Draft version August 11, 2023
Typeset using LaTeX default style in AASTeX631

The Foundations of Multi-Line Molecular Cloud Population Synthesis

Marissa Perry[1, 2] and Jens Kauffmann[2]

[1] The University of Texas at Austin, Department of Astronomy, Austin, TX
[2] Haystack Observatory, Massachusetts Institute of Technology, Westford, MA

ABSTRACT

To develop a thorough understanding of the structure and evolution of galaxies, we are required to investigate the fundamental process of molecular cloud formation. Although galactic molecular clouds have been studied in great detail, spatial resolution of current radio telescopes inhibits our ability to resolve individual molecular clouds in external galaxies. As a consequence, we receive an averaged signal over a range of physical environments in a number of molecular clouds. These environmental variations are thought to depend upon the life cycle of molecular clouds. This leads us to the hypothesis that each stage of cloud evolution formulates a populations of molecular clouds. Thus, molecular cloud population synthesis techniques are essential to understand the assembly of extragalactic systems. Here we theoretically model the decomposition of an extragalactic molecular line luminosity signal into its cloud population molecular line luminosity components. The identification of emission lines that can plausibly serve as tracers of specific physical environments are also explored in depth, developing the idea of multi-line cloud population synthesis.

Keywords: Galaxies: star formation — ISM: clouds — ISM: dense gas — ISM: tracer molecules

1. INTRODUCTION

Star formation (SF) occurs within turbulent molecular clouds, where masses of dense gas collapse under their own gravity. The star formation rate (SFR) in a nearby molecular cloud can be inferred by counting the number of newborn stars present within the dense gas. However, resolving individual stars is impractical for more distant molecular clouds. Thus, the dense gas component of a stellar nursery is often used as alternative physical indicator of SF. The mass of dense gas in molecular clouds is often defined to be the gas mass residing at visual extinctions of 7 mag and above, $M_{\rm dg} = M(A_V \geq 7\ {\rm mag})$, as this definition has been shown to result in a correlation between the SFR and the mass of dense gas as defined above, $\dot{M}_\star \propto M_{\rm dg}$ (e.g., Heiderman et al. 2010, Lada et al. 2010).

Although molecular clouds are primarily composed of H_2, under typical circumstances this molecule does not emit or absorb enough radiation for us to detect. To work around this, alternative tracer molecules excited by collisions with H_2 are studied. Transitions with high critical densities are often used to trace dense gas. For instance, Kauffmann et al. (2017) reasoned that the N_2H^+ ($J = 1$–0) molecular line transition emits at a high critical density and developed it into a potential tracer of dense gas, leading us to the suggested relationship $M_{\rm dg} \propto L_{\rm N_2H^+\,(1-0)}$. Thus, we can potentially study SF in molecular clouds through observations of particular molecular line transitions. Furthermore, ratios of these molecular line emissions can be used to identify cloud-to-cloud variations in physical conditions, such as the density and temperature of a molecular cloud. This is one of the areas which the Molecular Line Emission as a Tool for Galaxy Observations (LEGO) project investigates through mapping well-characterized star-forming regions in the Milky Way (Kauffmann & Barnes 2022).

Specific physical conditions within clouds have been theorized to correspond to evolutionary stages (Chevance et al. 2020). Furthermore, Foster et al. (2011) presents results shown in Table 1 indicating different molecules could be associated with different physical environments. If this is true, multi-line observations of early molecular cloud formation should differ from multi-line observations of a star-forming molecular cloud. With our current generation of radio telescopes and projects such as LEGO, multi-line observational data can be collected for entire local ($\leq$ 1.5 kpc) molecular clouds in the Milky Way (Barnes et al. 2020). Moving to extragalactic scales ($\geq$ 0.7 Mpc), these molecules are detectable over galaxy regions (Watanabe et al. 2014). Thus, we can potentially observe cloud populations within galaxies through multi-line cloud population synthesis.

In this project, the goal is to theoretically interpret such multi-line extragalactic observations. In this way, we can begin to overcome the resolution gap between local and extragalactic environments. With a single-dish telescope, we are able to spatially resolve local clouds. However, spatially resolving individual clouds in external galaxies is not achievable with current radio telescope technology. In this case, the line luminosity signal received is an average over a $\geq$ 100 pc diameter beam containing several molecular clouds. This leads us to an interesting question. *If we point the telescope beam on a region of molecular clouds in a nearby galaxy, can we meaningfully decompose the unresolved signal we receive?*

2. PROPERTIES OF LINE EMISSION

2.1. *Molecular Line Emission*

Understanding processes which lead to the production of emission lines is important to conceptualize the physical and chemical activity within a molecular cloud. The brightness temperature measured by a radio telescope beam $T_{\rm MB}$ characterizes the brightness of a particular emission line and can be represented by

$$T_{\rm MB} = T_{\rm ex} \cdot (1 - e^{-\tau}) \quad . \tag{1}$$

Optical depth τ quantifies the opaqueness of a molecular cloud observed in a particular emission line and is related to column density by $\tau \propto N$. Column density of a given molecule $N(\chi)$ quantifies the amount of the molecule resides along the line of sight and is defined as

$$N(\chi) = n({\rm H_2}) \cdot \chi \quad , \tag{2}$$

where $n({\rm H_2})$ is the number density of molecular hydrogen—the bulk of molecular cloud gas—and χ is the abundance of a given molecule relative to molecular hydrogen, which is defined as

$$\chi = \frac{n(\chi)}{n({\rm H_2})} \quad . \tag{3}$$

Intuitively, a molecule which resides in denser gas—higher $N({\rm H_2})$—or has a higher abundance in the column of gas—higher χ—have higher τ and vice versa. Thus, emission line visibility can potentially serve as a density and molecular abundance diagnostic (this is further discussed in Sec. 2.3).

