university of groningen / faculty of science and engineering / kapteyn astronomical institute

University of Groningen

Bachelor's Thesis

ReverseRADEX: a tool to quickly gauge global physical conditions of a gas cloud.

Supervisor:
Prof. Dr. Floris F.S. van der Tak (Kapteyn Astronomical Institute, University of Groningen)

2nd Examiner:
Dr. Kateryna Frantseva (Kapteyn Astronomical Institute, University of Groningen)

Author:
Filip van der Mooren (S3804011)

Abstract: Molecular emission line observations of interstellar gas clouds harbor information about the global physical components like kinetic temperature and densities. Current software to extract this information in a quick and practical manner is scarce and due for improvement. RADEX is used as the underlying radiative transfer code based on escape probability methodology, requiring molecular collision data following the format of the Leiden Atomic and Molecular Database (LAMDA). RADEX input consist of physical components and outputs line spectra and required inversion. Accordingly, a set of python wrappers for Fortran written RADEX was compared and SpectralRadex chosen. ReverseRADEX takes molecular spectral lines as input and optimizes for physical components using an algorithm chain consisting of three parts; a Brute-force method to estimate initial physical parameters, subsequently the Levenberg-Marquardt (LM) algorithm refines the parameter estimates, culminated by an Monte Carlo Markov Chain (MCMC) algorithm in order to determine parameter uncertainties. ReverseRADEX is available online and the design of ReverseRADEX is described here, with exemplifying tests showcasing the software's capabilities and limitations as a practical program to obtain physical parameter and uncertainty estimates quickly.

July 2, 2021

Contents

1 Introduction

Most astronomical observations are made with radiation, covering a large range of the electromagnetic spectrum. Specific parts of the spectrum are useful to study various objects, due to spectral line transitions of molecular[1] species appearing as emission or absorption on top of the continuum. The mid- and far-infrared (3 μm; 300 μm), (sub)millimeter (0.3 mm; 3 mm), and radio (0.3 cm; 30 cm) parts of the spectrum house diagnostic molecular spectral lines to study interstellar gas, both with Earth- and space-based telescopes.
Currently the most active fields within astrochemistry are protoplanetary disks, nurturing stellar and planetary formation within, among other interests like cataloging chemical constituents of diffuse and translucent clouds. For an overview of astrochemical study interests and their challenges e.g. see van Dishoeck 2017. Astrochemistry concerns itself with how molecules form and their influence on the surrounding medium, often determined from physical conditions revealed only through molecular line observations. Current state of the art observing instruments like the Atacama Large Millimeter Array provide such molecular line observations, and will be joined by upcoming instruments like the Large Latin American Millimeter Array, providing new research possibilities regarding astrochemistry. See e.g Mendoza et al. 2021 discussing; early universe chemistry, star forming regions, and asymptotic giant branch stars and circumstellar envelopes.

Observations however do not directly contain information about the physical conditions of the source and radiative transfer codes are required to interpret observations. One of such codes is RADEX (van der Tak et al. 2007), which can be used to compare molecular observations with a modeled spectrum. RADEX is an "intermediate-level" radiative transfer code, requiring the "basic-level" assumptions but in addition solves for the statistical equilibrium (SE) of (de-)excitation rates from and to a given state. The "basic-level" (e.g. "population diagram method" - Goldsmith et al. 1999) requires observations of molecular line strengths[2], which are then fit as a function of upper level energy. The excitation is described by a single temperature T_{ex} [K], often under the assumption of local thermodynamic equilibrium (LTE), in which case $T_{ex} = T_{kin}$, the kinetic temperature (van der Tak 2011). And in the case of similar beam sizes and low optical depths, or appropriate corrections are applied, physical conditions of the interstellar cloud can be determined.
Not assuming LTE, the "intermediate" level requires knowledge about molecular collision rates, somewhat limiting the usefulness of the method, for many species do not have (accurate) collisional rates available. RADEX utilizes molecular data, following the data file format of the Leiden Atomic and Molecular Database (LAMDA) (Schöier et al. 2005), also including quality labels for the molecular collisional data, presented in a recent update also mentioning prospects of the database (van der Tak et al. 2020). The SE is solved in RADEX using the escape probability approximation (Rybicki 1985), which assumes no information about internal structure variation, treating the gas cloud as a global entity.
The "sophisticated-level" forgoes the approximation of local excitation and solves for line strengths using both the depth into the cloud, as well as the velocity. An example of such a method is the Accelerated Lambda Iteration (ALI) method, incorporated e.g. into the LIME radiative transfer method (Brinch et al. 2010), where the local contribution to the radiation field is decoupled from the total radiation field by representing the cloud as a grid of points. Similarly, Monte Carlo based methods solve for radiative transfer by representing the cloud as an ensemble of cells, sending pho-

[1] In this thesis, the term "molecule(s)" refers to molecular, mono-atomic and ionic species.
[2] In this thesis, line strengths refers to both intensity T_R [K], and fluxes $\mathcal{F}$ [K km s^{-1}] and $\mathcal{F}$ [erg cm^{-2} s^{-1}].

ton packages in random directions from each cell (Hogerheijde 2000). For an overview of these "model-levels" see e.g. (van der Tak 2011) as well as e.g. (van Zadelhoff et al. 2002) for numerical performance and convergence characteristics for the non-LTE methods.

RADEX however takes the physical conditions sought after as input and outputs line strengths which can then be compared to observations. The inversion of this process is what ReverseRADEX offers by making multiple calls to RADEX with varying input parameters until a convergent solution is found, where model and observations agree best. A few programs[3] have already attempted this to various levels of accessibility. The need for such a program thus exists already and ReverseRADEX aims to serve as a quick and reliable tool to determine global physical conditions of observed interstellar clouds. Specifically, the following parameters can be determined if enough lines (parameters + 1) are observed; the kinetic temperature T_{kin} [K], the molecular column density N_{mol} [cm^{-2}] and volume densities of collision partners n_{col} [cm^{-3}].
The source code of RADEX is written in Fortran77 with bits of Fortran90 and hence runs fast. Over the years however, Fortran has not kept up as the primary programming language in the field of astronomy at 28 ± 2 % compared to python's 67 ± 2 % (Momcheva et al. 2015). Additionally, python is a prevalent language to teach freshman students. To abide by this raise in popularity of python, the Fortran source code is "wrapped" to allow for easy interfacing with python, as if the source code is a part of the python language, without losing the speed of Fortran. No new wrapper is developed specifically for this thesis, for various wrappers already exist and will be discussed in Section 5.
ReverseRADEX utilizes the SpectralRadex (Holdship et al. 2020) wrapper to make multiple calls on RADEX models which are compared to observational data, and determine the parameter and uncertainty estimates for the accompanying physical conditions of the interstellar gas that minimize the χ^2 statistic. In view of open science, the program is freely available at `gitlab.astro.rug.nl/mooren/ReverseRADEX` under the MIT license.

The thesis is structured as follows. Section 2 will briefly discuss the theory behind the formalism of RADEX to analyze observations. Section 3 discusses our attempt of incorporating MAGIX (Möller et al. 2013) into ReverseRADEX. Section 4 will provide a summary of RADEX for parts of the interest for ReverseRADEX. Section 5 examines the different python wrappers of RADEX and why SpectralRadex is chosen for use in ReverseRADEX. Section 6 covers the inversion process for RADEX and showcases results. Section 7 offers a discussion and prospects for various aspects of the thesis. Finally concluding the thesis in Section 8 with a summary of the thesis and prospects for ReverseRADEX.

[3] Listed here in no particular order: pyradexnest, radexcee, pyradex_mcmc specifically for (Kamenetzky et al. 2018).

2 Theory

This section is split in two subsections; firstly discussing the radiative transfer formalism applied to (Reverse)RADEX, and thereafter the algorithms used in Reverse(RADEX) for convergence are detailed.

2.1 Radiative transfer

The following section has been largely derived using (Draine 2011; Rybicki et al. 2004), unless specifically referenced otherwise. This section briefly discusses the formalism derived form the theory of radiative transfer adopted into (Reverse)RADEX. For a more in-depth look into the formalism used in (Reverse)RADEX to analyze molecular line observations see van der Tak et al. 2007.

2.1.1 Equation of radiative transfer

Radiative transfer is described by the emission, absorption and scattering of photons along a straight path from source to observer. Therefore, the specific intensity, I_ν [erg cm^{-2} s^{-1} Hz^{-1} sr^{-1}], is used for it is a quantity that is conserved along its path in the absence of any local absorption or emission, in addition to having a well defined direction of travel. Combining the specific intensity with the absorption α_ν [cm^{-1}] and emission j_ν [erg cm^{-3} s^{-1} Hz^{-1} sr^{-1}] coefficients, thus provide us with all components required to describe radiative transfer in the following differential form,

$$\frac{dI_\nu}{ds} = -\alpha_\nu I_\nu + j_\nu \quad [\text{erg cm}^{-3}\ \text{s}^{-1}\ \text{Hz}^{-1}\ \text{sr}^{-1}]. \tag{1}$$

This equation shows how the specific intensity varies as a function of absorption α_ν and emission j_ν and follows the conservation criteria when absorption and emission are absent or in equilibrium. Subsequently dividing through the absorption coefficient, the source function, $S_\nu = j_\nu/\alpha_\nu$ [erg cm^{-2} s^{-1} Hz^{-1} sr^{-1}] (describing the emissivity per unit optical depth), and optical depth in differential form along an infinitesimally thin path, $d\tau_\nu = \alpha_\nu ds$, can be defined and used to express the equation of radiative transfer in integral form,

$$I_\nu = I_\nu(s=0)e^{-\tau_\nu} + \int_0^{\tau_\nu} S_\nu(\tau'_\nu)e^{-(\tau_\nu-\tau'_\nu)}d\tau'_\nu \quad [\text{erg cm}^{-2}\ \text{s}^{-1}\ \text{Hz}^{-1}\ \text{sr}^{-1}] \tag{2}$$

where I_ν now describes the incoming radiation along the line of sight and $I_\nu(0)$ describes the "background" radiation entering the medium. Equation (2) is valid for both continuum radiation, covering a large bandwidth, and spectral lines, referring to drastic local absorption and emission alterations, covering a tiny frequency interval. The integral form of the radiative transfer equation is also usually the form used in radiative transfer codes (van Zadelhoff et al. 2002).

The specific intensity can then be used to derive the integrated mean intensity over the line profile,

$$\bar{J}_\nu = \int_0^\infty \left(\frac{1}{4\pi}\int I_\nu d\Omega\right)\phi_\nu(v)dv = \int_0^\infty J_\nu\phi_\nu(v)dv \quad [\text{erg cm}^{-3}\ \text{Hz}^{-1}] \tag{3}$$

where J_ν is the integrated mean intensity over the solid angle, Ω [sr], and v is the velocity [cm s^{-1}].

2.1.2 Gas emission

The spectral lines observed as spikes in the continuum spectrum are caused by molecules, either through emission or absorption. Local spontaneous emission for a molecule in upper level u with number density n_u occurs at a rate of the accompanying Einstein A_{ul} coefficient [s^{-1}], where u and l stand for upper and lower respectively, and can be linked to the emission coefficient,

$$j_\nu = \frac{h\nu_{ul}}{4\pi}\mathrm{n}_u A_{ul}\phi_\nu \quad [\mathrm{erg\ cm^{-3}\ s^{-1}\ Hz^{-1}\ sr^{-1}}] \tag{4}$$

where h [$\mathrm{erg\ Hz^{-1}}$] is Planck's constant, ν_{ul} [Hz] the central line frequency and ϕ_ν [$\mathrm{Hz^{-1}}$] the spontaneous emission line profile function.

The absorption coefficient is described using the Einstein B_{ul} and B_{lu} coefficients [$\mathrm{erg^{-1}\ cm^3\ s^{-1}}$ Hz], referring to induced emission and photon absorption respectively,

$$\alpha_\nu = \frac{h\nu_{ul}}{4\pi}\left(\mathrm{n}_l B_{lu}\varphi_\nu - \mathrm{n}_u B_{ul}\chi_\nu\right) \quad [\mathrm{cm^{-1}}] \tag{5}$$

where φ_ν and χ_ν are the line profile functions for absorption, and induced emission respectively (van der Tak et al. 2007). Notice how the induced emission has a negative contribution to the absorption coefficient and the line profiles, $\phi = \varphi = \chi$, only when collisional excitation dominates.

The collisional excitation can be described with the collision rate coefficient γ [$\mathrm{cm^{-3}\ s^{-1}}$] expressed using the Maxwellian average of the collisional cross section, σ [$\mathrm{cm^{-2}}$],

$$\gamma_{ul} = \left(\frac{8k_\mathrm{B}T_{kin}}{\pi\mu}\right)^{-\frac{1}{2}}\left(\frac{1}{kT_{kin}}\right)^2\int \sigma E \mathrm{e}^{-E/kT_{kin}}dE \quad [\mathrm{cm^3\ s^{-1}}] \tag{6}$$

where k_B [$\mathrm{erg\ K^{-1}}$] is the Boltzmann constant, E [erg] the collisional energy, T_{kin} [K] the kinetic temperature, μ [g] the reduced mass. Equation (6) is the downward collisional rate coefficient and its upward counterpart can be obtained from detailed balance,

$$\gamma_{lu} = \gamma_{ul}\frac{g_u}{g_l}\exp\left(\frac{-h\nu}{kT_{kin}}\right) \quad [\mathrm{cm^3\ s^{-1}}] \tag{7}$$

where g_i is the statistical weight of level i. These rate coefficients can be combined with the number density of the collision partner, $\mathrm{n_{col}}$, to find the collision rate (van der Tak et al. 2007),

$$C_{ul} = \mathrm{n_{col}}\gamma_{ul} \quad [\mathrm{s^{-1}}]. \tag{8}$$

2.1.3 Escape probability

The equation of statistical equilibrium can be written in the following compact form[4],

$$\frac{d\mathrm{n}_i}{dt} = 0 = \sum_{i\neq j}^{N}\mathrm{n}_j P_{ji} - \mathrm{n}_i\sum_{i\neq j}^{N}P_{ij} \quad [\mathrm{cm^{-3}\ s^{-1}}] \tag{9}$$

[4] (de Jong et al. 1980; Rybicki 1985; van der Tak et al. 2007)

where the destruction, P_{ij} [s^{-1}], and formation, P_{ji} [s^{-1}], coefficients of level i are,

$$P_{ij} = \begin{cases} A_{ij} + B_{ij}\bar{J}_\nu + C_{ij} & \text{for} \quad i > j \\ B_{ij}\bar{J}_\nu + C_{ij} & \text{for} \quad i < j \end{cases} \quad [s^{-1}] \tag{10}$$

and where, $B_{ij}\bar{J}_\nu$ is the transition absorption rate.

Equation (9) thus depends on both the level populations, n_i and n_j, and the local radiation field, equation (3), which in turn are interdependent on one another, posing a problem when trying to solve equation (9). The escape probability method mitigates this problem by considering only the global properties of the gas cloud, decoupling the interdependence by defining the radiation field in terms of the source function S_ν and optical depth dependent, geometrically averaged escape probability, $\beta(\tau_\nu)$, that a photon escapes the medium,

$$\bar{J}_\nu \approx S_\nu[1 - \beta(\tau_\nu)] \tag{11}$$

where the background radiation and any local continuum are ignored, and the relation between $\beta(\tau_\nu)$ and τ_ν is dependent on the adopted geometry, see Section 4 for the geometries provided by RADEX.

Equation (11) is a fundamental first-order relation of the simplest escape probability method (Rybicki 1985). When the gas cloud is completely opaque, the escape probability will equal zero and the radiation field equals the source function.

2.2 Algorithms

To estimate the physical conditions of the observed celestial object, three algorithms are utilized to constrain the parameters referring to the physical conditions. The algorithms are linked together in an algorithm chain, see Figure 1, with the first "algorithm" being the Brute-force method used to find the global minimum, secondly a non-linear least squares algorithm to refine the parameter estimates, Levenberg-Marquardt, and lastly an MCMC algorithm for uncertainty estimates. The benefit to chaining algorithms is that it reduces biases and the computation time for the computationally expensive MCMC algorithm can be reduced, as it will only have to obtain uncertainty estimates and not search the whole parameter space for the parameter estimates.

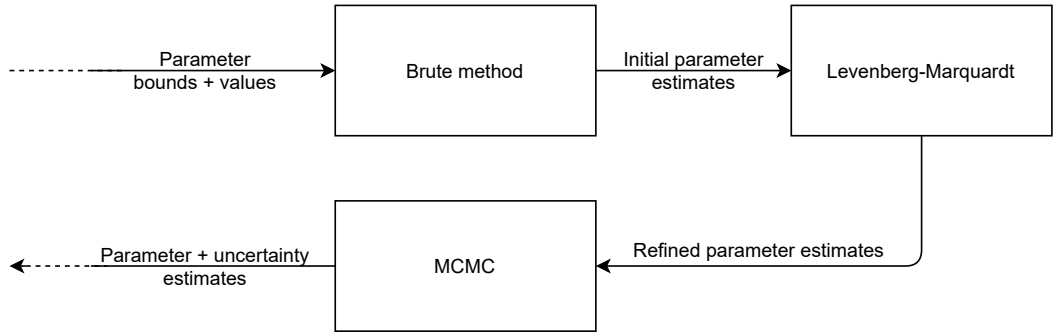


Figure 1: A flow diagram showcasing the algorithm chain used to converge the parameter estimates.

The parameter ranges of RADEX, Section 4.2, can span many orders of magnitude, making it difficult to define stopping criteria that are meaningful for all parameters. To circumvent this, a simple $\log_{10}$ scaling is applied to all parameters to be fit, reducing the maximum difference between parameters in most circumstances to approximately one order of magnitude.
There are also approximate timings taken for each respective algorithm with the following test parameter ranges; $10 < T_{kin}$ [K] < 1000, $10^{10} < N_{CO}$ [cm^{-2}] $< 10^{20}$, $100 < n_{H_2}$ [cm^{-3}] $< 10^8$. The timings were taken on a 4 core scientific Linux virtual machine running at 4.8 GHz and the combined run time for all algorithms (essentially the entire program) is $\lesssim 4.5$ minutes.

2.2.1 The Brute-force method

The Brute-force method used in ReverseRADEX is a simple grid search of the entire (user) specified parameter space. The complexity is thus $O(a^n)$ where min(number of evaluations per parameter) = $n > 1$ and, number of parameters to be fit = $a > 1$. The grid adaptively decides how many points per parameter to evaluate, based on the supplied user bounds and a predetermined difference between subsequent evaluations, as well as a minimum and maximum to ensure reasonable sampling and reduce computation time respectively.
Even though this complexity is unfavorable, all the grid search has to do is find reasonable initial parameter estimates and leave the rest to the next algorithm in the chain. This can be achieved only if enough observations are available to constrain the parameter degeneracy, see Section 7.1, in which case the grid does not have to be sampled very finely, for the parameter space has one clear minimized solution and is thus mono-modal.
Using the example detailed in the last paragraph of Section 2.2, the Brute-force method takes < 1

minute to sample the global minimum. This is partially due to the Brute-force method grid search being complemented by SpectralRadex's feature set, Section 5.1, utilizing both the built in grid calculation and multiprocessing capabilities.

2.2.2 Levenberg–Marquardt

After global parameter estimates have been determined by the Brute-force method, the derivative based LM algorithm (Moré 1978) refines the parameter estimates. The Brute-force method likely did not reach the optimal parameter combination but got close enough for the LM algorithm to further optimize the parameter estimates.
The LM algorithm is a non linear derivative least squares solver and implemented in ReverseRADEX through the SciPy python library[5] (Virtanen et al. 2020). LM is a robust and efficient algorithm that optimizes for parameters by estimating the Hessian matrix using the summed outer products of the gradients (Roweis 1996), but does not support bounds, which in most cases should not be a problem unless the initial parameter guesses are close to the (user) supplied or RADEX bounds, possibly causing the subsequent MCMC algorithm to fail or SpectralRadex to fatally fail respectively.
The LM algorithm had to trade in speed to attain its robustness but compared to the other algorithms used, speed was never a concern for the LM algorithm. Using the test case defined in the last paragraph of Section 2.2, the obtained run time was consistently found to be < 5 seconds and in extreme cases < 10 seconds.
One downside to using LM is that you need $n + 1$ data points to fit n parameters and in the case of the C atom for example, there are only a maximum of three spectral lines to be observed, of which all are forbidden and the $^3\mathrm{P}_2$—$^3\mathrm{P}_0$ has $\Delta J = 2$, making it much weaker and thus significantly harder to observe than the already forbidden $^3\mathrm{P}_1$—$^3\mathrm{P}_0$ and $^3\mathrm{P}_2$—$^3\mathrm{P}_1$ transitions. This means that you might only have two observed data points and can thus only constrain one parameter for these kinds of molecules.

2.2.3 MCMC

Now that the parameter estimates of the physical conditions have been established by the two prior algorithms in the chain, the MCMC algorithm is utilized to obtain parameter uncertainties. The MCMC algorithms works on the basis of minimizing the posterior distribution build up of the log likelihood and log prior. Given the observed line strengths y_i with uncertainties σ_i, measured at frequencies x_i, the log likelihood becomes,

$$\ln\left[\mathcal{L}(y_i|x_i,\sigma_i,\mathrm{p}_f,\mathrm{p}_c)\right] = -0.5\sum_{i=1}^{N}\left[\left(\frac{y_i - \mathrm{model}(x_i,\mathrm{p}_f,\mathrm{p}_c)}{\sigma_i}\right)^2 + 2\ln(\sigma_i) + \ln(2\pi)\right] \tag{12}$$

where p_f are the free parameters, p_c the constant parameters and model refers to a RADEX model.

A likelihood can however not be used to sample the free parameters outright, for the likelihood is a probability distribution governing the data y_i, conditioned by the parameters p_f. To draw parameter samples, a prior is needed to marginalize over the nuisance parameters (in this case all

[5] See for further information: docs.scipy.org/doc/scipy.optimize.least_squares.

but p_f) (Foreman-Mackey et al. 2013; Sivia et al. 2006). A uniform uninformative prior is used to marginalize over said nuisance parameters,

$$\ln\left[prior(\mathrm{p}_f)\right] = \mathrm{p}_l \leq \mathrm{p}_f \leq \mathrm{p}_u \tag{13}$$

where p_l and p_u are the (user) supplied lower and upper bounds for the free parameters respectively.

Combing equations (12) and (13), the posterior distribution is obtained,

$$\ln\left[p(\mathrm{p}_f|y_i,\sigma_i,\mathrm{p}_c)\right] \propto \ln\left[\mathcal{L}(y_i|x_i,\sigma_i,\mathrm{p}_f,\mathrm{p}_c)\right] \times \ln\left[prior(\mathrm{p}_f)\right]. \tag{14}$$

Equation (14) is used to optimize the model parameters p_f by checking on each iteration if the combination of parameters minimized the posterior more or less then the previous iteration and based on that accept or reject the new parameters respectively.

MCMC algorithms vary in how the updated parameters are calculated. The MCMC algorithm incorporated into ReverseRADEX is that of the emcee python package (Foreman-Mackey et al. 2013), and allows for various update algorithm schemes, referred to as *moves*. ReverseRADEX uses the following three moves to update the parameter coordinates; *StrechMove* (Foreman-Mackey et al. 2013; Goodman et al. 2010), *DEMove* (Nelson et al. 2013) and *DESnookerMove* (ter Braak et al. 2008) with 70%, 20% and 10% probability respectively. This entails that 70% of the MCMC walkers in parameter space are updated to the next set of parameters using the StretchMove, and 20% using DEMove and 10% using DESnookerMove, if the next set of parameter coordinates are accepted that is.
The run time was $\lesssim 3$ min for the test case defined in the last paragraph of Section 2.2.

3 MAGIX

Early on in this project, the solver of choice was the iterating algorithm engine MAGIX (Möller et al. 2013). Reason being, why develop an iterative solver specifically for this thesis when capable alternatives already exist.
The major benefit of MAGIX is that it serves as an interface to easily utilize a variety of different minimization algorithms, as well as smart support for chaining those algorithms. The smart aspect refers to the case when the first algorithm in the chain has a "BestSiteCounter" option, indicating multiple sites in parameter space that could be global minima, then the next algorithm in the chain will run for all those *sites*, instead of just one, and choose the best site to continue. The algorithm settings themselves (e.g. number of iterations, χ^2 limit, and a few algorithm specific options) are limited however. In other words MAGIX is a high level interface for iterative algorithms. A similar algorithm chain as currently implemented in ReverseRADEX, Section 2.2, could be achieved with MAGIX as well, with the difference being the specific settings of the algorithms.
Additionally, as MAGIX is an external program, it handles its own parallelization and provides algorithms that can benefit from said parallelization, requiring only that the model to iteratively solve for is parallelizable. Unfortunately MAGIX was developed on Linux, also supporting MACs but has not been tested on Windows and therefore is likely not completely supported on Windows.
I have seriously attempted to incorporate MAGIX into ReverseRADEX but the following two sections will clarify why it has been omitted.

3.1 Incorporating MAGIX

Since MAGIX is an external program, it requires a way to funnel the input from python and does this for all settings via xml files. This was translated in how the input of ReverseRADEX was tailored for MAGIX in an attempt to keep the code readable. Some of the code was also present in other parts of the code base and still remain after the removal of MAGIX from ReverseRADEX. See Section 7.3.1 for a more detailed discussion on the latent MAGIX code.
Although high level, MAGIX was not easily understood and the incomplete documentation (Möller et al. 2020) provided some headaches. However, the many examples for each algorithm exemplified enough of the undocumented settings to get MAGIX operational.
The aim of MAGIX is to be applicable in many situations but in pursuit of this generality suffers from performance loss. Each iteration of a RADEX model required a random folder to be generated, the model and input files to be copied there and the output to be copied from there into MAGIX for the solving, introducing a lot of overhead. This would not be much of an issue if the model itself would be the bottleneck but the run time of (Spectral)RADEX, see Table 2, is < 10 ms. In addition to the performance loss, the pursuit of generality likely also introduced the overwhelming amount of log and data bloat files[6].
In the hopes of incorporating MAGIX as a "quick and easy solution" to interface RADEX with iterative minimization algorithms, it ended up taking significantly more time than the python native alternatives currently implemented, see Section 2.2 and Appendix C.3 for the code.

[6] Depending primarily on the amount of algorithms, it could reach ±25 unnecessary files in ReverseRADEX's case for each time the program runs.

3.2 Results

Results obtained with MAGIX gradually improved over the weeks of development but were ultimately deemed unsatisfactory. Table 1 shows how the parameter and uncertainty estimates change for varying numbers of constraints (observed lines),

Table 1: A comparison using MAGIX of the parameter (+ uncertainty) estimates for T_{kin}, N_{CO}, and n_{H_2}, with all other parameters fixed, and using 4/40, 20/40, and 40/40 lines.

Run	Available data	T_{kin} [10^2 K]	N_{CO} [10^{15} cm^{-2}]	n_{H_2} [10^5 cm^{-3}]
	Input parameters	1.20	1	1
1	4/40 lines	1.12 ± 0.02	$0.278 \cdots \pm 2 \times 10^{-15}$	2.51189 ± 0.00002
2	20/40 lines	1.1 ± 0.5	$0.278 \cdots \pm 2 \times 10^{-10}$	2.51189 ± 0.00006
3	40/40 lines	1.1 ± 0.5	$0.278 \quad \pm 0.007$	2.51188 ± 0.00008

Table 1 shows a minimum of 4/40 data points for three parameters because the LM algorithm, Section 2.2.2, requires *data points* $\geq$ *parameters* + 1 in order to work. The results are purposefully not shown in $\log_{10}$ scale to showcase the absurdity of the uncertainties. Not only in numeric value, but also in data point dependence, where less data results in a irrationally more constrained parameter estimate with tiny uncertainties, especially in the case of N_{CO}. The uncertainties of n_{H_2}'s remain of the same order of magnitude but are tiny for all three runs. The most realistic numerical uncertainties are those of T_{kin} but still follow the wrong uncertainty–data-point relation. Generally it is expected that the uncertainties become larger when less data are available but the opposite trend is observed.

