TARDIS Documentation

Release 0.9.dev574

Stuart Sim and Wolfgang Kerzendorf

September 17, 2013

CONTENTS

This is the documentation for the TARDIS package.

CHAPTER

ONE

INTRODUCTION

This is a package for supernova radiative transfer called Temporal And Radiative Diffusion In Supernovae.

CHAPTER

TWO

INSTALLATION

very easy to install

CHAPTER

THREE

RUNNING TARDIS

To run TARDIS requires two files. The atomic database (for more info refer to *Atomic Data*) and a configuration file (more info at *Configuration File*).

Currently there is no script that can run TARDIS. However it is very easy to set one up:

```
from tardis import config_reader, model_radial_oned, simulation

tardis_config = config_reader.TARDISConfiguration.from_yaml('myconfig.yml')
radial1d_mdl = model_radial_oned.Radial1DModel(tardis_config)
simulation.run_radial1d(radial1d_mdl)
```

CHAPTER

FOUR

GRAPHICAL USER INTERFACE

TARDIS uses the PyQt4 framework for its cross-platform interface.

The GUI runs through the IPython Interpreter which should be started with the command `ipython-2.7 --pylab=qt`, so that it has acess to pylab.

Creating an instance of the `ModelViewer`-class requires that PyQt4 has already been initialized in IPython. The above command to start IPython accomplishes this.

4.1 GUI Layout and Features

CHAPTER

FIVE

CONFIGURATION FILE

TARDIS uses the YAML markup language for its configuration files. There are several sections which allow different settings for the different aspects of the TARDIS calculation. An example configuration file can be downloaded `here`.

Warning: One should note that currently floats in YAML need to be specified in a special format: any pure floats need to have a +/- after the e e.g. 2e+5

Every configuration file begins with the most basic settings for the model:

```
---
#Currently only simple1d is allowed
config_type: simple1d

#luminosity any astropy.unit convertible to erg/s
#special unit log_lsun(log(luminosity) - log(L_sun)
luminosity: 9.44 log_lsun

#time since explosion
time_explosion: 13 day

atom_data: ../atom_data/kurucz_atom_chianti_many.h5
```

The `config_type` currently only allows `simple1d`, but might be expanded in future versions of TARDIS. Many parameters in the TARDIS configuration files make use of units. One can use any unit that is supported by astropy units as well as the unit `log_lsun` which is $\log(L) - \log(L_{\odot})$. Time since explosion just takes a normal time quantity. `atom_data` requires the path to the HDF5 file that contains the atomic data (more information about the HDF5 file can be found here *Atomic Data*).

5.1 Plasma

The next configuration block describes the plasma parameters:

```
plasma:
    initial_t_inner: 10000 K
    initial_t_rad: 10000 K
    disable_electron_scattering: no
    plasma_type: nebular
    #radiative_rates_type - currently supported are lte, nebular and detailed
    radiative_rates_type: detailed
    #line interaction type - currently supported are scatter, downbranch and macroatom
    line_interaction_type : macroatom
    w_epsilon : 1.0e-10
```

`inital_t_inner` is temperature of the black-body on the inner boundary. `initial_t_rad` is the radiation temperature for all cells. For debugging purposes and to compare to *synapps* calculations one can disable the electron scattering. TARDIS will issue a warning that this is not physical. There are currently two `plasma_type` options available: `nebular` and `lte` which tell TARDIS how to run the ionization equilibrium and level population calculations (see *Plasma* for more information). The radiative rates describe how to calculate the J_{blue} needed for the *nlte* calculations and *macroatom* calculations. There are three options for `radiative_rates_type`: 1) `lte` in which $J_{\mathrm{blue}} = \mathrm{Blackbody}(T_{\mathrm{rad}})$, 2) `nebular` in which $J_{\mathrm{blue}} = W \times \mathrm{Blackbody}(T_{\mathrm{rad}})$, 3) `detailed` in which the J_{blue} are calculated using an estimator (this is described in ??????).

TARDIS currently supports three different kinds of line interaction: `scatter` - a resonance scattering implementation, `macroatom` - the most complex form of line interaction described in *macroatom* and `downbranch` a simplified version of `macroatom` in which only downward transitions are allowed.

Finally, `w_epsilon` describes the dilution factor to use to calculate J_{blue} that are 0, which causes problem with the code (so J_{blue} are set to a very small number).

NLTE:

```
nlte:
    coronal_approximation: True
    classical_nebular: False
```

The NLTE configuration currently allows setting `coronal_approximation` which sets all J_{blue} to 0. This is useful for debugging with *chianti* for example. Furthermore one can enable 'classical_nebular' to set all β_{Sobolev} to 1. Both options are used for checking with other codes and should not be enabled in normal operations.

5.2 Model

The next sections, describing the model, are very hierarchical. The base level is `model` and contains two subsections: `structure` and `abundances`. Both sections can either contain a `file` subsection which specifies a file and file type where the information is stored or a number of other sections.

```
model:
    structure:
        no_of_shells : 20

        velocity:
            type : linear
            v_inner : 1.1e4 km/s
            v_outer : 2e4 km/s


        density:
            #showing different configuration options separated by comments
            #simple uniform:
            #---------------
#           type : uniform
#           value : 1e-12 g/cm^3
            #---------------

            #branch85_w7 - fit of seven order polynomial to W7 (like Branch 85):
            #---------------
            type : branch85_w7
            #value : 1e-12
            # default, no need to change!
            #time_0 : 19.9999584 s
            # default, no need to change!
```

```
            #density_coefficient : 3e29
            #---------------


#        file:
#            type : artis
#            name : artis_model.dat
#            v_lowest: 10000.0 km/s
#            v_highest: 20000.0 km/s
```

In the `structure` section, one can specify a `file` section containing a `type` parameter (currently only `artis` is supported‘‘) and a `name` parameter giving a path top a file. For the `artis` type, one can specify the inner and outermost shell by giving a `v_lowest` and `v_highest` parameter. This will result in the selection of certain shells which will be obeyed in the abundance section as well if `artis` is selected there as well.

Warning: If a `file` section is given, all other parameters and sections in the `structure` section are ignored!

If one doesn't specify a `file` section, the code requires two sections (`velocities` and `densities`) and a parameter `no_of_shells`. `no_of_shells` is the requested number of shells for a model. The `velocity` section requires a `type`. Currently, only `linear` is supported and needs two parameters `v_inner` and `v_outer` with velocity values for the inner most and outer most shell.

