[54]

`plasmakin.py` – A Python extension module to handle chemical kinetics in plasma physics modeling

N. Pinhão*

Nuclear and Technological Institute, Phys. Dept.,

Estrada Nacional 10, 2685 Sacavém, Portugal

Abstract

PLASMAKIN [1] is a package to handle physical and chemical data used in plasma physics modeling and to compute gas-phase and gas-surface kinetics data: particle production and loss rates, photon emission spectra and energy exchange rates. It has no limits on the number of chemical species and reactions that can be handled, is independent of problem dimensions and can be used in both steady-state and transient problems. A large number of species properties and reaction types are supported, namely: gas or electron temperature dependent collision rate coefficients, vibrational and cascade levels, evaluation of branching ratios, superelastic and other reverse processes, three-body collisions, radiation imprisonment and photoelectric emission. Support of non-standard rate coefficient functions can be handled by a user-supplied shared library. The package includes a shared library with data reading and computational functions and a Python module providing Python function interfaces and classes.

*Electronic address: `npinhao@itn.pt`; URL: `http://www.itn.pt`

I. INTRODUCTION

A plasma is the state of ionized gases and is the most pervasive state of matter in Nature: It is found anywhere from stars and interstellar gas clouds to semi-conductor processing and fluorescent lamps. It encompasses a wide range of characteristic parameters: particle densities ($10^6 - 10^{21} particles/m^3$); length scale ($10^{-2} - 10^{16} m$); time scale ($10^{-10} - \infty\ s$); electron temperature ($10^{-1} - 10^5 eV$); magnetic field ($10^{-10} - 5T$) and involves the interaction of different types of particles – neutral atoms and molecules, excited species, electrons and photons – between them and with electro-magnetic fields [11].

Modeling of plasma physics problems requires the solution of conservation equations (mass, momentum and sometimes, energy) for the species involved together with the field equations, subject to appropriate boundary conditions. Source and sink terms in these equations include the chemical interactions of these particles in a wide range of processes.

Thus, from electron kinetics studies to collisional-radiative models or large multifluid and time dependent models, whichever the numerical method used, simulation of plasma and gas discharges invariably requires the reading, classification, sorting and manipulation of particles and reactions and, frequently, the evaluation of reaction rates and power transfer rates.

The handling of these data frequently requires a significant amount of code, development time and effort. It is clearly advantageous to have a package able to deal with that data regardless of the number or nature of the species and chemical reactions involved and of the problem being solved or the method used. Such a package can be used as a "black box" moving the description of particles and reactions from code to a data file, thus allowing the user to concentrate on the algorithm, and once the code is developed, to easily modify and test different chemical models.

The need for a "language" to write chemical reactions and to compute the kinetic terms in a generic way is a subject with very broad application of which several approaches have been developed. Several packages have been published that are directed to specific fields – plasma physics [2]; atmospheric chemistry [3, 4] – or to general purpose chemistry [5]. All of the above packages include ODE solvers. However, the first does not include surface reactions, the atmospheric chemistry codes do not cover the needs of plasma physics and the last, although quite complete, is a proprietary, commercial product. More recently, a

Chemical Markup Language schema to describe reactions in XML has been finalized [6].

The *PLASMAKIN* library is designed to provide a framework to handle species and reactions in a way that is not dependent on a user's program, the number of space dimensions or the nature of the problem being solved. At the same time, taking into account the rather different applications that can benefit from *PLASMAKIN* and the continuous improvement of numerical algorithms, an ODE solver was not included, leaving this choice to the user.

These design options allow *PLASMAKIN* to focus on the treatment of chemical kinetics data and be useful in a large range of codes – Boltzmann equation solvers, collisional-radiative codes, fluid and hybrid codes, Monte Carlo or PIC codes, etc.

The library has been extended "upwards" through a Python module – *Plasmakin.py* – with higher level functions and classes and allowing the development of complete programs.

The library can also be extended "downwards" to support non-standard reaction rate laws by a user-supplied library.

The development of a Python module allows integration with the broad range of services provided in Python (numeric libraries; plotting services; testing frameworks; graphical user interfaces) and a fast and efficient development of applications.