Furthermore, the parameter estimates themselves are far from the input parameters and most certainly do not fall within the tiny uncertainty estimates. Interestingly, the parameter estimates appear independent of the amount of data available, producing almost equal results for all three runs. For further discussion of MAGIX results see Section 7.3.1.

All things considered, I decided against implementing MAGIX into ReverseRADEX for it is an external dependency that is too generalized, slow, inconsistent and produces a lot of bloat as well as offers no support for the Windows platform.

4 RADEX

The model program used to compare with observational data in the iterative algorithm chain is RADEX. For a more detailed look into how and within what limits RADEX works and what it has to offer, see van der Tak et al. 2007. RADEX uses the "intermediate" escape probability method to iteratively solve equation (9) until a consistent solution exist for both the level populations and the radiation field. The solution is than used to derive other quantities explained later on. In (Reverse)RADEX, three escape probabilities, for different use cases, are included;

- The large velocity gradient (LVG) escape probability (de Jong et al. 1980; Mihalas 1978), primarily used for an expanding spherical shell,

$$\beta_{\rm LVG} = \frac{1}{\tau_\nu}\int_0^{\tau_\nu} {\rm e}^{-\tau'_\nu} d\tau'_\nu = \frac{1-{\rm e}^{-\tau_\nu}}{\tau_\nu}. \tag{15}$$

- For a spherically symmetric and homogeneous static gas cloud the escape probability (Osterbrock et al. 2006, Appendix 2) is,

$$\beta_{\rm sphere} = \frac{3}{2\tau_\nu}\left[1-\frac{2}{\tau_\nu^2}+\left(\frac{2}{\tau_\nu}+\frac{2}{\tau_\nu^2}\right){\rm e}^{-\tau_\nu}\right]. \tag{16}$$

- For shocks, a plane-parallel "slab" geometry escape probability (de Jong et al. 1975) is included,

$$\beta_{\rm slab} = \frac{1-{\rm e}^{-3\tau_\nu}}{3\tau_\nu}. \tag{17}$$

4.1 Capabilities useful for ReverseRADEX

RADEX requires the following set of inputs to run, see Table 3a; the molecular file, output file name, frequency range, kinetic temperature $T_{\rm kin}$ [K], the collision partners [name], density of collision partner $n_{\rm col}$ [cm^{-3}], background radiation temperature $T_{\rm bg}$ [K], molecular column density $N_{\rm mol}$ [cm^{-2}] and line width [km s^{-1}].
RADEX will then make an initial guess by solving equation (9) for the level populations of the molecular energy levels in the optically thin case, taking into account only the un-shielded background radiation field. From these initial level populations, the optical depth τ is calculated, in turn allowing for re-calculation of the the molecular excitation and incorporating the internal radiation field when solving equation (9). This step iteratively continues until a coherent solution exists between the level populations and radiation field.
When calculations finish, the following output gets returned, see Table 3b; re-iterates input, upper state energy [$E_{\rm up}$], line frequency [GHz], wavelength λ [μm], excitation temperature $T_{\rm ex}$ [K], optical depth τ, peak radiation temperature $T_{\rm R}$ [K], upper population density n_u [cm^{-3}], lower population density n_l [cm^{-3}], line flux $\mathcal{F}$ [K km s^{-1}] and line flux $\mathcal{F}$ [erg cm^{-2} s^{-1}]

The important inputs and outputs for ReverseRADEX are; $T_{\rm kin}$, $N_{\rm mol}$, $n_{\rm col}$ and $T_{\rm R}$, $\mathcal{F}$ [K km s^{-1}], $\mathcal{F}$ [erg cm^{-2} s^{-1}] respectively. The output is given as the background subtracted line intensity of the Rayleigh-Jeans limit equivalent peak radiation temperature,

$$T_{\rm R} = \frac{c^2}{2k\nu^2}\left(I_\nu^{\rm em} - I_\nu^{\rm bg}\right) \quad [{\rm K}] \tag{18}$$

where c [cm s^{-1}] is the speed of light, and ${I_\nu}^{\mathrm{em}}$ and ${I_\nu}^{\mathrm{bg}}$ are the total emission and background emission intensities.
From $\mathrm{T_R}$ the line fluxes are calculated using,

$$\mathcal{F} = \frac{\sqrt{\pi}}{2\sqrt{\ln 2}} \mathrm{T_R} \Delta V \quad [\mathrm{K\ km\ s^{-1}}] \tag{19}$$

where the fraction converts the square line profile used in calculations to a Gaussian line profile with ΔV [km s^{-1}] the FWHM of the Gaussian line profile.
and,

$$\mathcal{F} = \frac{\sqrt{\pi}}{2\sqrt{\ln 2}} 8\pi\tilde{\nu}^3 k_{\mathrm{B}} \mathrm{T_R} \Delta v \quad [\mathrm{erg\ cm^{-2}\ s^{-1}}] \tag{20}$$

where $\tilde{\nu} = \nu/c$ [cm^{-1}] is the wavenumber and Δv [cm s^{-1}] the FWHM.

4.2 Limitations and Assumptions

RADEX assumes an isothermal, homogeneous cloud that does not possess a large scale velocity field and is not in LTE but is in SE. The optical depth is also assumed to be independent of the velocity, causing the relation between $\mathrm{T_{ex}}$ and the line flux to break down for large τ. The ortho-para ratio of H_2 is handled internally by RADEX, if left unspecified, but can be indirectly formulated by the user when specifying the o–H_2 and p–H_2 densities. The limits on the input parameters of interest for RADEX range from the following; $0.1 < \mathrm{T_{kin}}$ [K] $< 10^4$, $10^5 < \mathrm{N_{mol}}$ [cm^{-2}] $< 10^{25}$, $10^{-3} < \mathrm{n_{col}}$ [cm^{-3}] $< 10^{13}$. The output of RADEX is not limited necessarily but warning is given to not always trust spectral lines with optically thick, $\tau \gtrsim 100$, results (van der Tak et al. 2007, sec: 4.1.2) or for -0.1 $\lesssim \tau$ (nonlinear amplification), relating to maser lines, which can negatively affect the non-maser lines (van der Tak et al. 2007, sec: 3.6).
There is also no information about the length scale or geometry of the source, as well as the assumption that the source fills the beam antenna, implying RADEX works with a beam-averaged column density (Mangum et al. 2015). The source's peak radiation temperature can then be directly compared to the observed antenna temperature, corrected for optical efficiency of the telescope.
Collisional excitation is assumed dominant, implying $\phi = \varphi = \chi = 1/\Delta v$, where RADEX utilizes a rectangular line shape. This is also the reason why optically thick lines are not modeled well, for τ is assumed constant over the line profile, and not both spectrally and spatially resolved.
The "intermediate" method, and especially the escape probability approximation, assume no knowledge of the internal structure of the cloud, as well as do not account for any dust or free-free opacity to the escape probability. For wavelengths $\gtrsim 1$ mm, dust continuum radiation is negligible, unless the source region has high column densities like in protoplanetary disks, but cannot be ignored for wavelengths $\lesssim 100$ μm (van der Tak et al. 2007, sec: 3.6). However, a tabulated dust or free-free radiation field can be included.
RADEX can only model one molecule (data file) at a time and thus does not support the modeling of multiple species or isotopologues. This is an issue in certain cases where line overlap occurs and possibly influence the excitation.
The molecular data files provided by the LAMDA contain collisional rate coefficients in tabulated form for certain temperatures. Interpolation takes place for all the temperatures in between but no extrapolation is done to avoid numbers blowing up, and the collisional rate coefficients plateau beyond the tabulated bounds.

5 Wrapper comparison

A wrapper serves the purpose of interfacing the RADEX Fortran source code with python to facilitate software development. A number of wrappers already exist (see Table 2 for all those found online), and this section compares their functionality, in order to determine their adaptability for use in ReverseRADEX. Numerical output is also compared to that of RADEX, in Table 3c, in order to ensure accurate values are obtained. The motivation behind the comparisons are that it would be redundant to develop a wrapper specifically for this thesis when capable alternatives are already available.
The wrapping method varied between; *F2PY*[7], which wraps Fortran source code to a python callable module, *RADEX binaries* where the RADEX executable is called from python, and *rewritten python* where the (entire) Fortran source code is ported to python.
Parallelization is also assessed in an effort to speed up the calculations beyond what RADEX is capable of. The RADEX source code utilizes Fortran77 common blocks, making RADEX not thread safe (Shults 2002) and thus not fit for parallelization outright. The source code can be recompiled with the inclusion of the "threadprivate" option for common blocks, to avoid unexpected behavior (Simulia Corp 2008, sec: 11.9.1) caused by multiple threads writing to the same point in memory at the same time. But replacing *common blocks* with *modules* is the much preferred approach to modern Fortran programming, seen in the source code for SpectralRadex (Holdship et al. 2020).
Another feature that is preferable is built in grid processing, to be used for the Brute-force method, Section 2.2.1. A built in grid calculation is likely tailor made to suit the wrappers ecosystem and efficient beyond what can be achieved if naively implemented after the fact for ReverseRADEX.

Table 2: General description of the wrappers's capabilities. Timings were obtained for the input parameters listed in Table 3a and are an average over 1000 runs, see Appendix B for the code.

wrapper	author	wrapping method	parallel processing	built in grid calculations	timings per run [ms]
RADEX	F.F.S van der Tak, et al.	original	no	no	8.12
SpectralRadex	J. Holdship, et al.	F2PY	yes	all parameters	3.96
ndRADEX	A. Taniguchi	binaries[a]	yes	all parameters	86.81[d]
pyradex	A. Ginsburg	F2PY	no[b]	no[c]	46.55
pythonradex	G. Cataldi	rewritten python	no[b]	no	32.18
myRadex	F. Du	F2PY	no[b]	no	x[e]
radexgrid	B. Svoboda	binaries[a]	yes[b]	yes	x

[a]The RADEX binaries get called from python.
[b]Not proclaimed by author or not (thoroughly) tested.
[c]Grid processing does not extend beyond a for loop for certain parameters.
[d]The timing is taken for 3x as many RADEX runs compared to other timings, see Section 5.2.
[e]'x' represents that no timings were conducted.

Timings were obtained, ensuring that the input and output was the *exact* same, see Appendix B for the code. The importance of the timings comes down to ReverseRADEX striving to be practical for scientific use by delivering quick estimates of the physical conditions from the observed cloud.
Another thing of note is that the wrappers in this section and Table 2 are ordered, primarily based on ease of use and feature set, while no definite qualitative benchmark is followed.

[7] See for further information: numpy.org/doc/stable/f2py.

5.1 SpectralRadex

The most recent of the wrappers is SpectralRadex (Holdship et al. 2020), wrapped using F2PY. The RADEX source code has been modernized substantially, with the major gain being the improved ability for parallelization, accomplished by forgoing the use of Fortran77 common blocks.
The parallelization functionality is (optionally) utilized by the built in grid processing, through means of a Pool object like multiprocessing.pool[8] from the python standard library. Furthermore, the use of the Pandas DataFrame[9] makes manipulating data a simple task, especially in the case of the built in grid processing used for the Brute-force method, Section 2.2.1.
Of all the wrappers tested, it is also the fastest wrapper, even beating out RADEX itself with a $\sim$2 x time speed increase, likely another benefit of the modernized Fortran. It even outperforms ndRADEX 20 fold for reasons that become clear in Section 5.2.
In Table 3c it is also shown that although the Fortran source code has been modernized, the numerical results remain comparable to RADEX, likely only deviating due to handling the rounding of the output differently. RADEX rounds most outputs to three decimal digits, whilst SpectralRadex outputs more digits. The largest variation is $\sim$0.00308% for the $\mathcal{F}$ [erg cm^{-2} s^{-1}], most definitely within acceptable deviation.
The recency of SpectralRadex comes accompanied with it being hosted by the UCL astronomy group and thus expected to receive prolonged support, at the very least beyond that of other wrappers, primarily backed by a lone author. The documentation reflects this, being ample and explanatory. In addition, the wrapper is supplied with an additional feature set for spectral modeling (Holdship et al. 2021) (see Section 7.5.3 for possible prospects of incorporating this into ReverseRADEX) as well as being the only wrapper claiming OS independence, improving accessibility. Considering all this, SpectralRadex is the RADEX wrapper utilized in ReverseRADEX.

5.2 ndRADEX

With a similar features set to SpectralRadex, there is ndRADEX (Taniguchi 2019). The primary difference being that the Fortran source code is left untouched and the RADEX binaries are used instead of wrapping the source code using F2PY.
Like SpectralRadex, ndRADEX supports parallelization, albeit not as effective as SpectralRadex's implementation, utilizing noticeably less processing power during run time. This is reflected in the timings (see Table 2) for ndRADEX, being the slowest out of all tested codes. This is unexpected for both pyradex and pythonradex do not claim to posses parallel processing capabilities yet beat out ndRADEX 2x and 3x respectively. The main reason for this discrepancy is that ndRADEX returns output for only one spectral line per RADEX run whereas all other codes return the full output. Naively speaking this amounts to an N times increase in run time, where N is the number of spectral lines in the molecular file. The timings were obtained using the parameters listed in Table 3a, with the code listed in Appendix B. The C atom only has three spectral lines in this case, implying that one run of a RADEX model takes $86.81/3 \approx 28.94$ ms, putting it on par with pythonradex's timings. However, this is still approximately a factor 3 slower than native RADEX at 8.12 ms.
Built in grid calculations are also supported and substantiated by Pandas DataFrames in addition

[8] See for further information: docs.python.org/3/multiprocessing.pool.
[9] See for further information: pandas.pydata.org/docs/DataFrame.

to a similar DataFrame, xarray's[10]. Xarray's operate similarly to Pandas but enable for "Handy I/O", with the particularly useful capability of saving and loading results using netCDF files.
As the RADEX binaries are used directly, it is of no surprise that the results obtained with ndRADEX and RADEX match to the order of at least $\sim 10^{-16}$, see Table 3c, making ndRADEX the most accurate. The small disparity should be nothing more than verisimilitude and can likely be attributed to a floating point error. The primairy reason for not using ndRADEX in ReverseRADEX is the unfavorable output limitation of ndRADEX, causing unnecessary significant computational overhead.

5.3 pyradex

On installation pyradex (Ginsburg 2014) downloads the RADEX Fortran source code, patches it, compiles the code and wraps it using F2PY. The patches are mostly minor with the exception of reassigning the ortho/para ratio calculation to python instead of keeping it in Fortran. Which is a possible cause of the dissimilarity in the numerical results of pyradex compared to RADEX, see Table 3c. Another cause, as with SpectralRadex, is the different handling of the rounding of output but certainly producing acceptable results. Using the latest version of pyradex, the fluxes are not directly computed, although the fluxes seem to be displayed for prior versions, based of the examples in the documentation. Unfortunately the installation experience for pyradex was the least user friendly of all the wrappers and prior versions are therefore not tested.
One of the reasons why the comparison is done for the C atom is because pyradex, using the CO molecule instead, would return 0.0 for the line strengths of higher spectral lines. Obviously this is unacceptable when these values have to be compared with observations of line strengths that, albeit tiny in some cases, are never zero. The exact cause, and if the same trend persists for other molecules, is untested.
Both grid processing and parallelization are not natively supported by pyradex and the timings are the slowest of all the codes (referring to only a single RADEX model run), see Table 2. Pyradex does have multiple contributors to the project as well as recent maintenance, in addition to the inclusion of astropy.units[11], ensuring that the output is exactly what is expected. The combination of the shortcomings however make pyradex unfit for use in ReverseRADEX.

5.4 Miscellaneous wrappers

The following wrappers differ from those above primarily because they solve the radiative transfer problem differently (pythonradex, myRadex) or are severely outdated (radexgrid). This is also the reason why no numeric comparison is conducted for these wrappers.

5.4.1 pythonradex

Pythonradex (Cataldi 2017) is not a wrapper but a purely pythonic implementation of RADEX that utilizes the "sophisticated" ALI[12] method to solve the system, whilst RADEX iteratively solves for

[10] See for further information: xarray.pydata.org.
[11] See for further information: docs.astropy.org/units.
[12] For further details on how ALI is implemented in pythonradex, see the documentation: pythonradex.readthedocs.io/accelerated-lamda-iteration-ali

statistical equilibrium until a consistent solution exist for both the level populations and radiation field (van der Tak et al. 2007, sec: 3.4). Pythonradex has not been explored in great detail, for a fully pythonic implementation of RADEX, in addition to a more sophisticated solver, is expected to suffer significant performance losses compared to the Fortran written version. Using the same comparison method as for the wrappers above (see Tables 2, 3 and Section 5), the timings for pythonradex are ∼32.18 ms per run, approximately 4x as slow as RADEX and nearly 8x slower then SpectralRadex.
Pythonradex also has no intrinsic support for grid and parallel processing and further issues are the inability to reproduce the following essential outputs for ReverseRADEX; T_R [K], $\mathcal{F}$ [K km s^{-1}] and $\mathcal{F}$ [erg cm^{-2} s^{-1}]. The absence of these outputs likely also affects the run time, making it appear faster than it would be if the same output were to be calculated and compared. One reason for this inability is that pythonradex calculates the line flux directly, radiation originating from the cloud only, whereas RADEX calculates what would be observed and output by a telescope; both the cloud, background and foreground radiation[13].
It does however have desirable features, lacking in RADEX or the other wrappers like; the ability to use molecular files that lack frequency data by calculating the frequencies from the energy levels, as well as more geometries to choose from, to name but a few features. Additionally, since the source code is 100% python and not also partially Fortran, and astronomers are more familiar with python than Fortran (Momcheva et al. 2015), it makes pythonradex more accessible towards user modifications to fit their needs and possibly implement as future contributions. And strengthening this accessibility is the inclusion of proper documentation.

5.4.2 myRadex

Also not a wrapper, myRadex (Du 2014), which is a software that solves the same problem as RADEX with one major difference. The difference being that myRadex utilizes an ODE solver that evolves the system towards statistical equilibrium, as opposed to RADEX's iterative method.
The source code is written in Fortran and there is both a command line, and F2PY wrapped version. Further things of note are the absence of built in parallel, and grid processing. Due to these deficiencies, the difference of approaches to solving the system and the lack of documentation, myRadex was deemed unfit for use in ReverseRADEX and hence no timings were performed either.

5.4.3 radexgrid

Radexgrid (Svoboda 2013) is the only wrapper that I was unable to get operational and therefore no timings are available. Reason being, radexgrid appears to be written in python 2 and after the initial release in 2013, has not seen an update past 2014. Radexgrid benefits from code from pyradex but as the name suggests and unlike pyradex, it does support grid processing for all parameters, excluding the geometry. Additionally, radexgrid also claims to support parallelization but the effectiveness is untested and likely not as profound, given that it wraps the RADEX binaries without meaningful modification to address the thread safety, like ndRadex.

[13] See for further information: pythonradex.readthedocs.io/difference-between-pythonradex-and-radex.

Table 3: Numeric wrapper comparison with RADEX as a baseline. See Appendix B for the code and Section 4.1 for a brief explanation on the columns.

(a) Input parameters for the numeric wrapper comparison and timings.

molecule	T_{kin} [K]	N_C [cm^{-2}]	n_{H_2} [cm^{-3}]	T_{bg} [K]	dv [km s^{-1}]	geometry
C atom[a]	100.0	1.0e14	1.0e5	2.73	1.0	uniform sphere

[a](Klein et al. 1998; Schroder et al. 1991; Yamamoto et al. 1991) and NIST for A-values. LAMDA accessed: 25/04/21.

(b) RADEX output for the input parameters listed in Table 3a.

Transition	E_{up} [K]	ν [GHz]	T_{ex} [K]	τ	T_R [K]	n_u [cm^{-3}]	n_l [cm^{-3}]	$\mathcal{F}$ [K km s^{-1}]	$\mathcal{F}$ [erg cm^{-2} s^{-1}]
$^3P_1$—$^3P_0$	2.36e+01	4.92e+02	4.86e+01	1.86e-04	7.02e-03	4.47e-01	2.42e-01	7.47e-03	1.15e-08
$^3P_2$—$^3P_1$	6.25e+01	8.09e+02	4.46e+01	2.18e-04	6.09e-03	3.11e-01	4.47e-01	6.48e-03	4.42e-08
$^3P_2$—$^3P_0$	6.25e+01	1.30e+03	4.60e+01	7.43e-12	1.61e-10	3.11e-01	2.42e-01	1.71e-10	4.86e-15

(c) Numeric comparison of wrapper output with the RADEX output of Table 3b as a baseline. The difference between RADEX and every wrapper is calculated according to: diff % = 100 × (wrapper − RADEX)/RADEX, where "wrapper" and "RADEX" refer to the (table) output columns.

wrapper	Transition	T_{ex} [%]	τ [%]	T_R [%]	n_u [%]	n_l [%]	$\mathcal{F}_{[\text{K km s}^{-1}]}$ [%]	$\mathcal{F}_{[\text{erg cm}^{-2}\text{ s}^{-1}]}$ [%]
pyradex	$^3P_1$—$^3P_0$	1.08e-04	1.55e-02	-6.77e-03	-5.65e-03	-1.77e-02	x[a]	x
	$^3P_2$—$^3P_1$	-3.83e-04	1.84e-02	-3.49e-03	-1.02e-02	-5.65e-03	x	x
	$^3P_2$—$^3P_0$	6.02e-04	-5.65e-03	6.92e-03	-1.02e-02	-1.77e-02	x	x
ndRADEX	$^3P_1$—$^3P_0$	0.00e+00	0.00e+00	-1.24e-14	0.00e+00	0.00e+00	1.16e-14	0.00e+00
	$^3P_2$—$^3P_1$	-1.59e-14	1.24e-14	1.42e-14	0.00e+00	0.00e+00	0.00e+00	0.00e+00
	$^3P_2$—$^3P_0$	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00
SpectralRadex	$^3P_1$—$^3P_0$	1.08e-04	1.55e-02	-6.85e-03	-5.65e-03	-1.77e-02	4.59e-03	3.08e-02
	$^3P_2$—$^3P_1$	-3.84e-04	1.84e-02	-3.63e-03	-1.02e-02	-5.65e-03	6.80e-03	1.08e-02
	$^3P_2$—$^3P_0$	6.01e-04	-5.65e-03	6.68e-03	-1.02e-02	-1.77e-02	-9.91e-03	4.86e-03

[a]'x' Represents that the wrapper did not directly output this quantity.

6 ReverseRADEX

ReverseRADEX is a program that takes molecular spectral line observations of interstellar gas clouds and returns the global physical conditions of said gas cloud based on an escape probability approximation, using the SpectralRadex python wrapper for RADEX. ReverseRADEX will solve for at most 8 parameters, including T_{kin}, N_{mol} and all[14] volume densities of collision partners; H_2, H, e^-, p–H_2, o–H_2, H^+ and He, and at minimum fits any one of the aforementioned parameters. The fitting will be done against user observations, Figure 2, and returns the parameter and uncertainty estimates, Table 4. ReverseRADEX is limited to a terminal application and or .ipynb notebook to operate and does not possess a graphical user interface (GUI), see Section 7.5.5 for further discussion on the UI.

6.1 Inverting RADEX

Section 4.1 lists what RADEX itself is capable of, and what the *important* output is for one run of the program. One of these *important* outputs; T_R, $\mathcal{F}$ [K km s^{-1}], $\mathcal{F}$ [erg cm^{-2} s^{-1}], in ReverseRADEX is taken as input, such that RADEX's input will become ReverseRADEX's output. An algorithm chain, Section 2.2, is employed to achieve this by first finding global estimates of the input parameters to be fit using a grid search, followed by the LM algorithm refining the parameter estimates, which is subsequently complemented by an MCMC algorithm in order to determine the uncertainties.

6.1.1 Input

The input for ReverseRADEX is thus the same as RADEX with the addition of bounds for fit parameters and observed molecular spectral lines for the interstellar cloud of interest. The same units are used and are clearly listed when prompted for input, Figure 3. The first input is the molecular data file following the file format from LAMDA and the second input is the observed data, see Figure 2 for how it should be formulated,

```
# 3
230.538  1.467483300199396e-07  1.467483300199396e-08
345.7959899  9.659722431032617e-07  9.659722431032618e-08
461.0407682  3.220545370547507e-06  3.220545370547507e-07
1267.014486  2.915051410930837e-06  2.9150514109308373e-07
```

Figure 2: The observed data file (new_test.dat, see Figure 3) where; the first line indicates the units used (1: T_R [K], 2: $\mathcal{F}$ [K km s^{-1}], 3: $\mathcal{F}$ [erg cm^{-2} s^{-1}], see Section 4.1), the first column contains the frequencies in GHz, the 2nd column indicates the line strengths in terms of the specified units in the header, and the last column is the uncertainty, in this case a flat 10%. These "observations" were generated with SpectralRadex and hence have an absurd number of decimal digits.

The remainder of the input is similar to RADEX's and is shown in Figure 3, where the first three inputs (after the file paths) are constants; background radiation field T_{bg} [K], line width dv [km s^{-1}] and the escape geometry, Section 4. Subsequent input follows for the parameters that can be

[14] It would be inappropriate to fit H_2 in combination with either p–H_2 or o–H_2, since RADEX internally always calculates the ortho/para ratio for H_2, unless p–H_2 or o–H_2 are specified, in which case H_2 gets ignored.

fit, starting with a prompt if the parameter is to be fit, followed by the upper and lower bounds if yes and the parameter value if no. For the volume densities, the same prompt asking if the parameter should be fit appears but the bounds are entered only once and are the same for all collision partners. Similarly to T_{kin} and N_{mol}, if the collision partner should not be fit bit is selected, the parameter value is required input.

```
[mooren@sl79 reverseRadex]$ python3.6 main.py
Enter molecular file path '*.dat': /home/mooren/BT/moldata/co.dat
Enter data file path '*.dat': /home/mooren/BT/reverseRadex/new_test.dat
Enter background radiation field [K]:
Enter line  width [km/s]:
Enter a geometry (1=sphere, 2=LVG, 3=slab): 1
Fit the kinetic temperature? (y/n): y
Enter minimum kinetic gas temperature [K]: 25
Enter maximum kinetic gas temperature [K]: 750
Fit the column density? (y/n): y
Enter minimum column density [cm^-2]: 1e10
Enter maximum column density [cm^-2]: 5e21
Enter (another) collision partner's name ['h2', 'h', 'e-', 'p-h2', 'o-h2', 'h+',
 'he'] or enter 'no' if not: h2
Fit h2's density? (y/n): y
Enter minimum volume density [cm^-3] for all collision partners: 1e3
Enter maximum volume density [cm^-3] for all collision partners: 1e8
Enter (another) collision partner's name ['h', 'e-', 'p-h2', 'o-h2', 'h+', 'he']
 or enter 'no' if not: n
```

Figure 3: The terminal input for the CO molecule, fitting $25 < T_{kin}$ [K] < 750, $10^{10} < N_{CO}$ [cm^{-2}] $< 5 \times 10^{21}$ and $10^3 < n_{H_2}$ [cm^{-3}] $< 10^8$. Blank entries indicate that default settings were used, $T_{bg} = 2.73$ [K] and $dv = 1$ [km s^{-1}].

Once the input sequence is completed, an overview is displayed before continuing to the fitting process, Figure 4. First it repeats back the file paths of the molecular and observation data file, followed by the units specified in the observed data file and whether uncertainties are included, Figure 2. Thereupon the numeric values for the constant parameters in addition to the numeric variable parameters, their bounds and fit status is reported. The overview finishes of with an indication of the frequency bounds within which all the observed spectral lines should be contained in addition to which escape probability geometry is selected, before prompting the user to either continue towards the fitting process or terminate.