In the `densities` section the `type` parameter again decides on the parameters. The type `uniform` only needs a `value` parameter with a density compatible quantity. The type `branch85_w7` uses a seven order polynomial fit to the W7 model and is parametrised by time since explosion. The parameters `time_0` and `density_coefficient` are set to sensible defaults and should not be changed.

```
#-- continued from model block before --
    abundances:
        #file:
        #    type : artis
        #    name : artis_abundances.dat

        nlte_species : [Si2]
        C: 0.01
        O: 0.01
        Ne: 0.01
        Mg: 0.01
        Si: 0.45
        S: 0.35
        Ar: 0.04
        Ca: 0.03
        Fe: 0.07
        Co: 0.01
        Ni: 0.01
```

The `abundance` section again has a possible `file` parameter with `type` (currently only `artis` is allowed) and a `name` parameter giving a path to a file containing the abundance information.

Warning: In contrast to the `structure` section, the `abundance` section will not ignore abundances set in the rest of the section, but merely will overwrite the abundances given in the file section.

In this section we also specify the species that will be calculated with our *nlte* formalism using the `nlte_species` parameter (they are specified in a list using astrophysical notation, e.g. [Si2, Ca2, Mg2, H1]). The rest of the section can be used to configure uniform abundances for all shells, by giving the atom name and a relative abundance fraction. If it does not add up to 1., TARDIS will warn - but normalize the numbers.

5.3 MonteCarlo

The `montecarlo` section describes the parameters for the MonteCarlo radiation transport and convergence criteria:

```
montecarlo:
    seed: 23111963171620
    no_of_packets : 2.e+4
    iterations: 100

    convergence_criteria:
        type: specific
        damping_constant: 0.5
        threshold: 0.05
        fraction: 0.8
        hold: 3

#    convergence_criteria:
#        type: damped
#        damping_constant: 0.5
#        t_inner:
#            damping_constant: 0.7
```

The `seed` parameter seeds the random number generator first for the creation of the packets (ν and μ) and then the interactions in the actual MonteCarlo process. The `no_of_packets` parameter can take a float number for input convenience and gives the number of packets normally used in each MonteCarlo loop. The parameters `last_no_of_packets` and `no_of_virtual_packets` influence the last run of the MonteCarlo loop when the radiation field should have converged. `last_no_of_packets` is normally higher than `no_of_packets` to create a less noisy output spectrum. `no_of_virtual_packets` can also be set to greater than 0 to use the Virtual Packet formalism (reference missing ?????). The `iterations` parameter describes the maximum number of MonteCarlo loops executed in a simulation before it ends. Convergence criteria can be used to make the simulation stop sooner when the convergence threshold has been reached.

The `convergence_criteria` section again has a `type` keyword. Two types are allowed: `damped` and `specific`. All convergence criteria can be specified separately for the three variables for which convergence can be checked (`t_inner`, `t_rad`, `ws`) by specifying subsections in the `convergence_criteria` of the same name. These override then the defaults.

1. `damped` only has one parameter `damping-constant` and does not check for convergence.
2. **`specific` checks for the convergence threshold specified in `threshold`. For `t_rad` and `w` only a given** fraction (specified in `fraction`) has to cross the `threshold`. Once a convergence threshold is read, the simulation needs to hold this state for `hold` number of iterations.

5.4 Spectrum

The spectrum section defines the

```
spectrum:
    start : 500 angstrom
    end : 20000 angstrom
    bins : 1000
    sn_distance : lum_density
    #sn_distance : 10 Mpc
```

Start and end are given as Quantities with units. If they are given in frequency space they are switched around if necessary. The number of bins is just an integer. Finally the `sn_distance` can either be a distance or the special

parameter `lum_density` which sets the distance to $\sqrt{\frac{1}{4\pi}}$ to calculate the luminosity density.

5.5 Config Reader

The YAML file is read by using a classmethod of the `from_yaml()`.

CHAPTER

SIX

ATOMIC DATA

The atomic data for tardis is stored in hdf5 files. TARDIS ships with a relatively simple atomic dataset which only contains silicon lines and levels. TARDIS also has a full atomic dataset which contains the complete Kurucz dataset (http://kurucz.harvard.edu/LINELISTS/GFALL/). This full dataset also contains recombination coefficients from the ground state (ζ – factor used in *Calculating Zeta*) and data for calculating the branching or macro atom line interaction (*macroatom*).

6.1 HDF5 Dataset

As mentioned previously, all atomic data is stored in hdf5 files which contain tables that include mass, ionization, levels and lines data. The atom data that ships with TARDIS is located in data/atom

The dataset `basic_atom_set` contains the Atomic Number, Symbol of the elements and average mass of the elements.

6.1.1 Basic Atomic Data

Name	Description	Unit
atomic_number	Atomic Number (e.g. He = 2)	z
symbol	Symbol (e.g. He, Fe, Ca, etc.)	None
mass	Average mass of atom	u

The ionization data is stored in `ionization_data`.

6.1.2 Ionization Data

Name	Description	Unit
atomic_number(z)	Atomic Number	1
ion_number	Ion Number	1
ionization_energy	Ionization Energy of atom	eV

The levels data is stored in `levels_data`.

6.1.3 Levels Data

Name	Description	Unit
atomic_number(z)	Atomic Number	1
ion_number	Ion Number	1
level_number	Level Number	1
energy	Energy of a particular level	eV
g		1
metastable		bool

All lines are stored in `lines_data`.

6.1.4 Lines Data

Name	Description	Unit
wavelength	Waveslength	angstrom
atomic_number(z)	Atomic Number	1
ion_number	Ion Number	1
f_ul	Upper level probability	1
f_lu	Lower level probability	1
level_id_lower	Upper level id	1
level_id_upper	Lower level id	1

The next three datasets are only contained in the full dataset available upon request from the authors.

The factor correcting for photo-ionization from excited levels (needed in *Calculating Zeta*) is stored in the dataset `zeta_data`. The data is stored in a special way as one large `numpy.ndarray` where the first two columns are Atomic Number and Ion Number. All further columns are the $\zeta -$ factors for different temperatures. The temperatures are stored in the attribute `t_rads`.

