T. E. A.

THERMOCHEMICAL EQUILIBRIUM ABUNDANCES

Code Description

Authors:
JASMINA BLECIC M. OLIVER BOWMAN

Programmers:
M. OLIVER BOWMAN
JASMINA BLECIC

Lead Scientist:
JASMINA BLECIC

Principal Investigator:
JOSEPH HARRINGTON

March 23, 2015

This document goes along with the TEA code, the TEA theory paper (Blecic et al., 2015), and the user manual. The documents are not peer-reviewed and the code is a test version only. They may not be redistributed to any third party. Please refer such requests to us. If you find this package useful for your research, please cite Blecic et al. (2015). The document and the program are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Our intent is to release this software under an open-source, reproducible-research license, once the code is mature and the first research paper describing the code has been accepted for publication in a peer-reviewed journal. We are committed to development in the open, and have posted this code on github.com so that others can test it and give us feedback. However, until its first publication and first stable release, we do not permit others to redistribute the code in either original or modified form, nor to publish work based in whole or in part on the output of this code. By downloading, running, or modifying this code, you agree to these conditions. We do encourage sharing any modifications with us and discussing them openly.

We welcome your feedback, but do not guarantee support. Please send feedback or inquiries to both:
Jasmina Blecic: jasmina@physics.ucf.edu
Joseph Harrington: jh@physics.ucf.edu

or alternatively,

Jasmina Blecic and Joseph Harrington
UCF PSB 441
4111 Libra Drive
Orlando, FL 32816-2385
USA

Thank you for testing TEA!

Contents

1 Code Overview

The Thermochemical Equilibrium Abundances (TEA) package (Figure 1 is composed of four different types of programs: `Library Programs` that read thermodynamic libraries and collect stoichiometric information, `TEA Drivers` that execute TEA for a single T, P point or a list of multiple T, P points, `Main Science Programs` that performs the iterative Lagrange minimization technique and Lambda correction algorithm, `File Control Programs` that manages input-output files and operations, and `Auxiliary Programs` that help the user to make proper input and plot the TEA output.