Section I gives a description of the problems *PLASMAKIN* is able to address. However, only a brief survey of the physics and chemistry bases is discussed. More detailed information can be found in [1]. The following sections discuss the architectural design including the library structure and the data model, the range of properties supported, the characteristics of the Python module, concluding with a summary and discussion of future developments. In the Appendix we present a sample data file.

II. PLASMA PHYSICS PROBLEMS

Plasma processes occur in a wide range of conditions. To understand these phenomena, the first set of information needed are – gas density or pressure; initial temperature; and whether the process occurs at constant pressure or at constant volume.

Depending on the problem being studied, the description of the plasma requires the solution of some form of Maxwell's equations together with conservation equations. For the

sake of clarity we will focus on the particle conservation equation [12]:

$$\frac{\partial n_i}{\partial t} + \nabla \cdot (n_i \vec{u_i}) = \mathcal{G}_i - n_i \mathcal{L}_i \tag{1}$$

where n_i is the density of particle i, $\vec{u}_i$ the mean particle velocity, $\mathcal{G}_i$ and $\mathcal{L}_i$ are gain and loss terms by collisions. These have the general expression:

$$\begin{aligned} \mathcal{G}_i = & \sum_{m \neq i} A^*_{mi} n_m + \sum_{m,n \neq i} k_{mn} n_m n_n + \sum_{mnl} k_{mnl} n_m n_n n_l \\ \mathcal{L}_i = & \sum_m A^*_{im} + \sum_{m \neq i} k_{im} n_m + \sum_{mn} k_{imn} n_m n_n \end{aligned} \tag{2}$$

A^*_{mn} are rates for unimolecular reactions (i.e. radiative processes) and k_{mn} and k_{mnl} rates for bimolecular and termolecular reactions, respectively. These reaction rates are, in general, functions of the electron or gas temperature.

To study the contribution of collisions to the energy balance in the plasma we are interested in three quantities: The power lost (or gained) by the electrons (P_e), the power converted to heat (P_H), and the radiated power (P_r). The computation of each of these terms is similar to the $\mathcal{G}_i$ and $\mathcal{L}_i$ terms above but slightly more complex and in most cases requires the use of the reaction enthalpy, H_r. As a simple example, the radiated power is given by $P_r = \sum_{m,n} \varepsilon_{mn} A^*_{mn} n_m$ where the sum is on all radiative transitions of all species and ε_{mn} is the photon energy.

The reaction enthalpy is calculated as $H_r(T) = \sum_p H^p_f(T) - \sum_r H^r_f(T)$, where $H_f(T)$ is the enthalpy of formation of the reaction products and reactants, respectively. We assume a simple linear dependency on temperature, $H_f(T) = H(T_0) + C_p \cdot \Delta T$, where T_0 is the standard temperature ($T_0 = 25\,^oC$) and C_p the specific heat, which is a reasonable approximation for gases at low pressure.

Finally we may also be interested in knowing the relative contribution of each reaction for the source and loss terms both in equation (1) and on the power loss terms.

The computation of the $\mathcal{G}_i$ and $\mathcal{L}_i$ terms, the power loss terms, (P_e, P_H, P_r) and the relative contribution of each reaction for these terms, are the main tasks of the *PLASMAKIN* library. For that purpose we need to consider the different properties of the chemical species and reactions involved.

A. Chemical Species

In a plasma, different types of species are present – atoms and molecules in different excited levels, positive and negative ions, electrons, and photons. In some applications (i.e. dusty plasmas), it is also necessary to consider bigger particles.

As we are interested in processes leading to excitation of different energy levels of the same atom or molecule, atoms or molecules in different excited levels are treated as different chemical species.

Some of the properties needed to characterize these species are common to any species – name, energy, mass, charge, formation enthalpy – while others are meaningful only to some species or needed only for some types of reactions.

Molecular species are a special case: In some problems it is important to consider the vibrational levels of molecules. The density of these levels can be estimated from a modified Treanor distribution [6, 7] provided the vibrational frequency, vibrational temperature and anharmonicity constant for those levels are known. This allows them to be considered as a group and to leave the details of each level to be handled by the library.

The full range of species properties considered can be found in [1].

B. Chemical Reactions

A large range of reaction processes can be handled by the library:

Unimolecular processes include radiative process, radiation imprisonment and the treatment of cascade levels.