Name	Description	Unit
atomic_number(z)	Atomic Number	1
ion_number	Ion Number	1
T_XXXX	Temperature for column	K
...	...	...
T_XXXX	Temperature for column	K

There are two datasets for using the macro atom and branching line interactions. The `macro_atom_data` and `macro_atom_references`:

The `macro_atom_data` contains blocks of transition probabilities, several indices and flags. The Transition Type flag has three states -1 for downwards emitting, 0 for downwards internal and 1 for upwards internal (for more explanations please refer to *macroatom*)

6.1.5 Macro Atom Data

Name	Description	Unit
atomic_number(z)	Atomic Number	1
ion_number	Ion Number	1
source_level_number	Source Level Number	1
destination_level_number	Destination Level Number	1
transition_type	Transition Type	1
transition_probability	Transition Probability	1
transition_line_id	Transition Line ID	1

Here's the structure of the probability block. The atomic number, ion number and source level number are the same within each block, the destination level number the transition type and transition probability are changing. The transition probabilities are only part of the final probability and will be changed during the calculation. For details on the macro atom please refer to *macroatom*.

Atomic Number	Ion Number	Source Level Number	Destination Level Number	Transition Type	Transition probabilities	Transition Line ID
Z_1	I_1	i_1	j_1	-1	$P_{\text{emission down } 1}$	k_1
Z_1	I_1	i_1	j_2	-1	$P_{\text{emission down } 2}$	k_2
...	...	...	...	...	...	...
Z_1	I_1	i_1	j_n	-1	$P_{\text{emission down } n}$	k_n
Z_1	I_1	i_1	j_1	0	$P_{\text{internal down } 1}$	k_1
Z_1	I_1	i_1	j_2	0	$P_{\text{internal down } 2}$	k_2
...	...	...	...	...	...	...
Z_1	I_1	i_1	j_n	0	$P_{\text{internal down } n}$	k_n
Z_1	I_1	i_1	j_1	1	$P_{\text{internal up } 1}$	k_1
Z_1	I_1	i_1	j_2	1	$P_{\text{internal up } 2}$	k_2
...	...	...	...	...	...	...
Z_1	I_1	i_1	j_n	1	$P_{\text{internal up } n}$	k_n

The `macro_references` dataset contains the numbers for each block:

6.1.6 Macro Atom References

Name	Description	Unit
atomic_number(z)	Atomic Number	1
ion_number	Ion Number	1
source_level_number	Source Level Number	1
count_down	Number of down transitions	1
count_up	Number of up transitions	1
count_total	Total number of transitions	1

6.2 The Atom Data Class

Atom Data is stored inside TARDIS in the `AtomData`-class. The class method `AtomData.from_hdf5()` will instantiate a new *AtomData*-class from an HDF5 file. If none is given it will automatically take the default HDF5-dataset shipped with TARDIS. A second function `AtomData.prepare_atom_data()` will cut the levels and lines data to only the required atoms and ions. In addition, it will create the intricate system of references needed by macro atom or branching line interactions.

6.3 Indexing fun

The main problem with the atomic data is indexing. Most of these references require multiple numbers, e.g. atomic number, ion number and level number. The **:py:module:'pandas'**-framework provides the ideal functions to accomplish this. In TARDIS we extensively use `pandas.MultiIndex`, `pandas.Series` and `pandas.DataFrame`

TO BE BETTER DOCUMENTED ...

CHAPTER

SEVEN

PLASMA

This module calculates the ionization balance and level populations in the BasePlasma, give a abundance fraction, temperature and density. After calculating the state of the plasma, these classes are able to calculate τ_{sobolev} for the supernova radiative transfer. The simplest BasePlasma (`BasePlasma`) only calculates the atom number densities, but serves as a base for all BasePlasma classes. The next more complex class is *LTEPlasma* which will calculate the aforementioned quantities in Local Thermal Equilibrium conditions (LTE). The `NebularPlasma`-class inherits from *LTEPlasma* and uses a more complex description of the BasePlasma (for details see *nebular_plasma*).

Note: In this documentation we use the indices i, j, k to mean atomic number, ion number and level number respectively.

All plasma calculations follow the same basic procedure in calculating the plasma state. This is always accomplished with the function `update_radiationfield`. This block diagram shows the basic procedure

7.1 Base Plasma

BasePlasma serves as the base class for all plasmas and can just calculate the atom number densities for a given input of abundance fraction.

$$N_{atom} = \rho_{\text{total}} \times \text{Abundance fraction} / m_{\text{atom}}$$

In the next step the line and level tables are purged of entries that are not represented in the abundance fractions are saved in *BasePlasma.levels* and *BasePlasma.lines*. Finally, the function *BasePlasma.update_t_rad* is called at the end of initialization to update the plasma conditions to a new $T_{\text{radiation field}}$ (with the give t_rad). This function is the same in the other plasma classes and does the main part of the calculation. In the case of *BasePlasma* this is only setting *BasePlasma.beta_rad* to $\frac{1}{k_{\text{B}} T_{\text{rad}}}$.

Here's an example how to instantiate a simple base plasma:

```
>>> from tardis import atomic, plasma
>>> atom_data = atomic.AtomData.from_hdf5()
>>> my_plasma = plasma.BasePlasma({'Fe':0.5, 'Ni':0.5}, 10000, 1e-13, atom_data)
>>> print my_plasma.abundances
atomic_number abundance_fraction number_density
------------- ------------------ --------------
           28                0.5  513016973.936
           26                0.5  539183641.472
```

7.2 Plasma Types

7.2.1 LTE Plasma

The *LTEPlasma* plasma class is the child of *BasePlasma* but is the first class that actually calculates plasma conditions. After running exactley through the same steps as *BasePlasma*, *LTEPlasma* will start calculating the partition functions.

$$Z_{i,j} = \sum_{k=0}^{max(k)} g_k \times e^{-E_k/(k_b T)}$$

, where Z is the partition function, g is the degeneracy factor, E the energy of the level and T the temperature of the radiation field.