6.1.2 Output

The output of ReverseRADEX will be five files, of which two figures, two data files and a summary file. The two plots produced are; a corner plot, Figure 6, and a spectrum plot, Figure 7a (unfortunately without the "truth" input parameters, of course). Moreover, a RADEX model run for the optimal parameter estimates in addition to the χ^2 values for each line are returned, see Appendix A.2.1. The user can thus inspect if any values appear nonphysical and possibly indicate grounds for dismissal of ReverseRADEX's results, as well as give information on the accompanying T_{ex}, τ, etc. of all the (un)observed spectral lines. Also returned are the sampled MCMC parameter estimates, Appendix A.2.2, to inspect the chains or e.g. if different looking plots are desirable.
Lastly, a summary file, Appendix A.2.3, is written that contains information about the input pa-

rameters and the fitting result.

```
Enter (another) collision partner's name ['h', 'e-', 'p-h2', 'o-h2', 'h+', 'he']
 or enter 'no' if not: n


Selected molfile path              : '/home/mooren/BT/moldata/co.dat'
Selected datafile path             : '/home/mooren/BT/reverseRadex/new_test.dat'
Selected line strength units       : FLUX (erg/cm2/s)
uncertainties included             : yes


[name of parameter, parameter value, (lower bound, upper bound), fit parameter?]
If a parameter is fit, "parameter value" is a dummy number and can be ignored.
If not fit, the boundaries are dummy numbers.
0.0 just indicates SpectralRadex to not use this collision partner.

Selected minimum and maximum
kinetic gas cloud temperature      : ['tkin', 362.5, (25.0, 750.0), True] K
Selected background radiation field: 2.73 K
Selected minimum and maximum
column densities                   : ['cdmol', 2.499999999995e+21, (10000000000.
0, 5e+21), True] cm^-2
Selected volume densities [cm^-3],
h2                                 : ['h2', 49999500.0, (1000.0, 100000000.0), T
rue]
h                                  : ['h', 0.0, (1000.0, 100000000.0), False]
e-                                 : ['e-', 0.0, (1000.0, 100000000.0), False]
p-h2                               : ['p-h2', 0.0, (1000.0, 100000000.0), False]
o-h2                               : ['o-h2', 0.0, (1000.0, 100000000.0), False]
h+                                 : ['h+', 0.0, (1000.0, 100000000.0), False]
he                                 : ['he', 0.0, (1000.0, 100000000.0), False]
Selected line width                : 1.0 km/s
Selected minimum and maximum
frequency                          : (230.30746200000002, 1268.2827687687688) GH
z
Selected geometry                  : uniform sphere

Continue to the fitting process? (y/n) y
```

Figure 4: An overview of the selected (user) settings, Figure 3, is displayed. The "name of the parameter" is primarily used internally, "parameter value" is the numeric selected value used in calculations (if not fit), "(upper bound, lower bound)" is self explanatory and "fit parameter?" is either True or False and either fit or not respectively.

In addition to the summary file, the terminal also displays information on the fitting progress during run time, Figure 5. Primarily to give insight into how much the parameter estimates differ per algorithm, especially important in the case of LM vs MCMC, for LM should have found the correct parameter estimates and any significant discrepancy between the MCMC and LM parameter estimates would likely indicate that the grid search was ineffective at finding the correct minima in parameter space.

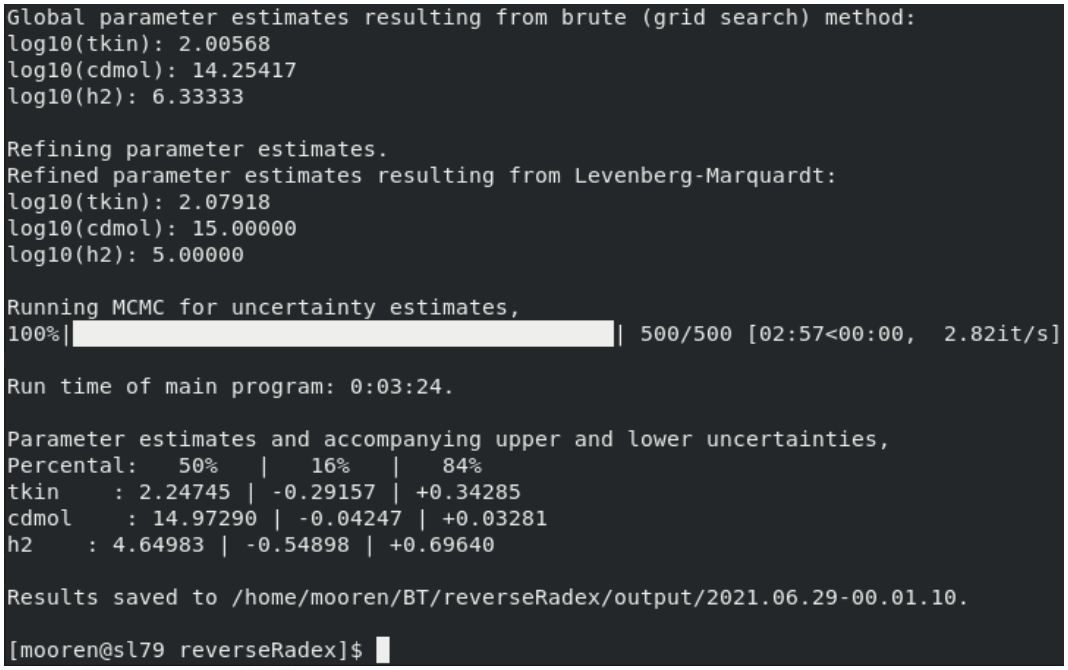


Figure 5: After the user answered "y" to the prompt aksing the user to continue to the fitting process, Figure 4, the resulting terminal output follows. From the top; first the initial estimate based on the global search algorithm, Section 2.2.1, followed by the refined parameter estimates using LM, Section 2.2.2, after which the MCMC algorithm runs with progression indicated by a progress bar, subsequently the run time, and lastly the parameter and uncertainty estimates are displayed, as well as where the results are saved.

6.2 Results

Results of ReverseRADEX will be shown for a single run using mock data generated by SpectralRadex, Figure 2, thus implying ReverseRADEX can find the exact parameter combination, which in real world observations is unlikely to be the case. A comparison is also performed to showcase how the results differ if different amounts of data are available.

6.2.1 Single run

For the input parameters summarized in Figure 4 and using generated data (4 CO spectral lines), Figure 2, the retrieved parameter estimates and uncertainties enclose the input parameters, see Table 4. The parameter estimate is the median of the MCMC chains and the upper and lower uncertainties enclose 1σ of uncertainty within them. The true kinetic temperature deviates the most from the input parameter but falls well within the uncertainties. The largest uncertainties are found for n_{H_2} followed shortly by T_{kin} and translate well to the parameter corner plot, Figure 6, where it is clearly shown that T_{kin} and n_{H_2} are highly correlated. N_{CO} is constrained much further with uncertainties one order of magnitude lower, even though the parameter value itself is one order of magnitude greater than T_{kin}.

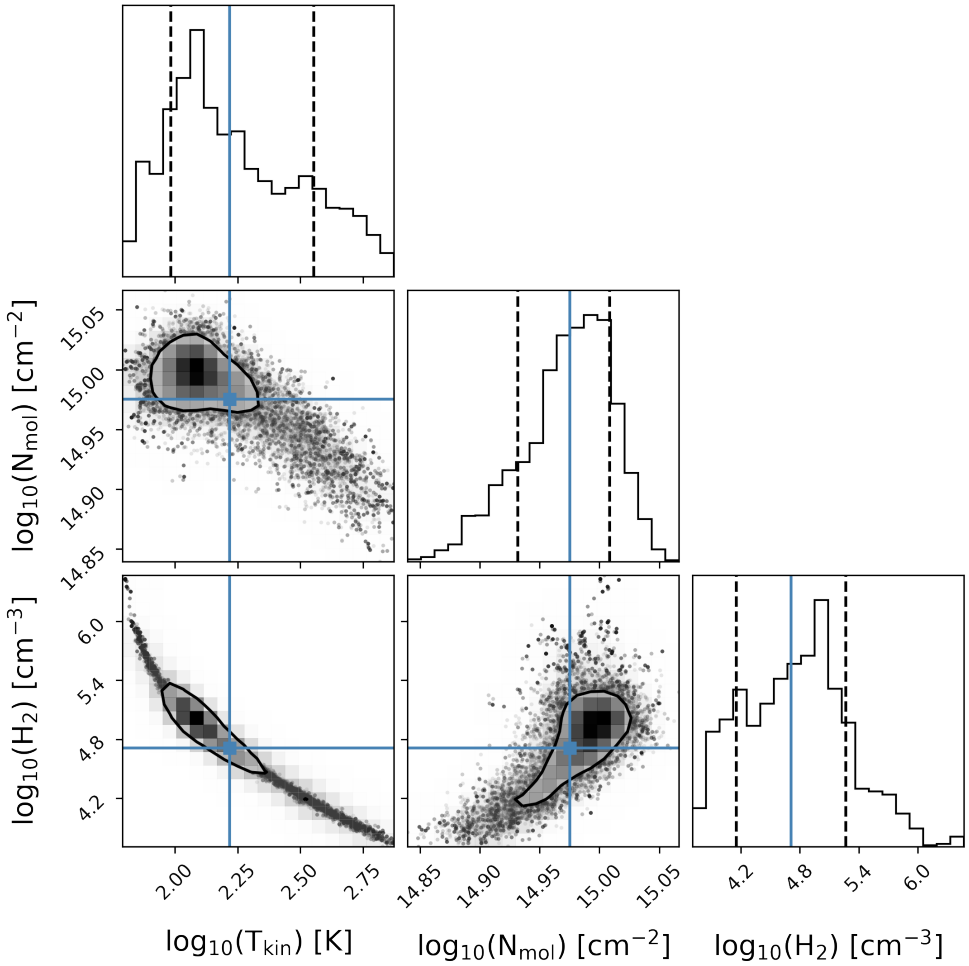
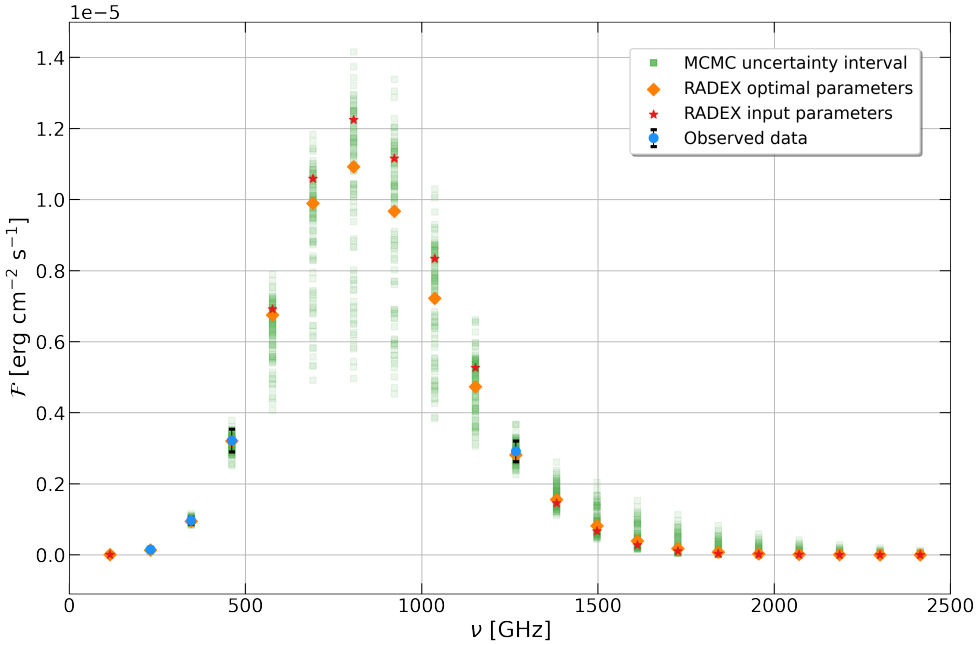
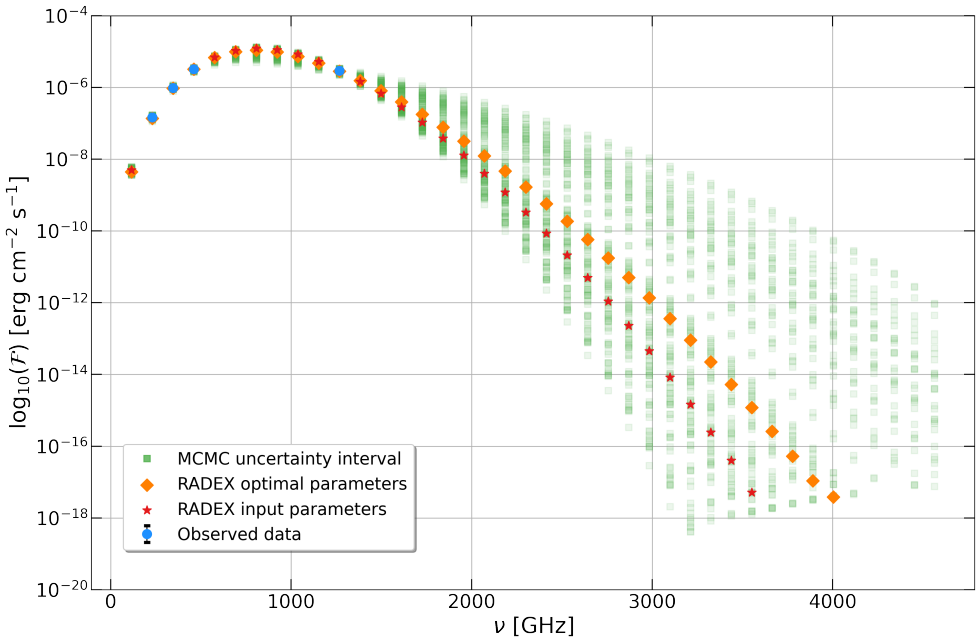


Figure 6: The input parameters are those of Figure 3. The blue lines indicate the median of the parameter distributions, the contours show the 1σ ($\sim$39% for 2D distributions) level and the dashed lines indicate the 1σ ($\sim$68% for 1D distribution) interval for the distributions on the diagonal.

The parameter distributions obtained, see Figure 6, from the MCMC run in the case of n_{H_2} and T_{kin} do not follow the expected Gaussian distribution very well, also indicative of why the upper and lower uncertainties for T_{kin} deviate, see Figure 5. In the case of N_{CO}, the Gaussian distribution is much more discernible. Furthermore, to a lesser extent than between T_{kin} and n_{H_2}, the remaining parameter combinations all show clear correlation. Interestingly, the correlation for all parameter pairs is not along a straight line in parameter space but more akin to a curved line.
Figure 7a shows how the MCMC algorithm sampled parameter estimates used to calculate the RADEX models confine the fit most stringent on the observed data points, especially clear for the CO spectral line at $\sim$1267 GHz where the "MCMC uncertainty interval" is tighter than for surrounding frequencies. The RADEX model for the input parameters is over-plotted and overlaps at the data points, diverging from the optimal parameter estimates primarily in the peak and by several dex at higher frequencies, Figure 7b.
A trace plot accompanying the corner plot of the MCMC chain is also made visible in Appendix A.2.4. The total run time for the settings used in Figure 4 is consistently $\lesssim$ 5 minutes.



(a) The spectrum on a linear scale to clearly show the results near the observed data points.



(b) The spectrum on a $\log_{10}$ scale to showcase how much the optimal parameters still deviate from the RADEX model using the input parameters at higher frequencies.

Figure 7: ReverseRADEX plot of the line flux vs. frequency for four spectral lines of the CO molecule, see Figure 3 for the full input and Figure 2 for which CO spectral lines. The RADEX model for the input parameters, Table 4, is over-plotted to showcase it falls withing the MCMC sampled region, see Figure 6. The MCMC "uncertainty interval" is simply 100 RADEX models selected from randomly drawn parameter combinations after burn-in.

6.2.2 Comparison

Table 4 also contains the results for the same input parameters, see Figure 3, if 8/40, 20/40 and 40/40 spectral lines are available to be fit,

Table 4: A comparison using ReverseRADEX of the parameter (+ uncertainty) estimates for T_{kin}, N_{CO}, and n_{H_2}, with all other parameters fixed, see Figure 3, and using 4/40, 8/40, 20/40, and 40/40 spectral transition lines. In all four cases, the input parameters are within the parameter estimates's 1σ uncertainty interval.

Run	Available data	$\log_{10}(T_{kin})$ [K]	$\log_{10}(N_{CO})$ [cm^{-2}]	$\log_{10}(n_{H_2})$ [cm^{-3}]
	Input parameters	~ 2.0791	15	5
1	4/40 spectral lines	$2.2^{+0.4}_{-0.3}$	$14.98^{+0.04}_{-0.05}$	$4.7^{+0.6}_{-0.6}$
2	8/40 spectral lines	$2.078^{+0.005}_{-0.005}$	$15.00^{+0.03}_{-0.03}$	$5.00^{+0.04}_{-0.04}$
3	20/40 spectral lines	$2.077^{+0.003}_{-0.004}$	$15.00^{+0.02}_{-0.03}$	$5.01^{+0.02}_{-0.02}$
4	40/40 spectral lines	$2.078^{+0.002}_{-0.002}$	$15.00^{+0.02}_{-0.02}$	$5.01^{+0.02}_{-0.02}$

For all four runs, the parameter uncertainties hold the input parameters within but from run 1 to run 2, the parameter estimates significantly differ compared to from run 2 to runs 3 and 4, where the parameter estimates are almost identical. Similarly for the uncertainties, from run 1 to runs 2, 3 and 4 they differ $\sim$2 dex in the case of T_{kin}, $\sim$1 dex in the case of n_{H_2} and only a factor $\sim$2 for N_{CO}. The uncertainties for run 4 are still meaningfully lower than those of runs 2 and 3 however. The excess of data thus does seem to offer diminishing returns when comparing run 1 to to runs 2, 3 and 4 and when comparing runs 2, 3 and 4 between themselves. For run 4 the Gaussian likelihood is resolved quite nicely as seen by the equality in magnitude of the upper and lower uncertainties. T_{kin} is thus constrained most stringently, 1 dex lower than N_{CO} and n_{H_2}, if enough data is available but matches the dex in uncertainty for n_{H_2} in the case of run 1. The column density seems to be indifferent towards change in the amount of data available.
Confirming the suspicion of a refined Gaussian likelihood for runs 2, 3 and 4, a corner plot, Figure 8, shows how the posterior parameter distributions differ based on the amount of available data.

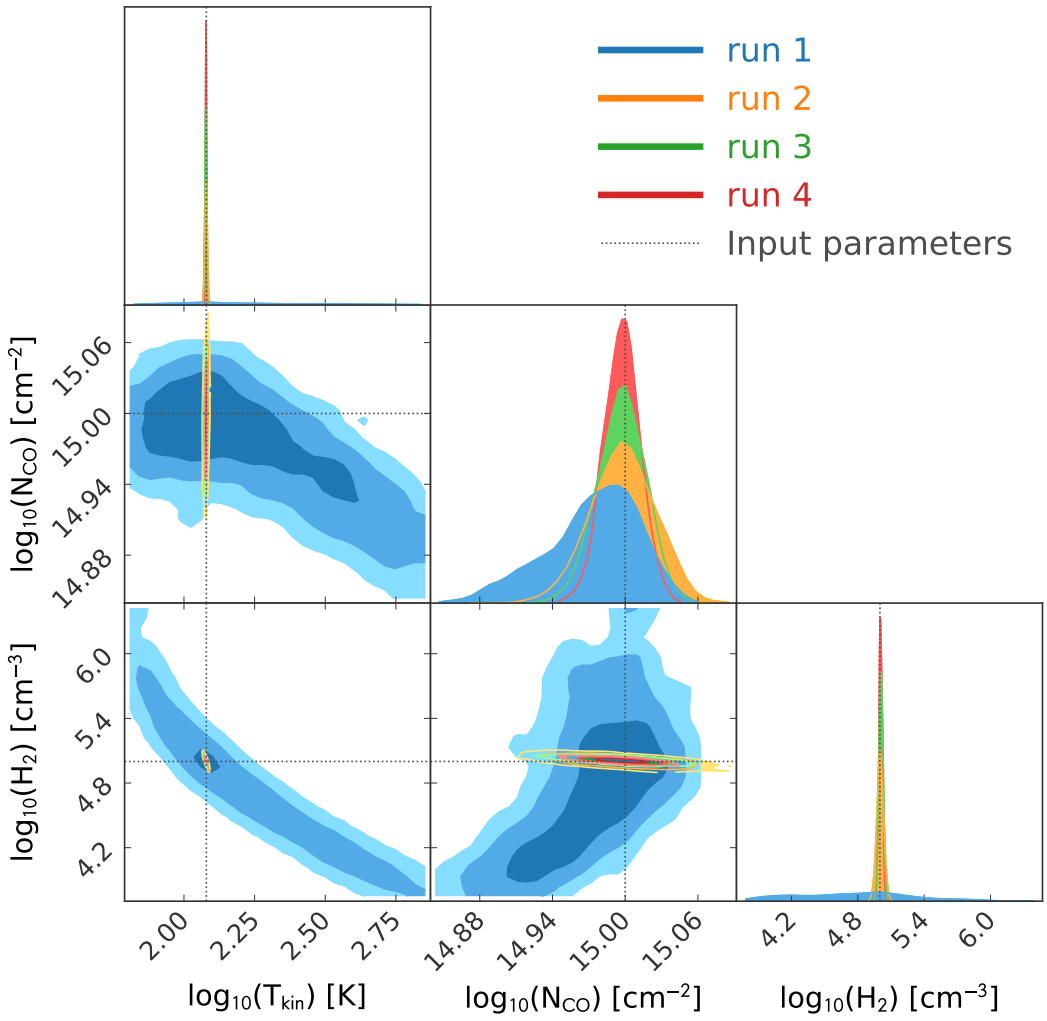


Figure 8: The input parameters are the same for each run, Figure 3, with the exception being that run 1 used "observed input" Figure 2, 4 CO spectral lines, and corollary according to Table 4, runs 2, 3 and 4 used 8, 20 and 40 CO spectral lines respectively. The dotted line indicates the input parameters of the parameter distributions; T_{kin} = 120 [K], N_{CO} = 1e15 [cm^{-2}], n_{H_2} = 1e5 [cm^{-3}]. And the contours show 3 σ contour levels (∼68%, ∼95%, ∼99% standard contour levels and not 2D σ levels).

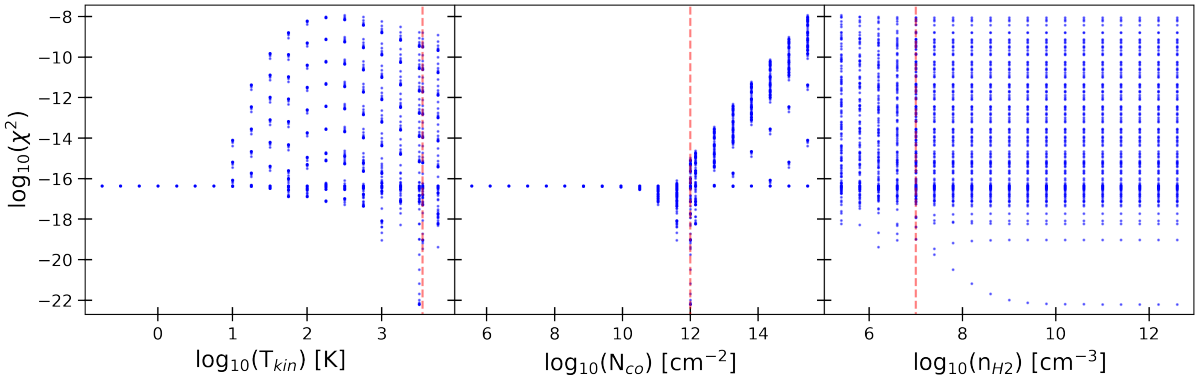
The extreme correlation between T_{kin} and n_{H_2} for run 1 is much less pronounced in the case of runs 2, 3 and 4 but still follow the same diagonal, albeit much tighter bound in terms of *magnitude*. Interestingly, the opposite is true for the other parameter correlations, where the correlation in *magnitude* remains comparably consistent between runs but the diagonals seem to have flipped, rotating ∼90°. For runs 2, 3 and 4, the distributions are much more established, as seen by the median falling dead in the middle of the 1D and 2D distributions, contrary to run 1 where the distributions are misshapen from the expected Gaussian distributions. Although regarding N_{CO}, the Gaussian distribution for run 1 is much more comparable to the other runs, primarily skewed towards lower values, as opposed to the distributions of T_{kin} and n_{H_2} where Gaussian structure is completely absent.

7 Discussion

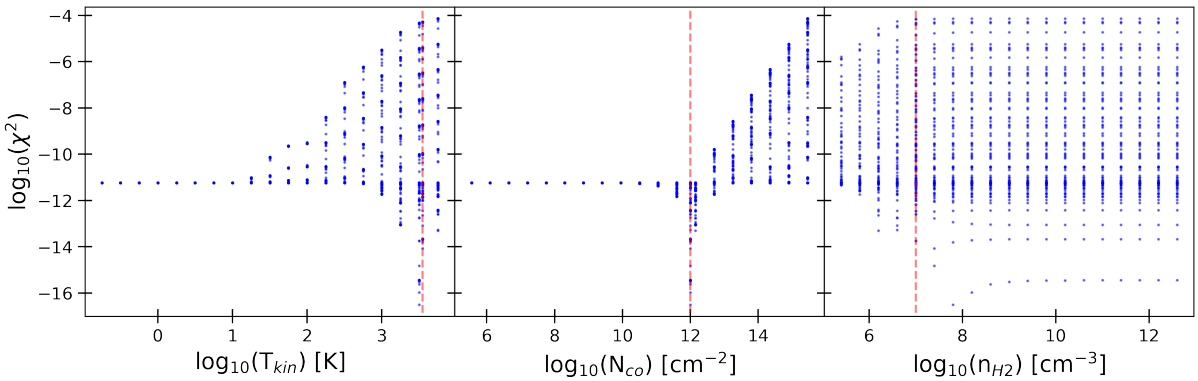
The discussion discloses certain topics of the thesis that may be ameliorated and provides prospects where possible in pursuit of this.

7.1 Parameter degeneracy

A grid in parameter space has been sampled for the CO[15] molecule within bounds of, and for the following parameters; $0.1 < \mathrm{T_{kin}}$ [K] $< 10^5$, $10^5 < \mathrm{N_{CO}}$ [cm^{-2}] $< 10^{16}$, $10^5 < \mathrm{n_{H2}}$ [cm^{-3}] $< 10^{13}$, which should be representative of other researchers's use case. The grid has been sampled for various combinations of lines to showcase the resulting mono- and multi-modal parameter space, see Figure 9, which implies one clear (global) minima or multiple possible (global) minima. The parameter estimates, as well as the true values, were generated using SpectralRadex, instead of using real world observations normally used in the χ^2 calculation. Reason being that the input parameters need to be known in order to distinguish the minima.