2.2. *Line Excitation*

Excited states of molecules store energy in the form of vibration and rotation, while interactions among molecules in a molecular cloud include collisions and absorption. A significant number will move into excited states when the mean time between molecular collisions is well below the radiative lifetime A_{jk}^{-1}. Intuitively, kinetic energy in the form of photons in low density regions with infrequent collisions do not get released. Critical density $n_{\rm cr}$ is one way of defining this, where the radiative (spontaneous) de-excitation rate equals the collisional de-excitation rate for a given transition Q from $j \rightarrow k$, and emission becomes efficient. Mathematically, this can be represented as

$$n_{\rm cr} = \frac{A_{jk}}{\gamma_{jk}} \quad , \tag{4}$$

where A_{jk} is the Einstein A coefficient and γ_{jk} is a characterization of de-excitation via collisions. In the low-density limit ($n < n_{\rm cr}$), there are insufficient collisions to excite the molecule into the upper level j and line emission is not efficient. In the high-density limit ($n > n_{\rm cr}$), line emission is more efficient. For a quantitative example, Barnes et al. (2020) found HCN to have $n_{\rm cr} = 3.0 \times 10^5\,{\rm cm}^{-3}$. The excitation temperature $T_{\rm ex}$ is generally several 10 % of the kinetic temperature $T_{\rm kin}$ at $n = n_{\rm cr}$.

The excitation of a particular transition occurs at an effective critical density n_{eff}. This becomes relevant if other means of excitation increase the effective collision rate of molecules. For instance, molecular absorption of photons have a similar effect to the molecule being subjected to an increased rate of collisions. This decreases the density needed to produce a certain excitation temperature T$_{\rm ex}$, as depicted in Fig. 1. Therefore, one has $n_{\rm eff} > n_{\rm cr}$ if only collisional excitation is significant to line emission and $n_{\rm eff} < n_{\rm cr}$ if a radiation means of excitation is also significant to

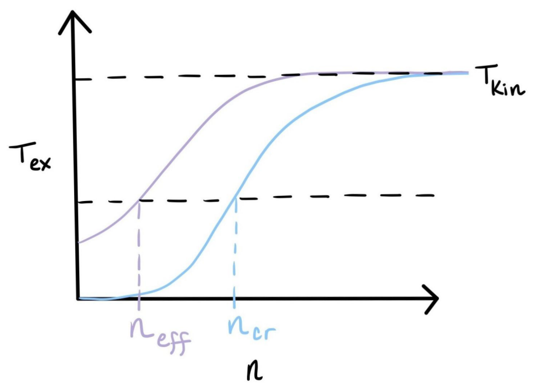


Figure 1. Comparison of $n_{\rm cr}$ and $n_{\rm eff}$ characterizations of molecular density in terms of $T_{\rm ex}$

line emission. In some cases, n_{eff} might be a better characterization of molecular density than n_{cr}. This is discussed in Shirley (2015), where tracing molecules which have a lower abundance within a gas cloud is found to be more accurate with n_{eff}.

2.3. *Astrochemistry*

As discussed in Sec. 2.1 and Sec. 2.2, gas density and temperature have a direct impact on line emission. However, a range of environmental factors also have a significant impact on molecular abundances χ, and therefore, the presence of emission lines associated with a given molecule.

By modeling a molecular cloud's reaction to environmental changes, we can understand how physical parameters influence the chemistry of clouds. An example calculation of this type was performed in Harada et al. (2019), where they considered a range of environmental parameters, including gas density n, visual extinction A_V, temperature T, cosmic-ray ionization rate ξ, and sulfur abundance S/H. For instance, they show that while the abundance of N_2H^+—a theorized tracer of cold dense gas—generally increases with time and molecular density, as shown in Fig. 2, the abundance of C_2H—a theorized tracer of hot sparse gas—stays approximately the same over time and various molecular densities. Although this work constructively outlines how changes in each physical parameter impacts molecular abundance, the setup is idealized with long timescale calculations being made with fixed physical conditions. A more realistic setup is presented in Ruaud et al. (2018), where the movement of gas packages in a molecular cloud is simulated. However, this has its own limitations as it is not trivial to tease out how a given physical property affects a particular molecular abundance.

Observations have proven helpful in constraining these theoretical notions of environmental influences on molecular clouds. In Watanabe et al. (2014), a high abundance of CH_3OH was found in two regions (P1 and P2) of a spiral arm in M51. It is theorized that when CO molecules freeze out onto interstellar dust grains, astrochemical reactions can lead to CH_3OH formation on the surface of dust grains. Temperatures of order 100K are needed to desorb CH_3OH from the dust grains. Such conditions can be found in shocked regions of gas, so that CH_3OH can serve as a tracer of energetic processes.

A list of molecules and their hypothesized physical tracings are shown in Table 1. In laying out a number of molecules and their physical tracings, theoretical and observational findings are considered in Jackson et al. (2013). Their findings suggest that (a) a molecular cloud might look different in different emission lines and (b) that such line emission trends might be different in different clouds. However, these molecular tracers may not be effective in all situations. This holds true for more recent observations on larger spatial scales, as in Kauffmann et al. (2017). It was shown that the HCN molecule is not a reliable tracer of high gas densities, in contrast to findings by Jackson et al. (2013). This highlights that although our current understanding of astrochemical models and observational trends allow us to identify molecules that can potentially serve as tracers of physical properties, this knowledge is constantly progressing with new simulations and observations.