The next step is to calculate the ionization balance using the Saha ionization equation. and then calculating the Number density of the ions (and an electron number density) in a second step. First $g_e = \left(\frac{2\pi m_e k_B T_{rad}}{h^2}\right)^{3/2}$ is calculated (in *LTEPlasma.update_t_rad*), followed by calculating the ion fractions (*LTEPlasma.calculate_saha*).

$$\frac{N_{i,j+1} \times N_e}{N_{i,j}} = \Phi_{i,(j+1)/j}$$

$$\Phi_{i,(j+1)/j} = g_e \times \frac{Z_{i,j+1}}{Z_{i,j}} e^{-\chi_{j \to j+1}/k_B T}$$

In the second step (*LTEPlasma.calculate_ionization_balance*), we calculate in an iterative process the electron density and the number density for each ion species.

$$N(X) = N_1 + N_2 + N_3 + \dots$$

$$N(X) = N_1 + \frac{N_2}{N_1} N_1 + \frac{N_3}{N_2}\frac{N_2}{N_1} N_1 + \frac{N_4}{N_3}\frac{N_3}{N_2}\frac{N_2}{N_1} N_1 + \dots$$

$$N(X) = N_1 (1 + \frac{N_2}{N_1} + \frac{N_3}{N_2}\frac{N_2}{N_1} + \frac{N_4}{N_3}\frac{N_3}{N_2}\frac{N_2}{N_1} + \dots)$$

$$N(X) = N_1 \underbrace{(1 + \frac{\Phi_{i,2/1}}{N_e} + \frac{\Phi_{i,2/2}}{N_e}\frac{\Phi_{i,2/1}}{N_e} + \frac{\Phi_{i,4/3}}{N_e}\frac{\Phi_{i,3/2}}{N_e}\frac{\Phi_{i,2/1}}{N_e} + \dots)}_{\alpha}$$

$$N_1 = \frac{N(X)}{\alpha}$$

Initially, we set the electron density (N_e) to the sum of all atom number densities. After having calculated the ion species number densities we recalculated the electron density by weighting the ion species number densities with their ion number (e.g. neutral ion number densities don't contribute at all to the electron number density, once ionized contribute with a factor of 1, twice ionized contribute with a factor of two, ...).

Finally we calculate the level populations (*LTEPlasma.calculate_level_populations*), by using the calculated ion species number densities:

$$N_{i,j,k} = \frac{g_k}{Z_{i,j}} \times N_{i,j} \times e^{-\beta_{rad} E_k}$$

This concludes the calculation of the plasma. In the code, the next step is calculating the $\tau_{Sobolev}$ using the quantities calculated here.

Example Calculations

```
import os
from matplotlib import pyplot as plt
from matplotlib import colors
from tardis import atomic, plasma, util
import numpy as np
import pandas as pd

#Making 2 Figures for ionization balance and level populations

plt.figure(1).clf()
ax1 = plt.figure(1).add_subplot(111)

plt.figure(2).clf()
ax2 = plt.figure(2).add_subplot(111)

# expanding the tilde to the users directory
atom_fname = os.path.expanduser('~/.tardis/si_kurucz.h5')

# reading in the HDF5 File
atom_data = atomic.AtomData.from_hdf5(atom_fname)

#The atom_data needs to be prepared to create indices. The Class needs to know which atomic numbers a
#calculation and what line interaction is needed (for "downbranch" and "macroatom" the code creates s
atom_data.prepare_atom_data([14], 'scatter')

#Initializing the NebularPlasma class using the from_abundance class method.
#This classmethod is normally only needed to test individual plasma classes
#Usually the plasma class just gets the number densities from the model class
lte_plasma = plasma.LTEPlasma.from_abundance(10000, {'Si': 1}, 1e-13, atom_data, 10.)


#Initializing a dataframe to store the ion populations  and level populations for the different tempe
ion_number_densities = pd.DataFrame(index=lte_plasma.ion_populations.index)
level_populations = pd.DataFrame(index=lte_plasma.level_populations.ix[14, 1].index)
t_rads = np.linspace(2000, 20000, 100)

#Calculating the different ion populations and level populuatios for the given temperatures
for t_rad in t_rads:
    lte_plasma.update_radiationfield(t_rad, w=1.0)
    #getting total si number density
    si_number_density = lte_plasma.number_density.get_value(14)
    #Normalizing the ion populations
    ion_density = lte_plasma.ion_populations / si_number_density
    ion_number_densities[t_rad] = ion_density

    #normalizing the level_populations for Si II
    current_level_population = lte_plasma.level_populations.ix[14, 1] / lte_plasma.ion_populations.ix
    #normalizing with statistical weight
    current_level_population /= atom_data.levels.ix[14, 1].g

    level_populations[t_rad] = current_level_population

ion_colors = ['b', 'g', 'r', 'k']

for ion_number in [0, 1, 2, 3]:
    current_ion_density = ion_number_densities.ix[14, ion_number]
```

```
    ax1.plot(current_ion_density.index, current_ion_density.values, '%s-' % ion_colors[ion_number],
             label='Si %s W=1.0' % util.int_to_roman(ion_number + 1).upper())


#only plotting every 5th radiation temperature
t_rad_normalizer = colors.Normalize(vmin=2000, vmax=20000)
t_rad_color_map = plt.cm.ScalarMappable(norm=t_rad_normalizer, cmap=plt.cm.jet)

for t_rad in t_rads[::5]:
    ax2.plot(level_populations[t_rad].index, level_populations[t_rad].values, color=t_rad_color_map.t
    ax2.semilogy()

t_rad_color_map.set_array(t_rads)
cb = plt.figure(2).colorbar(t_rad_color_map)

ax1.set_xlabel('T [K]')
ax1.set_ylabel('Number Density Fraction')
ax1.legend()

ax2.set_xlabel('Level Number for Si II')
ax2.set_ylabel('Number Density Fraction')
cb.set_label('T [K]')

plt.show()
```

7.2.2 Nebular Plasma

The *NebularPlasma* class is a more complex description of the Plasma state than the *LTEPlasma*. It takes a dilution factor (W) into account, which deals with the dilution of the radiation field due to geometric, line-blocking and other effects.

The calculations follow the same steps as *LTEPlasma*, however the calculations are different and often take into account if a particular level is *meta-stable* or not. *NebularPlasma* will start calculating the partition functions.

$$Z_{i,j} = \underbrace{\sum_{k=0}^{max(k)_{i,j}} g_k \times e^{-E_k/(k_b T)}}_{\text{metastable levels}} + \underbrace{W \times \sum_{k=0}^{max(k)_{i,j}} g_k \times e^{-E_k/(k_b T)}}_{\text{non-metastable levels}}$$

, where Z is the partition function, g is the degeneracy factor, E the energy of the level, T the temperature of the radiation field and W the dilution factor.