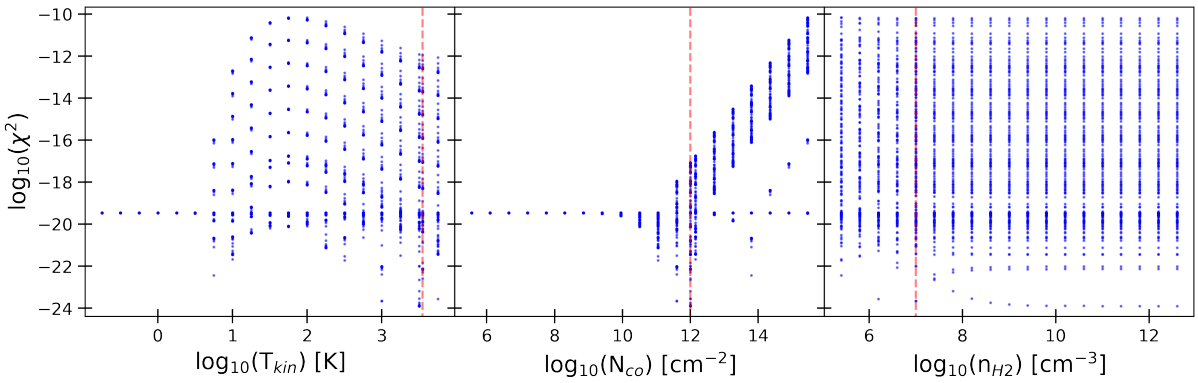


(a) The χ^2 statistic in multi-modal parameter space using CO spectral lines; 5–4, 6–5, 7–6, 8–7.



(b) The χ^2 statistic in mono-modal parameter space using CO spectral lines; 5–4, 18–17, 35–34.

[15] The collisional data for the CO molecule in LAMDA (accessed: 25/04/21) is provided by (Yang et al. 2010) and the spectroscopic data by CDMS and JPL.



(c) The χ^2 statistic in multi-modal parameter space using CO spectral line 4–3.

Figure 9: The un-normalized χ^2 statistic for the CO molecule. The data was generated in a logspace grid where the blue points indicate the χ^2 value obtained for each point in parameter space, and the red vertical dashed line indicates the input parameters; T_{kin} = 3500 [K], N_{CO} = 1e12 [cm^{-2}], n_{H_2} = 1e7 [cm^{-3}].

The parameter space of only a very limited subset of all available molecules in the LAMDA database have been looked at but all showcased a mono-modal parameter space when enough data was used, $4 \gtrsim$ spectral lines, preferably spaced apart over the frequency band. However, this likely varies for each molecule and their degeneracy dependence. Additionally, for the sake of comprehensible plots, not the entire parameter space has been sampled, as certain combinations of parameters resulted in χ^2 values in excess of 10^{250} but the mono-modality of the parameter space still applied. This is possibly due to mis-modeling of results like maser lines, Section 4.2, that occur for some combinations of parameters.

When three, non-adjacent, spectral lines are observed and compared, Figure 9b, the parameter space is non-degenerate, whereas if four adjacent lines are used, see Figure, 9a, the volume density is degenerate. This is also showcased by Figure 7 where the MCMC sampled uncertainty is constrained slightly further at each "observed" spectral line then the surrounding unobserved spectral lines. Figure 7b also reveals that for the higher frequencies, not constrained by "observations", the model with the best parameter estimates still deviates multiple orders of magnitude. Figure 9 also indicates how fine the grid should be sampled and thus what step size to use in the grid search algorithm, Section 2.2.1, in order to find the global minimum in the case of non-adjacent lines.

In the case of only one spectral line, see Figure 9c, the parameter degeneracy (Spilker et al. 2014) becomes unavoidably apparent, especially for T_{kin} and n_{H_2}. n_{H_2} spans a wide range of possible minima as well as a multiple minima for T_{kin}. But ReverseRADEX requires *data points* $\geq$ *parameters* + 1 to run at all, Section 2.2.2 and a single spectral line is never enough to constrain both temperature and density.

One tidbit to add is that the parameter degeneracy mainly persists for and between T_{kin} and n_{H_2} which also produces a different joint posterior contour compared to the similar, albeit orientated differently, T_{kin}–N_{CO} and N_{CO}–n_{H_2} contours, see Figure 8.

The degeneracies are problematic, for the first algorithm in the chain is a simple grid search, not tailored to find appropriate solutions of degenerate, multi-modal, parameter space. To mitigate this, the user can try to enforce strict bounds, defeating the purpose of a solver, or provide more data to constrain the parameters further, preferably spaced out over the frequency band. Although more data is usually always better, expect diminishing results, see Section 6.2.2. Alternatively, on the side of ReverseRADEX, the program would require either a finer grid search, possibly imple-

ment MAGIX's "BestSiteCounter", or different global optimization method e.g. the bees algorithm (Pham et al. 2006), or particle swarm optimization (Kennedy et al. 1995). These algorithms can find the global minimum in such circumstances, whilst likely not significantly increasing computation time, if not diminishing it.

7.2 MCMC

The chains of the MCMC algorithm and the moves within those chains are elaborated upon. Indicating why certain choices were made and where room for improvement lies.

7.2.1 Chains

Continuing where Section 7.1 left off, one possibility is that the MCMC algorithm will migrate from the degenerate false minima, settled on by algorithms prior in the chain, to another (global or local) minima. Although, it is unlikely for the MCMC walkers to always jump, as many times as needed, to find the global minimum, given the sampling period used in ReverseRADEX is only 500 steps. The sampling period is short because the prior algorithms in the chain should have found the parameter estimates already and the MCMC algorithm is primarily used to determine the uncertainties. There is also no thinning of the chains, in favor of efficiency (Link et al. 2012), which does make all determined parameter estimates and uncertainties dependently drawn.
In the limited testing for which the chains did migrate, it happened for $\lesssim 80$ steps, after which the chain settles again. Therefore a burn-in is applied of 100 steps, leaving 400 steps, to help ensure that the determined parameter estimates and uncertainties are drawn from convergent chains.
The user can edit the source files, Appendix C.3.5, to change the sampling and burn-in period. And further testing will have to be conducted in order to settle on a step size that will work for all molecules, and perhaps more notably, for different (more) parameter combinations. Additionally providing the user with the option of setting their own sampling period would be preferred but remains omitted to avoid cluttering the terminal input, but would be added if a GUI was available, Section 7.5.5.

7.2.1.1 Moves

All the moves used in ReverseRADEX are ensemble moves that benefit from parallelization as described in (Foreman-Mackey et al. 2013). The current number of walkers is 35 but should preferably be a multiple of the number of processors used in order to most efficiently make use of the computation time and avoid having to wait for a process that updates less walkers than the processors can handle. In other words, if 4 processors are available and 35 walkers are used, once the $4 \times 8 = 32$ walker moves (updated in sets of 4) have been processed, $35 - 32 = 3$ walkers still remain and only 3 of the 4 processors will be utilized in updating the three walker moves. Not enough testing has been performed to set a confident interval where the number of walkers will always be sufficient for any system.

The StretchMove is not particularly tailored for high dimensional or multi-modal parameter spaces but is indifferent towards them and has the advantage of quickly moving from false minima, following an affine-invariant updating scheme (Allison et al. 2013; Goodman et al. 2010). This is to

combat parameter degeneracy to a certain extent and likely why the burn-in only has to be ∼100 steps before the sampling takes place for convergent chains only. Affine-invariant MCMC in addition to this flexibility also benefits from robustness (Allison et al. 2013).
The differential evolution moves; DEMove and DESnookerMove are fairly similar to each other and are used primarily to overcome the multi-modal issue as they are effective at exploring multi-modal regions (ter Braak et al. 2008).
Regarding the modality of the parameter space there has not been much issue with the current implementation of moves, Section 2.2.3, but testing has been limited. The higher dimensionality has not been tested at all beyond three parameters. The combination of moves for ReverseRADEX has been established primarily based on trial and error of the different moves[16], rather than grounded in research, extending beyond what is mentioned in the paragraphs above. The "correct" combination of moves is thus room for improvement.

7.2.2 Distributions

The MCMC algorithm utilizes both a log likelihood and log prior in order to marginalize over nuisance parameters and combine to form a posterior in order to determine the uncertainties of parameters of interest. The distributions can be manually altered by going to Appendix C.3.2.

7.2.2.1 The likelihood

The (log) likelihood, equation (12), is assumed to be a simple Gaussian likelihood as this is thought to be the general case, suitable for most users. If the user is aware of any underestimates of the variance that are unaccounted for, then this underestimate is not propagated in the uncertainty estimations of the parameters without determining the uncertainty of the underestimate. Additionally, if your object of interest is not modeled well by a Gaussian likelihood for whatever reason, then it is advisable to manually change the likelihood, for ReverseRADEX will not accurately describe the parameter and uncertainty estimates in that case.

7.2.2.2 The prior

The (log) prior, equation (13), used is a simple uninformative uniform prior, while the prior algorithms in the chain already contain information on the parameter space and thus parameter boundaries, beyond those that the user supplied. This entails that the boundaries could be constrained further if the information by prior algorithms is adequate, which in the case of a simple grid search as a global optimization is not deemed adequate. If a global optimization technique like e.g. bees or pso is used that greatly improves the chances of finding the global minimum, then the prior could constrain the posterior further to primarily envelope the global minima and yield more accurate and precise results.
Additionally, the prior need not constrict only the parameters to be fit, but could also constrain other quantities output by SpectralRadex, like τ, T_{ex}, n_u and n_l, or (molecular) mass and length of the object of interest (Kamenetzky et al. 2018), but that would require information not directly supported by (Reverse)RADEX. Caution is advised though, for the prior only applies to the MCMC algorithm in the algorithm chain, thus if your prior is very constricting, it is unlikely to obtain a

[16] See: emcee.readthedocs.io/moves for information about all available moves in emcee.

good fit in the short sampling period used in ReverseRADEX.

7.3 MAGIX

The issues that arose (and still persist) whilst working with MAGIX and further reasons why it was dismissed are detailed here.

7.3.1 Latent MAGIX

The earliest version of ReverseRADEX contained code to support the iterating engine MAGIX which was ultimately dropped in favor for simpler, native to python, implementations of iterative solvers, Section 2.2. Parts of this code are entangled with other parts of the code-base, deemed necessary to operate MAGIX, a separate program to python. Some of this code still remains, primarily in user input, Appendix C.2, and the subsequent grid search algorithm, Appendix C.3.3, compared to the MAGIX free much clearer; fitting helper functions, Appendix C.3.2, LM code, Appendix C.3.4, or MCMC code, Appendix C.3.5, albeit hard to spot without knowing what to look for.
The performance impact of the latent code should be quite minimal, as it is not code that has to run for each iteration of a RADEX model, but possibly still causes slight overhead and is primarily a nuisance in terms of readability of the code in some parts.

7.3.2 MAGIX results

The number of data points does not affect the parameter estimate in any significant way, meaning either the combination of lines randomly converge to the same physical conditions, the 4/40 data points dominate the fitting process over all other data points, or something else under the hood of MAGIX renders void the dependence on data points.
The case presented in Section 3.2 showcases parameter estimates that do not equal the input parameters and consistently produces them from run to run as long as no parameters were altered. Changing up the input parameters slightly however, resulted in different parameter estimates and in some cases also return the input parameters. The uncertainties were mostly the problematic factor in MAGIX in addition to the inconsistent results on slightly varying parameter input estimates, making it unfit for real world use cases where slightly different input parameters by a user are to be expected. It may very well have been mis-use of MAGIX that produces these results for MAGIX has found success in other use cases[17].

7.4 Wrapper comparison

The comparison of the timings could be improved by randomizing the input for each iteration of the 1000 runs but keeping it consistent for every single wrapper. This way any form of caching should mitigated and slight variations in the results might be expected.
Additionally, multiple molecular files with various amounts of lines could have been used to perform the numerical comparison but displaying the numerical results for the CO molecule with 40 lines

[17] See for further information: magix.astro.uni-koeln.de/publications.

would not have been pleasing in tabular form but is difficult without it.

7.5 ReverseRADEX

The continuously updated LAMDA database will make ReverseRADEX and other "intermediate-level" radiative transfer codes passively increase in usefulness. More (accurate) collisional data will becomes available (van der Tak et al. 2020), and the drawback of not being able to model every species will diminish over time.
A discussion of ReverseRADEX is offered in terms of program specific *concepts* rather than the specific components, which are discussed in other sections of the discussion.

7.5.1 Limitations and Assumptions

All the limitations of RADEX, Section 4.2, also apply to ReverseRADEX, as well as some newly introduced ones.
Since ReverseRADEX works with grid calculations (built in by SpectralRadex) there is e.g. no option to control or fix ortho/para ratios if these would be known e.g. from prior observations. An attempted solution could be to enforce the ratio by adding it manually to the prior, 7.2.2.2, although only the MCMC algorithm takes into account the prior (in addition to the likelihood and resulting posterior), possibly converging on vastly different parameter estimates than the LM algorithm. It could conceivably be worked into the grid search however. And since the MCMC walkers only take 500 steps, it is unlikely that the input parameters guided by the ortho/para ratio is not obtained within this sample period.
Most of the testing has been conducted with the CO molecule and in lesser amount with the C atom. There are therefore no guarantees it will work for all molecules present in the LAMDA but ReverseRADEX should offer support for most of the molecular files. No testing has been performed with real observed data either in view of time constraints unfortunately.
Some molecular files from the LAMDA might not be supported. Although the LAMDA has a particular file format, the structure of the molecular files is not always the same, most notably seen in the absence of frequencies or unordered frequencies. Problems thus might ensue but should be *obvious* when they occur. The absence of frequencies can be addressed by ReverseRADEX in the future by calculating them from the energy levels.
ReverseRADEX should be OS independent for the only compatibility trouble might come from RADEX but SpectralRadex claims to be OS independent. I myself however have not been able to get SpectralRadex running properly in Windows.
Currently only T_{kin}, N_{mol} and n_{col} are supported to be fit, to avoid cluttering the UI but once a GUI would be added, it should be easy to introduce the other parameters like dv as fit candidates as well.
The tabulated background radiation field that can be added to RADEX has not been tested for ReverseRADEX and is not currently supported by ReverseRADEX, although support could be added relatively easily. The question remains if SpectralRadex has support for it.
A prospect for (Reverse)RADEX is the ability to model multiple molecules at once or at least model isotopologue lines (Tunnard et al. 2016). Improving the containment of physical parameters by observing more data.
See also the other sections of the discussion, regarding limitations and assumptions not specifically

mentioned here.

7.5.2 Figures

The "MCMC uncertainty interval" in Figure 7a is used instead of an 1σ uncertainty interval. The MCMC samples more clearly show a measure of how different the model could really look with slightly different parameters and where the model is sampled most densely, information that would be lost with a simple 1σ uncertainty interval.

7.5.3 Fitting

Currently, the observed peak line strength is matched against (Sepctral)RADEX's line strength output and thus does not account for the line profile and possible (hyper-fine) line overlap. This might be ameliorated by utilizing SpectralRadex's spectral modeling capabilities (Holdship et al. 2021) that can be used to fit molecular spectra instead of peak line strengths.
Line overlap is also addressed by using an opacity weighted radiation temperate, following (Hsieh et al. 2015) but would preferably be addressed in the source code of RADEX directly.
Alternatively, line ratios would be a better candidate for the χ^2 statistic, for the calibration uncertainties of the telescope will divide out between the two spectral lines and thus diminish.

Regarding the bounds of the volume densities of the collision partners, currently they are set for all partners at once. This is not much of an issue when the bounds of the all parameters to be fit is unknown but when once parameter is constrained prior to running it through ReverseRADEX for whatever reason, then it would cause unnecessary computational overhead, in addition to parameter degeneracy, to have global bounds in order to accommodate for the most uncertain collision partner. Alternatively if bounds are know to be strict for multiple collision partners but also vastly different for said collision partners, similar computational overhead and parameter degeneracy is introduced. The primary reason it is omitted from the current implementation of ReverseRADEX is to reduce cluttering of the (terminal) UI, as well as the limited foreseen use of such a feature following the limited amounts of available collisional data for certain molecules. But this data is only said to improve in the coming years and a GUI with the option of individual bounds for individual collision partners would be the preferred solution.

7.5.4 Code

The code has been written in pursuit of (Wilson et al. 2014)'s *best practices*, in order to write *good* code, and following (Prlić et al. 2012) in view of open science, making ReverseRADEX accessible and clear, both for the user and (future) developer(s). In both cases, the cutting of MAGIX from the codebase at a later stage of the project was problematic, with latent MAGIX code still persisting, Section 7.3.1. This is also indicative of the code not being modular enough to swap out one algorithm chain, albeit and external one, for another one and continues to serve as a point of improvement that can be solved by refactoring the code.
Additional improvements in no particular order, primarily in favor of code maintainability and readability are; type hinting, provide and use built in (unit)tests, add assertions to check validity

of code operation, provide online external documentation as well as improve in-software documentation, extend in-software documentation with examples, reduce repeated code. See Appendix B of (van der Tak et al. 2007) for the coding standards of RADEX itself.

7.5.5 User interface

An effort was made to make the terminal UI as user friendly as possible, following (Prlić et al. 2012), by being lenient where possible and allowing the user to re-enter input if invalid instead of terminating the program. Once invalid input is detected, a clear warning should be displayed that explains what was invalid and the user should in most cases be able to continue without much backtrack. Invalid input should in all cases be caught before running the program as well, to avoid confusing about what might have gone wrong. There might still be cases where a warning by (Spectral)RADEX occurs, "Warning: Calculation did not converge in 9999 iterations.", but these can be safely ignored based on my experience. The parameter combinations that result in no convergence are unlikely to be candidates for observations. A likely candidate for failure are the observed and molecular data files that are only checked by extension in ReverseRADEX and not much effort goes into verifying their contents before run time. Sometimes this failure will be invisible and no warning or error is returned even-though the program is either stuck, or terminated.
Alternative methods to running ReverseRADEX could be to allow for an input file and preferably using a GUI but those options where omitted in view of time constraints. What is included however is a Jupyter notebook[18] that allows for manual user input, see Appendix A.1. The notebook needs to remain in the root folder of ReverseRADEX or be prepared to run into at the very least issues with relative imports. The manual input is not checked like that of the terminal based input so try and at least keep the type of the input the same as in Appendix A.1.

[18] See for further information: https://jupyter.org.

8 Conclusion

A program was developed to quickly gauge physical conditions of interstellar gas clouds from molecular spectral line observations, with RADEX as the underlying radiative transfer code based on an escape probability formalism. RADEX takes physical parameters as input and outputs molecular spectral line data. ReverseRADEX inverts this process by taking molecular spectral line data as input, as well as molecular collisional data, provided it follows the LAMDA format, and outputting estimated global physical conditions. To this end, a variety of python wrappers for RADEX were compared and SpectralRadex was picked for its suitable feature set, including grid search and accompanying parallelization capabilities, as well as the prospect of incorporating the spectral modeling capabilities into ReverseRADEX.

ReverseRADEX utilizes three algorithms chained together; the Brute-force method examines parameter space and determines initial parameter estimates, followed by the LM algorithm in order to refine the parameter estimates, finishing of with an MCMC algorithm to determine uncertainties. MAGIX was considered instead for optimization purposes but the implementation turned out futile.

Synthetic results of ReverseRADEX prove to be promising, finding the *true* physical conditions within a practical time of under 5 minutes, Section 6.2.1. The best results are obtained if more data is available, but primarily if spaced out over the frequency band as opposed to clumped together.

Limitations and prospects for ReverseRADEX are discussed in Section 7, most notably; replacing and adjusting the Brute-force method and MCMC algorithm respectively, adding a GUI, and refactoring the code for maintainability.

Acknowledgments

This brings me to the end of my bachelors research project and another learning experience richer. I would like to thank Prof. Dr. Floris F.S. van der Tak for allowing me to pursue this bachelor project and aiding me along the way. Laying emphasis on the aiding aspect as Floris really allowed me to put forth my own project and served as a guiding role more than a demanding one being open to my own suggestions as well. Nonetheless, the project would not have been completed without his help.
I would also like to thank Dr. Kateryna Frantseva for being willing to serve as the 2nd examiner for this bachelor project, as well as Drs. Martin G.R Vogelaar for providing me with the necessary information on where and how to publish ReverseRADEX.
Software: RADEX (van der Tak et al. 2007), SpectralRadex (Holdship et al. 2020), ndRADEX (Taniguchi 2019), pyRadex (Ginsburg 2014), pythonradex (Cataldi 2017), NumPy (Harris et al. 2020), SciPy (Virtanen et al. 2020), Pandas (McKinney 2010; Reback et al. 2020), emcee (Foreman-Mackey et al. 2013), Matplotlib (Hunter 2007), corner (Foreman-Mackey 2016), pyGTC (Bocquet et al. 2016)

References

Allison, R. and J. Dunkley (2013). In: *MNRAS* 437.4, 3918–3928. DOI: 10.1093/mnras/stt2190.

Bocquet, S. and F. W. Carter (2016). In: *J. Open Source Softw.* 1.6. DOI: 10.21105/joss.00046.

Brinch, C. and M. R. Hogerheijde (Nov. 2010). In: *A&A* 523, A25. DOI: 10.1051/0004-6361/201015333.

Cataldi, G. (2017). Version used: 0.1 (Oct 2020); original release/first (known) commit (Dec 2017). URL: https://github.com/gica3618/pythonradex.

de Jong, T., W. Boland, and A. Dalgarno (1980). In: *A&A* 91, pp. 68–84. URL: https://ui.adsabs.harvard.edu/abs/1980A&A....91...68D.

de Jong, T., S. Chu, and A. Dalgarno (1975). In: *ApJ* 199, pp. 69–78. DOI: 10.1086/153665.

Draine, B. T. (2011). first print. PUP. ISBN: 978-1-400-84732-7.

Du, F. (2014). Current version: (Jun 2020); original release/first (known) commit (Jan 2014). URL: https://github.com/fjdu/myRadex.

Foreman-Mackey, D. (2016). In: *J. Open Source Softw.* 1.2, p. 24. DOI: 10.21105/joss.00024.

Foreman-Mackey, D., D. W. Hogg, D. Lang, et al. (2013). In: *Publ. Astron. Soc. Pac.* 125.925, 306–312. DOI: 10.1086/670067.

Ginsburg, A. (2014). Version used: 0.4.2.dev (Aug 2020); original release/first (known) commit (Feb 2014). URL: https://github.com/keflavich/pyradex.

Goldsmith, P. F. and W. D. Langer (1999). In: *ApJ* 517.1, pp. 209–225. DOI: 10.1086/307195.

Goodman, J. and J. Weare (Jan. 2010). In: *Commun. Appl. Math. Comput. Sci.* 5.1, pp. 65–80. DOI: 10.2140/camcos.2010.5.65.

Harris, C. R., K. J. Millman, S. J. van der Walt, et al. (Sept. 2020). In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

Hogerheijde Michiel R and van der Tak, F. F. S. (2000). In: *A&A*. arXiv: astro-ph/0008169.

Holdship, J. and et al. (2021). In prep.

Holdship, J. and the UCL Astronomy Group (2020). Version used: 0.3.2 (Mar 2021); original release/first (known) commit (Jul 2020). URL: https://github.com/uclchem/SpectralRadex.

Hsieh, T.-H., S.-P. Lai, A. Belloche, et al. (2015). In: *ApJ* 802.2, p. 126. DOI: 10.1088/0004-637x/802/2/126.

Hunter, J. D. (2007). In: *Comput. Sci. Eng.* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Kamenetzky, J., G. C. Privon, and D. Narayanan (2018). In: *ApJ* 859.1, p. 9. DOI: 10.3847/1538-4357/aab3e2.

Kennedy, J. and R. Eberhart (1995). In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4, pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968.

Klein, H., F. Lewen, R. Schieder, et al. (1998). In: *ApJ* 494.1, pp. L125–L128. DOI: 10.1086/311169.

Link, W. A. and M. J. Eaton (2012). In: *Methods in Ecology and Evolution* 3.1, pp. 112–115. DOI: 10.1111/j.2041-210X.2011.00131.x.

Mangum, J. G. and Y. L. Shirley (2015). In: *Publ. Astron. Soc. Pac.* 127.949, 266–298. DOI: 10.1086/680323.

McKinney, W. (2010). In: *Proceedings of the 9th Python in Science Conference*. Ed. by S. van der Walt and J. Millman, pp. 56 –61. DOI: 10.25080/Majora-92bf1922-00a.

Mendoza, E., N. Duronea, D. Ronsó, et al. (2021). In: *Frontiers in Astronomy and Space Sciences* 8. DOI: 10.3389/fspas.2021.655450.

Mihalas, D. (1978). 2nd ed. San Francisco: W. H. Freeman and Co.

Möller, T., Bernst, I., Panoglou, D., et al. (2013). In: *A&A* 549, A21. arXiv: 1210.6466 [astro-ph].

Möller, T. and Panoglou, D. (May 2020). User manual, PDF. Version 2.1.0. I. Physikalisches Institut, Universität zu Köln. 50937 Köln, Germany. URL: https://magix.astro.uni-koeln.de/sites/magix/files/files/MAGIX_Manual.pdf.

Momcheva, I. and E. Tollerud (2015). arXiv: 1507.03989 [astro-ph.IM].

Moré, J. J. (1978). In: *Numerical Analysis*. Ed. by G. A. Watson. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 105–116. ISBN: 978-3-540-35972-2.

Nelson, B., E. B. Ford, and M. J. Payne (2013). In: *ApJS* 210.1, p. 11. DOI: 10.1088/0067-0049/210/1/11.

Osterbrock, D. E. and G. J. Ferland (2006). 2nd ed. University science books.

Pham, D., A. Ghanbarzadeh, E. Koç, et al. (2006). In: *Intelligent Production Machines and Systems*. Ed. by D. Pham, E. Eldukhri, and A. Soroka. Oxford: ESL, pp. 454–459. ISBN: 978-0-08-045157-2. DOI: 10.1016/B978-008045157-2/50081-X.

Prlić, A. and J. B. Procter (Dec. 2012). In: *PLOS Computational Biology* 8.12, pp. 1–3. DOI: 10.1371/journal.pcbi.1002802.

Reback, J., W. McKinney, jbrockmendel, et al. (Dec. 2020). Version v1.1.5. DOI: 10.5281/zenodo.4309786.

Roweis, S. (1996). Notes, PDF. New York University, Department of Computer Science, Courant Institute of Mathematical Sciences. New York, USA. URL: https://cs.nyu.edu/~roweis/notes/lm.pdf.

Rybicki, G. B. (1985). In: NATO ASI Series (Series C: Mathematical and Physical Sciences), vol 152. Springer, Dordrecht, pp. 199–206. DOI: 10.1007/978-94-009-5372-7_16.

Rybicki, G. B. and A. P. Lightman (2004). New York, USA: Wiley. ISBN: 978-0-471-82759-7.

Schöier, F. L., van der Tak, F. F. S., van Dishoeck, E. F., et al. (2005). In: *A&A* 432.1, pp. 369–379. DOI: 10.1051/0004-6361:20041729.

Schroder, K, V Staemmler, M. D. Smith, et al. (1991). In: *J. Phys. B: At. Mol. Opt. Phys.* 24.10, pp. 2487–2502. DOI: 10.1088/0953-4075/24/10/007.

Shults, B. (2002). In: *32nd Annual Frontiers in Education*. IEEE, T1G–14–T1G–20. ISBN: 0-7803-7444-4. DOI: 10.1109/FIE.2002.1157918.

Simulia Corp (Apr. 2008). User guide, PDF. Version 6.8. (Accessed on 08/06/2021) URL: simulia.com/Analysis-Manual/11.9.1-Parallel-execution-in-Abaqus. Dassault Systèmes Simulia Corp. Waltham, MA 02451 - USA.

Sivia, D. and J. Skilling (2006). URL: global.oup.com/Data-Analysis-A-Bayesian-Tutorial. Oxford University Press. ISBN: 978–0–19–856831–5.

Spilker, J., D. P. Marrone, J. Aguirre, et al. (2014). In: *ApJ* 785, p. 149.

