt. *Fundamentals of Plasma Physics*. 3rd edition. New York: Springer, 2004 (cited on page 12).

[GK08] Dan M. Goebel and Ira Katz. *Fundamentals of Electric Propulsion: Ion and Hall Thrusters*. 1st edition. Pasadena: California Institute of Technology, 2008 (cited on page 16).

Articles

[Bae+12] M. Baeva et al. "Modeling of microwave-induced plasma in argon at atmospheric pressure". In: *Physical Review E* 85.5 (May 2012), page 056404. ISSN: 1539-3755. DOI: 10.1103/PhysRevE.85.056404. URL: http://link.aps.org/doi/10.1103/PhysRevE.85.056404 (cited on page 47).

[Bak12] Daniel N Baker. "New Twists in Earth's radiation belts". In: *American Scientist* 102 (2012) (cited on page 21).

[GB13] W Van Gaens and A Bogaerts. "Kinetic modelling for an atmospheric pressure argon plasma jet in humid air". In: *Journal of Physics D: Applied Physics* 46.27 (2013), page 275201. URL: http://stacks.iop.org/0022-3727/46/i=27/a=275201 (cited on page 48).

[Shp+13] Yuri Y. Shprits et al. "Unusual stable trapping of the ultrarelativistic electrons in the Van Allen radiation belts". In: *Nature Physics* 9.11 (2013), pages 699–703. ISSN: 1745-2473. DOI: 10.1038/nphys2760. URL: http://www.nature.com/doifinder/10.1038/nphys2760 (cited on page 21).

Online resources

[FEI10] FEI. *Introduction to Electron Microscopy*. July 2010. URL: http://www.fei.com/documents/introduction-to-microscopy-document/ (cited on page 15).

[Mar13] Stefano Markidis. *Lectures on Computational Plasma Physics*. Sept. 2013. URL: https://www.pdc.kth.se/education/computational-plasma-physics/computational-plasma-physics (cited on page 21).

[Sci13] Scitechdaily. *Scientists Explain the Formation of the Unusual Third Van Allen Radiation Ring*. Sept. 2013. URL: http://scitechdaily.com/scientists-explain-formation-unusual-third-van-allen-radiation-ring/ (cited on page 21).