The next step is to calculate the ionization balance using the Saha ionization equation. and then calculating the Number density of the ions (and an electron number density) in a second step. In the first step, we calculate the ionization balance using the LTE approximation ($\Phi_{i,j}$(LTE)). Then we adjust the ionization balance using two factors ζ and δ.

Calculating Zeta

ζ is read in for specific temperatures and then interpolated for the target temperature.

Calculating Delta

δ is a radiation field correction factors which is calculated according to Mazzali & Lucy 1993 ([3]; henceforth ML93)

In ML93 the radiation field correction factor is denoted as δ and is calculated in Formula 15 & 20

The radiation correction factor changes according to a ionization energy threshold χ_{T} and the species ionization threshold (from the ground state) χ_0.

For $\chi_{\mathrm{T}} \geq \chi_0$

$$\delta = \frac{T_{\mathrm{e}}}{b_1 W T_{\mathrm{R}}} \exp\left(\frac{\chi_{\mathrm{T}}}{kT_{\mathrm{R}}} - \frac{\chi_0}{kT_{\mathrm{e}}}\right)$$

For $\chi_{\mathrm{T}} < \chi_0$

$$\delta = 1 - \exp\left(\frac{\chi_{\mathrm{T}}}{kT_{\mathrm{R}}} - \frac{\chi_0}{kT_{\mathrm{R}}}\right) + \frac{T_{\mathrm{e}}}{b_1 W T_{\mathrm{R}}} \exp\left(\frac{\chi_{\mathrm{T}}}{kT_{\mathrm{R}}} - \frac{\chi_0}{kT_{\mathrm{e}}}\right),$$

where T_{R} is the radiation field Temperature, T_{e} is the electron temperature and W is the dilution factor.

Now we can calculate the ionization balance using equation 14 in [3]:

$$\Phi_{i,j} = \frac{N_{i,j+1} n_e}{N_{i,j}}$$

$$\Phi_{i,j} = W \times [\delta\zeta + W(1-\zeta)] \left(\frac{T_{\mathrm{e}}}{T_{\mathrm{R}}}\right)^{1/2} \Phi_{i,j}(\mathrm{LTE})$$

In the last step, we calculate the ion number densities according using the methods in `LTEPlasma`

Finally we calculate the level populations (`NebularPlasma.calculate_level_populations()`), by using the calculated ion species number densities:

$$N_{i,j,k}(\text{not metastable}) = W \frac{g_k}{Z_{i,j}} \times N_{i,j} \times e^{-\beta_{\mathrm{rad}} E_k}$$

$$N_{i,j,k}(\text{metastable}) = \frac{g_k}{Z_{i,j}} \times N_{i,j} \times e^{-\beta_{\mathrm{rad}} E_k}$$

This concludes the calculation of the nebular plasma. In the code, the next step is calculating the τ_{Sobolev} using the quantities calculated here.

Example Calculations

```
import os

from matplotlib import colors
from tardis import atomic, plasma, util
from matplotlib import pyplot as plt

import numpy as np
import pandas as pd

#Making 2 Figures for ionization balance and level populations

plt.figure(1).clf()
ax1 = plt.figure(1).add_subplot(111)

plt.figure(2).clf()
ax2 = plt.figure(2).add_subplot(111)
```

```
# expanding the tilde to the users directory
atom_fname = os.path.expanduser('~/.tardis/si_kurucz.h5')

# reading in the HDF5 File
atom_data = atomic.AtomData.from_hdf5(atom_fname)

#The atom_data needs to be prepared to create indices. The Class needs to know which atomic numbers a
#calculation and what line interaction is needed (for "downbranch" and "macroatom" the code creates s
atom_data.prepare_atom_data([14], 'scatter')

#Initializing the NebularPlasma class using the from_abundance class method.
#This classmethod is normally only needed to test individual plasma classes
#Usually the plasma class just gets the number densities from the model class
nebular_plasma = plasma.NebularPlasma.from_abundance(10000, 0.5, {'Si': 1}, 1e-13, atom_data, 10.)


#Initializing a dataframe to store the ion populations  and level populations for the different tempe
ion_number_densities = pd.DataFrame(index=nebular_plasma.ion_populations.index)
level_populations = pd.DataFrame(index=nebular_plasma.level_populations.ix[14, 1].index)
t_rads = np.linspace(2000, 20000, 100)

#Calculating the different ion populations and level populuatios for the given temperatures
for t_rad in t_rads:
    nebular_plasma.update_radiationfield(t_rad, w=1.0)
    #getting total si number density
    si_number_density = nebular_plasma.number_density.get_value(14)
    #Normalizing the ion populations
    ion_density = nebular_plasma.ion_populations / si_number_density
    ion_number_densities[t_rad] = ion_density

    #normalizing the level_populations for Si II
    current_level_population = nebular_plasma.level_populations.ix[14, 1] / nebular_plasma.ion_popula
    #normalizing with statistical weight
    current_level_population /= atom_data.levels.ix[14, 1].g

    level_populations[t_rad] = current_level_population

ion_colors = ['b', 'g', 'r', 'k']

for ion_number in [0, 1, 2, 3]:
    current_ion_density = ion_number_densities.ix[14, ion_number]
    ax1.plot(current_ion_density.index, current_ion_density.values, '%s-' % ion_colors[ion_number],
             label='Si %s W=1.0' % util.int_to_roman(ion_number + 1).upper())


#only plotting every 5th radiation temperature
t_rad_normalizer = colors.Normalize(vmin=2000, vmax=20000)
t_rad_color_map = plt.cm.ScalarMappable(norm=t_rad_normalizer, cmap=plt.cm.jet)

for t_rad in t_rads[::5]:
    ax2.plot(level_populations[t_rad].index, level_populations[t_rad].values, color=t_rad_color_map.t
    ax2.semilogy()

#Calculating the different ion populations for the given temperatures with W=0.5
ion_number_densities = pd.DataFrame(index=nebular_plasma.ion_populations.index)
for t_rad in t_rads:
    nebular_plasma.update_radiationfield(t_rad, w=0.5)
    #getting total si number density
```

```
    si_number_density = nebular_plasma.number_density.get_value(14)
    #Normalizing the ion populations
    ion_density = nebular_plasma.ion_populations / si_number_density
    ion_number_densities[t_rad] = ion_density

    #normalizing the level_populations for Si II
    current_level_population = nebular_plasma.level_populations.ix[14, 1] / nebular_plasma.ion_popula
    #normalizing with statistical weight
    current_level_population /= atom_data.levels.ix[14, 1].g

    level_populations[t_rad] = current_level_population

#Plotting the ion fractions

for ion_number in [0, 1, 2, 3]:
    print "w=0.5"
    current_ion_density = ion_number_densities.ix[14, ion_number]
    ax1.plot(current_ion_density.index, current_ion_density.values, '%s--' % ion_colors[ion_number],
             label='Si %s W=0.5' % util.int_to_roman(ion_number + 1).upper())

for t_rad in t_rads[::5]:
    ax2.plot(level_populations[t_rad].index, level_populations[t_rad].values, color=t_rad_color_map.t
             linestyle='--')
    ax2.semilogy()

t_rad_color_map.set_array(t_rads)
cb = plt.figure(2).colorbar(t_rad_color_map)

ax1.set_xlabel('T [K]')
ax1.set_ylabel('Number Density Fraction')
ax1.legend()

ax2.set_xlabel('Level Number for Si II')
ax2.set_ylabel('Number Density Fraction')
cb.set_label('T [K]')

plt.show()
```