Svoboda, B. (2013). Current version: 0.1 (Oct 2014); original release/first (known) commit (Oct 2013). URL: https://github.com/autocorr/radexgrid.

Taniguchi, A. (2019). Version used: 0.2.2 (Oct 2020); original release/first (known) commit (May 2019). DOI: 10.5281/zenodo.4139707. URL: https://github.com/astropenguin/ndradex.

ter Braak, C. J. F. and J. A. Vrugt (Oct. 2008). In: *Stat. Comput.* 18.4, pp. 435–446. DOI: 10.1007/s11222-008-9104-9.

Tunnard, R. and T. R. Greve (2016). In: *ApJ* 819.2, p. 161. DOI: 10.3847/0004-637x/819/2/161.

van der Tak, F. F. S., Black, J. H., Schöier, F. L., et al. (2007). In: *A&A* 468.2, pp. 627–635. DOI: 10.1051/0004-6361:20066820.

van der Tak, F. (June 2011). In: *Proc. Int. Astron. Union* 7.S280, pp. 449–460. DOI: 10.1017/s1743921311025191.

van der Tak, F., F. Lique, A. Faure, et al. (Apr. 2020). In: *Atoms* 8.2, p. 15. DOI: 10.3390/atoms8020015.

van Dishoeck, E. F. (2017). In: *Proc. Int. Astron. Union* 13.S332, 3–22. DOI: 10.1017/s1743921317011528.

van Zadelhoff, G.-J., Dullemond, C. P., van der Tak, F. F. S., et al. (2002). In: *A&A* 395.1, pp. 373–384. DOI: 10.1051/0004-6361:20021226.

Virtanen, P., R. Gommers, T. E. Oliphant, et al. (2020). In: *Nat. Methods* 17, pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

Wilson, G, D. Aruliah, C. Brown, et al. (2014). In: *PLoS Biol* 12 (1), e1001745. DOI: 10.1371/journal.pbio.1001745.

Yamamoto, S. and S. Saito (1991). In: *ApJ* 370, pp. L103–L105. DOI: 10.1086/185987.

Yang, B., P. C. Stancil, N. Balakrishnan, et al. (2010). In: *ApJ* 718.2, pp. 1062–1069. DOI: 10.1088/0004-637x/718/2/1062.

Appendices

A ReverseRADEX

A.1 main.ipynb

An Jupyter notebook to be used instead of the terminal version of ReverseRADEX. The third cell is Figure 10 and should be the only cell a user has to change any values in order to get ReverseRADEX working for their data.

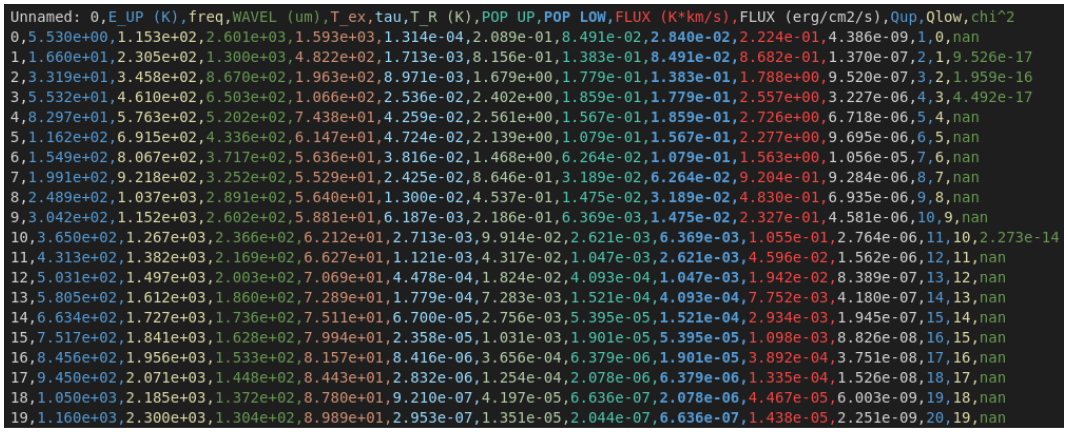
```
#### If you would like to set the input manually, uncomment the cell ####


user_molfile  = '/home/mooren/BT/moldata/co.dat'
user_datfile  = '/home/mooren/BT/reverseRadex/new_test.dat'
freq_indices  = data_retrieval.get_molfile_frequency_index(user_datfile,
                                                           user_molfile)
freq          = data_retrieval.get_frequencies(freq_indices, user_molfile)
user_mol_frequencies, freq_min, freq_max, number_of_lines_total = freq
freq_range    = (freq_min, freq_max)


units             = data_retrieval.get_user_units(user_datfile)
uncertainties     = data_retrieval.uncertainties_included(user_datfile)
(y_observed,
 y_uncertainties) = data_retrieval.line_strengths(user_datfile,
                                                  uncertainties)

# variable parameters.
# [name parameter, init guess, (bound_low, bound_upp), fit parameter?]
# 0.1 < tkin < 1e4 [K]
temp_kin = ['tkin', 131, (10.0, 500.0), True]
# 1e5 < cdmol 1e25 [cm^-2]
coldens  = ['cdmol', 3e16, (1e10, 5e21), True]
# coll partner:(init guess, fit parameter?)
# 1e-3 < coll partner < 1e13 [cm^-3]
voldens  = {'h2':(3e4, True), 'h':(0.0, False), 'e-':(0.0, False),
            'p-h2':(0, False), 'o-h2':(0.0, False), 'h+':(0.0, False),
            'he':(0.0, False), 'min_max':(5e3, 5e8)}

# constant parameters.
Tbg  = 2.73  # K
dv   = 1.0   # km s^-1
geom = 1     #(1=sphere, 2=LVG, 3=slab)
# just for displaying purposes,
geom_name = 'uniform sphere'


#### If you would like to set the input manually, uncomment the cell ####
```

Figure 10: One cell of the main.iypnb is displayed here that concerns how manual user input should be entered, if preferred over working from the terminal.

A.2 Output

A.2.1 RADEX.csv

"RADEX.csv" Is saved for the user to see if RADEX produces any physically feasibly results for the selected input, and determined output parameters. Additionally, it also provides information specific to (un)observed spectral lines e.g. T_{ex}, optical depth τ and number densities of the upper and lower state. The un-normalized χ^2 values for each observed line is calculated for the user to see which line(s) were most important in the fitting process and if a particular (set) of lines possibly originates from vastly different global physical conditions.

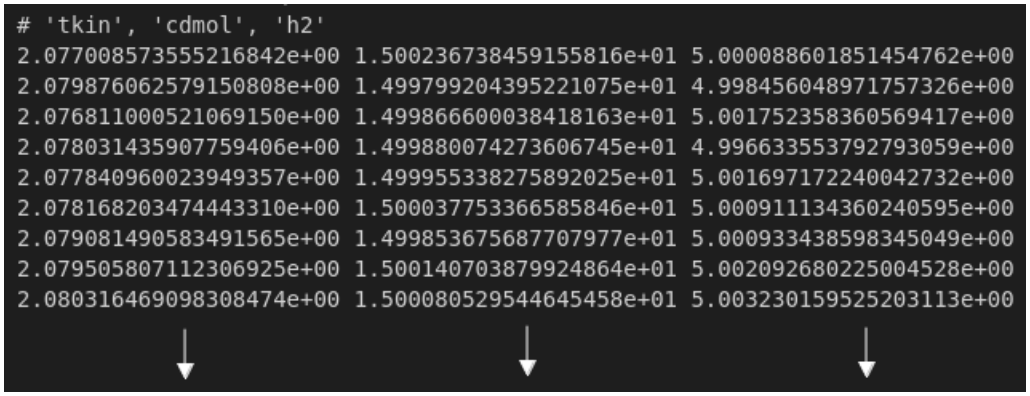


Unnamed: 0	E_UP (K)	freq	WAVEL (um)	T_ex	tau	T_R (K)	POP UP	POP LOW	FLUX (K*km/s)	FLUX (erg/cm2/s)	Qup	Qlow	chi^2
0	5.530e+00	1.153e+02	2.601e+03	1.593e+03	1.314e-04	2.089e-01	8.491e-02	2.840e-02	2.224e-01	4.386e-09	1	0	nan
1	1.660e+01	2.305e+02	1.300e+03	4.822e+02	1.713e-03	8.156e-01	1.383e-01	8.491e-02	8.682e-01	1.370e-07	2	1	9.526e-17
2	3.319e+01	3.458e+02	8.670e+02	1.963e+02	8.971e-03	1.679e+00	1.779e-01	1.383e-01	1.788e+00	9.520e-07	3	2	1.959e-16
3	5.532e+01	4.610e+02	6.503e+02	1.066e+02	2.536e-02	2.402e+00	1.859e-01	1.779e-01	2.557e+00	3.227e-06	4	3	4.492e-17
4	8.297e+01	5.763e+02	5.202e+02	7.438e+01	4.259e-02	2.561e+00	1.567e-01	1.859e-01	2.726e+00	6.718e-06	5	4	nan
5	1.162e+02	6.915e+02	4.336e+02	6.147e+01	4.724e-02	2.139e+00	1.079e-01	1.567e-01	2.277e+00	9.695e-06	6	5	nan
6	1.549e+02	8.067e+02	3.717e+02	5.636e+01	3.816e-02	1.468e+00	6.264e-02	1.079e-01	1.563e+00	1.056e-05	7	6	nan
7	1.991e+02	9.218e+02	3.252e+02	5.529e+01	2.425e-02	8.646e-01	3.189e-02	6.264e-02	9.204e-01	9.284e-06	8	7	nan
8	2.489e+02	1.037e+03	2.891e+02	5.640e+01	1.300e-02	4.537e-01	1.475e-02	3.189e-02	4.830e-01	6.935e-06	9	8	nan
9	3.042e+02	1.152e+03	2.602e+02	5.881e+01	6.187e-03	2.186e-01	6.369e-03	1.475e-02	2.327e-01	4.581e-06	10	9	nan
10	3.650e+02	1.267e+03	2.366e+02	6.212e+01	2.713e-03	9.914e-02	2.621e-03	6.369e-03	1.055e-01	2.764e-06	11	10	2.273e-14
11	4.313e+02	1.382e+03	2.169e+02	6.627e+01	1.121e-03	4.317e-02	1.047e-03	2.621e-03	4.596e-02	1.562e-06	12	11	nan
12	5.031e+02	1.497e+03	2.003e+02	7.069e+01	4.478e-04	1.824e-02	4.093e-04	1.047e-03	1.942e-02	8.389e-07	13	12	nan
13	5.805e+02	1.612e+03	1.860e+02	7.289e+01	1.779e-04	7.283e-03	1.521e-04	4.093e-04	7.752e-03	4.180e-07	14	13	nan
14	6.634e+02	1.727e+03	1.736e+02	7.511e+01	6.700e-05	2.756e-03	5.395e-05	1.521e-04	2.934e-03	1.945e-07	15	14	nan
15	7.517e+02	1.841e+03	1.628e+02	7.994e+01	2.358e-05	1.031e-03	1.901e-05	5.395e-05	1.098e-03	8.826e-08	16	15	nan
16	8.456e+02	1.956e+03	1.533e+02	8.157e+01	8.416e-06	3.656e-04	6.379e-06	1.901e-05	3.892e-04	3.751e-08	17	16	nan
17	9.450e+02	2.071e+03	1.448e+02	8.443e+01	2.832e-06	1.254e-04	2.078e-06	6.379e-06	1.335e-04	1.526e-08	18	17	nan
18	1.050e+03	2.185e+03	1.372e+02	8.780e+01	9.210e-07	4.197e-05	6.636e-07	2.078e-06	4.467e-05	6.003e-09	19	18	nan
19	1.160e+03	2.300e+03	1.304e+02	8.989e+01	2.953e-07	1.351e-05	2.044e-07	6.636e-07	1.438e-05	2.251e-09	20	19	nan

Figure 11: The "RADEX.csv" file containing the RADEX model results for the input parameters and optimal parameter values obtained in the fitting process, see Figure 13. Not all output is displayed for there are 40 lines in the LAMDA CO file. The colors are a result of the text viewer used to easily distinguish between columns.

A.2.2 sampler.dat

"sampler.dat" is the saved version of an emcee EnsembleSampler.flatchain object, ideal for plotting Giant Triangle Confusograms (GTC) with e.g. the pyGTC python module for instance, see Figure 8. The file consists of 35 (walkers) $\times$ 500 (steps) = 17500 rows and columns for every parameter fit.



```
# 'tkin', 'cdmol', 'h2'
2.077008573555216842e+00 1.500236738459155816e+01 5.000088601851454762e+00
2.079876062579150808e+00 1.499799204395221075e+01 4.998456048971757326e+00
2.076811000521069150e+00 1.499866600038418163e+01 5.001752358360569417e+00
2.078031435907759406e+00 1.499880074273606745e+01 4.996633553792793059e+00
2.077840960023949357e+00 1.499955338275892025e+01 5.001697172240042732e+00
2.078168203474443310e+00 1.500037753366585846e+01 5.000911134360240595e+00
2.079081490583491565e+00 1.499853675687707977e+01 5.000933438598345049e+00
2.079505807112306925e+00 1.500140703879924864e+01 5.002092680225004528e+00
2.080316469098308474e+00 1.500080529544645458e+01 5.003230159525203113e+00
```

Figure 12: The "sampler.dat" file for the input of Figure 3, containing the parameter values for the walkers of the total MCMC chain, including burn-in.

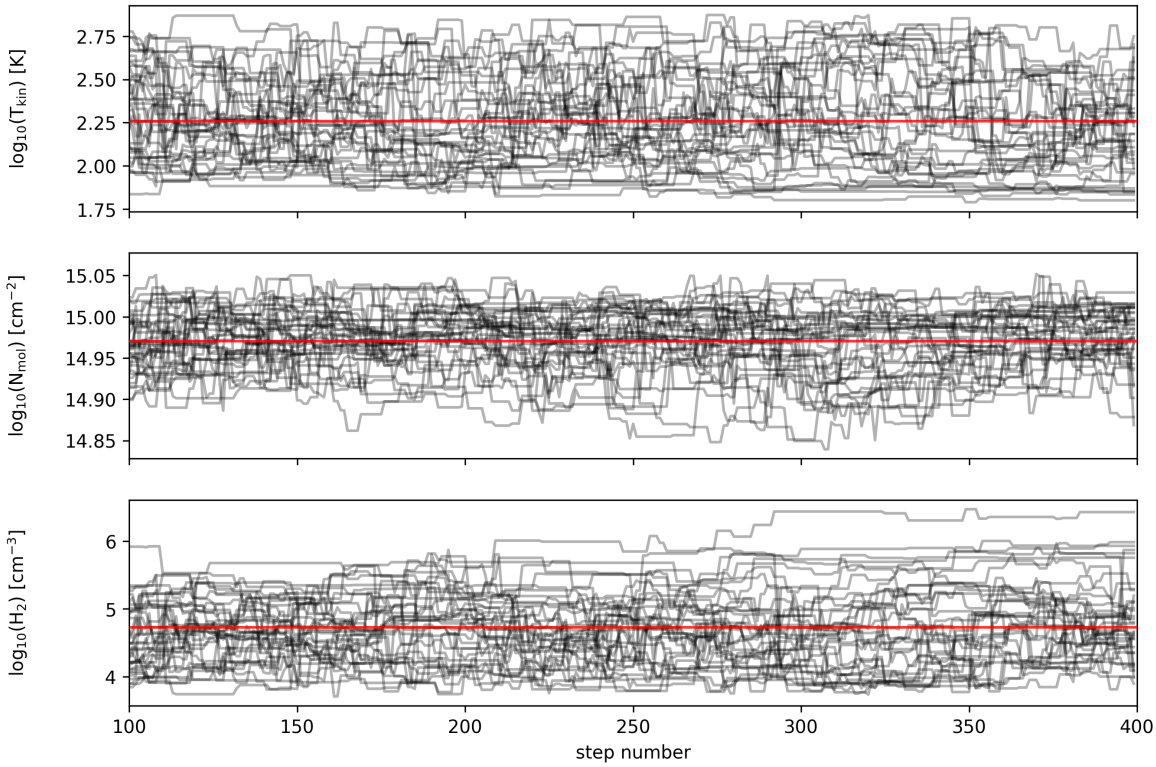
A.2.3 parameters.txt

```
Data file used: /home/mooren/BT/reverseRadex/new_test.dat
Line (frequencies) used: 230.538, 345.7959899, 461.0407682, 1267.014486
tbg = 2.73
fmin = 230.30746200000002
fmax = 1268.2827687687688
linewidth = 1.0
geometry = 1
molfile = /home/mooren/BT/moldata/co.dat
h = 0.0
e- = 0.0
p-h2 = 0.0
o-h2 = 0.0
h+ = 0.0
he = 0.0
tkin's parameter boundaries: (1.3979400086720377, 2.8750612633917)
cdmol's parameter boundaries: (10.0, 21.69897000433602)
h2's parameter boundaries: (3.0, 8.0)

Percental:   50%   |   16%   |   84%
tkin    : 2.21368 | -0.25136 | +0.38304
cdmol    : 14.97493 | -0.04701 | +0.03271
h2    : 4.71784 | -0.62480 | +0.60441
```

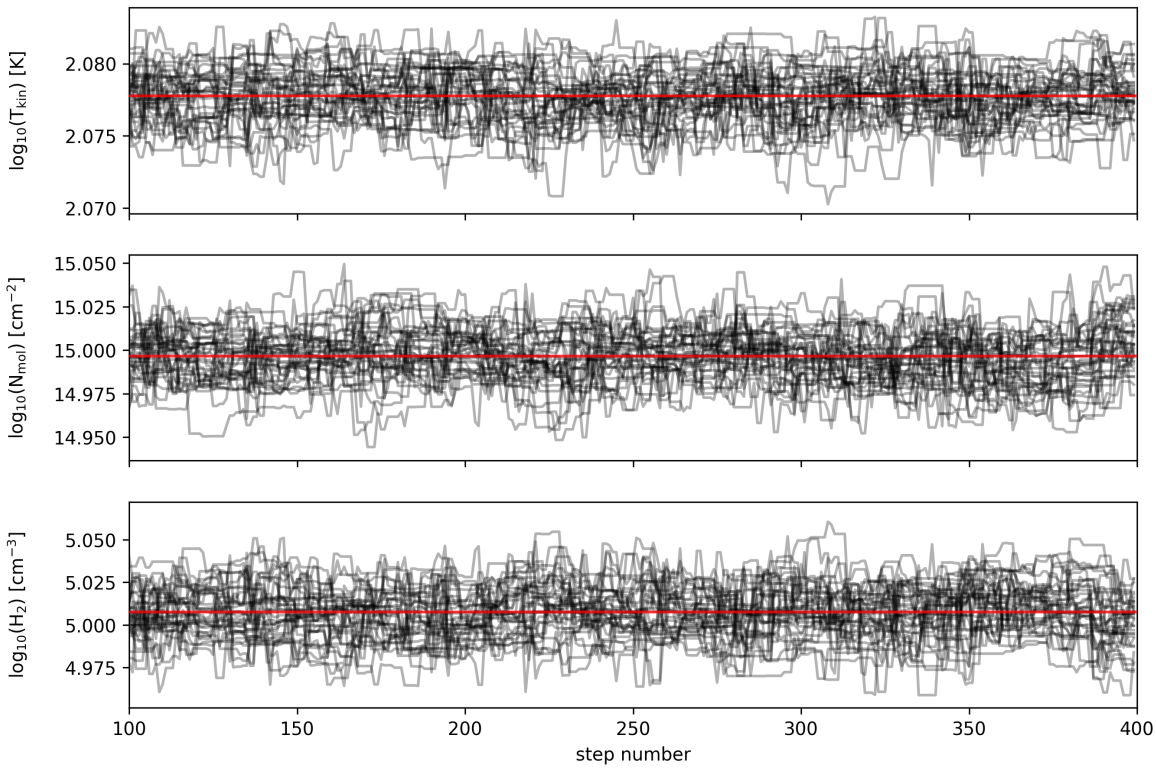
Figure 13: The "parameters.txt" file containing a summary of the input parameters and resulting parameter and uncertainty estimates of the fitted parameters. The parameter median (50%) is given as well as the upper (84%) and lower (16%) uncertainties constraining the values encapsulated within to 1σ.

A.2.4 Trace plots

The trace plots for run 1 and run 4, Table 4, are graphed here to show how more data will benefit the convergence of the parameter and uncertainty estimation process conducted by the MCMC algorithm. A much tighter trace plot is obtained using more data, Figure 14b, as expected for the distribution can be constrained more, and as a consequence is devoid of outlier walkers that often reject parameter updates. This "tightness" translates similarly to the corner plots, Figure 8. For 14b the trace plots appear stationary with no real trend observed, but for 14a there are a few walkers, showing an unwanted trend, near the bottom and top for $\log_{10}(T_{kin})$ and $\log_{10}(n_{H_2})$ respectively, that remain near constant and thus often reject parameter updates.



(a) The trace plot after burn-in of the MCMC algorithm for the input shown in Figure 4 (4 CO lines).



(b) The trace plot after burn-in of the MCMC algorithm for the input shown in Figure 4 but with all available 40 CO lines.

Figure 14: All the input parameters for both subplots is the same, Figure 3, with the exception being that 14a used "observed input" Figure 2, 4 CO lines, and 14b used all available 40 CO lines. The red line indicates the mean of the trace for the respective parameters. The trace plots show the steps after burn-in.

B Wrapper comparison code

The wrapper comparison code is used to conduct the timings seen in Table 2 and the numeric comparison, 3c, that ensures the input and output for all wrappers is as similar as possible: a numpy array in the same shape and containing the same contents.
All comments that look like "#%%" are indicative of code *cells*, like those in Jupyter notebooks, but specifically for the Virtual Studio Code IDE[19], allowing you to run cells straight from .py files.

```python
#!/usr/bin/env python3
#%%
from numpy import (
    concatenate,
    loadtxt,
    savetxt,
    array,
    ones
)


from time import perf_counter
def timeit(function):
    """a timer function to be used as a decorator that runs 1000
    iterations of the function that is decorated and prints the average
    time per run.

    Args:
        function (function): function to time.


    Returns:
        innder: function to be passed through and execute as normal.
    """
    def timeit_inner(*args, **kwargs):
        """function doing the actual timing.


        Returns:
            function: function to be passed through and execute as normal.
        """
        time = []
        runs = 1000
        for i in range(runs):
            tic   = perf_counter()
            fnc   = function(*args, **kwargs)
            toc   = perf_counter()
            time += [toc - tic]

        runtime = sum(time)/len(time)
        print(f"\nThis code took ~{runtime:0.5f} seconds per run " +
              f"for a total of {runs} runs.\n")

        return fnc

    return timeit_inner
```