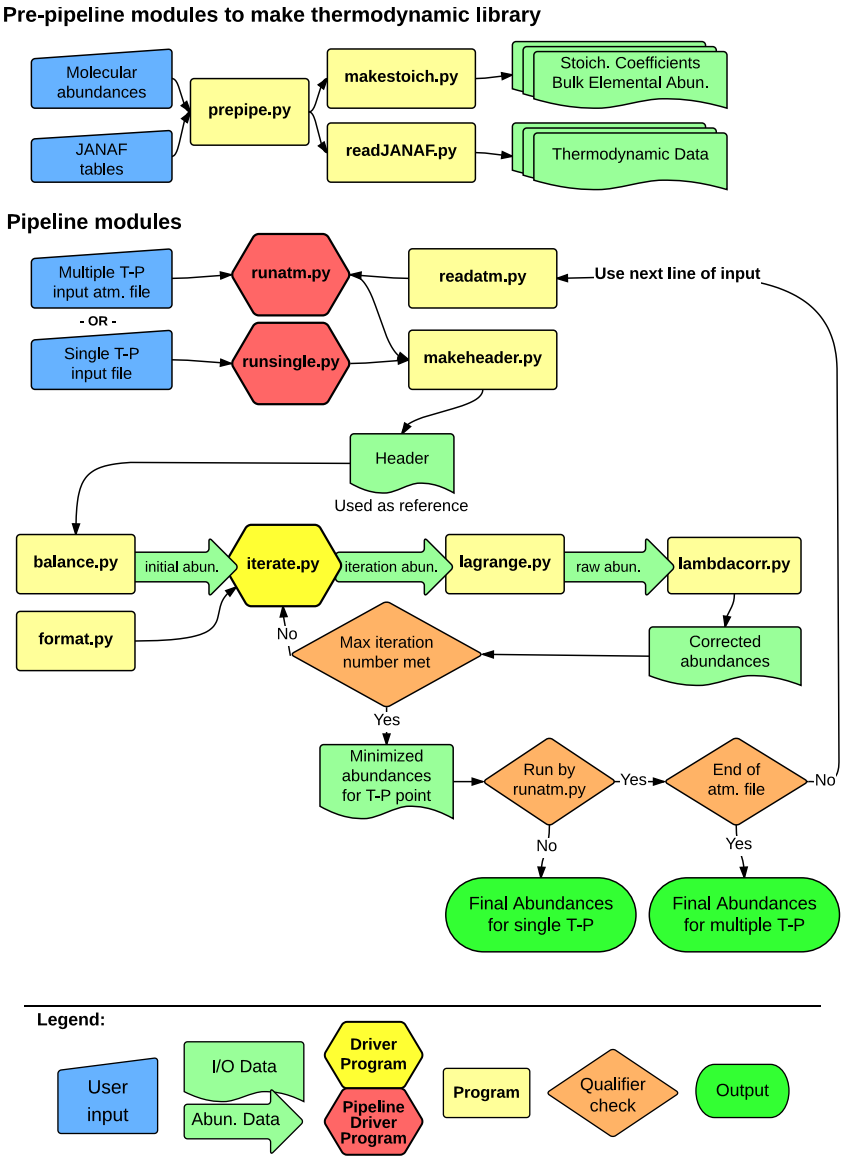

Figure 1: Layout of the pre-pipeline and pipeline packages.

The code is built in a modular fashion where each program performing calculations can be easily controlled by a driver program or be otherwise replaced as long as the appropriate inputs and outputs are conserved. This modular work-flow facilitates strict control over what calculations are performed and where, and allows future users easy access to code manipulations or additions. As the TEA code is an open-source package, such modifications are encouraged if the user wishes

to fine-tune any mechanics or apply new techniques in order to reach equilibrium abundances.

This is a third part of a three-part document describing the TEA code. The first part, the TEA theory document (Blecic et al., 2015), presents the theoretical basis for the method applied, the second part is a user manual, and this document contains programs description to allow the user future modifications. If you find this package useful for your research, please cite Blecic et al. (2015).

This project was completed with the support of the NASA Earth and Space Science Fellowship Program, grant NNX12AL83H, held by Jasmina Blecic, PI Joseph Harrington. Project development included graduate student Jasmina Blecic and undergraduate M. Oliver Bowman.

2 Code Description

The text is divided in 5 sections, each describing the aforementioned types of programs with the detailed description of each routine.

2.1 Library Programs

The TEA `Library Programs` read the thermodynamic tables and the elemental abundances file and store necessary thermochemical data and stoichiometric information in files. These files are further used by the TEA `Driver programs` and `Main Scientific Programs`. The library programs consists of three modules: `prepipe.py`, `readJANAF.py` and `makestoich.py`. Currently, these modules process the JANAF tables and the elemental abundances file made based on Asplund et al. (2009), both provided with the code. The user can feed the code with its own library, granted the format that TEA can process is obeyed.

These programs are executed prior to the release, and the thermochemical data and stoichiometric information needed for TEA to run are provided with the code. However, if updated JANAF tables are obtained, the pre-pipeline execution will populate the files with new information.

2.1.1 prepipe.py

This program sets and/or executes the pre-pipeline TEA routines `readJANAF.py` and `makestoich.py`. It consists of two functions: `comp()` and `setup()`. The `comp()` function counts the number of each element in a chemical species, while the `setup()` function reads the JANAF tables and allows sharing of common routines for `readJANAF.py` and `makestoich.py`. If executed as `prepipe.py`, it will run both routines. If desired, user can run each routine separately.

The `comp()` function is the species counting function. It counts the number of each element in a chemical species, by taking in a string of a chemical species (i.e., ”H2O”) and returning an array containing every element with corresponding counts found in that species. It is called by `makestoich.py` and the `setup()` function. If desired, user can return stoichiometric array containing only the elements found in the input species. Otherwise, it returns the full array of all 113 available elemental stoichiometric data.