7.3 Sobolev optical depth

This function calculates the Sobolev optical depth τ_{Sobolev}

$$C_{\text{Sobolev}} = \frac{\pi e^2}{m_e c}$$

$$\tau_{\text{Sobolev}} = C_{\text{Sobolev}} \, \lambda \, f_{\text{lower}\rightarrow\text{upper}} \, t_{\text{explosion}} \, N_{\text{lower}} (1 - \frac{g_{\text{lower}}}{g_{\text{upper}}} \frac{N_{\text{upper}}}{N_{\text{lower}}})$$

7.4 Macro Atom

The macro atom is described in detail in [1]. The basic principal is that when an energy packet is absorbed that the macro atom is on a certain level. Three probabilities govern the next step the P_{up}, P_{down} and $P_{down\ emission}$ being the probability for going to a higher level, a lower level and a lower level and emitting a photon while doing this respectively (see Figure 1 in [1]).

The macro atom is the most complex idea to implement as a data structure. The setup is done in *~tardisatomic*, but we will nonetheless discuss it here (as *~tardisatomic* is even less documented than this one).

For each level we look at the line list to see what transitions (upwards or downwards are possible). We create a two arrays, the first is a long one-dimensional array containing the probabilities. Each level contains a set of probabilities, The first part of each set contains the upwards probabilities (internal upward), the second part the downwards probabilities (internal downward), and the last part is the downward and emission probability.

each set is stacked after the other one to make one long one dimensional *~numpy.ndarray*.

The second array is for book-keeping it has exactly the length as levels (with an example for the Si II level 15):

Level ID	Probability index	N_{up}	N_{down}	N_{total}
14001015	???	17	5	17 + 5*2 = 27

We now will calculate the transition probabilites, using the radiative rates in Equation 20, 21, and 22 in [1]. Then we calculate the downward emission probability from Equation 5, the downward and upward internal transition probabilities in [2].

$$
\begin{aligned}
p_{\text{emission down}} &= \mathcal{R}_{\text{i}\rightarrow\text{lower}}\,(\epsilon_{\text{upper}} - \epsilon_{\text{lower}})/\mathcal{D}_i \\
p_{\text{internal down}} &= \mathcal{R}_{\text{i}\rightarrow\text{lower}}\,\epsilon_{\text{lower}}/\mathcal{D}_i \\
, p_{\text{internal up}} &= \mathcal{R}_{\text{i}\rightarrow\text{upper}}\,\epsilon_i/\mathcal{D}_i \\
&,
\end{aligned}
$$

where i is the current level, ϵ is the energy of the level, and $\mathcal{R}$ is the radiative rates.

We ignore the probability to emit a k-packet as TARDIS only works with photon packets. Next we calculate the radidative rates using equation 10 in [2].

$$
\begin{aligned}
\mathcal{R}_{\text{upper}\rightarrow\text{lower}} &= A_{\text{upper}\rightarrow\text{lower}}\beta_{\text{Sobolev}}n_{\text{upper}} \\
\mathcal{R}_{\text{lower}\rightarrow\text{upper}} &= (B_{\text{lower}\rightarrow\text{upper}}n_{\text{lower}} - B_{\text{upper}\rightarrow\text{lower}}n_{\text{upper}})\beta_{\text{Sobolev}}J^b_\nu \\
&,
\end{aligned}
$$

with $\beta_{\text{Sobolev}} = \frac{1}{\tau_{\text{Sobolev}}}(1 - e^{-\tau_{\text{Sobolev}}})$.

using the Einstein coefficients

$$
\begin{aligned}
A_{\text{upper}\rightarrow\text{lower}} &= \frac{8\nu^2\pi^2e^2}{m_ec^3}\,\frac{g_{\text{lower}}}{g_{\text{upper}}}\,f_{\text{lower}\rightarrow\text{upper}} \\
A_{\text{upper}\rightarrow\text{lower}} &= \underbrace{\frac{4\pi^2e^2}{m_ec}}_{C_{\text{Einstein}}}\,\frac{2\nu^2}{c^2}\,\frac{g_{\text{lower}}}{g_{\text{upper}}}\,f_{\text{lower}\rightarrow\text{upper}} \\
B_{\text{lower}\rightarrow\text{upper}} &= \frac{4\pi^2e^2}{m_eh\nu c}\,f_{\text{lower}\rightarrow\text{upper}} \\
B_{\text{lower}\rightarrow\text{upper}} &= \underbrace{\frac{4\pi^2e^2}{m_ec}}_{C_{\text{Einstein}}}\,\frac{1}{h\nu}f_{\text{lower}\rightarrow\text{upper}} \\
B_{\text{upper}\rightarrow\text{lower}} &= \frac{4\pi^2e^2}{m_eh\nu c}\,f_{\text{lower}\rightarrow\text{upper}} \\
B_{\text{upper}\rightarrow\text{lower}} &= \underbrace{\frac{4\pi^2e^2}{m_ec}}_{C_{\text{Einstein}}}\,\frac{1}{h\nu}\,\frac{g_{\text{lower}}}{g_{\text{upper}}}f_{\text{lower}\rightarrow\text{upper}}
\end{aligned}
$$

we get

$$\mathcal{R}_{\text{upper}\rightarrow\text{lower}} = C_{\text{Einstein}} \frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{upper}}} f_{\text{lower}\rightarrow\text{upper}} \beta_{\text{Sobolev}} n_{\text{upper}}$$

$$\mathcal{R}_{\text{lower}\rightarrow\text{upper}} = C_{\text{Einstein}} \frac{1}{h\nu} f_{\text{lower}\rightarrow\text{upper}} (n_{\text{lower}} - \frac{g_{\text{lower}}}{g_{\text{upper}}} n_{\text{upper}}) \beta_{\text{Sobolev}} J^b_\nu$$