Both forward and reverse bimolecular reactions are supported.

A large number of forward rate coefficients have an Arrhenius temperature dependence,

$$k_i = K_i T_g^{\beta_i} \exp\left(-\frac{\epsilon_i}{k_B T_g}\right) \tag{3}$$

where K_i, β_i and ϵ_i characterize the reaction.

Reactions in plasma, however, can have more complex temperature dependencies or, as is the case for electron collision reactions, depend on the electron temperature. To accommodate this, *PLASMAKIN* assumes a power series dependence on temperature in the exponential term of the rate coefficients:

$$k_i = \alpha_i^0 T^{\beta_i^0} exp\left(\sum_{j=1}^{5} \frac{\alpha_i^j}{T^{\beta_i^j}}\right) \tag{4}$$

where T is the electron temperature T_e for electron collision reactions and T_g for other cases. The rate coefficients are characterized by a maximum of twelve parameters (α^j, β^j, $j = 0, ..., 5$).

The rate coefficient for forward and reverse reactions are related through the principle of detailed balancing. Hence, once the forward rate, k_f, is known, the reverse rate, k_r (for a two-body collision, $A + B \rightleftharpoons C + D$) is determined from

$$\frac{k_f}{k_r} = \left(\frac{\mu_r}{\mu_f}\right)^{3/2} \frac{g_C g_D}{g_A g_B} \exp\left(-\frac{\epsilon_t}{k_B T_g}\right) \tag{5}$$

where μ_f and μ_r are the reduced masses for particles in "forward" and "reverse" reactions, g the degeneracies of the energy levels and ϵ_t the energy change.

Thermolecular reactions are supported both as standard reactions and as pressure-dependent reactions where different 3rd-body species can have different reaction efficiencies.

Finally, to account for more complex type of reaction rate coefficients, *PLASMAKIN* can call a dynamic library or can be linked with a user's routine.

The full range of reactions, including surface reactions, can be found in [1].

III. *PLASMAKIN* ARCHITECTURE

The *PLASMAKIN* package is composed of three units: a Python module, a Fortran 95 module, and a dummy routine. The discussion of the Python module is postponed to section V.

The development of the Fortran module was influenced by the application of Object Oriented methodologies to Fortran [8, 9]. The programming paradigms of *abstraction, information hiding, data encapsulation* and *function overloading* guided the design of data types and routines and have simplified the future development of *PLASMAKIN* and of bindings for other languages.

The main unit is a Fortran module with public and private data and procedures. This module is a collection of subroutines organized in five groups of tasks:

Data reading and processing: Parse the datafile, test the correctness of the data and build species and reactions;

Inquiry routines: Allow the user to inquire about plasma, species or reaction properties and to compute the source and loss terms for the conservation equations;

Setting routines: Allow the user to set some of the plasma, species or reaction properties (i.e. the gas or electron temperatures. In this case all the temperature dependent rate coefficients are recomputed.)

Error processing and diagnostics: Depending on user's choice, errors are handled either by returning an error code or printing an error message and stopping the program.

Interface routines: These routines allow the calling of module routines from C programs solving calling conventions differences between C and Fortran such as the use of a null character in C strings and the handling of assumed-shape arrays in the Fortran routine interfaces.

All names of public procedures follow a simple convention: `pk<action><subject>`, where `pk` is used to identify *PLASMAKIN* procedures, `<action>` is a verb or an inquiry clause (`Read, Get, Set, Clean, Is`) and `<subject>` is the data acted on by the procedure (`Data, Value, Species, Reactions, ReverseReaction, Sources, PhotonEmission, PowerLosses`).

Details of the procedure interfaces can be found in [1].

The dummy routine is included to allow the computation of reaction rates through non-standard expressions. The routine included in the package is just a template with the proper calling convention that a user has to adopt whenever non-standard rate expressions are needed. Ideally this routine should be compiled as a shared library. The arguments passed from *PLASMAKIN* include the index of the reaction in the datafile, arrays with the vibrational quantum number of reactant and product species, the gas and electron temperatures and the densities of species involved in 3-body reactions.