[19] See for further information: visualstudio.com/jupyter-support-py#_jupyter-code-cells.

```
47
48
49 import cProfile, pstats, io
50 # this import appears to be python >=3.7
51 from pstats import SortKey
52
53
54 # This is not a function to be used as a benchmark tool, only profiling.
55 def profile(fnc):
56     """https://youtu.be/8qEnExGLZfY
57
58     Args:
59         fnc (function): function to profile
60
61
62     Returns:
63         function: function to be passed through and execute as normal.
64     """
65     def profile_inner(*args, **kwargs):
66         """https://docs.python.org/3/library/profile.html#profile.Profile
67
68
69         Returns:
70             function: function to be passed through and execute as normal.
71         """
72         pr = cProfile.Profile()
73         pr.enable()
74         retval = fnc(*args, **kwargs)
75         pr.disable()
76         s = io.StringIO()
77         sortby = SortKey.CUMULATIVE
78         ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
79         ps.print_stats()
80         print(s.getvalue())
81
82         return retval
83
84     return profile_inner
85
86
87 #%%
88 # parameters
89 tkin        = 50.0
90 cdmol       = 1e14
91 h2          = 1e4
92 h           = 0
93 e           = 0
94 ph2         = 0
95 oh2         = 0
96 hplus       = 0
97 he          = 0
98 molfile     = '/home/mooren/BT/moldata/catom.dat'
99 tbg         = 2.73
100 linewidth   = 1.0
101 # geometry    = (unifrom sphere) differs for every wrapper
102 # some manual settings required for ndRadex (which lines) and for
103 # pythonRadex (the collision densities).
```

```
104
105
106 #%%
107 # normal Radex
108 from os import system
109
110
111 # @profile
112 @timeit
113 def radex_call():
114     radex_in = [molfile.split('/')[-1], 'comparison.out', '0 0', tkin,1,
115                 'h2', h2, tbg, cdmol, linewidth, 0]
116
117     with open('comparison.in', 'w') as infile:
118         for parameter_input in radex_in:
119             infile.write(f"{parameter_input}\n")
120
121     system('radex_sphere < comparison.in')
122
123     radex_output = loadtxt('comparison.out', skiprows=13,
124                            usecols=[3,4,6,7,8,9,10,11,12])
125
126     return radex_output.T
127
128 radex_output = radex_call()
129 # print(radex_output)
130 (E_up_radex, freq_radex, T_ex_radex, tau_radex, T_R_radex, n_u_radex,
131  n_l_radex, FLUX_Kkms_radex, FLUX_ergcm2s_radex) = radex_output
132
133
134 #%%
135 # pyRadex
136 import pyradex
137
138
139 # @profile
140 @timeit
141 def pyradex_call():
142     pyradex_run = pyradex.Radex(
143         temperature=tkin, column=cdmol,
144         collider_densities={'H2':h2, 'H':h, 'e':e, 'pH2':ph2, 'oH2':oh2,
145                             'H+':hplus, 'He':he,},
146         species=molfile[:-4], tbackground=tbg, deltav=linewidth)
147     pyradex_output = pyradex_run(escapeProbGeom='sphere')
148
149     # pretty print.
150     # print(pyradex_output.to_pandas())
151     pyradex_output_cut = pyradex_output.to_pandas().to_numpy().T
152
153     return pyradex_output_cut
154
155 pyradex_output_cut = pyradex_call()
156 (T_ex_pyradex, tau_pyradex, freq_pyradex, E_up_pyradex, _, _, n_u_pyradex,
157  n_l_pyradex, brightness_pyradex, T_R_pyradex) = pyradex_output_cut
158 # print(pyradex_output_cut)
159
160
```

```
#%%
# ndRadex
from ndradex import run as ndradex_run


# @profile
@timeit
def ndradex_call():
    # ndRadex.run only takes/prints the radex output for the highest frequency
    # (as per the documentation) thus you need to specify every line (and run
    # the model again for every line) to get the full radex output.
    # nd_radex_lines = ['1-0', '2-1', '3-2', '4-3', '5-4', '6-5', '7-6', '8-7',
    #                   '9-8', '10-9', '11-10', '12-11', '13-12', '14-13',
    #                   '15-14', '16-15', '17-16', '18-17', '19-18', '20-19',
    #                   '21-20', '22-21', '23-22', '24-23', '25-24', '26-25',
    #                   '27-26', '28-27', '29-28', '30-29', '31-30', '32-31',
    #                   '33-32', '34-33', '35-34', '36-35', '37-36', '38-37',
    #                   '39-38', '40-39']  # co.dat
    nd_radex_lines = ['1-0', '2-1', '2-0']  # catom.dat/oatom.dat

    nd_radex_output = ndradex_run(molfile, QN_ul=nd_radex_lines, T_kin=tkin,
                                  N_mol=cdmol, n_H2=h2, n_H=None, n_e=None,
                                  n_oH2=None, n_pH2=None, n_Hp=None, n_He=None,
                                  T_bg=tbg, dv=linewidth, geom='uni',
                                  progress=False, n_procs=3)

    # pretty print.
    # print(nd_radex_output.to_dataframe())
    nd_radex_output_cut = nd_radex_output.to_dataframe().to_numpy().T[6:]

    return nd_radex_output_cut

nd_radex_output_cut = ndradex_call()
(E_up_ndradex, freq_ndradex, _, T_ex_ndradex, tau_ndradex, T_R_ndradex,
 n_u_ndradex, n_l_ndradex, I_ndradex, F_ndradex, _) = nd_radex_output_cut
# print(nd_radex_output_cut)


#%%
# spectralRadex
from spectralradex.radex import run as spectral_radex_run


# @profile
@timeit
def spectral_call():
    spectral_radex_parameters = {}
    spectral_radex_parameters['tkin']       = tkin
    spectral_radex_parameters['cdmol']      = cdmol
    spectral_radex_parameters['h2']         = h2
    spectral_radex_parameters['h']          = h
    spectral_radex_parameters['e-']         = e
    spectral_radex_parameters['p-h2']       = ph2
    spectral_radex_parameters['o-h2']       = oh2
    spectral_radex_parameters['h+']         = hplus
    spectral_radex_parameters['he']         = he
    spectral_radex_parameters['molfile']    = molfile
```

```
    spectral_radex_parameters['tbg']        = tbg
    spectral_radex_parameters['linewidth']  = linewidth
    spectral_radex_parameters['geometry']   = 1


    spectralRadex_output = spectral_radex_run(spectral_radex_parameters)

    # pretty print.
    # print(spectralRadex_output)
    spectralRadex_output_cut = spectralRadex_output.to_numpy().T

    return spectralRadex_output_cut

spectralRadex_output_cut = spectral_call()
(E_up_spectral, freq_spectral, _, T_ex_spectral,
 tau_spectral,T_R_spectral, n_u_spectral, n_l_spectral,
 FLUX_Kkms_spectral, FLUX_ergcm2s_spectral, _, _) = spectralRadex_output_cut
# print(spectralRadex_output_cut)


#%%
#pythonRadex
from pythonradex import nebula, helpers
from scipy import constants


# @profile
@timeit
def pythonradex_call():
    ext_background = helpers.generate_CMB_background(0)

    #FIXME cannot use all densities propely due to KeyErrors. It seems
    # that only 'ortho-H2' and 'para-H2' are allowed, even thought the
    # documentation appears to mention all 7 collision partners should
    # work?
    # coldens = {
    #         'H2':h2/constants.centi**3, 'h':h/constants.centi**3,
    #         'e':e/constants.centi**3, 'para-H2':ph2/constants.centi**3,
    #         'ortho-H2':oh2/constants.centi**3, 'H+':hplus/constants.centi**3,
    #         'He':he/constants.centi**3
    #     }
    # manually get these from "comparison.out" radex output.
    coldens = {'para-H2':7.711/constants.centi**3,
               'ortho-H2':2.289/constants.centi**3}
    dv = linewidth*constants.kilo
    cd = cdmol/constants.centi**2

    python_radex_run = nebula.Nebula(
        data_filepath=molfile, geometry='uniform sphere RADEX',
        ext_background=ext_background, Tkin=tkin, line_profile='square',
        coll_partner_densities=coldens, Ntot=cd, width_v=dv)
    python_radex_run.solve_radiative_transfer()

    # capture the output from the printed results, since direct access
    # is not clear.
    python_radex_run.print_results()
    pyradexTex    = python_radex_run.Tex
```

```
    pyradexlevels = python_radex_run.level_pop
    pyradextau    = python_radex_run.tau_nu0
    python_radex_run.compute_line_fluxes(4*3.14)
    pyradexflux   = array(python_radex_run.obs_line_fluxes) * 1000

    return

pythonradex_call()


# %%
# # if a radex output column is missing from a wrapper's output, said output
# will be equated to "empty".
empty = ones(radex_output.shape[1])
def wrap_rad_difference(wrapper_array, radex_array):
    """calculate the percentage difference of specific wrapper output
    compared to native radex output.

    Args:
        wrapper_array (numpy.array): contains the output columns (every
        single spectral line) and specific output (E_up, freq, etc.) of
        a wrapper as a 2D array.
        radex_array (numpy.array): contains the output columns (every
        single spectral line) and specific output (E_up, freq, etc.) of
        a wrapper as a 2D array.


    Returns:
        float: the difference expressed as a percentage.
    """

    if wrapper_array.all() != 1:
        diff = (wrapper_array - radex_array) / radex_array
        return  diff.astype('float64') * 100
    else:
        # just an indicator in the table to show that the wrapper had
        # no output for this column.
        return empty


pyradex_table = array([
    E_up_pyradex, freq_pyradex, T_ex_pyradex, tau_pyradex,
    T_R_pyradex, n_u_pyradex, n_l_pyradex, empty, empty
])

ndRadex_table = array([
    E_up_ndradex, freq_ndradex, T_ex_ndradex, tau_ndradex,
    T_R_ndradex, n_u_ndradex, n_l_ndradex, I_ndradex, F_ndradex
])

spectral_table = array([
    E_up_spectral, freq_spectral, T_ex_spectral,
    tau_spectral, T_R_spectral, n_u_spectral, n_l_spectral,
    FLUX_Kkms_spectral, FLUX_ergcm2s_spectral
])

# python_radex_table = ['pythonRadex', ]
```

```
332
333 wrappers = [pyradex_table, ndRadex_table, spectral_table]#,
334 #              # python_radex_table]
335
336 # hacky way of getting LaTeX table output written to file.
337 with open('intermediate_comparison_table.txt', 'w') as table:
338     for wrapper in wrappers:
339         difference = array([wrap_rad_difference(wrap, rad)
340                             for wrap, rad
341                             in zip(wrapper, radex_output)]).T
342         savetxt(table, difference, fmt='%.2e', delimiter='\t')
343
344 temp = loadtxt('intermediate_comparison_table.txt')
345
346 intermediate = array([array([f'{temp_value:.2e}', '&'])
347                       for temp_value
348                       in concatenate((radex_output.T.flatten(),
349                                       temp.flatten()))]).flatten()
350
351 final_table = intermediate.reshape(temp.shape[0] + radex_output.shape[1],
352                                    (temp.shape[1] + radex_output.shape[0]))
353
354 savetxt('comparison_table.txt', final_table, fmt='%s')
355
356
357 # %%
```

C ReverseRADEX (main program code)

The file tree fore the ReverseRADEX program is seen in Figure 15 and in this appendix the sub-sections; user_input, Appendix C.2, fitting, Appendix C.3, and save_plot, Appendix C.4 are listed in chronological run time order.

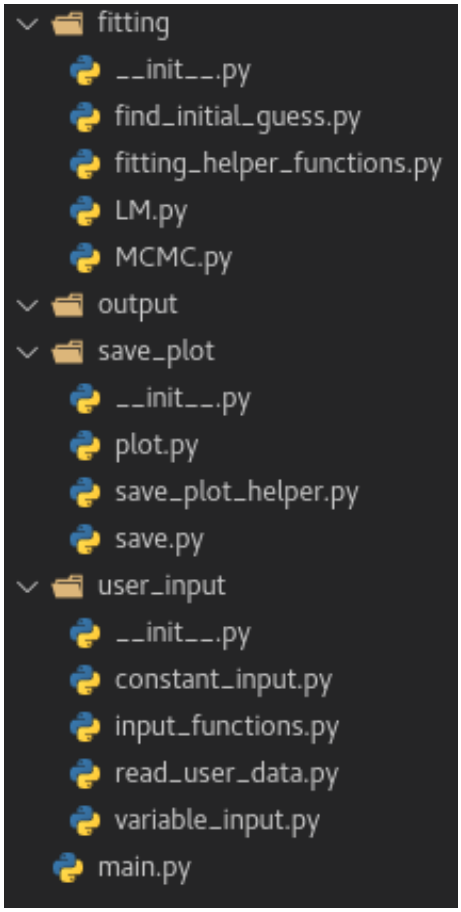


Figure 15: The file tree for ReverseRADEX.

C.1 main

```
#!/usr/bin/env python3
#%%
# relative imports.
from user_input import (
    ConstantParamaters,
    VariableParamters,
    DataRetrieval,
    yay_or_nay
)
from fitting import (
    AlgorithmHelpers,
```

```
    find_initial_parameter_guesses,
    run_levenberg_marquardt,
    run_monte_carlo,
)
from save_plot import Plotting, SaveResults

# module imports.
from numpy import (
    append,
    array,
    log10,
    full,
)
from datetime import timedelta, datetime
from pathlib import Path
from time import time
from os import getcwd


input_constant = ConstantParamaters()
input_variable = VariableParamters()
data_retrieval = DataRetrieval()


#%%
#### Catch user input from terminal ####
user_molfile = input_constant.molfile_input()
user_datfile = input_constant.datafile_input()
# user_molfile  = '/home/mooren/BT/moldata/co.dat'
# user_datfile  = '/home/mooren/BT/reverseRadex/new_test.dat'

# (matching) frequencies with molfile.
freq_indices  = data_retrieval.get_molfile_frequency_index(user_datfile,
                                                           user_molfile)
freq          = data_retrieval.get_frequencies(freq_indices, user_molfile)
user_mol_frequencies, freq_min, freq_max, number_of_lines_total = freq
freq_range    = (freq_min, freq_max)

# checking for units and uncertainties.
units             = data_retrieval.get_user_units(user_datfile)
uncertainties     = data_retrieval.uncertainties_included(user_datfile)
(y_observed,
 y_uncertainties) = data_retrieval.line_strengths(user_datfile,
                                                  uncertainties)

# constant parameters.
Tbg             = input_constant.background_radiation_input()
dv              = input_constant.line_width_input()
geom, geom_name = input_constant.geometry_input()

# variable parameters.
temp_kin = input_variable.kinetic_temperature_input()
coldens  = input_variable.column_density_input()
voldens  = input_variable.collision_densities_input()
#### Catch user input from terminal ####


```

```
#%%
#### If you would like to set the input manually, uncomment the cell ####


# user_molfile  = '/home/mooren/BT/moldata/co.dat'
# user_datfile  = '/home/mooren/BT/reverseRadex/new_test.dat'
# freq_indices  = data_retrieval.get_molfile_frequency_index(user_datfile,
#                                                            user_molfile)
# freq          = data_retrieval.get_frequencies(freq_indices, user_molfile)
# user_mol_frequencies, freq_min, freq_max, number_of_lines_total = freq
# freq_range    = (freq_min, freq_max)


# units             = data_retrieval.get_user_units(user_datfile)
# uncertainties     = data_retrieval.uncertainties_included(user_datfile)
# (y_observed,
#  y_uncertainties) = data_retrieval.line_strengths(user_datfile,
#                                                   uncertainties)

# # variable parameters.
# # [name parameter, init guess, (bound_low, bound_upp), fit parameter?]
# # 0.1 < tkin < 1e4 [K]
# temp_kin = ['tkin', 131, (10.0, 500.0), True]
# # 1e5 < cdmol 1e25 [cm^-2]
# coldens  = ['cdmol', 3e16, (1e10, 5e21), True]
# # coll partner:(init guess, fit parameter?)
# # 1e-3 < coll partner < 1e13 [cm^-3]
# voldens  = {'h2':(3e4, True), 'h':(0.0, False), 'e-':(0.0, False),
#             'p-h2':(0, False), 'o-h2':(0.0, False), 'h+':(0.0, False),
#             'he':(0.0, False), 'min_max':(5e3, 5e8)}

# # constant parameters.
# Tbg   = 2.73  # K
# dv    = 1.0   # km s^-1
# geom = 1     #(1=sphere, 2=LVG, 3=slab)
# # just for displaying purposes,
# geom_name = 'uniform sphere'


#### If you would like to set the input manually, uncomment the cell ####

##### No user input required beyond this point #####
##### Unless you want to tweak the algorithms  #####


#%%
# printing the chosen settings for user to check.
print(f"\n\nSelected molfile path              : '{user_molfile}'")
print(f"Selected datafile path             : '{user_datfile}'")
print(f"Selected line strength units       : {units}")
print(f"uncertainties included             : {uncertainties}")
print("\n\n[name of parameter, parameter value, (lower bound, upper" +
      " bound), fit parameter?]")
print('If a parameter is fit, "parameter value" is a dummy number and ' +
      'can be ignored.\nIf not fit, the boundaries are dummy numbers.\n' +
      "0.0 just indicates SpectralRadex to not use this collision " +
      "partner.\n")
```

```
126 print(f"Selected minimum and maximum \n" +
127       f"kinetic gas cloud temperature      : {temp_kin} K")
128 print(f"Selected background radiation field: {Tbg} K")
129 print(f"Selected minimum and maximum \n" +
130       f"column densities                   : {coldens} cm^-2")
131
132
133 constant_parameters = {
134     'tbg':Tbg, 'fmin':freq_min, 'fmax':freq_max, 'linewidth':dv,
135     'geometry':geom, 'molfile':user_molfile
136 }
137
138 # handeling the kinetic temperatrue and column density.
139 lim_low = array([])
140 lim_upp = array([])
141 tkin_cd = [temp_kin, coldens]
142 lims_to_save = {}
143 for parameter in tkin_cd:
144     prm_name, prm_value, prm_bounds, prm_fit = parameter
145     if prm_fit == True:
146         prm_low, prm_upp = prm_bounds
147         lim_low = append(lim_low, log10(prm_low))
148         lim_upp = append(lim_upp, log10(prm_upp))
149         lims_to_save[prm_name] = (lim_low[-1], lim_upp[-1])
150     else:
151         constant_parameters[prm_name] = prm_value
152
153 # handeling the collision partners (volume densities).
154 str_voldens = "Selected volume densities [cm^-3],  "
155 print(str_voldens)
156 vol_dens_summary = []
157 min_max = 'min_max'
158 for collision_partner in voldens:
159     if collision_partner != min_max:
160         blank = ''.ljust(len(str_voldens) -
161                          len(collision_partner) - 1) + ":"
162         param_value, param_fit = voldens[collision_partner]
163         voldens_min, voldens_max = voldens[min_max]
164         vol_dens_summary += [[collision_partner, param_value,
165                               (voldens_min, voldens_max), param_fit]]
166         print(f"{collision_partner}" + blank +
167               f" {vol_dens_summary[-1]}")
168         if param_fit == False:
169             constant_parameters[collision_partner] = param_value
170         else:
171             lim_low = append(lim_low, log10(voldens_min))
172             lim_upp = append(lim_upp, log10(voldens_max))
173             lims_to_save[collision_partner] = (lim_low[-1], lim_upp[-1])
174
175 # printing remaining input.
176 print(f"Selected line width                : {dv} km/s")
177 print(f"Selected minimum and maximum \n" +
178       f"frequency                          : {freq_range} GHz")
179 print(f"Selected geometry                  : {geom_name}")
180
181
182 # check if any of the parameters is set to be fit.
```

```
if lim_low.shape[0] == 0:
    raise AssertionError("No parameter is set to be fit.")

# check if more data is available than parameters to fit.
# getting the names of the parameters to be fit, the order is important.
all_parameter_names = ['molfile', 'tkin', 'cdmol', 'tbg', 'h2', 'p-h2',
                       'o-h2', 'e-', 'h', 'he', 'h+', 'fmin', 'fmax',
                       'linewidth', 'geometry']
fit_parameters_names = []
for parameter_name in all_parameter_names:
    if parameter_name not in constant_parameters.keys():
        fit_parameters_names += [parameter_name]

len_data     = len(y_observed)
len_fit_prms = len(fit_parameters_names)
if (len_data > len_fit_prms) != True:
    raise AssertionError(f"{len_data} observed data points is not "
                         f"enough data to fit {len_fit_prms} parameters" +
                         ". Need: 'data > parameters + 1'.")

# prompt user to either continue to the fitting process or terminate.
user_prms_check = yay_or_nay("\nContinue to the fitting process? (y/n) ")
if user_prms_check == '' or user_prms_check == True:
    pass
else:
    raise KeyboardInterrupt("User terminated the program.")



#%%
# since the frequency range is used to limit the radex output, the
# indices have to be shifted to accommodate for that (for instance, the
# range might start at 300 GHz while lines exist < 300 GHz. Therefore, the
# index needs to be shifted).
matching_index = list(map(lambda add: add - freq_indices[0], freq_indices))

# create an index array to be used for cutting the SpectralRadex output
# to match the spectral lines present in the user supplied data file.
matching_lines = full(number_of_lines_total, False)
matching_lines[matching_index] = True


#%%
### start of main program ###
start_time = time()

#### global parameter search ####
print("\nEstimating initial parameters.")
# use the brute method global search to find initial estimates for
# paremeters to be fit.
cst_prms = [user_molfile, Tbg, dv, freq_min, freq_max, geom, units,
            matching_index, user_datfile, uncertainties]
global_parameter_estimates = find_initial_parameter_guesses(
    temp_kin, coldens, voldens, vol_dens_summary, cst_prms
)

grid_time = time()
```

```
grid_duration  = grid_time - start_time
grid_duration_HHMMSS = str(timedelta(seconds=grid_duration)).rpartition('.')[0]
print(f"Time elapsed: {grid_duration_HHMMSS}")
print("Global parameter estimates resulting from brute " +
      "(grid search) method:")
for name, value in zip(fit_parameters_names, global_parameter_estimates):
    print(f"log10({name}): {value:.5f}")


#%%
#### setting up Levenberg-Marquardt and MCMC parameters ####
alg_help = AlgorithmHelpers(
    y_observed,
    y_uncertainties,
    units,
    lim_low,
    lim_upp,
    constant_parameters,
    matching_lines,
    fit_parameters_names
)


#%%
#### Levenberg-Marquardt least squares to refine parameter estimates ####
print("\nRefining parameter estimates.")
initial_parameters = run_levenberg_marquardt(global_parameter_estimates,
                                             alg_help.RADEX_model,
                                             y_observed,
                                             y_uncertainties)

LM_time = time()
LM_duration  = LM_time - start_time
LM_duration_HHMMSS = str(timedelta(seconds=LM_duration)).rpartition('.')[0]
print(f"Time elapsed: {LM_duration_HHMMSS}")
print("Refined parameter estimates resulting from Levenberg-Marquardt:")
for name, value in zip(fit_parameters_names, initial_parameters):
    print(f"log10({name}): {value:.5f}")


#%%
#### MCMC for uncertainty estimates ####
N = 500  # number of steps the MCMC algorithm takes.
print("\nRunning MCMC for uncertainty estimates,")
MCMC_output, ndim = run_monte_carlo(initial_parameters,
                                    alg_help.log_probability,
                                    number_of_steps=N)


#%%
### end of main program ###
end_time = time()
duration = end_time - start_time
duration_HH_MM_SS = str(timedelta(seconds=duration)).rpartition('.')[0]
print(f"\nRun time of main program: {duration_HH_MM_SS}.")


```

```
#%%
##### plotting and saving of results #####
date_time = datetime.now().strftime("%Y.%m.%d-%H.%M.%S")
#FIXME add which molecule is used?
# create output directory.
cwd         = getcwd()
output_path = cwd + f'/output/{date_time}'
Path(output_path).mkdir()


### saving ###
saving = SaveResults(
    MCMC_output,
    output_path,
    constant_parameters,
    fit_parameters_names
)

# saving MCMC ensamble.
saving.save_MCMC_sampler()

# saving RADEX.csv output and obtaining parameter medians.
prms_50s = saving.RADEX_for_optimal_parameters(
    user_datfile, user_mol_frequencies, y_observed, y_uncertainties,
    freq_indices, units, lims_to_save
)
### saving ###


### plotting ###
plot = Plotting(
    MCMC_output,
    output_path,
    prms_50s,
    fit_parameters_names
)

# Plotting and saving the corner plot.
plot.plot_corner()

# Plotting the molecular spectrum.
plot.plot_spectrum(
    units, y_observed, y_uncertainties, constant_parameters,
    user_mol_frequencies
)
### plotting ###


print(f"\nResults saved to {output_path}.\n")


# %%
```

C.2 user_input

Code used for capturing user input and translating it to input that the subsequent parts of the program expect.

C.2.1 __init__.py

```
from .fitting_helper_functions import *
from .find_initial_guess import *
from .MCMC import *
from .LM import *
```

C.2.2 input_functions.py

```
#!/usr/bin/env python3


def re_enter_wrapper(fnc):
    """a wrapper that recalls the function if input is invalid or
    unsatisfactory.

    Args:
        fnc (function): the function to be wrapped and checked for
        validity of input and recalled if necessary.


    Returns:
        funcion: the wrapper function
    """

    def re_enter_or_return(*args, **kwargs):
        """function that creates a while loop of function calls to
        the function to be wrapped, until the returned value is
        satisfactory.


        Returns:
            function: recalls the function.
        """

        while True:
            function_return = fnc(*args, **kwargs)
            if function_return == 'dummy':
                continue
            else:
                return function_return

        return

    return re_enter_or_return


```

```
@re_enter_wrapper
def numeric_input(query):
    """function that is called to either return user input or standard
    parameter if input is omitted. If user input is called,
    the input is checked to be either int or float.

    Args:
        query (str): prompt for user input.

    Raises:
        TypeError: Input is not an integer or float.


    Returns:
        str, float: empty string (read by python as None) prompt or
        user input as a float.
    """

    entry = input(query)
    if entry == '':
        return entry

    try:
        # to allow for scientific/exponential notation.
        if float(entry):
            return float(entry)
    except ValueError:
        pass

    # if input is not '' (empty) or in scientific notation, check if
    # input is numeric (float/int)
    entry_check = entry.replace(' ', '').strip()
    if_conditions =(
        entry_check.isdigit() or
        (entry_check.replace('.', '', 1).isdigit() and
         entry_check.count('.') < 2)
    )
    if not if_conditions:
        print("Input is not an integer or float, or is " +
              f"negative. User input was '{entry}'")
        return 'dummy'

    #TODOXME what should be the case for entry == 0? Currently it
    # seems to ignore the entry (read by python as "False"?)
    # especially important for Tbg and adding a tabulated radiation
    # background.
    # if int(entry) == 0 and float(entry) == 0.0:
    #     raise ValueError(f"Input must be positive. User input was {entry}")

    return float(entry)


def min_max_check(parameter_min, parameter_max, parameter_name):
    """checks if minimum of parameter is less than maximum
    of parameter.

    Args:
```

```
96          prms_min (float): lower bound of parameter.
97
98          prms_max (float): upper bound of parameter.
99
100         parameter_name (str): name of parameter to check.
101
102
103     Raises:
104         ValueError: minimum is greater than maximum.
105     """
106
107     if parameter_min > parameter_max:
108         print(f"The minimum {parameter_name} selected" +
109               f" '{parameter_min}' is greater than" +
110               f" the maximum '{parameter_max}'.")
111         return 'dummy'
112
113     return
114
115
116 def in_between_check(parameter_min, parameter_max,
117                      parameter_value, parameter_name):
118     """check if the parameter value is within the parameter limits.
119
120     Args:
121         prms_min (float): lower bound of parameter.
122
123         prms_max (float): upper bound of parameter.
124
125         parameter_value (float): value of parameter to be checked.
126
127         parameter_name (str): name of parameter to check.
128
129
130     Raises:
131         ValueError: parameter value is not within the boundary limits.
132     """
133
134     #FIXME make it clear when it talks about user defined limits and when
135     # it discusses radex limits?
136     if not (parameter_min < parameter_value < parameter_max):
137         print(f"{parameter_name} = {parameter_value}, " +
138               "is not within the limits of " +
139               f"[{parameter_min}; {parameter_max}].")
140         return 'dummy'
141
142     return
143
144
145 @re_enter_wrapper
146 def yay_or_nay(query):
147     """check if user wants to fit (a specific) parameter(s).
148
149     Args:
150         query (str): prompt for user input.
151
152     Raises:
```

```
153         ValueError: raises error if the fit is not declared properly.
154 
155 
156     Returns:
157         bool: True (fit parameter) False (do not fit parameter).
158     """
159 
160     entry = input(query)
161 
162     if entry == '':
163         return entry
164 
165     entry = entry.replace(' ', '')
166 
167     yes = ['y', 'yay', 'yes']
168     no  = ['n', 'nay', 'no']
169     while True:
170         if entry in yes:
171             return True
172         elif entry in no:
173             return False
174         else:
175             print(
176                 f"\n'{entry}' is not a valid input. Try entering " +
177                 f"one of either {yes} if you want to fit the " +
178                 f"parameter(s) or {no} if you do not want to fit " +
179                 "the parameter(s).\nThe default is 'no'"
180             )
181             return 'dummy'
182 
183     return
184 
185 
186 @re_enter_wrapper
187 def collision_check(query, collision_partner_names, no):
188     """check if user entered a valid collision partner name. This only
189     checks all the names that RADEX supports, not necessarily the names
190     of collision partners actually present in the molecular data file.
191 
192     Args:
193         query (str): query to raise user and request input.
194 
195 
196     Returns:
197         str: name of collision partner.
198     """
199 
200     # collision_partner_names = ['h2', 'h', 'e-', 'p-h2', 'o-h2', 'h+', 'he']
201     collision_partner_name = input(query).replace(' ', '').lower()
202     conditions = (
203         collision_partner_name not in collision_partner_names
204         and collision_partner_name not in no
205     )
206     if conditions:
207         print(
208             f"\n'{collision_partner_name}' is not in " +
209             f"'{collision_partner_names}'."
```

```
210         )
211         return 'dummy'
212     else:
213         return collision_partner_name
214 
215     return
```