3. MULTI-LINE CLOUD POPULATION SYNTHESIS

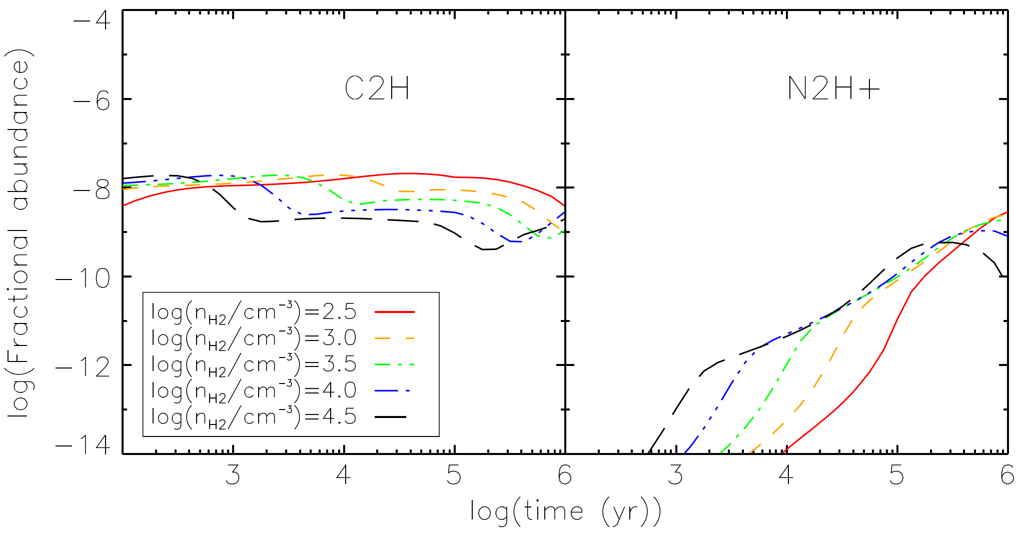


Figure 2. Comparison of C_2H and N_2H^+ fractional abundances calculations over time based on Fig. 1 of Harada et al. (2019).

Table 1. Theorized tracer molecules taken from Foster et al. (2011).

Species	Transition	ν_0 (GHz)	Primary Information Provided
N_2H^+	$J = 1 - 0$	93.173772	High column density, depletion resistant, optical depth
^{13}CS	$J = 2 - 1$	92.494303	High column density
H	41α	92.034475	Ionized gas
CH_3CN	$J_K = 5_1 - 4_1$	90.979020	Hot core
HC_3N	$J = 10 - 9$	91.199796	Hot core
$^{13}C^{34}S$	$J = 2 - 1$	90.926036	High colum density
HNC	$J = 1 - 0$	90.663572	High column density, cold gas
$HC^{13}CCN$	$J = 10 - 9, F = 9 - 8$	90.593059	Hot core
HCO^+	$J = 1 - 0$	89.188526	High column density, kinematics
HCN	$J = 1 - 0$	88.631847	High column density, optical depth
HNCO	$J_{K_a,K_b} = 4_{0,4} - 3_{0,3}$	88.239027	Hot core
HNCO	$J_{K_a,K_b} = 4_{1,3} - 3_{1,2}$	87.925238	Hot core
C_2H	$N = 1 - 0, J = 3/2 - 1/2, F = 2 - 1$	87.316925	Photodissociation region
SiO	$J = 2 - 1$	86.847010	Shock/outflow
$H^{13}CO^+$	$J = 1 - 0$	86.754330	High column density, optical depth
$H^{13}CN$	$J = 1 - 0$	86.340167	High column density, optical depth

Using all of this chemical and physical information, we can attempt to piece together a life cycle for molecular clouds. Three main stages have been theorized: 1) a molecular cloud envelope forms and is mostly inactive, 2) efficient SF begins and a photo disassociation region (a high temperature region which dissolves nearby gas) forms, 3) gas gets dispersed even more, leaving behind isolated stars. This process is illustrated in Fig. 3, where molecular tracers are indicated for each chemical environment a stage produces. The relative time a cloud spends in each phase is quantified in in Fig. 4 taken from Chevance et al. 2020.

Since each stage of cloud evolution produces a different physical environment, using the idea of physical tracers from Sec. 2.3 we can hypothesize that each stage has a specific combination of molecular emission lines. For instance, molecules such as CH_3OH, HC_3N, $HC^{13}CCN$, and HNCO are thought to be associated with protostars and HII regions and can indicate an early evolutionary stage (Jackson et al. 2013). In other words, each evolutionary stage potentially

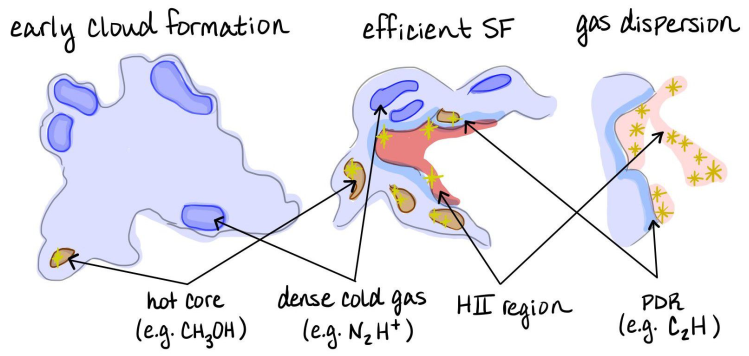


Figure 3. Illustration of molecular cloud evolution. Left: early cloud formation. Middle: efficient SF phase. Right: gas dispersion phase. Molecules theorized to be associated with these physical properties are labeled. Young stellar objects are embedded within the HII (ionized molecular hydrogen) region.

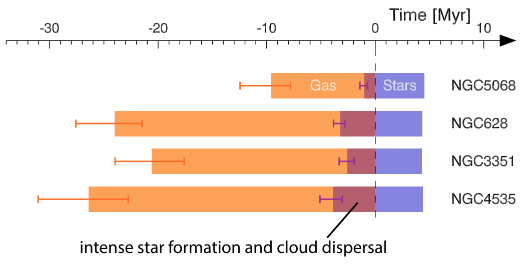


Figure 4. The relative time molecular clouds spend in their evolutionary stages taken from Chevance et al. 2020.