IV. *PLASMAKIN* DATA INPUT

Data input files are ASCII files using Fortran NAMELIST structures to take advantage of Fortran native IO support. NAMELIST structures are annotated lists of values that provide a simple and convenient mechanism of data input. Each record is specified by a namelist name preceded by an ampersand, followed by pairs of names and values separated by the equals sign, and is terminated by a slash character. These records can appear anywhere and in any number in the file and can include comments started by a '!' character.

PLASMAKIN recognizes three NAMELIST:

PLASMAKIN_DATA: used to read plasma initial values for gas density or pressure, units and gas and electron temperatures. See the example below:

```
&PLASMAKIN_DATA Pressure=5,'mbar', Gas_n=,'cm-3', Gas_T=350,'K' /
```

In this case the gas density is computed from the pressure and temperature values and the output will be specified in cm^{-3};

CHEM_SPECIES: used to input individual species or groups of species in the case of vibrational levels. The amount of data that has to be written is minimized as all properties have default values and the parser is able to deduce missing values from the information already available. Three examples are shown below:

```
&CHEM_SPECIES name='Ne', constant=T, mass=20.2, initial_conc=50,'%'/
&CHEM_SPECIES name='Ne[3P2]', energy=16.6, g=5, data_file='trp.dat'/
&CHEM_SPECIES name = 'A2[X,v]', constant = T, mass = 70
  v = 0,45, omega = 1580.19,'cm-1', vib_T = 1000,'K',
  anharmonicity = 7.58e-3, initial_conc = 50.0,'%' /
```

In the first case the species `Ne` has a default value of zero for the energy . The second species, `Ne[3P2]` inherit the mass value from its parent, `Ne`. Additional data can be read from the file `trp.dat`. In this case the library just passes the file name to the calling program. Finally in the last case we have a group of vibrational species. The library converts the group into individual species from `A2[X,v= 0]` to `A2[X,v=45]` and computes the initial concentrations as refered in section II A;

CHEM_REACTION: used to input individual reactions or reaction groups in the case of reactions involving vibrational levels.

```
&CHEM_REACTION ! Values from V.A. Ivanov J. Phys. B 31 [1998] 1765
  reaction = 'e + Ne[3P2] -> e + Ne[3P1]'
  value = 1.603e-6,-0.3,-6.0e2,1, units = 'cm3s-1' /
&CHEM_REACTION ! 2025 reactions
  reaction = 'A2[X,v] + A2[X,w-1] -> A2[X,v-1] + A2[X,w]'
  data_file = '',5 /
```

The first namelist describes a single reaction following an Arrhenius law: $k = 1.603\times10^{-6}\cdot T_g^{-0.3}\exp\left(+\frac{6.0\times10^2}{k_B T_g}\right)$. The second example represents a group of reactions with reaction rates obtained from the 5th equation of a user's routine. Taking into account the combinations of vibrational numbers in the example above, this reaction group symbolizes 2025 reactions.

V. THE PYTHON MODULE

The most recent addition to the *PLASMAKIN* package is a Python wrapper based on the `ctypes` foreign function library.

The development of a Python module serves several purposes:

1. It allows a broader audience to use the library since it does not require knowledge of Fortran;

2. It integrates the library with the rich set of Python modules. Of special importance to plasma modeling is the integration with other modules for scientific computing as `Scipy` [10] or `Matplotlib`; and

3. It allows the use of Python development tools as `doctest` or `unittest` that have proved useful to track errors in the library itself.

The module is organized into three levels:

1. The lower level comprises the loading of the *PLASMAKIN* dynamic link library and function definitions to call the library C compatible routines. The arguments and

return values of these functions are `ctypes` data types. While in Fortran the public subroutines have overloaded interfaces that accept different types of data, `ctypes` do not include any mechanism to support a similar behavior. Thus the number of functions defined had to be increased. These routines are not meant to be called directly thus all the routines names start with two underscore characters;

2. The second level are functions that use only Python built-in types as function arguments or as return values. These functions call the lower level functions to access the *PLASMAKIN* library. The naming convention is similar to the equivalent library subroutines (`<action><subject>`). A closer similarity can even be obtained if the Python module is imported as `pk` (making the function calls `pk.<action><subject>`). The functions have the following characteristics:

   (a) They accept different type of data allowing the recovery of the level of abstraction of the Fortran routines that had been lost with the first level functions;

   (b) The argument list is simplified. In the Fortran library most of the procedures are subroutines, returning values in the argument list. Because in Python we deal with functions, all the arguments in Fortran that are used only to return values have been removed from the argument list or, when these arguments were optional arguments, they have been transformed into logical flags;

   (c) They use as much as possible, arguments with default values which further simplify the use of these functions;

   (d) Have proper exception handling.