The `setup()`routine reads raw JANAF tables placed in the appropriate directory (default: `janaf/`) and extracts thermodynamic and stoichiometric data of interest. It serves as a setup for the `readJANAF.py` and `makestoich.py` routines. The program takes the names of the raw JANAF tables directory, thermodynamic output directory, and stoichiometric output directory, and

the names of the output stoichiometric file and the pre-written file containing abundance data (default: `abundances.txt`). It takes number of elements from `comp()` and loops over all JANAF data and abundance data, performing various thermodynamic calculations and stoichiometric functions on the appropriate species of interest.

2.1.2 makestoich.py

This code makes the `stoich_out` file (default: `stoich.txt`) that carries stoichiometric values of all species that appear in the JANAF tables. It reads the chemical formula from each JANAF file to obtain the number of atomic weights of each element in each species. Also reads in bulk elemental abundances from `abundances.txt` that currently uses Asplund et al. (2009) solar photosphere abundances. The code creates a temporary directory where JANAF tables are converted into stoichiometric tables. The tables carry a unique name and state, given by the top line in each JANAF table: (i.e., original JANAF table `Al-001.txt` is converted to `Al_ref.txt`). If desired, user can preserve this directory (in `TEA_config.py`, doprint = `True`).

The names of the files in the temporary directory have the following format:

1. If a species appears just once in JANAF tables, it gets a unique name of the compound and its state and is defined as 'originals' in the code (example: $CH4_g$).
2. If a species appears several times, it is defined as 'redundant' in the code and an additional string is added to differentiate among them. (example: Al2O3_cr_Alpha, Al2O3_cr_Beta, Al2O3_cr_Kappa).

The setup of this code is made in `prepipe.py` inside `setup()`. The code retrieves the setup information, creates a temporary directory to store the converted files, allocates space for these files with a unique, original names and files with redundant names, then loops over all JANAF tables to write stoichiometric files.

To write the `stoich_out` file (default: `stoich.txt`), it again loops over all elements (listed in `comp()` inside `prepipe.py`) and matches them with the abundance data from abundance file (default: abundances.txt). If a match is not found, the abundance is set to zero. This information is written at the top of the `stoich_out` file.

For each species in the JANAF tables and elements in `comp()`, the species name and stoichiometric values are written into the `stoich_out` file. The code ignores redundant compounds as they carry the same stoichiometric values.

The `prepipe.py` code can execute this code together with the `readJANAF.py`. `makestoich.py` can also be executed on its own with the simple command: `makestoich.py`.

2.1.3 readJANAF.py

This code makes the `thermo_dir` (default: `/lib/gdata`) directory that carries converted JANAF tables with only the data needed for TEA to run: the temperature, T in (K), free energy function $(-[G^o - H^o(T_r)]/T)$ in (J/K/mol), and the heat of formation $(\Delta_f H^o)$, in (kJ/mol). This program will remove intermittent commentary in the JANAF files and will produce separate files for stoichiometrically identical species with unique descriptors. Output data files are produced in the format /lib/gdata/SPECIES_STATE.txt. The code also makes `conversion_record.txt` that gives the names of the original, raw JANAF files and the new names given by the

`readJANAF.py`. To sort the file list in alphabetical order, execute in terminal: `sort conversion_record. > conversion_record_sorted.txt`

The names of the files have the following format:

1. Each converted file carries a unique name and state given by the top line in each JANAF file.
2. If a species appears just once in JANAF tables, it gets a unique name of the compound and its state and is defined as 'originals' in the code (example: $CH4_g$).
3. If a species appears several times, it is defined as 'redundant' in the code and an additional string is added to differentiate among them. (example: Al2O3_cr_Alpha, Al2O3_cr_Beta, Al2O3_cr_Kappa).
4. If a species is an ion, additional string is added (example: Al-007.txt and Al-008.txt became Al_ion_n_g, Al_ion_p_g)

The setup of this code is made in `prepipe.py` inside the `setup()` function. The code retrieves pre-pipeline setup information, creates a directory for converted thermodynamic files, checks whether a species is redundant in JANAF tables, and creates `conversion_record.txt`. If a species is redundant, an additional string is added to the file name. The module loops over all JANAF tables and writes data into the correct columns with the correct labels.