This results in the transition probabilities:

$$p_{\text{emission down}} = C_{\text{Einstein}} \frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{i}}} f_{\text{lower}\rightarrow\text{i}} \beta_{\text{Sobolev}} n_{\text{i}} \left(\epsilon_{\text{i}} - \epsilon_{\text{lower}}\right) / \mathcal{D}_i$$

$$p_{\text{internal down}} = C_{\text{Einstein}} \frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{i}}} f_{\text{lower}\rightarrow\text{i}} \beta_{\text{Sobolev}} n_{\text{i}} \, \epsilon_{\text{lower}} / \mathcal{D}_i$$

$$p_{\text{internal up}} = C_{\text{Einstein}} \frac{1}{h\nu} f_{\text{i}\rightarrow\text{upper}} (n_{\text{i}} - \frac{g_{\text{i}}}{g_{\text{upper}}} n_{\text{upper}}) \beta_{\text{Sobolev}} J^b_\nu \, \epsilon_i / \mathcal{D}_i$$

,

and as we will normalise the transition probabilities numerically later, we can get rid of C_{Einstein}, $\frac{1}{\mathcal{D}_i}$ and number densities.

$$p_{\text{emission down}} = \frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{i}}} f_{\text{lower}\rightarrow\text{i}} \beta_{\text{Sobolev}} \left(\epsilon_{\text{i}} - \epsilon_{\text{lower}}\right)$$

$$p_{\text{internal down}} = \frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{i}}} f_{\text{lower}\rightarrow\text{i}} \beta_{\text{Sobolev}} \, \epsilon_{\text{lower}}$$

$$p_{\text{internal up}} = \frac{1}{h\nu} f_{\text{i}\rightarrow\text{upper}} \underbrace{(1 - \frac{g_{\text{i}}}{g_{\text{upper}}} \frac{n_{\text{upper}}}{n_i})}_{\text{stimulated emission}} \beta_{\text{Sobolev}} J^b_\nu \, \epsilon_i$$

,

There are two parts for each of the probabilities, one that is pre-computed by ~*tardisatomic* and is in the HDF5 File, and one that is computed during the plasma calculations:

$$p_{\text{emission down}} = \underbrace{\frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{i}}} f_{\text{lower}\rightarrow\text{i}} (\epsilon_{\text{i}} - \epsilon_{\text{lower}})}_{\text{pre-computed}} \beta_{\text{Sobolev}}$$

$$p_{\text{internal down}} = \underbrace{\frac{2\nu^2}{c^2} \frac{g_{\text{lower}}}{g_{\text{i}}} f_{\text{lower}\rightarrow\text{i}} \epsilon_{\text{lower}}}_{\text{pre-computed}} \beta_{\text{Sobolev}}$$

$$p_{\text{internal up}} = \underbrace{\frac{1}{h\nu} f_{\text{i}\rightarrow\text{upper}}}_{\text{pre-computed}} \beta_{\text{Sobolev}} J^b_\nu \, (1 - \frac{g_{\text{i}}}{g_{\text{upper}}} \frac{n_{\text{upper}}}{n_i}) \, \epsilon_i.$$

7.5 NLTE treatment

NLTE treatment of lines is available both in ~LTEPlasma and the ~NebularPlasma class. This can be enabled by specifying which species should be treated as NLTE with a simple list of tuples (e.g. [(20,1)] for Ca II).

First let's dive into the basics:

There are two rates to consider from a given level.

$$
\begin{aligned}
R_{\text{upper}\to\text{lower}} &= \underbrace{A_{ul} n_u}_{\text{spontaneous emission}} + \underbrace{B_{ul} n_u \bar{J}_\nu}_{\text{stimulated emission}} + \underbrace{C_{ul} n_u n_e}_{\text{collisional deexcitation}} \\
&= n_u \underbrace{(A_{ul} + B_{ul} \bar{J}_\nu + C_{ul} n_e)}_{r_{ul}} \\
R_{\text{lower}\to\text{upper}} &= \underbrace{B_{lu} n_l \bar{J}_\nu}_{\text{stimulated absorption}} + \underbrace{C_{lu}\, n_l\, n_e}_{\text{collisional excitation}} \\
&= n_l \underbrace{(B_{lu} \bar{J}_\nu + C_{ul} n_e)}_{r_{lu}},
\end{aligned}
$$

where $\bar{J}_\nu$ (in LTE this is $B(\nu, T)$) denotes the mean intensity at the frequency of the line and n_e the number density of electrons.

Next, we calculate the rate of change of a level by adding up all outgoing and all incoming transitions from level j.

$$
\frac{dn_j}{dt} = \underbrace{\sum_{i \neq j} R_{ij}}_{\text{incoming rate}} - \underbrace{\sum_{i \neq j} R_{ji}}_{\text{outgoing rate}}
$$

In a statistical equilibrium all incoming rates and outgoing rates add up to 0 ($\frac{dn_j}{dt} = 0$). We use this to calculate the level populations using the rate coefficients ($r_i j, r_j i$).

$$
\begin{pmatrix}
-(\mathcal{R}_{\infty\in} + \cdots + \mathcal{R}_{\infty|}) & \cdots & \mathcal{R}_{|\infty} \\
\vdots & \ddots & \vdots \\
\mathcal{R}_{\infty|} & \cdots & -(\mathcal{R}_{|\infty} + \cdots + \mathcal{R}_{|,|-\infty})
\end{pmatrix}
\begin{pmatrix} n_1 \\ \vdots \\ n_j \end{pmatrix} =
\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
$$

with the additional constrained that all the level number populations need to add up to the current ion population N we change this to

$$
\begin{pmatrix}
1 & 1 & \cdots \\
\vdots & \ddots & \vdots \\
\mathcal{R}_{\infty|} & \cdots & -(\mathcal{R}_{|\infty} + \cdots + \mathcal{R}_{|,|-\infty})
\end{pmatrix}
\begin{pmatrix} n_1 \\ \vdots \\ n_j \end{pmatrix} =
\begin{pmatrix} N \\ 0 \\ 0 \end{pmatrix}
$$