3. The higher level comprises class definitions. The following classes are defined:

   PK – a class to keep the information on global properties. The values of the gas temperature (`GasTemp`), electron temperature (`eTemp`) and gas density (`GasN`) can be set. All the other properties are fixed. In this case, an attempt to assign a new value raises an exception.

   Species – a class representing the chemical species. The density (`n`) or concentration can be changed by the user (changing also the values in the library) but all

other properties have fixed values. An attempt to change these values raises an exception;

Reaction – a class representing reactions. The value of the reaction coefficient can be changed. This class defines a method `Update()` to update the value of the rate coefficient. This is necessary because if the value of rate coefficents in the library are changed, i.e. due to a change in temperature, the only way to update the corresponding values in the class instance is to call this function;

Phys_Property – a helper class to describe a physics property, containing a float value and a string for the units;

Data_Column – a helper class to hold the string for a filename and an integer index.

VI. A SAMPLE CASE

As an illustration of the usage of this package we study the passage of an electron swarm through a gas.

The electrons have a Gaussian profile and along their passage, they excite and ionize the gas, producing several chemical species. These species take part in several reactions among themselves and with the electrons, and are transported either by diffusion of advection to the walls where they are neutralized or de-excited.

Here we consider an hypothetical gas with diffusion coefficients and rate coefficients adjusted for this example.

The species density is obtained from the solution of a system of ordinary differential equations.

The listing of the program can be found in Appendix A. To solve this problem `plasmakin` is used together with `scipy` and `pylab` for computational and plotting services, respectively.

The calls to `plasmakin` are limited to a few lines:

- In line 27 the datafile with the description of species and reaction (shown in Appendix B) is read;
- In lines 29 – 31 we define instances of the classes `PK` and `Species`; and
- In line 52 all the "magic" is done when the $\mathcal{G}_i$ and $\mathcal{L}_i$ terms are computed.