C.2.3 read_user_data.py

```
1  #!/usr/bin/env python3
2  
3  # module imports
4  from numpy import loadtxt, ones
5  
6  # relative imports
7  from user_input.input_functions import in_between_check
8  
9  
10 class DataRetrieval:
11     #FIXME things that are used multiple times should go into __init__?
12     #def __init__(self):
13     #    return
14 
15 
16     #FIXME just use numpy.loadtxt, array.T for transposing and .astype(float)
17     # to transform the elements of the array from str to float instead of
18     # "transpose_float_convert_list"? runtime is really not an issue here.
19 
20 
21     ### general functions ###
22     def get_file_lines(self, data_file_location):
23         """get the file lines of user supplied data file in a list.
24 
25         Args:
26             data_file_location (str): file location on system of data file.
27 
28 
29         Returns:
30             list: list of file lines of the user supplied data file.
31         """
32 
33         with open(data_file_location, 'r') as data_file:
34             #TODO add a check to see if file is empty,
35             # perhaps if possible also if there are enough data points
36             # for the number of parameters chosen to fit for LM to still
37             # work --> N + 1 data points required (N is number of fit
38             # parameters)
39             data_file_lines = data_file.readlines()
40         return data_file_lines
41 
42 
43     def transpose_float_convert_list(self, input_list, selected_row):
44         """transpose N x M list and convert elements type of a single
45         row to float.
46 
```

```
47          Args:
48              input_list (list): list to be transposed and converted.
49              selected_row (int): the index of the row that should be returned
50
51
52          Returns:
53              list: transposed list with float elements.
54          """
55
56          #FIXME try and use zip(*list)?
57          transposed_list = list(map(lambda *untransposed: list(untransposed),
58                                     *input_list))
59
60          transposed_list = transposed_list[selected_row]
61
62          #FIXME take this out when I also include lineID and just do it
63          # where it is needed outside of this function.
64          transposed_float_list = list(map(float, transposed_list))
65
66          return transposed_float_list
67      ### general functions ###
68
69
70      ### data retrieving functions ###
71      def get_user_units(self, data_file_location):
72          """retrieves the line strength units from the user supplied data file.
73
74          Args:
75              data_file_location (str): file location on system of data file.
76
77          Raises:
78              ValueError: checks if unit selection is read by python
79              properly.
80
81
82          Returns:
83              int: integer describing which line strength units are to be used.
84          """
85
86          units_line = self.get_file_lines(data_file_location)[0]
87          #FIXME  perhaps use units_line.find(1,2,3) or something?
88          # or maybe a for loop and check
89          # "for units in [1,2,3]: for line in lines: if units in line: break?
90          units_retrieved = int(units_line.strip()[-1])
91
92          valid_units = ['T_R (K) => # 1',
93                         'FLUX (K*km/s) => # 2',
94                         'FLUX (erg/cm2/s) => # 3']
95          valid_units_int  = [1,2,3]
96          valid_units_name = ['T_R (K)', 'FLUX (K*km/s)', 'FLUX (erg/cm2/s)']
97          if units_retrieved not in valid_units_int:
98              raise ValueError("Units selected in the header of " +
99                               f'"{data_file_location}" are invalid. ' +
100                              "Please enter one of the following units, " +
101                              f"{valid_units}, in the header by typing " +
102                              '"# int".')
103
```

```
104         return valid_units_name[units_retrieved - 1]
105
106
107     def get_molfile_frequencies(self, molecular_file):
108         """get all the frequencies as floats in a list from the
109         selected molfile.
110
111         Args:
112             molecular_file (str): file location on system of molecular file.
113
114
115         Returns:
116             list: list of frequencies with float type.
117         """
118
119         contents_molfile = self.get_file_lines(molecular_file)
120         #FIXME this only works if all molfiles are structured the same way
121         # and some inconsistencies in LAMDA files definitley ocurred.
122         trans = ('!TRANS', '! TRANS')
123         numbr = ('!NUMBER', '! NUMBER')
124         index_lower = None
125         index_upper = None
126         for molfile_line in contents_molfile:
127             if molfile_line.startswith(trans):
128                 index_lower = contents_molfile.index(molfile_line)
129
130             if molfile_line.startswith(numbr):
131                 index_upper = contents_molfile.index(molfile_line)
132
133             # since we are after the first occurrence of '!TRANS' and
134             # subsequent '!NUMBER' in the molfile, this if statement
135             # should suffice in finding the indices
136             if ((index_lower and index_upper) != None
137                 and index_upper > index_lower):
138                 break
139
140         radiative_transitions = contents_molfile[index_lower + 1:index_upper]
141         radiative_transitions_split = [split_line.split()
142                                        for split_line
143                                        in radiative_transitions]
144
145         # "transposing" the list to select the column (now row) of frequencies.
146         # index 4 indicates the frequency column (now row) in the molfile.
147         molfile_frequencies = self.transpose_float_convert_list(
148             radiative_transitions_split, 4)
149
150         return molfile_frequencies
151
152
153     #TODOFIXME if LAMDA file does not contain frequencies, this will not work
154     # (spectralRadex itself will also not work) change the LAMDA file to
155     # include frequencies in the correct column (get it from energy levels)?
156     def get_molfile_frequency_index(self, data_file_location, molecular_file):
157         """get the index of molecular file frequencies that match user
158         supplied data file frequencies. The indices are to be used for
159         cutting radex output to match user input later on (this is important)
160         for MAGIX to work properly).
```

```
161
162         Args:
163             data_file_location (str): file location on system of user data.
164             molecular_file (str): file location on system of molecular file.
165
166         Raises:
167             EOFError: checks if a specific line in user supplied data file
168                     does not match any line in the selected molecular file.
169
170
171         Returns:
172             list: indices of molfile that match user supplied data.
173         """
174
175         data_lines = self.get_file_lines(data_file_location)[1:]
176
177         data_lines_split = [data_line.split()
178                             for data_line
179                             in data_lines]
180
181         # index 0 selects first row (frequencies) in user supplied data file.
182         user_freqs = self.transpose_float_convert_list(data_lines_split, 0)
183
184         # gets the frequencies (float) of the selected molfile in a list.
185         molfile_frequencies = self.get_molfile_frequencies(molecular_file)
186
187         # bw ("bandwith") to within which a user supplied frequency should
188         # match the molfile frequency.
189         #TODO dynamically change "bw" based on molfile by seeing what is
190         # the closest frequency discrepancy and take half that?
191         # For now just with 0.001, 0.01 or 0.1 or something as max "bw"?
192         bw = 0.001
193         matching_index   = []
194         for user_freq in user_freqs:
195             for iter_index, molfile_freq in enumerate(molfile_frequencies):
196                 if (user_freq * (1-bw) <= molfile_freq <= user_freq / (1-bw)):
197
198                     matching_index += [iter_index]
199                     break
200
201             #FIXME make it so it returns all the lines that do not match.
202             # ALSO be clearer about how a line is matched (using bw)?
203             #FIXME find a better error to return, instead of EOFError?
204             else:
205                 raise EOFError(f"The line corresponding to {user_freq}" +
206                                f" GHz in '{data_file_location}'" +
207                                " does not match any frequency in the " +
208                                f"selected molfile: '{molecular_file}'.")
209
210         return matching_index
211
212
213     def get_frequencies(self, frequency_indices, molecular_file):
214         """get the minimum and maximum frequency [GHz] from the molfile and
215         user supplied data. it is ensured that the minimum and maximum
216         frequency are included in the frequency range by means of a
217         "bw" (bandwidth).
```

```
218
219         Args:
220             frequency_indices (list): indices of matching frequencies.
221             molecular_file (str): file location on system of molecular file.
222
223
224         Returns:
225             tuple: minimum and maximum frequency
226         """
227
228         all_frequencies = self.get_molfile_frequencies(molecular_file)
229         number_of_freqs = len(all_frequencies)
230
231         frequencies_that_match = [all_frequencies[index_match]
232                                   for index_match
233                                   in frequency_indices]
234
235         # ensuring that all frequencies are within the minimum and maximum.
236         #NOTE this might prove problematic when frequencies in LAMDA
237         # file are unordered and certain output will be cut by SpectralRadex
238         # and thus not matched to user data. (see line ~217 of main.py)
239         # works fine for ordered frequency files.
240         bw = 0.001
241         freq_min = min(frequencies_that_match) * (1 - bw)
242         freq_max = max(frequencies_that_match) / (1 - bw)
243
244         # check if frequencies are within RADEX limits.
245         in_between_check(0, 3e7, freq_min, 'minimum frequency read from ' +
246                          'molecular file (expects GHz)')
247
248         in_between_check(0, 3e7, freq_max, 'maximum frequency read from ' +
249                          'molecular file (expects GHz)')
250
251         return frequencies_that_match, freq_min, freq_max, number_of_freqs
252
253
254     def uncertainties_included(self, data_file_location):
255         """determine if there are uncertainties included in the user
256         supplied data file by checking the amount of columns.
257
258         Args:
259             data_file_location (str): file location on system of data file.
260
261         Raises:
262             Exception: if the number of columns is invalid. The excpected
263             number is 2 (frequencies and line strengths) or 3 (frequencies,
264             line strengths and line strength uncertainties).
265
266
267         Returns:
268             str: either 'yes' or 'no' to be used as input for MAGIX to tell
269             it that uncertainties are included or not.
270         """
271
272         #TODO add support for if there are some lines with, and some
273         # lines without uncertainties (for those without, just set it
274         # equal to one, as they are only used in a chi2 calculation?)
```

```
275
276         # [1:] to exclude header.
277         data_lines = self.get_file_lines(data_file_location)[1:]
278
279         number_of_columns = len(data_lines[0].split())
280         if number_of_columns == 3:
281             return 'yes'
282         elif number_of_columns == 2:
283             return 'no'
284         else:
285             raise Exception(f"The number of columns = {number_of_columns}" +
286                             f" in {data_file_location} is invalid. " +
287                             "The excpected number is 2 (frequencies and " +
288                             "line strengths) or 3 (frequencies, line " +
289                             "strengths and line strength uncertainties).")
290
291         return
292
293
294     def line_strengths(self, user_data_file, uncertainty):
295         """extract the line strength column (with uncertainties) from the
296         user supplied data file. These uncertainties are only used for
297         calculating the chi^2 values so the default uncertainties = 1 (or
298         any other constant) since they have no effect then.
299
300         Args:
301             user_data_file (str): user supplied data file directory.
302
303             uncertainty (str): uncertainties included ('yes' -OR- 'no')
304
305
306         Returns:
307             tuple: 1 numpy array with line strenghts and 1 numpy array
308             with line strength uncertainties.
309         """
310
311         data = loadtxt(user_data_file).T
312         if uncertainty == 'no':
313             line_strenghts = data[1]
314             return (line_strenghts, ones(line_strenghts.shape[0]))
315         else:
316             line_strenghts, line_strenght_uncertanties = data[1:]
317             return (line_strenghts, line_strenght_uncertanties)
318
319         return
320
321
322
323     ### data retreiving functions ###
```

C.2.4 constant_input.py

```
1 #!/usr/bin/env python3
2 #%%
3 from user_input.input_functions import (
```

```
    numeric_input,
    in_between_check,
    re_enter_wrapper
)


class ConstantParamaters:
    #def __init__(self):
    #    return


    def molfile_input(self):
        """function to ask for molecular file path.

        Raises:
            FileNotFoundError: User did not supply a molecular file.


        Returns:
            str: string of name refering to the molecular file used.
        """

        user_molfile = input(
            "Enter molecular file path '*.dat': ").replace(' ', '')
        if user_molfile == '':
            raise FileNotFoundError("User did not supply a " +
                                    "molecular file")

        extension = '.dat'
        if not user_molfile.endswith(extension):
            user_molfile += extension

        return user_molfile


    def datafile_input(self):
        """function to ask for data file path.

        Raises:
            FileNotFoundError: User did not supply a data file.


        Returns:
            str: string of name refering to the molecular file used.
        """

        user_datafile = input(
            "Enter data file path '*.dat': ").replace(' ', '')
        if user_datafile == '':
            raise FileNotFoundError("User did not supply a " +
                                    "data file path")

        extension = '.dat'
        if not user_datafile.endswith(extension):
            user_datafile += extension

        return user_datafile
```

```
61
62
63      @re_enter_wrapper
64      def background_radiation_input(self):
65          """function to set background radiation field based on user input,
66          or return a default value.
67
68
69          Returns:
70              float: either user input or standard parameter.
71          """
72          #TODO/FIXME how is the option of a user supplied radiation field
73          # handeled by (spectral) radex, since that is what needs to be
74          # the input then instead of background temperature?
75
76          temp_background = (numeric_input(
77              "Enter background radiation field [K]: ") or 2.73)
78
79          # check if 'temp_background' within bounds that RADEX operates.
80          btw_check = (
81              in_between_check(-1e4, 1e4, temp_background,
82                               'background radiation field') == None
83          )
84          if not btw_check:
85              return 'dummy'
86
87          return temp_background
88
89
90      @re_enter_wrapper
91      def line_width_input(self):
92          """function to set line width [km/s] based on user input,
93          or return a default value.
94
95
96          Returns:
97              float: either user input or standard parameter.
98          """
99
100         line_width = (numeric_input("Enter line  width [km/s]: ") or 1.0)
101
102         # check if 'line_width' within bounds that RADEX operates.
103         btw_check = (
104             in_between_check(1e-3, 1e3, line_width, 'line width') == None
105         )
106         if not btw_check:
107             return 'dummy'
108
109         return line_width
110
111
112     @re_enter_wrapper
113     def geometry_input(self):
114         """function set geometry based on user input, or return uniform
115         sphere as default geometry.
116
117         Raises:
```

```
            ValueError: not a valid geometry.


        Returns:
            int: integer referring to geometry.
        """
        sphere = ['1', 'sphere', 'uni']
        lvg    = ['2', 'lvg']
        slab   = ['3', 'slab']
        cloud_geometry = input(
            "Enter a geometry (1=sphere, 2=LVG, 3=slab): "
        ).replace(' ', '').lower()

        #FIXME take out default geometry and always ask user?
        if cloud_geometry == '':
            return int(1), 'uniform sphere'
        elif cloud_geometry in sphere:
            return 1, 'uniform sphere'
        elif cloud_geometry in lvg:
            return 2, 'LVG'
        elif cloud_geometry in slab:
            return 3, 'slab'
        else:
            print("Not a valid geometry. Choose one " +
                  "of three geometries;\nuniform sphere: "+
                  f"               {sphere}\n" +
                  "large velocity gradient " +
                  f"(LVG): {lvg}\nplane-parallel " +
                  f"slab:           {slab}")
            return 'dummy'

        return


# %%
```

C.2.5 variable_input.py

```
#!/usr/bin/env python3
#%%
# relative imports
from user_input.input_functions import (
    numeric_input,
    min_max_check,
    in_between_check,
    yay_or_nay,
    re_enter_wrapper,
    collision_check
)


class VariableParamters:
    #def __init__(self):
    #    return

```

```
18
19      @re_enter_wrapper
20      def kinetic_temperature_input(self):
21          """function to set kinetic temperature boundaries based on
22          user input, or return default values.
23
24
25          Returns:
26              float, tuple, bool: parameter value, (minimum and maximum
27              kinetic temperature), fit parameter.
28          """
29
30          temp_kin_name = 'tkin'
31
32          temp_kin_fit = (yay_or_nay("Fit the kinetic temperature? (y/n): ")
33                          or False)
34
35          if temp_kin_fit == False:
36              temp_kin = (
37                  numeric_input("Enter kinetic gas temperature [K]: ")
38              )
39              if temp_kin == '':
40                  print("No kinetic temperature is entered. Enter " +
41                        "a kinetic temperature in Kelvin.")
42                  return 'dummy'
43
44              # check if 'temp_kin' is within RADEX boundary limits.
45              btw_check = (
46                  in_between_check(0.1, 1e4, temp_kin, temp_kin_name) == None
47              )
48              if not btw_check:
49                  return 'dummy'
50
51              return [temp_kin_name, temp_kin, (0.1, 1e4), temp_kin_fit]
52
53
54          temp_kin_min = (numeric_input(
55              "Enter minimum kinetic gas temperature [K]: ")
56                  or 10.0)
57          # check if 'temp_kin_min' within bounds that RADEX operates.
58          btw_check = (
59              in_between_check(0.1, 1e4, temp_kin_min,
60                               'minimum kinetic temperature') == None
61          )
62          if not btw_check:
63              return 'dummy'
64
65          temp_kin_max = (numeric_input(
66              "Enter maximum kinetic gas temperature [K]: ")
67                  or 500.0)
68          # check if 'temp_kin_max' within bounds that RADEX operates.
69          btw_check = (
70              in_between_check(0.1, 1e4, temp_kin_max,
71                               'maximum kinetic temperature') == None
72          )
73          if not btw_check:
74              return 'dummy'
```

```
75
76          # checks if maximum > minimum.
77          mm_check = (
78              min_max_check(temp_kin_min, temp_kin_max, temp_kin_name) == None
79          )
80          if not mm_check:
81              return 'dummy'
82
83          # will not be used by program and serves a dummy purpose to keep
84          # the return in the expected shape.
85          temp_kin = (temp_kin_max - temp_kin_min)/2
86
87          return [temp_kin_name, temp_kin, (temp_kin_min, temp_kin_max),
88                  temp_kin_fit]
89
90
91      @re_enter_wrapper
92      def column_density_input(self):
93          """function to set column density boundaries based on
94          user input, or return default values.
95
96
97          Returns:
98              float, tuple, bool: parameter value, (minimum and maximum
99              column density), fit parameter.
100         """
101
102         cd_name = 'cdmol'
103
104         cd_fit = (yay_or_nay("Fit the column density? (y/n): ") or False)
105
106         if cd_fit == False:
107             cd = (
108                 numeric_input("Enter column density [cm^-2]: ")
109             )
110             if cd == '':
111                 print("No column density is entered. Enter " +
112                       "a column density in cm^-2.")
113                 return 'dummy'
114
115             # check if 'tcd' is within RADEX boundary limits.
116             btw_check = (
117                 in_between_check(1e5, 1e25, cd, cd_name) == None
118             )
119             if not btw_check:
120                 return 'dummy'
121
122             return [cd_name, cd, (1e5, 1e25), cd_fit]
123
124
125         cd_min = (numeric_input(
126             "Enter minimum column density [cm^-2]: ")
127                 or 1e11)
128         # check if 'cd_min' within bounds that RADEX operates.
129         btw_check = (
130             in_between_check(1e5, 1e25, cd_min,
131                              'minimum column density') == None
```

```
        )
        if not btw_check:
            return 'dummy'

        cd_max = (numeric_input(
            "Enter maximum column density [cm^-2]: ")
                or 1e16)
        # check if 'cd_max' within bounds that RADEX operates.
        btw_check = (
            in_between_check(1e5, 1e25, cd_max,
                             'maximum column density') == None
        )
        if not btw_check:
            return 'dummy'

        # checks if maximum > minimum.
        mm_check = (
            min_max_check(cd_min, cd_max, cd_name) == None
        )
        if not mm_check:
            return 'dummy'

        cd = (cd_max - cd_min)/2

        return [cd_name, cd, (cd_min, cd_max), cd_fit]


    @re_enter_wrapper
    def collision_densities_input(self):
        """function to set volume densities (and boundaries) based on
        user input, or return default values.


        Returns:
            dict[tuple]: dictionary with collision partner volume
            densities (float) and matching "fit parameter?" indicator
            (bool).
        """

        # collision partners,
        h2    = (0.0, False)
        h     = (0.0, False)
        e     = (0.0, False)
        ph2   = (0.0, False)
        oh2   = (0.0, False)
        hplus = (0.0, False)
        he    = (0.0, False)

        nmin  = 1e-3
        n_min = nmin
        nmax  = 1e13
        n_max = nmax

        densities = {'h2':h2, 'h':h, 'e-':e, 'p-h2':ph2, 'o-h2':oh2,
                     'h+':hplus, 'he':he, 'min_max':(n_min, n_max)}

        vol_dens_names = ['h2', 'h', 'e-', 'p-h2', 'o-h2', 'h+', 'he']
```

```
189         no = ['n', 'no', 'nah', 'nay', 'nope']
190         # input loop that gets recalled on invalid input (@re_enter_wrapper).
191         while True:
192             if vol_dens_names == []:
193                 break
194 
195             collision_key = collision_check(
196                 "Enter (another) collision partner's name " +
197                 f"{vol_dens_names} or enter 'no' if not: ",
198                 vol_dens_names, no
199             )
200             if collision_key in no:
201                 break
202             else:
203                 collision_fit = yay_or_nay(
204                     f"Fit {collision_key}'s density? (y/n): "
205                 )
206                 if collision_fit == True:
207                     # checks if bounds have already been entered.
208                     if (n_min is nmin and n_max is nmax):
209                         #TODO have individual limits for all
210                         # collision partners.
211                         n_min = (numeric_input(
212                             "Enter minimum volume density" +
213                             " [cm^-3] for all collision " +
214                             "partners: "
215                             ) or n_min
216                         )
217                         nmin = n_min
218                         # check if 'n_min' within bounds that RADEX operates.
219                         btw_check = (
220                             in_between_check(
221                                 1e-3, 1e13, n_min, 'minimum volume density'
222                             ) == None
223                         )
224                         if not btw_check:
225                             return 'dummy'
226 
227                         n_max = (numeric_input(
228                             "Enter maximum volume density" +
229                             " [cm^-3] for all collision " +
230                             "partners: "
231                             ) or n_max
232                         )
233                         # check if 'n_max' within bounds that RADEX operates.
234                         btw_check = (
235                             in_between_check(
236                                 1e-3, 1e13, n_max, 'maximum volume density'
237                             ) == None
238                         )
239                         if not btw_check:
240                             return 'dummy'
241 
242                         # checks if maximum > minimum.
243                         mm_check = (
244                             min_max_check(n_min, n_max,
245                                           'volume density') == None
```

```
                    )
                    if not mm_check:
                        return 'dummy'

                    densities['min_max'] = (n_min, n_max)
                    collision_value = (n_max - n_min)/2
            else:
                collision_value = numeric_input(
                    f"Enter density of {collision_key}: "
                )
                if collision_value == '':
                    print("Collision partner density is not set. " +
                          f"User entered '{collision_value}'.")
                    return 'dummy'
                # check if 'collision_value' within RADEX bounds.
                btw_check = (
                    in_between_check(
                        1e-3, 1e13, collision_value, f'{collision_key}'
                    ) == None
                )
                if not btw_check:
                    return 'dummy'


            collision_value_dummy = (n_max - n_min)/2
            if collision_value != collision_value_dummy:
                pass
            else:
                collision_value = collision_value_dummy

            densities[collision_key] = (collision_value, collision_fit)

            vol_dens_names.remove(collision_key)

    # check if a density or bounds are entered for any
    # collision partner.
    densities_copy = densities.copy()
    del densities_copy['min_max']
    density_values = [collision_partner[0]
                      for collision_partner
                      in densities_copy.values()]
    # check if any entry in density_values is a float != 0.0
    # since python reads bool(0.0) = False and nonzero floats
    # as True.
    if any(density_values) != True:
        print("No volume density is set.")
        return 'dummy'

    # check if parameters are within boundary limits.
    vol_invalid = []
    for key, value in densities.items():
        conditions = [
            value[1] == True,
            value[0] != 0.0,
            value != densities['min_max'],
            not (n_min < value[0] < n_max)
        ]
```

```
303             if all(conditions):
304                 vol_invalid += [(n_min, n_max, value[0], key)]
305 
306         if vol_invalid != []:
307             for col_partner in vol_invalid:
308                 n_min, n_max, value, key = col_partner
309                 print(f"{key} = {value} is not within limits [{n_min};" +
310                       f" {n_max}].")
311             return 'dummy'
312 
313         return densities
314 
315 
316 # %%
```

C.3 fitting

Code used to do the fitting, containing all the algorithms used.

C.3.1 __init__.py

```
from .fitting_helper_functions import *
from .find_initial_guess import *
from .MCMC import *
from .LM import *
```

C.3.2 fitting_helper_functions.py

```
#!/usr/bin/env python3


# module imports
from numpy import (
    inf,
    log,
    pi
)
from numpy import sum as np_sum
from spectralradex.radex import run


class AlgorithmHelpers:
    def __init__(self,
                 observed_line_strengths,
                 observed_line_strengths_uncertainties,
                 unit_key,
                 bounds_low,
                 bounds_upp,
                 constant_parameters,
                 matching_lines,
                 fit_parameters_names):
        """constant variables/parameters required by the fitting
        algorithms but not necessarily able to be passed through outright.

        Args:
            observed_line_strengths (numpy.array[float]): observed line
            strengths obtained from the user supplied data file,
            units => T_R (K) -OR- FLUX (K*km/s) -OR- FLUX (erg/cm2/s). To
            be used in "log_likelihood()".

            observed_line_strengths_uncertainties (numpy.array[float]):
            observed line strengths uncertainties obtained from the user
            supplied data file,
            units => T_R (K) -OR- FLUX (K*km/s) -OR- FLUX (erg/cm2/s). To
            be used in "log_likelihood()".

            unit_key (str): string of what units the user supplied data
```

```
            consists of => T_R (K) -OR- FLUX (K*km/s) -OR- FLUX (erg/cm2/s).
            To be used in "RADEX_model()" as a "key" to slice SpectralRadex
            output.

            bounds_low (numpy.array[float]): lower bounds of all parameters
            to be fit => [par_1_low, par_2_low, ..., par_n_low], to be used
            in the prior calculation. To be used in "log_prior()".

            bounds_upp (numpy.array[float]): upper bounds of all parameters
            to be fit => [par_1_upp, par_2_upp, ..., par_n_upp], to be used
            in the prior calculation. To be used in "log_prior()".

            constant_parameters (dict): all parameters not (able to) fit
            stored in a dictionary, as SpectralRadex takes a dictionary
            as input. To be used in "RADEX_model()".

            matching_lines (numpy.array[bool]): array of booleans
            indicating which lines of the molecular file and SpectralRadex
            output are present in the user supplied data file, and can thus
            be compared. True if present, False if not present. To be used
            in "RADEX_model()" to cut the output.

            fit_parameters_names (list[str]): names, recognized by
            SpectralRadex, of parameters to be fit. To be used in
            "RADEX_model()".


        Retrun:
            None
        """

        self.y_obs                  = observed_line_strengths
        self.y_err                  = observed_line_strengths_uncertainties
        self.unit_key               = unit_key
        self.bounds_low             = bounds_low
        self.bounds_upp             = bounds_upp
        self.parameters             = constant_parameters
        self.matching_lines         = matching_lines
        self.fit_parameters_names   = fit_parameters_names

        return


    def RADEX_model(self, fit_parameters_values):
        """Calculates a RADEX model.

        Args:
            fit_parameters_values (nd.array): contains the parameter values
            of the parameters to be fit.


        Returns:
            nd.array: RADEX line strength output for matching lines.
        """

        variable_parameters = {
            variable_parameter_name:variable_parameter_value
```

```
97              for variable_parameter_name, variable_parameter_value
98              in zip(self.fit_parameters_names, 10.0**fit_parameters_values)
99          }
100
101         self.parameters.update(variable_parameters)
102
103         #NOTE if I use this to cut the output (for performance?), then
104         # "output_matching_observation" has to be reworked as well since it
105         # expects the full output.
106         # output_limit = (
107         #     self.matching_line_indeces[0] + self.matching_line_indeces[-1] + 1
108         # )
109         radex_output = run(self.parameters)#.head(output_limit)
110
111         # cut (Spectral)RADEX output to match the user observed
112         # lines, provided in the datafile.
113         output_matching_observation = radex_output.loc[
114             self.matching_lines, self.unit_key
115         ].to_numpy()
116
117         return output_matching_observation
118
119
120 ### MCMC functions only ###
121
122     #FIXME optimize this for speed? define a residuals function?
123     def log_likelihood(self, fit_parameters_values):
124         """logarithm of the likelihood distribution over datasets
125         for the RADEX model.
126
127         Args:
128             fit_parameters_values (numpy array): values of the variable
129             parameters to be fit in the MCMC algorithm.
130
131
132         Returns:
133             float: logarithm of the likelihood distribution.
134         """
135
136         y_RADEX = self.RADEX_model(fit_parameters_values)
137
138         return - 0.5 * (log(2 * pi) + np_sum(
139             2 * log(self.y_err) + ( (self.y_obs - y_RADEX) / self.y_err )**2
140             )
141         )
142
143
144     def log_prior(self, fit_parameters_values):
145         """logarithm of the uniform prior that solely checks if the
146         walkers from the MCMC chain are within the supplied limits.
147
148         Args:
149             fit_parameters_values (numpy array): values of the variable
150             parameters to be fit in the MCMC algorithm.
151
152
153         Returns:
```

```
154             float: "-inf" (unlikely to be true a.k.a. outside limits) or
155             "0.0" within limits (possibly true parameter values).
156         """
157 
158         check = (
159             (fit_parameters_values >= self.bounds_low) ==
160             (fit_parameters_values <= self.bounds_upp)
161         )
162 
163         # check if any of the parameters are outside specified bounds.
164         if False in check:
165             return -inf
166         else:
167             return 0.0
168 
169         return
170 
171 
172     def log_probability(self, fit_parameters_values):
173         """the logarithm of the probability distribution for the
174         parameter uncertainty estimation using MCMC to sample the
175         parameters.
176 
177         Args:
178             fit_parameters_values (numpy array): values of the variable
179             parameters to be fit in the MCMC algorithm.
180 
181 
182         Returns:
183             float: logarithm of the probability distribution.
184         """
185 
186         logprior = self.log_prior(fit_parameters_values)
187         if logprior != 0.0:
188             return -inf
189         else:
190             return logprior + self.log_likelihood(fit_parameters_values)
191 
192         return
193 
194 ### MCMC functions only ###
```

C.3.3 find_initial_guess.py

```
1  #!/usr/bin/env python3
2 
3 
4 
5  #FIXME this whole file should be rewritten pretty much, latent MAGIX
6  # code, long function, not utilizing pandas like in
7  # "fitting_helper_function.py", how parameters that are either fit or not
8  # are handeled, etc.
9  # pretty much make this function similar to LM.py or MCMC.py but above all,
10 # probably should look into the bees algorithm or pso algorithm or other
11 # global search algorithm besides grid search to combat parameter
```

```
# degeneracy.



#%%
# module imports
from numpy import (
    concatenate,
    geomspace,
    linspace,
    loadtxt,
    append,
    array,
    where,
    log10,
    full,
    ones,
    ix_
)
from spectralradex.radex import run_grid
from multiprocessing import cpu_count, Pool
import warnings


def data_file_extraction(user_data_file, uncertainty):
    """extract the line strength column (with uncertainties) from the
    user supplied data file. These uncertainties are only used for
    calculating the chi^2 values so the default uncertainties = 1 (or
    any other constant) since they have no effect then.

    Args:
        user_data_file (str): user supplied data file directory.

        uncertainty (str): uncertainties included ('yes' -OR- 'no')


    Returns:
        tuple: 1 numpy array with line strenghts and 1 numpy array
        with line strength uncertainties.
    """
    data = loadtxt(user_data_file).T
    if uncertainty == 'no':
        line_strenghts = data[1]
        return (line_strenghts, ones(line_strenghts.shape[0]))
    else:
        line_strenghts, line_strenght_uncertanties = data[1:]
        return (line_strenghts, line_strenght_uncertanties)

    return


#FIXME use *args for y_err based on uncertainty?
def chi_squared(y_fit, y_obs, y_err, uncertainty):
    """calculate the chi^2 values between the user data file and the
    spectralRadex grid calculations (y_err as a default is equal to an
    array of ones).