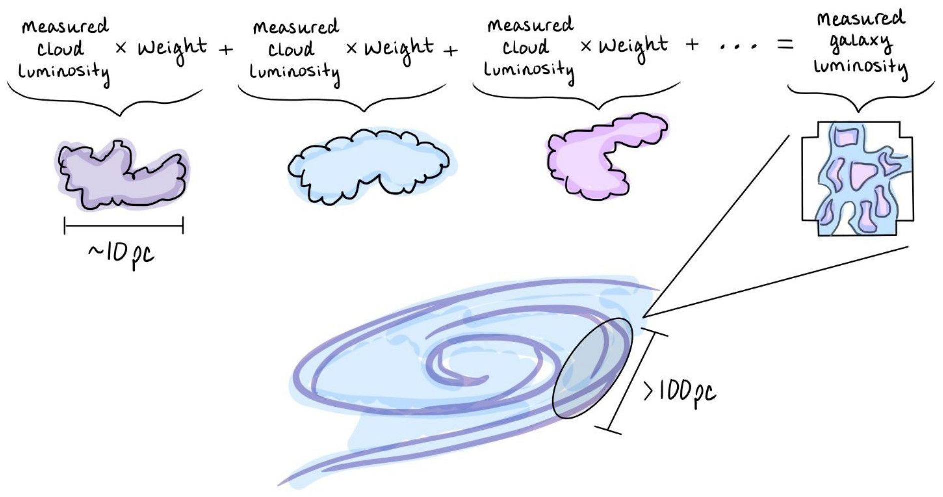

Measured cloud luminosity × weight +
Measured cloud luminosity × weight +
Measured cloud luminosity × weight + . . . =
Measured galaxy luminosity
~10 pc
>100 pc

Figure 5. Components of the line luminosity signal from an external galaxy. The measured galaxy line luminosity is represented as a data cube of a mapped region (indicated by the black box) within position-position-velocity space.

forms a population of molecular clouds, distinguishable from one another via multi-line observations. These multi-line cloud populations are represented in Fig. 5 as an observation of a nearby galaxy region.

3.1. *Single Cloud Observation*

Observing a molecular cloud with a single-dish radio telescope, we can let Q be an index running over n molecular emission lines. Let i further be an index running over a total of m molecular clouds. Then the luminosity of cloud i in emission line Q can be expressed as $L_{i,Q}$. This information can be expressed via vectors

$$\vec{L}_i = \begin{pmatrix} L_{i,1} \\ \vdots \\ L_{i,n} \end{pmatrix} \quad . \tag{5}$$

Thus, a single molecular cloud can be observed in several molecular emission lines. Such an observation highlights physical regions a given molecular cloud population contains. For instance, let us say the first cloud population shown in Fig. 5 has lots of stellar outflows, causing nearby gas to become shocked. Upon inspecting Table 1, we would expect the SiO emission line luminosity element $L_{1,\,\mathrm{SiO}}$ to be relatively high. In other words, different clouds populations have different line luminosities $\vec{L}_i$.

3.2. *Galaxy Region Observation*

Observing the line luminosity of a galaxy with a single-dish telescope, we can represent our beam signal as a sum over several clouds within an area of several 10^2 pc diameter, as shown in Fig. 5. As discussed in Sec. 3, groups of molecular clouds potentially have similar emission line properties, forming cloud categories. Let us assume that these clouds fall into m categories indicated by index i, and that w_i clouds of category i exist within the studied area. Then the galaxy's line luminosity in transition Q can be expressed as

$$L_{\mathrm{gal},Q} = L_{\mathrm{a},Q} \cdot w_\mathrm{a} + \ldots + L_{m,Q} \cdot w_m \quad , \tag{6}$$

or as

$$\begin{aligned} \vec{L}_\mathrm{gal} &= \vec{L}_\mathrm{a} \cdot w_\mathrm{a} + \ldots + \vec{L}_m \cdot w_m && (7)\\ &= \sum_i^m \vec{L}_i \cdot w_i && (8)\\ &= \hat{L}_\mathrm{cloud} \cdot \vec{w} \quad , && (9) \end{aligned}$$

where $\hat{L}_\mathrm{cloud}$ is an $(m \times n)$-dimensional matrix in which each matrix element represents cloud i observed in molecular emission line Q.

3.2.1. *Line Luminosity Ratios*

Now let us select a reference emission line R. We can use the corresponding reference line luminosity for a given cloud, $L_{i,R}$, to rewrite Eq. (9) as

$$L_{\mathrm{gal},R} \cdot \begin{pmatrix} L_{\mathrm{gal},1}/L_{\mathrm{gal},R} \\ \vdots \\ L_{\mathrm{gal},n}/L_{\mathrm{gal},R} \end{pmatrix} = \sum_i^m L_{i,R} \cdot \begin{pmatrix} L_{i,1}/L_{i,R} \\ \vdots \\ L_{i,n}/L_{i,R} \end{pmatrix} \cdot w_i \quad . \tag{10}$$

We may further divide Eq. (10) by $L_{\mathrm{gal},R}$, which yields

$$\begin{pmatrix} L_{\mathrm{gal},1}/L_{\mathrm{gal},R} \\ \vdots \\ L_{\mathrm{gal},n}/L_{\mathrm{gal},R} \end{pmatrix} = \sum_i^m \frac{L_{i,R}}{L_{\mathrm{gal},R}} \cdot \begin{pmatrix} L_{i,1}/L_{i,R} \\ \vdots \\ L_{i,n}/L_{i,R} \end{pmatrix} \cdot w_i \quad . \tag{11}$$

At this point we may introduce a new weight,

$$w_i^* = (L_{i,R}/L_{\mathrm{gal},R}) \cdot w_i \quad , \tag{12}$$

and line ratio vectors,

$$\vec{S}_i = \begin{pmatrix} L_{i,1}/L_{i,R} \\ \vdots \\ L_{i,n}/L_{i,R} \end{pmatrix} \quad . \tag{13}$$

This allows to write Eq. (11) as

$$\vec{S}_{\rm gal} = \vec{S}_{\rm a} \cdot w^*_{\rm a} + \ldots + \vec{S}_m \cdot w^*_m \tag{14}$$

$$= \sum_i^m \vec{S}_i \cdot w^*_i \quad . \tag{15}$$

In other words, the line ratio vector of a galaxy can be described as a weighted sum of cloud line ratio vectors, provided the galaxy's cloud population is composed of m categories. These line ratios gives us the ability to connect variations in molecular species—and therefore, physical properties—across multiple cloud complexes. These are not implemented in the remainder of this work, but the theoretical experiments explored come to the same conclusion whether or not they take luminosity input in the form of line ratios.