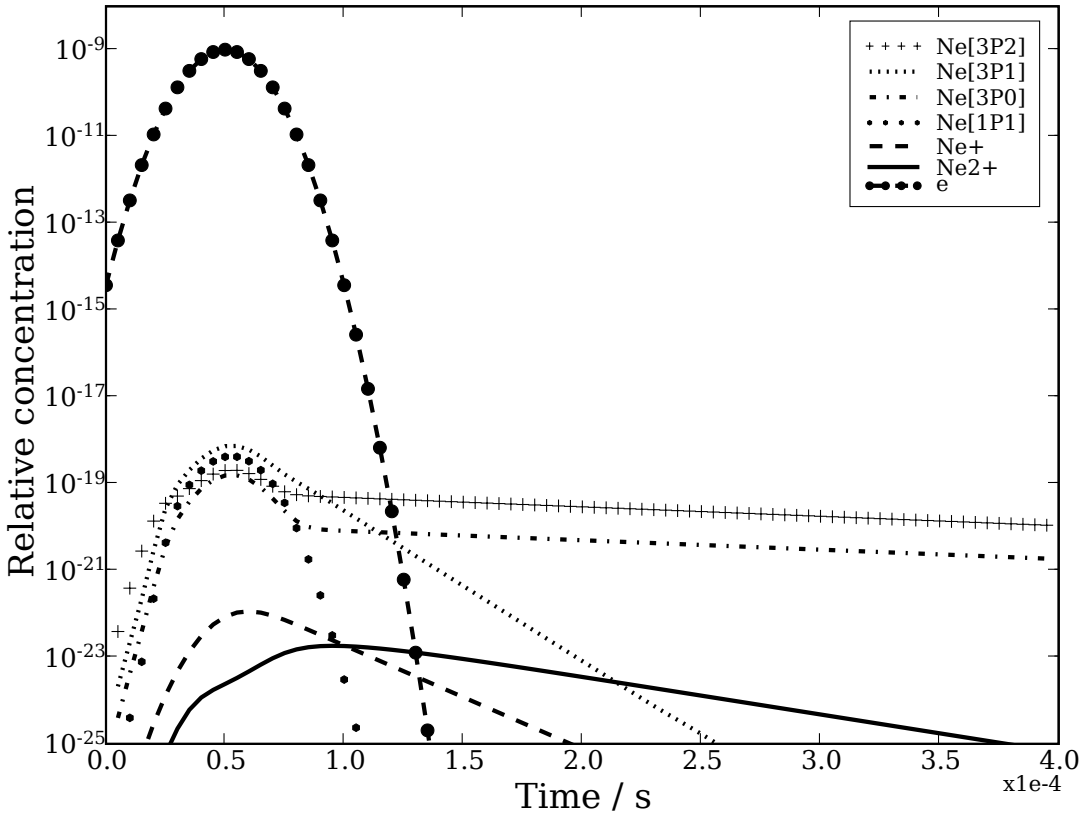

Figure 1: Time dependency of species formed during the passage of an electron swarm through a gas.

Changing the values in the datafile and running the again the program allows a quick study of the influence of several parameters – gas density, reactions included in the model, values of the rate coefficients. The result of one simulation is shown in Figure 1.

VII. CONCLUSIONS

A Python extension module has been developed to access the *PLASMAKIN* chemical kinetics library.

The Python module, however, is not just a wrapper around the library and adds a "Pythonistic" way of problem solving to the analysis of plasma physics problems.

Taking advantage of other Python modules for debugging, numerical computation and data analysis and representation is possible to build programs quickly and reliably.

Further work will continue both on the extension of the *PLASMAKIN* library and the `plasmakin.py` module. Future plans include the simulation of photon emission spectra taking into account line broadening; the introduction of a database for species properties and reactions, and the migration of the datafile format to XML.

Acknowledgments

We wish to express our thanks to D. Baracol for assistance in the revision of this article.

Appendix A: SAMPLE CODE

```
#!/usr/bin/env python

"""Example showing the use of Plasmakin.py: Electron pulse in a gas.

Description:
This program evaluates the interaction of a short electon pulse in an ideal
gas.
The distribution of the electron pulse in time has a Gaussian profile with
tMax=5.e-5 s and Std Dev= 1.e-5 s.
The electron pulse excites and ionizes the gas producing several species.
These species react through several processes and drift or difuse to the
walls, where they are neutralized or deexcited. The values of rate coefficients
are not realistic as sometimes they are too high or too low comparing with the
range of values found in real gases but serve the purpose of testing the
Plasmakin module.
Changing the values in this file or in the datafile the user can quickly see
the results. In this way this programs serves a pedagogic purpose.

Owner: N. Pinhao, ITN - Physics Dept. - PORTUGAL
Date: June 2007"""

import plasmakin as pk
from scipy import *
from pylab import *

pk.ReadData('test.dat')

gas = pk.PK()
gas.eTemp = 10.              # mean electron energy = 10 eV
```

```
30 sp = [pk.Species(i) for i in range(gas.NnC,gas.NnTV+gas.NnC)]
31
32 # Time
33 dt = 5.e-6; t = arange(0.0, 4.e-4, dt)
34
35 # Initial density for non-constant species, except electrons
36 n0 = array([sp[i].n for i in range(gas.NnTV-1)])
37
38 # Electron density
39 ne = stats.norm.pdf(t,5.e-5,1.e-5)
40 norm = max(ne)*1.e3
41 ne = where(ne>1e-20,ne/norm,0)
42
43 # Artificial difusion / advection for conservation equations
44 Dv = array([5.e+3,5.e+3,5.e+3,5.e+3,4.e+4,2.e+4])
45
46 def dndt(y,x):
47     net = gas.GasN*stats.norm.pdf(x,5.e-5,1.e-5)/norm
48     net = net>1e-20 and net or 0
49     tt = list(y)
50     tt.append(net)
51     SrC, SrP = pk.GetSources(tt)
52     return array(SrC[:gas.NnTV-1]) - y*(array(SrP[:gas.NnTV-1]+Dv))
53
54 # Integration of the ode system
55 z = integrate.odeint(dndt,n0,t,h0=1.e-10)
56
57 # Plot the results
58 lines = ['k+','k:','k-.','k.','k--','k-','ko--']
59 for i in range(gas.NnTV-1):
```

```
60        semilogy(t,z[:,i]/gas.GasN,lines[i],label=sp[i].name, lw=2)
61
62  # ... add the electrons scaled by 1e-6
63  semilogy(t,ne/1.e6,lines[gas.NnTV-1], label=sp[gas.NnTV-1].name, lw=2)
64  ylim((1.e-25,1.e-8))
65  xlabel('Time / s',size=16); xticks(size=12)
66  ylabel('Relative concentration',size=16); yticks(size=12)
67  legend()
68  savefig('test.eps')
69  #show()
```