The `prepipe.py` code can execute this code together with the `readJANAF.py`. `readJANAF.py` can also be executed on its own with the simple command: `readJANAF.py`.

2.2 TEA Drivers

TEA is executed by one of two drivers, depending a single T, P point or a list of T, P points are provided. Both drivers follow the same general flow of execution.

2.2.1 runsingle.py

This program runs TEA over an input file that contains only one T, P point (see Section **??**. The code retrieves the input file and the current directory name given by the user, and sets locations of all necessary modules and directories that will be used. Then, it executes the modules in the following order: `makeheader.py, balance,py`, and `iterate.py`. The final results with the input and the configuration files are saved in the em results/ directory.

This module prints on screen the code progress: the current T, P line from the pre-atm file, the current iteration number, and informs the user that minimization is done. Example: 100 Maximum iteration reached, ending minimization.

The program is executed with in-shell inputs:
`runsingle.py <input file> <name of the result directory>`
Example: `runsingle.py inputs/Examples/inp_Example.txt Single_Example`

2.2.2 runatm.py

This program runs TEA over a pre-atm file that contains multiple T, P's. The code first retrieves the pre-atm file, and the current directory name given by the user. Then, it sets locations of all necessary modules and directories of files that will be used. It allocates an array to store the final abundances for each species of each T, P run. The program loops over all lines (T, P's) in the pre-atm file and executes the modules in the following order: `readatm.py,`

`makeheader.py, balance,py, iterate.py`, and `readoutput.py`. `Iterate.py` executes `lagrange.py` and `lambdacorr.py`. `readoutput.py` reads results from the TEA iteration loop executed in `iterate.py`. The abundances are calculated and stored in an abundance array. Then, the rad() function is called from the `radpress.py` module to calculate radii for each pressure in the atmosphere. The code then opens the final atm file to write the results. It takes first common lines from the pre-atm file and writes the data from the stored radii, temperature, pressure and abundances array. The code has a condition to save or delete all intermediate files, time stamps for checking the speed of execution, and is verbose for debugging purposes. If these files are saved, the function will create a unique directory for each T, P point. This functionality is controlled in the TEA_config file. The final results with the input and the configuration files are saved in the results/ directory.

This module prints on screen the current T, P line from the pre-atm file, the current iteration number, and informs the user that minimization is done. Example: 5 100 Maximum iteration reached, ending minimization.

The program is executed with in-shell inputs:
`runatm.py <pre-atm file> <name of the result directory>`
Example: `runatm.py inputs/Examples/atm_Example.atm Atm_Example`

2.3 Main Scientific Programs

These programs perform scientific calculations explained in the Blecic et al. (2015), Sections 2, 3, and 4. The mass balance equation, the Lagrange optimization system of equations, the Lambda correction procedure, and the iterative minimization approach are separated in four modules, respectively: `balance.py`, `lagrange.py`, `lambdacorr.py`, and `iterate.py`. Including condensates or solids can be easily implemented inside these modules, by modifying the equations listed in the code comments.

2.3.1 balance.py

This code produces an initial guess for the first TEA iteration by fulfilling the mass balance condition, $\sum_{i=1}^{n} a_{ij}\, x_i = b_j$ (equation (17) in Blecic et al., 2015), where i is species index, j is element index, a's are stoichiometric coefficients, and b's are elemental fractions by number, i.e., ratio of number densities of element 'j' to the total number densities of all elements in the system (see the end of the Section 2 in Blecic et al., 2015). The code writes the result into machine- and human-readable files.