4. BENCHMARKING OF POPULATION SYNTHESIS APPROACHES

4.1. *Synthetic Data Generation and Analysis*

In this work, we attempt two methods of theoretically modeling and decomposing a galaxy beam signal. For both methods, we generate synthetic cloud line luminosities $\hat{L}_{\rm cloud}$ as well as cloud population weight coefficients $\vec{w}$, giving us a galaxy line luminosity $\vec{L}_{\rm gal}$ from Eq. 9. Measurement errors $\delta\vec{L}_{\rm gal}$ and $\delta\hat{L}_{\rm cloud}$ are added to these line luminosities to produce our synthetic observations as

$$\hat{L}_{\rm cloud,obs} = \hat{L}_{\rm cloud} + \delta\hat{L}_{\rm cloud} \quad , \tag{16}$$

$$\vec{L}_{\rm gal,obs} = \vec{L}_{\rm gal} + \delta\vec{L}_{\rm gal} \quad . \tag{17}$$

The accuracy of the predicted weights outputted by our optimization are calculated as

$$w_{{\rm err},i} = \frac{|w_{{\rm model},i} - w_{{\rm obs},i}|}{w_{{\rm obs},i}} \quad , \tag{18}$$

where $\vec{w}_{\rm obs}$ is the observed weight vector and $\vec{w}_{\rm model}$ is the model weight vector. An evaluation metric (e.g. mean, median, or maximum) can be applied to $\vec{w}_{\rm err}$ to result in a single representative error value.

4.2. *Numerical Optimization*

A galaxy line luminosity model can be defined as

$$\vec{L}_{\rm model} = \hat{L}_{\rm cloud,obs} \cdot \vec{w}_{\rm model} \quad , \tag{19}$$

where $\vec{w}_{\rm model}$ are unknown cloud population weight coefficients. Minimizing the distance between a galaxy model $\vec{L}_{\rm model}$ and a synthetic observation $\vec{L}_{\rm gal,obs}$, we can approximate the unknown weights $\vec{w}_{\rm model}$. This distance can be expressed as

$$F_{\vec{L}} = (\vec{L}_{\rm gal,obs} - \vec{L}_{\rm model})^2 \quad . \tag{20}$$

This was performed using the `scipy.optimize.minimize` function by passing Eq. 16 and Eq. 17 through a minimization routine of Eq. 20, solving a system of n equations with m unknowns. The performance of this method is presented in Fig. 6, where each pixel represents the median error in the optimizer's predicted weights $\vec{w}_{\rm model}$ calculated from Eq. 18 for a given m and n value indicated on the axes.

An assumption was made that cloud population weight coefficients should be positive ($\vec{w}_{\rm model} > 0$). However, when running the minimization routine with no constraints we found negative weights were being predicted. To fix this, a

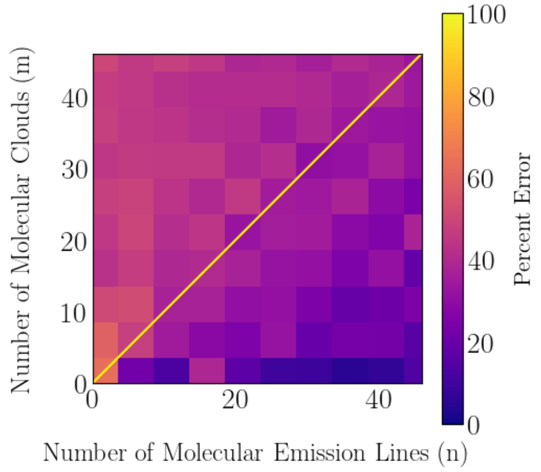


Figure 6. Minimization run for multiple experiments of galaxy observations. Ten values for m and n were passed (1, 6, 11, 16, 21, 26, 31, 36, 41, 46) through the optimization routine. The $\vec{w}_{\rm err}$ values calculated in Eq. 18 were averaged over 50 iterations for each pixel. Darker pixels represent instances where $\vec{w}_{\rm model}$ is at a farther distance from $\vec{w}_{\rm obs}$, indicating a higher percent error in the optimization.

constrained solver and bounds parameter within the minimization function were implemented. Further details of this are explored in Appendix A.

For a realistic system of 5 molecular clouds ($m = 5$) and 15 emission lines ($n = 15$), this method produced a median error of 37.95%. Furthermore, there is significantly higher median error in the upper triangle region of Fig. 6. Thus, we find that *molecular cloud population weights $\vec{w}_{obs}$ can be approximated in observations where there are more emission lines (n) than molecular clouds (m).* This finding makes sense, since our problem is essentially a high-dimensional system of linear equations, where system is only solvable when there are more constraints than unknowns.

4.3. *Probabilistic Programming*

While this galaxy line luminosity signal decomposition problem can be approached deterministically with numerical optimization, it is not a favorable method. Since we are dealing with a high-dimensional parameter space it is likely for deterministic methods to get stuck in local extrema. Here, we also approach this problem as a Bayesian linear regression problem, relying on the Markov Chain Monte Carlo (MCMC) technique with the `pymc3` Python library.

Our MCMC model was initialized with the same realistic system ($m = 5$ and $n = 15$) as with the numerical optimization method in Sec. 4.2. Priors of unknown cloud population weights $\vec{w}_{\rm model}$ and cloud luminosities $\hat{L}_{\rm cloud}$, as well as two likelihood functions from Eq. 16 and Eq. 17 were defined within the model. This setup ensured that measurement errors $\delta\hat{L}_{\rm cloud}$ and $\delta\vec{L}_{\rm gal}$ were accounted for in the determination of the true cloud population weights $\vec{w}_{\rm obs}$.