```

```
    Args:
        y_fit (numpy array): the spectralRadex grid fit line strengths
        for all transition lines [T_R (K) -OR- FLUX (K*km/s) -OR-
        FLUX (erg/cm2/s)].

        y_obs (numpy array): the observed line strengths read from data
        file [T_R (K) -OR- FLUX (K*km/s) -OR- FLUX (erg/cm2/s)].

        y_err (numpy array): the observed line strength uncertainties
        read from data file [T_R (K) -OR- FLUX (K*km/s) -OR-
        FLUX (erg/cm2/s)].

        uncertainty (str): are uncertainties included ('yes', 'no').


    Returns:
        [numpy array]: chi_squared values
    """
    if uncertainty == 'no':
        return ( (y_obs - y_fit)**2 ).sum(axis=1)
    else:
        return ( ( (y_obs - y_fit) / y_err )**2 ).sum(axis=1)

    return


def find_initial_parameter_guesses(kinetic_temperature, column_density,
                                   voldens, volume_density,
                                   constant_parameters,
                                   core_count=cpu_count()):
    """calculate the initial parameter guesses to be used by MAGIX
    based on user supplied parameter fit information (bounds,
    fit=True/False, observed data). This is done with spectralRadex's
    grid running function that runs one (large) grid, from which the
    parameter values with the lowest chi2 are chosen as initial
    estimates. This global search method is not very optimized and is
    sometimes referred to as the "brute method". Alternatives
    would be the bees algorithm or particle swarm optimization amongst
    other.

    Args:
        summary = [name [str], value [float], (lim_low, lim_upp) [floats],
        fit (bool)]

        kinetic_temperature (list): summary of kinetic temperature.

        column_density (list): summary of column density.

        voldens (list): list of summaries of all collision partners.

        volume_density (list): list of summaries of all collision partners.

        constant_parameters (list): list of required constant parameters
        for spectralRadex and user data file information.


    Returns:
```

```
126         list: list of lists of parameters that now contains the initial
127         parameter guesses for the values to be written to "parameters.xml"
128         for MAGIX.
129     """
130     _, Tkin_value, Tkin_limits, Tkin_fit = kinetic_temperature
131     _, cd_value, cd_limits, cd_fit     = column_density
132     (user_molfile, Tbg, dv, freq_min, freq_max, geom,
133      units, matching_index, user_datfile,
134      uncertainties) = constant_parameters
135
136
137     # counting the number of parameters to fit (will later be used for
138     # masking spectralRadex output when comparing to observational data
139     # and when retreiving the best initial parameter guesses).
140     # excluding the volume densities for now (the next for loop accounts
141     # for those).
142     parameters_to_fit = [Tkin_fit, cd_fit]
143     number_of_parameters_to_fit = 0
144     for fit in parameters_to_fit:
145         if fit == True:
146             number_of_parameters_to_fit += 1
147
148     Tkin_min, Tkin_max       = Tkin_limits
149     cd_min, cd_max           = cd_limits
150     voldens_min, voldens_max = voldens['min_max']
151
152     num_points_tkin = int((Tkin_max - Tkin_min) / 40)
153     if num_points_tkin < 5:
154         num_points_tkin = 5
155     elif num_points_tkin > 30:
156         num_points_tkin = 30
157
158     num_points_cd = int(log10(cd_max) - log10(cd_min)) - 1
159     if num_points_cd < 5:
160         num_points_cd = 5
161
162     #FIXME what if multiple collision partners are fit, it will be a very
163     # large grid and take a long time just to find initial parameters.
164     num_points_voldens = int(log10(voldens_max) - log10(voldens_min)) + 3
165     if num_points_voldens < 7:
166         num_points_voldens = 7
167
168     #TODO if num_points_tkin + num_points_cd + num_points_voldens > 40?
169     # than decrease it for all points to a more reasonable grid by limiting
170     # the largest num_points_*** first, until the total < 40? again.
171     # num_points_list = [num_points_tkin + num_points_cd + num_points_voldens]
172     # if sum(num_points_list) > 50:
173     #     sorted(num_points_list)
174
175
176     # be sure to exclude the first and last point of the chosen limits
177     # by taking endpoint=False and [1:] to exclude the starting point.
178     if Tkin_fit == True:
179         Tkin_grid = linspace(Tkin_min, Tkin_max,
180                              num_points_tkin + 1, endpoint=False)[1:]
181     else:
182         Tkin_grid = Tkin_value
```

```
183
184     if cd_fit == True:
185         cd_grid = geomspace(cd_min, cd_max,
186                             num_points_cd + 1, endpoint=False)[1:]
187     else:
188         cd_grid = cd_value
189
190
191     grid_guess_parameters = {}
192     for collision_partner in voldens:
193         # exclude the volume density bounds
194         if collision_partner != 'min_max':
195             value, fit = voldens[collision_partner]
196             if fit == False:
197                 grid_guess_parameters[collision_partner] = value
198             else:
199                 number_of_parameters_to_fit += 1
200                 grid_guess_parameters[collision_partner] = geomspace(
201                     voldens_min, voldens_max, num_points_voldens + 1,
202                     endpoint=False
203                 )[1:]
204
205     grid_guess_parameters['tkin']      = Tkin_grid
206     grid_guess_parameters['cdmol']     = cd_grid
207     grid_guess_parameters['molfile']   = user_molfile
208     grid_guess_parameters['tbg']       = Tbg
209     grid_guess_parameters['linewidth'] = dv
210     grid_guess_parameters['fmin']      = freq_min
211     grid_guess_parameters['fmax']      = freq_max
212     grid_guess_parameters['geometry']  = geom
213     # print(grid_guess_parameters)
214
215
216     #FIXME use psutil.cpu_count(logical=True) instead?
217     pool = Pool(processes=core_count)
218
219     grid_output_DataFrame = run_grid(grid_guess_parameters,
220                                      target_value=units,
221                                      pool=pool)
222     #FIXME instead of ".to_numpy()" use pandas dataframe namings for
223     # columns instead? might make code clearer but speed should not
224     # be an issue to begin with (see example in SpectralRadex code
225     # or in LM.py/MCMC.py).
226     grid_output = grid_output_DataFrame.to_numpy()
227
228     # "number_of_parameters_to_fit" accounts for the variable parameters
229     # used in the grid calculations (and also output) by spectralRadex
230     # and thus in need of removal when compared to user data.
231     grid_output_cut = grid_output[:,number_of_parameters_to_fit:]
232
233
234     # only take the spectralRadex output for matching (observed) lines
235     # taken from the data file.
236     grid_output_to_compare = grid_output_cut[
237         ix_(full(grid_output_cut.shape[0], True), matching_index)
238     ]
239
```

```
240     y_observed, y_uncertainties = data_file_extraction(user_datfile,
241                                                        uncertainties)
242 
243     # using '[None,:]' to "match" the dimensionality of
244     # 'grid_output_to_compare' and be able to easily vectorize the chi2
245     # calculation.
246     chi2 = chi_squared(grid_output_to_compare,
247                        y_observed[None,:], y_uncertainties[None,:],
248                        uncertainties)
249 
250     min_chi2_index = int(where(chi2 == chi2.min())[0][0])
251 
252     ## ignore UserWarning ##
253     # user warning is that irrelevant columns have the same name.
254     def usr():
255         warnings.warn("user warning", UserWarning)
256 
257     with warnings.catch_warnings():
258         warnings.simplefilter("ignore")
259         usr()
260         grid_output_dict = grid_output_DataFrame.to_dict()
261     ## ignore UserWarning ##
262 
263 
264     global_parameter_estimates = array([])
265     if Tkin_fit == True:
266         global_tkin = grid_output_dict['tkin'][min_chi2_index]
267         global_parameter_estimates = append(
268             global_parameter_estimates, log10(global_tkin)
269         )
270 
271     if cd_fit == True:
272         global_cd = grid_output_dict['cdmol'][min_chi2_index]
273         global_parameter_estimates = append(
274             global_parameter_estimates, log10(global_cd)
275         )
276 
277 
278     ## suppress deprecation warning ##
279     def depr():
280         warnings.warn("deprecated", DeprecationWarning)
281 
282     with warnings.catch_warnings():
283         warnings.simplefilter("ignore")
284         depr()
285         # save the initial parameter guesses (vol_dens) to appropriate lists.
286         for collision_partner in volume_density:
287             col_partner_name, *_, col_partner_fit = collision_partner
288             if col_partner_fit == True:
289                 collision_partner_index = int(
290                     where(
291                         array(volume_density).T[0] == col_partner_name
292                         )[0][0]
293                 )
294                 volume_density[collision_partner_index][1] = (
295                     grid_output_dict[col_partner_name][min_chi2_index]
296                 )
```

```
297                 global_vd = grid_output_dict[col_partner_name][min_chi2_index]
298                 global_parameter_estimates = append(
299                     global_parameter_estimates, log10(global_vd)
300                 )
301     ## suppress deprecation warning ##
302 
303     return global_parameter_estimates
```

C.3.4 LM.py

```
1  #!/usr/bin/env python3
2  
3  # module import.
4  from scipy.optimize import least_squares
5  
6  
7  def run_levenberg_marquardt(parameter_estimates, model, y_obs, y_err):
8      """run the Levenberg-Marquardt least squares algorithm on the RADEX
9      model for the initial parameter estimates supplied by the global
10     search algorithm. The least squares algorithm is used to refine the
11     estimates of the minimum found by the global search algorithm, after
12     which said parameters will be subject to an MCMC run for uncertainty
13     estimates.
14 
15     Args:
16         parameter_estimates (numpy.array): parameter estimates
17         obtained via the global search algorithm.
18 
19         model (function): RADEX model calculated with SpectralRadex.
20 
21         y_obs (numpy.array): line strengths from the user supplied data
22         file to be used in calculating the residuals.
23 
24         y_err (numpy.array): line strength uncertaintiesfrom the user
25         supplied data file to be used in calculating the residuals.
26 
27 
28     Return:
29         numpy.array: the refined parameter estimates to be subject to
30         an MCMC run.
31     """
32 
33     def residuals(parameters_to_optimize):
34         """a function to calculate the residuals of SpectralRadex
35         output compared with the observed data from the user supplied
36         data file.
37 
38         Args:
39             parameters_to_optimize (numpy.array): parameter estimates
40             obtained via the global search algorithm.
41 
42 
43         Returns:
44             numpy.array: the residuals for the RADEX model, either with
45             the inclusion of uncertainties or not.
```

```
        """

        y_RADEX = model(parameters_to_optimize)
        if y_err.all() == 1:
            return y_RADEX - y_obs
        else:
            return (y_RADEX - y_obs) / y_err

        return


    ls_solution = least_squares(residuals, parameter_estimates, method='lm',
                                ftol=1e-10, xtol=1e-10, gtol=1e-10,)

    return ls_solution.x
```

C.3.5 MCMC.py

```
#!/usr/bin/env python3

# module imports
from emcee.moves import (
    DESnookerMove,
    StretchMove,
    DEMove,
)
from emcee import EnsembleSampler
from multiprocessing import Pool, cpu_count
from numpy.random import randn


# required/suggested by emcee when using automatic parallelization done by
# numpy using MKL linear algebra for instance to disable it and let
# other implementations like Pool in this case take care of parallelization.
# see https://emcee.readthedocs.io/en/stable/tutorials/parallel
import os
os.environ["OMP_NUM_THREADS"] = "1"


#FIXME set walkers as multiple of cpu_count()?
def run_monte_carlo(initial_parameters,
                    log_probability_function,
                    number_of_walkers=35,
                    number_of_steps=500,
                    number_of_burnin_steps=100,
                    number_of_walker_steps=200,
                    core_count=cpu_count()):
    """
    Args:
        initial_parameters (nd.array): initial parameters obtained by prior
        algorithms (grid search --> LM) in the chain.

        log_probability_function (function): logarithm (base10) of posterior.

        number_of_walkers (int): number of walkers. Defaults to 35.
```

```
38
39          number_of_steps (int): number of steps. Defaults to 500.
40
41          number_of_burnin_steps (int): number of burnin steps. Defaults to 100.
42
43          number_of_walker_steps (int): number of walker steps. Defaults to 200.
44
45          core_count (int): number of processors. Defaults to cpu_count().
46
47
48      Returns:
49          EnsambleSampler, int: emcee sampler object and number of parameters.
50      """
51
52      # Initialize the walkers in a Gaussian "ball" around the best initial
53      # parameter estimates found by prior algorithms in the chain.
54      pos = initial_parameters + 1e-3 * randn(number_of_walkers,
55                                              initial_parameters.shape[0])
56      nwalkers, ndim = pos.shape
57
58      # run the MCMC algorithm.
59      with Pool(processes=core_count) as pool:
60          #FIXME figure out the best set of moves for all molecules?
61          sampler = EnsembleSampler(
62              nwalkers, ndim, log_probability_function, pool=pool,
63              moves=[(StretchMove(a=3), 0.7),
64                     (DEMove(), 0.2),
65                     (DESnookerMove(), 0.1),]
66          )
67
68          sampler.run_mcmc(pos, number_of_steps, progress=True)
69
70          #FIXME separate burnin and uncertainty sampling?
71          # # calculate burnin chain.
72          # idk = sampler.run_mcmc(pos, number_of_burnin_steps, progress=True)
73
74          # # calculate walker (uncertainties?) chain.
75          # sampler.run_mcmc(idk[???], number_of_walker_steps, progress=True)
76
77      return sampler, ndim
```

C.4 save_plot

Code used to save an plot results.

C.4.1 __init__.py

```
from .fitting_helper_functions import *
from .find_initial_guess import *
from .MCMC import *
from .LM import *
```

C.4.2 save_plot_helper.py

```
#!/usr/bin/env python3

#module imports
from spectralradex.radex import run
from numpy import array


def RADEX_model_plot(fit_parameter_names, parameters,
                     fit_parameters_values):
    """calculate RADEX model.

    Args:
        fit_parameter_names (list): list of names of parameters to fit.

        parameters (dict): constants.

        fit_parameters_values (array/list): fitted parameters' values.


    Returns:
        pd.DataFrame: full RADEX output + for which input parameters.
    """

    variable_parameters = {
        variable_parameter_name:variable_parameter_value
        for variable_parameter_name, variable_parameter_value
        in zip(fit_parameter_names, 10.0**array(fit_parameters_values))
    }

    parameters.update(variable_parameters)
    parameters['fmin']=0
    parameters['fmax']=3e7

    radex_output = run(parameters)

    return radex_output
```

C.4.3 plot.py

```
#!/usr/bin/env python3

#relative imports
from .save_plot_helper import RADEX_model_plot

# module imports
from matplotlib.pyplot import figure, show
import matplotlib.pyplot as plt
from numpy import exp
from numpy.random import randint
import warnings
import corner


class Plotting:
    def __init__(self,
                 sampler,
                 output_path,
                 parameter_50s,
                 fit_parameter_names):
        """class used for plotting.

        Args:
            sampler (emcee:EnsembleSampler): MCMC parameter coordinates.

            output_path (str): output folder.

            parameter_50s (nd.array): MCMC parameter medians.

            fit_parameter_names (list): Names of fitted parameters.


        Returns:
            None
        """

        self.sampler             = sampler
        self.output_path         = output_path
        self.parameter_50s       = parameter_50s
        self.fit_parameter_names = fit_parameter_names

        #FIXME put in the molecule name for column density.
        plot_names = {
            'tkin':r"log$_{10}$(T$_{\mathrm{kin}}$) [K]",
            'cdmol':r"log$_{10}$(N$_{\mathrm{mol}}$) [cm$^{-2}$]",
            'h2':r"log$_{10}$(H$_2$) [cm$^{-3}$]",
            'p-h2':r"log$_{10}$(p-H$_2$) [cm$^{-3}$]",
            'o-h2':r"log$_{10}$(o-H$_2$) [cm$^{-3}$]",
            'e-':r"log$_{10}$(e$^-$) [cm$^{-3}$]",
            'h':r"log$_{10}$(H) [cm$^{-3}$]",
            'he':r"log$_{10}$(He) [cm$^{-3}$]",
            'h+':r"log$_{10}$(H$^+$) [cm$^{-3}$]"
        }

```

```
        self.plot_labels = [plot_names[plot_name]
                            for plot_name
                            in self.fit_parameter_names]

        return


    def plot_corner(self):
        """Make and save a corner plot of the MCMC sampled posterior
        distributions of the parameters that are fit. both 2D contours
        between parameters and 1D distributions.


        Returns:
            None
        """
        flat_samples = self.sampler.get_chain(discard=100, flat=True)
        fig = corner.corner(
            flat_samples, labels=self.plot_labels, truths=self.parameter_50s,
            quantiles=(0.16, 0.84), levels=(1 - exp(-0.5),), smooth=True,
            label_kwargs={'fontsize':15}
        )

        plt.close(fig)
        fig.savefig(f'{self.output_path}/MCMC_corner_plot.png',
                    dpi=300, bbox_inches='tight')

        return


    def plot_spectrum(self,
                      unit_name,
                      line_strength_y,
                      line_strength_err,
                      constant_parameters,
                      frequencies):
        """plot the observed data points, as well as the RADEX model
        spectrum for the best estimates and an "uncertainty" interval
        using 100 random MCMC results.

        Args:
            unit_name (str): dict key of units selected by user.

            line_strength_y (nd.array): user line strenghts.

            line_strength_err (nd.array): user line strength uncertainties.

            constant_parameters (dict): RADEX input for SpectralRadex for
                                        the constant input.

            frequencies (nd.array): molfile frequencies matching user
                                    frequencies.


        Returns:
            None
        """
```

```
112
113         unit_labels = {
114             'T_R (K)':r'T$_{\mathrm{R}}$ [K]',
115             'FLUX (K*km/s)':r'$\mathcal{F}$ [K km s$^{-1}$]',
116             'FLUX (erg/cm2/s)':r'$\mathcal{F}$ [erg cm$^{-2}$ s$^{-1}$]'
117         }
118
119         fig   = figure(figsize=(15,10.5))
120         frame = fig.add_subplot(1,1,1)
121
122
123         # plot 100 randomly drawn RADEX models to showcase uncertainty
124         # interval loosely.
125         flat_samples = self.sampler.get_chain(discard=100, flat=True)
126         inds = randint(len(flat_samples), size=100)
127         for ind in inds:
128             sample = flat_samples[ind]
129             rnd_output = RADEX_model_plot(
130                 self.fit_parameter_names, constant_parameters, sample
131             )
132             rnd_freqs          = rnd_output['freq']
133             rnd_line_strengths = rnd_output[unit_name]
134             frame.scatter(rnd_freqs, rnd_line_strengths, color='#4daf4a',
135                           alpha=0.1, marker='s')
136         #FIXME add to legend without dummy plot.
137         # a dummy plot to add MCMC "uncertainty interval" to legend.
138         frame.plot(frequencies[0], line_strength_y[0], color='#4daf4a',
139                    marker='s', scalex=False, scaley=False, alpha=0.8,
140                    zorder=0, label='MCMC uncertainty interval',
141                    linestyle = 'None')
142
143
144         # plot RADEX model for the optimal parameter estimates
145         output_50 = RADEX_model_plot(
146             self.fit_parameter_names, constant_parameters,
147             self.parameter_50s
148         )
149         freq_50           = output_50['freq']
150         line_strengths_50 = output_50[unit_name]
151         frame.scatter(freq_50, line_strengths_50,
152                       color='#ff7f00', marker='D',
153                       label='RADEX optimal parameters', s=70)
154
155
156         ## ignore UserWarning ##
157         # The warning says "fmt" is redundant when "marker" is defined
158         # (or vice versa?) but this does not seem to be the case.
159         def usr():
160             warnings.warn("user warning", UserWarning)
161
162         with warnings.catch_warnings():
163             warnings.simplefilter("ignore")
164             usr()
165             # plot observations.
166             frame.errorbar(frequencies, line_strength_y,
167                            yerr=line_strength_err,
168                            marker='.', fmt=',', mew=3, ms=13, linewidth=1,
```

```
                            capsize=3, capthick=1,
                            color='dodgerblue', ecolor='black',
                            label='Observed data', zorder=10)
        ## ignore UserWarning ##


        #FIXME make sure that this legend placement is sufficient
        # for every molecule?
        frame.legend(fontsize=16, fancybox=True, shadow=True, ncol=1,
                     loc='upper right', bbox_to_anchor=(0.975, 0.9678))
        frame.set_xlabel(r'$\nu$ [GHz]', fontsize=21)
        frame.set_ylabel(unit_labels[unit_name], fontsize=21)
        frame.yaxis.offsetText.set_fontsize(18)
        frame.set_axisbelow(True)
        frame.grid(True)

        frame.tick_params(axis='both', direction='in', which='major',
                          length=10, width=1, labelsize=18)

        fig.savefig(f'{self.output_path}/spectrum.png',
                    dpi=300, bbox_inches='tight')
        show()

        return
```

C.4.4 save.py

```
#!/usr/bin/env python3


#relative imports
from .save_plot_helper import RADEX_model_plot

#module imports
from numpy import (
    percentile,
    savetxt,
    array,
    diff,
    full,
    NaN
)
from pandas import read_csv


class SaveResults:
    def __init__(self,
                 sampler,
                 output_path,
                 constant_parameters,
                 fit_parameters_names):
        """
        Args:
            sampler (EnsambleSampler): emcee sampler object containing
            all MCMC sampler parameter coordinates.
```

```
29
30              output_path (str): output directory.
31
32              constant_parameters (dict): dictionary of the constant
33              parameter inputs of SpectralRadex.
34
35              fit_parameters_names (list): parameter names of parameters
36              to be fit.
37
38
39          Returns:
40              None
41          """
42
43          self.sampler              = sampler
44          self.output_path          = output_path
45          self.constant_parameters  = constant_parameters
46          self.fit_parameters_names = fit_parameters_names
47          return
48
49
50      def print_parameter_uncertainty_estimates(self,
51                                                user_datfile,
52                                                user_frequencies,
53                                                limits):
54          """prints and saves fit information.
55
56          Args:
57              user_datfile (str): observed user data file location.
58
59              user_frequencies (list): observed user line frequencies.
60
61              limits (dict): dictionary of the limits for fit parameters.
62
63
64          Returns:
65              list: parameter medians
66          """
67
68          parameter_50s = []
69          print("\nParameter estimates and accompanying upper and "
70                "lower uncertainties,")
71          prms_sum_dir = f'{self.output_path}/parameters.txt'
72          with open(prms_sum_dir, 'w') as prms_txt:
73              prms_txt.write(
74                  'Data file used: ' +
75                  user_datfile +
76                  '\n'
77              )
78              prms_txt.write(
79                  'Line (frequencies) used: ' +
80                  str(user_frequencies)[1:-1] +
81                  '\n'
82              )
83              for i, parameter_name in enumerate(self.constant_parameters):
84                  if parameter_name in self.fit_parameters_names:
85                      prms_txt.write(
```

```
                        f"{parameter_name}'s parameter boundaries: " +
                        f'{limits[parameter_name]}\n'
                    )
                else:
                    prms_txt.write(
                        f'{parameter_name} = ' +
                        f'{self.constant_parameters[parameter_name]}\n'
                    )


            header = f"Percental:   50%   |   16%   |   84%   "
            print(header)
            prms_txt.write('\n' + header)
            for i, parameter_name in enumerate(self.fit_parameters_names):
                # obtaining the median and upper and lower uncertainties
                # that enclose 1 sigma.
                parameter_uncertainty_estimates = percentile(
                    self.sampler.get_chain(discard=100, flat=True)[:, i],
                    q=[16, 50, 84]
                )
                uncertainties = diff(parameter_uncertainty_estimates)

                median = parameter_uncertainty_estimates[1]
                prm_16, prm_84 = uncertainties

                parameter_50s += [median]

                parameter_summary = (
                    f"{parameter_name}    : {median:.5f} | -{prm_16:.5f}" +
                    f" | +{prm_84:.5f}"
                )
                print(parameter_summary)
                prms_txt.write('\n' + parameter_summary)


        return parameter_50s


    def save_MCMC_sampler(self):
        """save emcee EnsembleSampler.flatchain object.
        """

        savetxt(
            f'{self.output_path}/sampler.dat',
            self.sampler.get_chain(flat=True),
            header=str(self.fit_parameters_names)[1:-1]
        )

        return


    def RADEX_for_optimal_parameters(self,
                                     user_datfile,
                                     user_frequencies,
                                     y_obs,
                                     y_err,
                                     matching_indices,
```

```
                                    units,
                                    limits):
        """saves RADEX.csv, which is a RADEX model output for
        the parameter medians estimated with the MCMC algorithm.
        Addintionally, the chi^2 for each observed line is
        calculated and saved as well.

        Args:
            user_datfile (str): user observed data file location.

            user_frequencies (list): user observed frequencies.

            y_obs (list): user observed line strengths.

            y_err (list): user observed line strengths uncertainties.

            matching_indices (list): incides that match the user
            observations with the RADEX output.

            units (str): dictunary key that indicates the units used
            in user_datfile.

            limits (dict): dictionary of the limits for fit parameters.


        Returns:
            list: parameter medians
        """

        # this is the only way the function below is called.
        params_50 = self.print_parameter_uncertainty_estimates(
            user_datfile, user_frequencies, limits
        )
        optimal_RADEX = RADEX_model_plot(
            self.fit_parameters_names, self.constant_parameters, params_50
        )


        # define to chi2 column to be added to RADEX.csv.
        y_RADEX = optimal_RADEX.loc[matching_indices, units].to_numpy()
        if y_err.all() == 1:
            chi2_calc = (y_obs - y_RADEX) ** 2
        else:
            chi2_calc = ( (y_obs - y_RADEX) / y_err ) ** 2

        chi2 = full(optimal_RADEX[units].shape[0], NaN)
        chi2[matching_indices] = chi2_calc

        csv_path = f'{self.output_path}/RADEX.csv'
        # write RADEX output to csv.
        optimal_RADEX.to_csv(
            csv_path, sep=',', na_rep=NaN, float_format='%.3e'
        )
        # read the RADEX output.
        csv_file = read_csv(csv_path)
        # add the chi2 values as the last column.
        csv_file['chi^2'] = array(chi2)
```

```
200         # save the new RADEX .csv file.
201         csv_file.to_csv(
202             csv_path, index=False, na_rep=NaN, float_format='%.3e'
203         )
204 
205         return params_50
```