The code begins by making a directory for the output results. Then, it reads the header file and imports all relevant chemical data from it. To satisfy the mass balance equation, some y_i variables remain as free parameters. The number of free parameters is set to the number of total elements in the system, thus ensuring that the mass balance equation can be solved for any number of input elements and output species the user chooses. The code locates a chunk of species (y_i) containing a sum of ai_j values that forbids ignoring any element in the system (sum of the ai_j values in a column must not be zero). This chunk is used as a set of free variables in the system. The initial scale for other y_i variables are set to a known, arbitrary number. Initially, starting values for the known species are set to 0.1 moles, and the mass balance equation is calculated. If this value does not produce all positive mole numbers, the code automatically sets known parameters to 10 times smaller and tries again. Actual mole numbers for the initial guesses of y_i are arbitrary, as TEA only requires a balanced starting point to initialize minimization. The goal of this code

is to find a positive set of non-zero mole numbers to satisfy this requirement. Finally, the code calculates $\bar{y}$, initializes the iteration number, Δ, and $\bar{\Delta}$ to zero and writes results into machine- and human-readable output files.

This code is called by `runatm.py` and `runsingle.py` and can be executed alone with in-shell input: `balance.py <HEADER_FILE> <DIRECTORY_NAME>`

2.3.2 lagrange.py

This code applies Lagrange's method and calculates minimum based on the methodology elaborated in Blecic et al. (2015) in Section (3). Equations in this code contain both references and an explicitly written definitions. The program reads the last iteration's output and data from the last header file, creates variables for the Lagrange equations, sets up the Lagrange equations, and calculates final x_i mole numbers for the current iteration cycle. Note that the mole numbers that result from this function are allowed to be negative. If negatives are returned, lambda correction `lambdacorr.py` is necessary. The final x_i values, as well as $\bar{x}$, $\bar{y}$, Δ, and $\bar{\Delta}$ are written into machine- and human-readable output files. This function is executed `iterate.py` and can be run independently.

2.3.3 lambdacorr.py

This module applies lambda correction method (see Section 4 in Blecic et al., 2015). When input mole numbers are negative, the code corrects them to positive values and pass them to the next iteration cycle. The code reads the values from the last lagrange output, the information from the header file, performs checks, and starts setting basic equations. It sets a smart range so it can efficiently explore the lambda values from [0,1]. Half of the range is sampled exponentially, and the other half linearly, totalling 150 points. The code retrieves the last lambda value before first derivative becomes positive (equation (34) in TEA theory document), and corrects negative mole numbers to positive.

The code works without adjustments and with high precision for the fractional abundances (mixing fractions) up to 10^{-14} and the temperature range of 1000 - 4000 K. For temperatures below 1000 K and mixing fractions below 10^{-14}, the code produces results with low precision. To improve the precision, adjust the lambda exploration variables `lower` and `steps` to larger magnitudes (i.e., `lower` = -100, `steps` = 1000). This will lengthen the time of execution.

This function is executed `iterate.py` and can be run independently.

2.3.4 iterate.py

This program executes the iteration loop for TEA. It repeats Lagrangian minimization (`lagrange.py`) and lambda correction (`lambdacorr.py`) until the maximum iteration is reached. The code has time stamps for checking the speed of execution and is verbose for debugging purposes. Both are controlled in `TEA_config.py` file.

The flow of the code goes as follows: the current header, output, and result directory are read; physical properties are retrieved from the header, and the `balance.py` output is read as the initial iteration input and passed to `lagrange.py`. Lagrange x_i values are then checked for negative values: the next iteration starts either with lambda correction output (if negative x_i's are found) or with the output produced by `lagrange.py` (if all x_i's are positive). This procedure is repeated until the maximum iteration is reached, which stops the loop. Intermediate results from

each iteration step are written in the machine- and human-readable output files on the user's request in `TEA_config.py`.