This model was run with 10,000 draws, 5,000 burn-in samples, and 3 chains. In Fig. 7, the histograms on the diagonal show the marginalized posterior densities for each model weight $w_{{\rm model},i}$. Performance of the MCMC model is shown in Fig. 8, where $\vec{w}_{\rm obs}$ and $\vec{w}_{\rm model}$ are directly compared.

When approximating the true cloud population weights $\vec{w}_{\rm obs}$, this method resulted in a median error of 5.96 % calculated with Eq. 18. We find that when accounting for uncertainties outputted by MCMC, each predicted weight $\vec{w}_{\rm model}$ falls within the range of it's corresponding true weight $\vec{w}_{\rm obs}$.

5. SUMMARY

We explore the idea of multi-line cloud population synthesis and formulate theoretical methods of decomposing a galaxy line luminosity signal into its cloud population line luminosity components. We find that it is possible to decompose an unresolved galaxy line luminosity signal in observations where there are more molecular emission lines n than molecular clouds m. This decomposition can be performed more effectively with probabilistic programming methods, such as MCMC, as they are capable of dealing with high-dimensional parameter spaces and accounting for known measurement errors in the approximation of molecular cloud population weights $\vec{w}_{\rm obs}$. In the future, these

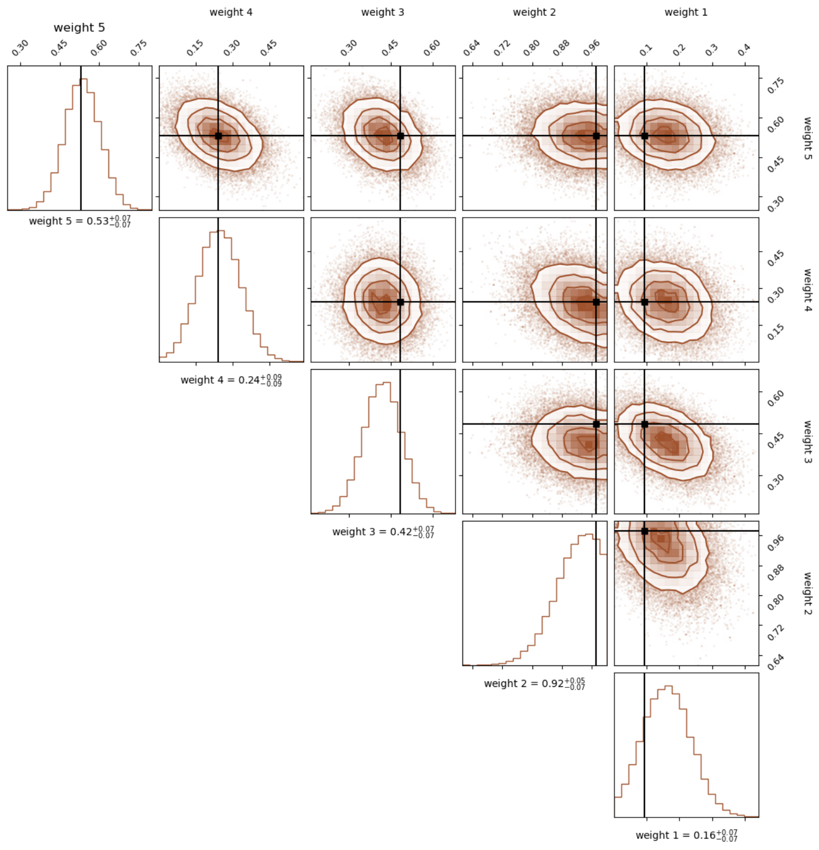


Figure 7. Corner plot of $\vec{w}_{\mathrm{model}}$, where the true weight values $\vec{w}_{\mathrm{obs}}$ are indicated by overlaid black lines. The contour levels are defined as (0.5, 1, 1.5, 2)-sigma equivalent. The levels for the corner Python module are further explained in the sigmas documentation.

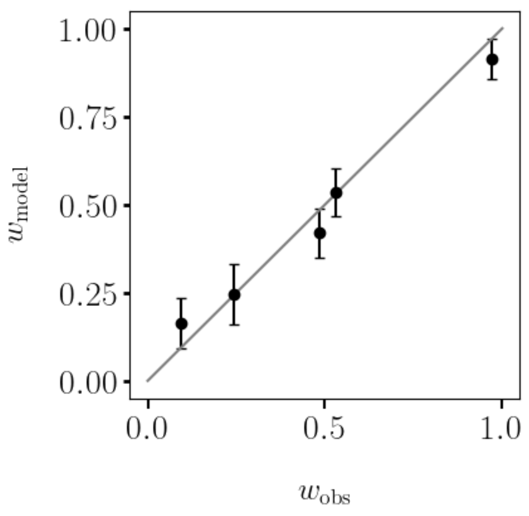


Figure 8. MCMC model performance. The standard deviation for $\vec{w}_{\mathrm{model}}$ probability is represented by error bars. The expected slope value for the scatter is shown by the grey solid line.

investigations point us in the direction of overcoming the resolution gap between local and extragalactic environments in radio astronomy.

REFERENCES

Barnes, A. T., Kauffmann, J., Bigiel, F., et al. 2020, MNRAS, 497, 1972, doi: 10.1093/mnras/staa1814

Chevance, M., Kruijssen, J. M. D., Vazquez-Semadeni, E., et al. 2020, SSRv, 216, 50, doi: 10.1007/s11214-020-00674-x

Foster, J. B., Jackson, J. M., Barnes, P. J., et al. 2011, ApJS, 197, 25, doi: 10.1088/0067-0049/197/2/25

Harada, N., Nishimura, Y., Watanabe, Y., et al. 2019, ApJ, 871, 238, doi: 10.3847/1538-4357/aaf72a

Heiderman, A., Evans, Neal J., I., Allen, L. E., Huard, T., & Heyer, M. 2010, ApJ, 723, 1019, doi: 10.1088/0004-637X/723/2/1019