Appendix B: SAMPLE DATAFILE

```
! ************************************************************************
!                                test.dat
! Datafile for the Python example program
! ************************************************************************

&PLASMAKIN_DATA Pressure=10.,'mbar', Gas_n=,'cm-3', Gas_T=350,'K'/

! 1. Gas species
&CHEM_SPECIES name='Ne', constant=T, mass=20.18, initial_conc=100,'%'/
&CHEM_SPECIES name='Ne[3P2]', energy=16.61, g=5, data_file='NeTransp.txt'/
&CHEM_SPECIES name='Ne[3P1]', energy=16.67, g=3/
&CHEM_SPECIES name='Ne[3P0]', energy=16.71, g=1/
&CHEM_SPECIES name='Ne[1P1]', energy=16.85, g=3/
&CHEM_SPECIES name='Ne[3p]',  energy=18.38, cascade=T/
&CHEM_SPECIES name='Ne[3pM]', energy=18.97, cascade=T/
&CHEM_SPECIES name='Ne[4s]',  energy=19.66, cascade=T/
&CHEM_SPECIES name='Ne+',     energy=21.56, charge=+1/
&CHEM_SPECIES name='Ne2+', mass=40.36,    charge=+1/
&CHEM_SPECIES name='photon', v=1,13/
&CHEM_SPECIES name='e', charge=-1  /

! 2. Electron excitation and ionization

! 2.1. Excitation of 3s levels
&CHEM_REACTION reaction='e + Ne -> e + Ne[3P2]', value=5.e-4, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[3P1]', value=3.e-5, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[3P0]', value=5.e-6, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[1P1]', value=1.e-6, units='cm3s-1'/
```

```
! 2.2. Excitation of 3p and 4s+upper levels
&CHEM_REACTION reaction='e + Ne -> e + Ne[3p]',  value=5.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[3pM]', value=3.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[4s]',  value=1.e-7, units='cm3s-1'/

! 2.5. Ionization
&CHEM_REACTION reaction='e + Ne -> 2*e + Ne+', value=1.e-8, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P2] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P1] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P0] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[1P1] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/

! 2.3. s_j-s_i transitions
&CHEM_REACTION reaction='e + Ne[3P2] -> e + Ne[3P1]'
  value=1.603e-6,-0.3,-6.0e2,1, units='cm3s-1'/ [coeff in K]
&CHEM_REACTION reaction= 'e + Ne[3P1] <-> e + Ne[3P0]'
  value=3.1e-8,,-5.176e3,1 units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P2] <-> e + Ne[3P0]'
  value=8.2e-9,,-1.118e3,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P2] <-> e + Ne[1P1]'
  value=5.e-9,,2.658e3,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P1] <-> e + Ne[1P1]'
  value=5.e-9,,2.054e3,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P0] <-> e + Ne[1P1]'
  value=2.3e-7,,1.543e3,1, units='cm3s-1'/

! 3. Radiative transitions
! 3.1. 3s radiative levels
&CHEM_REACTION reaction='Ne[3P1] -> Ne + photon1', value=0.486e8/
&CHEM_REACTION reaction='Ne[1P1] -> Ne + photon2', value=6.11e8/
! 3.2 4s radiative levels
```

```
&CHEM_REACTION  reaction='Ne[4s] -> Ne + photon11',  value=1.21e8/

! 3.3. Radiation imprisonment
&CHEM_REACTION reaction='Ne + photon1 -> Ne[3P1]', value=1.539e-3/
&CHEM_REACTION reaction='Ne + photon2 -> Ne[1P1]', value=1.746e-3/
&CHEM_REACTION reaction='Ne + photon11 -> Ne[4s]', value=1.0e-3/

! 3.4. Cascade reactions
&CHEM_REACTION reaction='Ne[3p]  -> Ne[3P2] + photon3',  value=9.24e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[3P2] + photon4',  value=6.128e7/
&CHEM_REACTION reaction='Ne[3p]  -> Ne[3P1] + photon5',  value=6.722e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[3P1] + photon6',  value=9.397e7/
&CHEM_REACTION reaction='Ne[3p]  -> Ne[3P0] + photon7',  value=1.691e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[3P0] + photon8',  value=2.49e7/
&CHEM_REACTION reaction='Ne[3p]  -> Ne[1P1] + photon9',  value=6.689e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[1P1] + photon10', value=9.208e7/
&CHEM_REACTION reaction='Ne[4s]  -> Ne[3p]  + photon12', value=1.034e7/
&CHEM_REACTION reaction='Ne[4s]  -> Ne[3pM] + photon13', value=1.434e7/

! 3. Heavy species kinetics
! 3.1 Pooling reactions
&CHEM_REACTION reaction='2*Ne[3P2] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P1] -> Ne + Ne+ + e'
  value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P0] -> Ne + Ne+ + e'
  value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[1P1] -> Ne + Ne+ + e'
  value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P1] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[3P0] -> Ne + Ne+ + e'
  value=3.2e-10, units='cm3s-1'/
```

```
&CHEM_REACTION reaction='Ne[3P1] + Ne[1P1] -> Ne + Ne+ + e'
  value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P0] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P0] + Ne[1P1] -> Ne + Ne+ + e'
  value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[1P1] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P2] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P0] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[1P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[3P0] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[1P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P0] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P0] + Ne[1P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[1P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/

! 3.2. Molecular ion formation
&CHEM_REACTION reaction='Ne+ + Ne + M -> Ne2+ + M' value=3.5e-31, units='cm6s-1'/

! 4. electron-ion recombination
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3p] + Ne', value=1.338e-6,-0.67, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3pM] + Ne', value=1.338e-6,-0.67, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3P2] + Ne', value=6.693e-7,-0.67, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3P1] + Ne', value=6.693e-7,-0.67, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3P0] + Ne', value=6.693e-7,-0.67, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[1P1] + Ne', value=6.693e-7,-0.67, units='cm3s-1'/
```