The program is executed by `runatm.py` and can be executed alone with in-shell input:
`iterate.py <header file> <name of the result directory>`

2.4 File Control Programs

These programs generate inputs for the `Main Scientific Programs`, manage reading and processing the input information, and produce machine-readable and human readable output files.

2.4.1 format.py

This module allows each program to read the output of the previous step so the data can be used in the next step. It also manages the format for each output file and produces both machine-readable and human-readable files.

It contains the following functions:

1. **readheader()**: This function reads the current header file (one T, P) and returns data common to each step of TEA. It searches only for the required chemical data from the header file and fills out the output arrays appropriately. The function is used by `balance.py, lagrange.py, lambdacorr.py`, and `iterate.py`.
2. **readoutput()**: This function reads output files made by the `balance.py, lagrange.py` and `lambdacorr.py`. It reads any iteration's output and returns the data in an array.
3. **output()**: This function produces machine-readable output files. The files are saved only if `saveout = True` in `TEA_config.py` file. The function is used by the `balance.py, lagrange.py`, and `lambdacorr.py`. The function writes the name of the header, current iteration number, species list, starting mole numbers of species for current iteration, final mole numbers of molecular species after the iteration is done, difference between starting and final mole numbers, total sum of initial mole numbers, total sum of final mole numbers and the change in total mole numbers of all species.
4. **fancyout()**: This function produces human readable output files. The files are saved only if `saveout = True` in `TEA_config.py` file. The function is used by the `balance.py, lagrange.py`, and `lambdacorr.py`. The function writes the name of the header, current iteration number, species list, starting mole numbers of species for current iteration, final mole numbers of molecular species after the iteration is done, difference between starting and final mole numbers, total sum of initial mole numbers, total sum of final mole numbers and the change in total mole numbers of all species. If `doprint = True`, all data written to the file is presented on-screen.
5. **fancyout_results()**: This function produces the final result output for each T, P in the human-readable format. The final mole number for each species is divided by the total mole numbers of all species in the mixture. This gives our final results, which is the mole fraction abundance for each species. This function is called by the `iterate.py` module.
6. **printout()**: Prints iteration progress number or other information in one line of terminal.

2.4.2 makeheader.py

This module contains functions to write headers containing all necessary chemical data for a single T, P and multiple T, P TEA runs. It consists of two main functions, `make_singleheader()` and `make_atmheader()` called by the `runsingle.py` and `runatm.py` modules, respectively. The `header_setup()`, `atm_headarr()`, `single_headarr()`, and `write_header()` are supporting functions for the main functions. This module is imported by `runatm.py` and `runsingle.py` to create the header files.

header_setup() - This function is a common setup for both single T, P and multiple T, P TEA runs. Given the thermochemical data and stoichiometric table, this function returns stoichiometric values and an array of chemical potentials for the species of interest at the current temperature and pressure. Using the data provided in the JANAF tables, and the equations (10) and (11) from Blecic et al. (2015), it calculates the free energies (chemical potentials) of each species at a certain temperature. It also returns an array of booleans that marks which information should be read from `stoich_file` for the current species. It is executed by the `make_atmheader()` and `make_singleheader()` functions.

single_headarr() - This function gathers data needed for TEA to run in a single T, P case. These are: elemental abundances, species names, and their stoichiometric values. For the list of elements and species used, it takes the abundances and stoichiometric values and puts them in the final array. It converts logarithmic (dex) elemental abundances into number densities. This function is run by `make_singleheader()` and is dependent on results from `header_setup()`.

atm_headarr() - This function gathers data needed for TEA to run in a multiple T, P case. These are: elemental abundances, species names, and their stoichiometric values. For the list of elements and species used, it takes the abundances and stoichiometric values and puts them in the final array. This function is run by `make_atmheader()` and is dependent on results from `header_setup()`.

write_header() - This function writes a header file that contains all necessary data for TEA to run. It is run by `make_atmheader()` and `make_singleheader()`.