For a three level atom we have:

$$
\begin{aligned}
\frac{dn_1}{dt} &= \underbrace{n_2 r_{21} + n_3 r_{31}}_{\text{incoming rate}} - \underbrace{(n_1 r_{12} + n_1 r_{13})}_{\text{outgoing rate}} = 0 \\
\frac{dn_2}{dt} &= \underbrace{n_1 r_{12} + n_3 r_{32}}_{\text{incoming rate}} - \underbrace{(n_2 r_{21} + n_2 r_{23})}_{outgoingrate} = 0 \\
\frac{dn_3}{dt} &= \underbrace{n_1 r_{13} + n_2 r_{23}}_{\text{incoming rate}} - \underbrace{(n_3 r_{32} + n_3 r_{31})}_{\text{outgoing rate}} = 0,
\end{aligned}
$$

which can be written in matrix from:

$$
\begin{pmatrix}
-(r_{12} + r_{13}) & r_{21} & r_{31} \\
r_{12} & -(r_{21} + r_{23}) & r_{32} \\
r_{13} & r_{23} & -(r_{31} + r_{32})
\end{pmatrix}
\begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} =
\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
$$

To solve for the level populations we need an additional constraint: $n_1 + n_2 + n_3 = N$. By setting $N = 1$: we can get the relative rates:

$$\begin{pmatrix} 1 & 1 & 1 \\ r_{12} & -(r_{21} + r_{23}) & r_{32} \\ r_{13} & r_{23} & -(r_{31} + r_{32}) \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Now we go back and look at the rate coefficients used for a level population - as an example $\frac{dn_2}{dt}$:

$$\begin{aligned} \frac{dn_2}{dt} &= n_1 r_{12} - n_2(r_{21} + r_{23}) + n_3 r_{32} \\ &= n_1 B_{12}\bar{J}_{12} + n_1 C_{12} n_e - n_2 A_{21} - n_2 B_{21}\bar{J}_{21} - n_2 C_{21} n_e \\ &\quad - n_2 B_{23}\bar{J}_{23} - n_2 C_{23} n_e + n_3 A_{32} + n_3 B_{32}\bar{J}_{32} + n_3 C_{32} n_e, \\ &\quad + n_3 A_{32} + n_3 C_{32} n_e, \end{aligned}$$

Next we will group the stimulated emission and stimulated absorption terms as we can assume $\bar{J_{12}} = \bar{J_{21}}$:

$$\frac{dn_2}{dt} = n_1(B_{12}\bar{J}_{12} \underbrace{(1 - \frac{n_2}{n_1}\frac{B_{21}}{B_{12}})}_{\text{stimulated emission term}} + C_{12} n_e) - n_2(A_{21} + C_{23} n_e + n_2 B_{23}\bar{J}_{23} \underbrace{(1 - \frac{n_3}{n_2}\frac{B_{32}}{B_{23}})}_{\text{stimulated emission term}}) + n_3(A_{32} + C_{32} n_e)$$

CHAPTER

EIGHT

RADIATIVE MONTE CARLO

The radiative monte carlo is initiated once the model is constructed.

Different line interactions

line_interaction_id == 0: scatter line_interaction_id == 1: downbranch line_interaction_id == 2: macro

8.1 Radiationfield estimators

During the monte-carlo run we collect two estimators for the radiation field:

$$J_{\text{estimator}} = \sum \epsilon l$$
$$\bar{\nu}_{\text{estimator}} = \sum \epsilon \nu l,$$

where ϵ, ν are comoving energy and comoving frequency of a packet respectively.

To calculate the temperature and dilution factor we first calculate the mean intensity in each cell ($J = \frac{1}{4\pi\,\Delta t\,V} J_{\text{estimator}}$)., [2].

The weighted mean frequency is used to obtain the radiation temperature. Specifically, the radiation temperature is chosen as the temperature of a black body that has the same weighted mean frequency as has been computed in the simulation. Accordingly,

$$\frac{h\bar{\nu}}{k_B T_R} = \frac{h}{k_B T_R} \frac{\bar{\nu}_{\text{estimator}}}{J_{\text{estimator}}} = 24\zeta(5)\frac{15}{\pi^4},$$

where the evaluation comes from the mean value of

$$\bar{x} = \frac{\int_0^\infty x^4/(\exp x - 1)dx}{\int_0^\infty x^3/(\exp x - 1)dx} = 24\zeta(5)\frac{15}{\pi^4} = 3.8322\ldots$$

and so

$$\begin{aligned} T_R &= \frac{1}{\bar{x}} \frac{h}{k_B} \frac{\bar{\nu}_{\text{estimator}}}{J_{\text{estimator}}} \\ &= 0.260945 \frac{h}{k_B} \frac{\bar{\nu}_{\text{estimator}}}{J_{\text{estimator}}}. \end{aligned}$$

With the radiation temperature known, we can then obtain our estimate for for the dilution factor. Our radiation field model in the nebular approximation is

$$J = WB(T_R) = W\frac{\sigma_{SB}}{\pi}T_R^4,$$

i.e. a dilute blackbody. Therefore we use our value of the mean intensity derrived from the estimator (above) to obtain the dilution factor

$$W = \frac{\pi J}{\sigma_{SB} T_R^4} = \frac{1}{4\sigma_{SB} T_R^4 \, \Delta t \, V} J_{\text{estimator}}.$$

There endeth the lesson.

CHAPTER

NINE

GLOSSARY

meta-stable A level is considered meta-stable if there's no line that has this level as the upper state. This means that these levels can not be reached by absorbing a photon and are only reached when decaying.

synapps simple radiative transport code for supernovae. Please refer to Synapps for more information

CHAPTER

TEN

REFERENCES

BIBLIOGRAPHY

[1] L. B. Lucy. Monte Carlo transition probabilities. *aap*, 384:725–735, March 2002.

[2] L. B. Lucy. Monte Carlo transition probabilities. II.. *aap*, 403:261–275, May 2003.

[3] P. A. Mazzali and L. B. Lucy. The application of Monte Carlo methods to the synthesis of early-time supernovae spectra. *aap*, 279:447–456, November 1993.

INDEX