Jackson, J. M., Rathborne, J. M., Foster, J. B., et al. 2013, PASA, 30, e057, doi: 10.1017/pasa.2013.37

Kauffmann, J., & Barnes, A. 2022, in European Physical Journal Web of Conferences, Vol. 265, European Physical Journal Web of Conferences, 00019, doi: 10.1051/epjconf/202226500019

Kauffmann, J., Goldsmith, P. F., Melnick, G., et al. 2017, A&A, 605, L5, doi: 10.1051/0004-6361/201731123

Lada, C. J., Lombardi, M., & Alves, J. F. 2010, ApJ, 724, 687, doi: 10.1088/0004-637X/724/1/687

Ruaud, M., Wakelam, V., Gratier, P., & Bonnell, I. A. 2018, A&A, 611, A96, doi: 10.1051/0004-6361/201731693

Shirley, Y. L. 2015, PASP, 127, 299, doi: 10.1086/680342

Watanabe, Y., Sakai, N., Sorai, K., & Yamamoto, S. 2014, ApJ, 788, 4, doi: 10.1088/0004-637X/788/1/4

APPENDIX

A. CLOUD POPULATION SYNTHESIS: IMPLEMENTATION VIA NUMERICAL OPTIMIZATION

The relative weighting between cloud populations can be derived via direct numerical optimization. This appendix documents an implementation of this analysis strategy in form of a Jupyter notebook.

Appendix_A

August 11, 2023

Appendix A

Scipy Optimize Theoretical Toy Model

The following two cells enable automatic equation numbering for this document using MathJax.

[3]:
```
%%javascript
MathJax.Hub.Config({
    TeX: { equationNumbers: { autoNumber: "AMS" } }
});
```

```
<IPython.core.display.Javascript object>
```

[4]:
```
%%javascript
MathJax.Hub.Queue(
  ["resetEquationNumbers", MathJax.InputJax.TeX],
  ["PreProcess", MathJax.Hub],
  ["Reprocess", MathJax.Hub]
);
```

```
<IPython.core.display.Javascript object>
```

[3]:
```
fig_num = 1
```

1 Fundamental problem

Observing a galaxy's line luminosity with a single-dish telescope, we can represent this value as a weighted sum over several molecular clouds. If we say these clouds fall into m cloud categories and we are observing n molecular emission lines, the $(m \times n)$-dimensional cloud luminosity can be represented by $\hat{L}_{\text{cloud}}$ and the n-dimensional galaxy luminosity can be represented as $\vec{L}_{\text{gal}}$. Thus, the galaxy's line luminosity in emission line j can be expressed as

$$L_{\text{gal},\,j} = L_{\text{a},\,j} \cdot w_{\text{a}} + ... + L_{m,\,j} \cdot w_m \quad , \tag{1}$$

or as

$$\vec{L}_{\text{gal}} = \sum_{i}^{m} \vec{L}_i \cdot w_i \tag{2}$$

$$= \hat{L}_{\text{cloud}} \cdot \vec{w} \quad , \tag{3}$$

where $\vec{w}$ is an m-dimensional vector representing the weight for each molecular cloud category. Thus, our question is *if the galaxy luminosity $\vec{L}_{\text{gal}}$ and molecular cloud luminosity $\hat{L}_{cloud}$ are known, can we determine the weight w_i that each cloud category has on the combined signal?*

In theory, this problem is a high-dimensional system of linear equations, where there are n constraints and m unknowns. So, it's reasonable to assume the weight coefficients $\vec{w}$ can be accurately predicted only when the system has fewer constraints than unknowns.

Here, we approach this question as a theoretical optimization problem, where we minimize the difference between a galaxy model $\vec{L}_{\text{model}}$ and our synthetic observations $\vec{L}_{\text{gal}}$. In other words, the "distance" between both vectors is minimized to predict weights $\vec{w}$ which produce our observations. This distance can be expressed as

$$F_{\vec{L}} = (\vec{L}_{\text{gal}} - \vec{L}_{\text{model}})^2 \quad . \tag{4}$$

2 Setting up the model

The minimization is performed using the `scipy.optimize.minimize` function. In some cases, a penalty term is desirable to ensure the minimization accurately represents system we are modeling. In this document, we implemented a penalty term $g(\vec{w})$ to satisfy $w_i \geq 0$ since negative weights can not physically exist in our system. The penalty term $g(\vec{w})$ can be added as

$$F_{\vec{L}}* = F_{\vec{L}} + g(\vec{w}) \quad , \text{ where } g(\vec{w}) = \begin{cases} 0 & w_i \geq 0 \\ \infty & w_i < 0 \end{cases} \quad . \tag{5}$$

To analyze the accuracy of the weights outputted by our optimization, we calculate the percent error between the observed weight vector $\vec{w}_{\text{obs}}$ and model weight vector $\vec{w}_{\text{model}}$ as a vector $\vec{w}_{\text{err}}$, where

$$w_{\text{err},\ i} = \frac{|w_{\text{model},\ i} - w_{\text{obs},\ i}|}{w_{\text{obs},\ i}} \quad . \tag{6}$$

An evaluation metric (e.g. mean, median, or maximum) is applied to $\vec{w}_{\text{err}}$, resulting in a single representative error value.

This document utilizes NumPy, SciPy, Astropy, Matplotlib, and tqdm libraries.

[4]:
```python
import numpy as np
import scipy.optimize as opt
from scipy.optimize import Bounds
from astropy.table import Table, vstack
import matplotlib.pyplot as plt
```

```
from tqdm import tqdm
import time
import warnings # removing warnings
warnings.filterwarnings('ignore')

# plotting specifications
from cycler import cycler
plt.rc('axes',prop_cycle=(cycler('color', ['k','b','g','r','c','m','y'])))
plt.rcParams['text.usetex']= True
plt.rcParams['mathtext.fontset']= 'custom'
plt.rcParams['mathtext.default']= 'rm'
plt.rcParams['axes.formatter.use_mathtext']=False
#-----------------------------------------
plt.rcParams['font.size']= 15.0
plt.rcParams['axes.labelsize']= 16.0
plt.rcParams['axes.unicode_minus']= False
plt.rcParams['xtick.major.size']= 6
plt.rcParams['xtick.minor.size']= 3
plt.rcParams['xtick.major.width']= 1.5
plt.rcParams['xtick.minor.width']= 1.0
plt.rcParams['axes.titlesize']= 20
plt.rcParams['xtick.labelsize'] = 20
plt.rcParams['ytick.labelsize']= 20
plt.rcParams['ytick.major.width']= 2.0
plt.rcParams['ytick.minor.width']= 1.0

%matplotlib inline
```

First, we define the function we intend to minimize (Eq. 4). An optional `penalty` term from Eq. 5 can be passed through this function.