make_singleheader() - This is the main function that creates single-run TEA header. It reads the input T, P file and retrieves necessary data. It then calls the `header_setup()`, `single_headarr()`, and `write_header()` functions to create a header for the single T, P point. This function is called by the `runsingle.py` module.

make_atmheader() - This is the main function that creates a TEA header for one T, P of a pre-atm file. It retrieves number of elements and species used for only the q-th T, P point in the pre-atm file. It then calls the `header_setup()`, `atm_headarr()`, and `write_header()` functions to create a header for this point. This function is called by the `runatm.py` module.

2.4.3 readatm.py

This function reads a pre-atm file and returns data that TEA will use. It opens a pre-atmosphere file to find markers for species and TEA data, retrieves the species list, reads data below the markers, and fills out data into corresponding arrays. It also returns number of runs TEA must execute for each T, P. The function is used by `runatm.py`.

2.4.4 readconf.py

This code reads the TEA config file, `TEA.cfg`. There are two sections in `TEA.cfg`: the TEA section and the PRE-ATM section. The TEA section carries parameters and booleans to run and debug TEA. The PRE-ATM section carries paramaters to make pre-atmospheric file (see Section 2.5.1).

2.5 Auxiliary Programs

These programs are provided for the user convenience. They allow easy generation of the multiple T, P pre-atmospheric files (input for the `runatm.py` module), and plotting of the desired abundances profiles from the final multiple T, P atmospheric files (output of the `runatm.py` module).

2.5.1 makeatm.py

This module produces a pre-atmospheric file in the format that TEA can read it. Before running `makeatm.py`, the `TEA.cfg` file needs to be edited with the following information: location to the PT file, the desired name of the pre-atmospheric file, desired input elemental species, and desired output molecular species.

To run the code type in the terminal: `makeatm.py` <DIRECTORY_NAME>.
The <DIRECTORY_NAME> is the name of the folder where the user wants the current run to be placed. The pre-atmopsheric file will be placed in the atm_inputs directory under the <DIRECTORY_NAME> folder.

This module consists of 2 functions:

1. **readPT() -** reads pressure-temperature ($T - P$) profile from the PT file provided. If custom made must be in the format provided in doc/examples/ folder.
2. **makeatm()** - produces a pre-atmospheric file in the format that TEA can read it. The file will be placed in `atm_inputs/` directory. It calls `readPT()` function to take pressure and temperature array and reads the elemental abundance data file (default: `abundances.txt`, Asplund et al. 2009**FINDME**). The code trims the abundance data to the elements of interest, converts species dex abundances (logarithmic abundances, dex stands for decimal exponent) into number densities and divides them by the hydrogen number densities fractional abundances. It writes data (pressure, temperature, elemental abundances) into a pre-atmospheric file. The config file, pressure and temperature file, and the abundances file are copied to the `atm_inputs/` directory.

2.5.2 plotTEA.py

This code plots a figure of temperature vs. abundances for the final atmospheric file produced by TEA. It needs 2 arguments on the command line: the path to the atmospheric file name and the names of the species user wants to plot.

Arguments given should be in the following format:

1. **plotTEA.py** - calls the plotTEA module.
2. **filename** - string. Full path to the atmospheric file.

3. **species -** list of strings. List of species that user wants to plot. Species names should be given with their symbols (without their states) and no breaks between species names (e.g., CH4,CO,H2O).

To run the code do: `plotTEA.py <RESULT_ATM_FILE> <SPECIES_NAMES>`
Example: `plotTEA.py results/atm_Example/atm_Example.atm CO,CH4,H2O,NH3`
The plot is opened once execution is completed and saved in the `plots/` subdirectory.

The lower range on the y axis (mixing fraction) when the temperatures are below 600 K should be set at most $10e^{-14}$.

References

Blecic, J., Harrington, J., & Bowman, M. O. 2015, in prep for ApJSupp