[1] N. Pinhao, *Comp. Phys. Commun.* **135** (2001) 105

[2] S. A. Roberts, PLASKEM, *Comp. Phys. Commun.* **18** (1979) 363

[3] C. J. Aro, CHEMSODE, *Comp. Phys. Commun.* **97** (1996) 304

[4] G. D. Carver, P.D. Brown and O. Wild, ASAD, *Comp. Phys. Commun.* **105** (1997) 197

[5] R. J. Kee, F.M. Rupley, E. Meeks and J.A. Miller, CHEMKIN - III, Sandia National Laboratories, 1996

[6] Gemma L. Holliday, Peter Murray-Rust, and Henry S. Rzepa *J. Chem. Inf. Model.* **46(1)** (2006) 145 - 157

[6] C. E. Treanor, J. W. Rich and R. G. Rehm, *J. Chem. Phys.* **48** (1968) 1768

[7] B. F. Gordiets, S. S. Mamedov and L. A. Shelepin, *JETP* **40** (1972) 640-646

[8] C. D. Norton, *Object Oriented Programming Paradigms in Scientific Computing*, PhD thesis, Renssleaer Polytechnic Institute, Troy, New York, August 1996

[9] C. D. Norton, B. K. Szymanski and V. K. Decyk, *Communications of the ACM*, **38(10)** (1995) 88-100

[10] E. Jones, T. Oliphant, P. Peterson et al. *SciPy: Open Source Scientific Tools for Python*, (2001–) URL: http://www.scipy.org

[11] In some problems the gravitational field has also to be taken into account.

[12] Both the momentum and the energy conservation equations also include collisional terms. These terms, however, are small in the momentum conservation equation and thus, frequently neglected. This is not always true for the energy conservation equation although a good number of problems can be solved without including this equation.