[5]:
```
def dist_func(weights, m, gal_obs, cloud_obs, penalty):
    '''
    SUMMARY:
    Computes the Euclidean distance between observed and model galaxy␣
 ↪luminosity vectors

    PARAMETERS:
    weights (arr): m-dimensional weight vector
    m (int): number of molecular clouds
    g_luminosity (arr): n-dimensional galaxy luminosity vector
    luminosities (arr): (m x n)-dimensional molecular cloud luminosity vector
    penalty (bool): determines whether penalty variable is used for constraint
    '''
    # ---- penalty term ----
    g = 0
    if penalty:
        for i in range(m):
            if weights[i] < 0:
                g = np.inf
    # --------------------------
    lModel = np.dot(weights,cloud_obs)
    dist = np.linalg.norm(gal_obs - lModel)**2
    return dist + g
```

Next, we define a function which generates our synthetic observations satisfying Eq. 3 and performs the minimization. Synthetic observations w_i and $L_{i,j}$ are sampled from random instances within [0,1], which produce $L_{\text{gal},j}$. Measurement uncertainties $\delta L_{i,j}$ and $\delta L_{\text{gal},j}$ are sampled from random instances of a normal distribution with a standard deviation of 0.1, corrupting $L_{i,j}$ and $L_{\text{gal},j}$. These corruptions are added as

$$\hat{L}_{\text{cloud, obs}} = \hat{L}_{\text{cloud}} + \delta\hat{L}_{\text{cloud}} \quad , \tag{7}$$

$$\vec{L}_{\text{gal, obs}} = \vec{L}_{\text{gal}} + \delta\vec{L}_{\text{gal}} \quad . \tag{8}$$

These synthetic observations are not necessarily realistic, as luminosity signals normally vary within a broader range of values than [0,1]. In this work, we use small values to serve as a toy model approach to the theoretical problem.

We then pass $\vec{L}_{\text{gal, obs}}$ and $\hat{L}_{\text{cloud, obs}}$ on a routine that predicts $\vec{w}$ through the minimization of Eq. 4. There is an optional `bounds_constraint` term passed through this minimization function. This is an alternative way of applying a penalty, where bounds on $\vec{w}_{\text{model}}$ are applied directly within the `scipy.optimize.minimize` function.

```
[6]: def minimize(m, n, solver, bounds_constraint, penalty):
         '''
         SUMMARY:
         Generates synthetic observation
         Determines corresponding weights for this observation

         PARAMETERS:
         m (int): number of molecular clouds
         n (int): number of emission lines
         solver (str): type of SciPy minimize solver
         bounds_constraint (bool): determines whether bounds are used for constraint
         penalty (bool): determines whether penalty variable is used for constraint
         '''
         # true values
         cloud_luminosity = np.random.rand(m,n)
         weights = np.random.rand(m)
         gal_luminosity = np.dot(weights, cloud_luminosity)
         # known uncertainty
         cloud_err = np.random.normal(0, 0.1, size=(m,n))
         gal_err = np.random.normal(0, 0.1, size=(n))
```

```
    # observations
    cloud_obs = np.add(cloud_luminosity, cloud_err)
    gal_obs = np.add(gal_luminosity, gal_err)

    if bounds_constraint:
        # applying constraint to weights
        ub = np.full(m, 1)
        lb = np.full(m, 0)
        bnds = Bounds([lb],[ub])
        # minimization
        result = opt.minimize(dist_func,
                              x0=np.full(m, 0.5),
                              args=(m, gal_obs, cloud_obs, penalty),
                              method = solver,
                              bounds=bnds)
    else:
        # minimization
        result = opt.minimize(dist_func,
                              x0=np.full(m, 0.5),
                              args=(m, gal_obs, cloud_obs, penalty),
                              method = solver)
    return {"input_luminosities": cloud_luminosity,
            "input_weights": weights,
            "galaxy_luminosity": gal_luminosity,
            "output_weights": result.x,
            "F(l)": result.fun}
```

To calculate how far off $\vec{w}_{\text{model}}$ is from $\vec{w}_{\text{obs}}$, we define the following function. An evaluation metric term `eval_metric` is applied to the error vector $\vec{w}_{\text{err}}$ outputted by Eq. 6, resulting in a single representative error value.

[7]:
```
def getPercentError(original, test, eval_metric):
    '''
    SUMMARY:
    Applies specified percent error metric to weights

    PARAMETERS:
    original (arr): initial weight vector
    test (arr): new weight vector
    eval_metric (str): specified statistical metric
    '''
    absolute = np.absolute(np.subtract(test, original))
    divided_absolute = np.divide(absolute, original) * 100

    if eval_metric == "mean":
        averaged_divided_absolute = np.average(divided_absolute)
    elif eval_metric == "max":
        averaged_divided_absolute = np.amax(divided_absolute)
    elif eval_metric == "median":
        averaged_divided_absolute = np.median(divided_absolute)
    return averaged_divided_absolute
```

Then, we define a function which runs a single instance of m and n through `minimize` iteratively. The `iterate` term determines the number of times the experiment is performed. Averaging over several iterations of the experiment, we reduce the chances of outlier results. In other words, a higher `iterate` term gives a more accurate representation of error in the model.

The output `summarized_data` is given as a table of a single row containing the inputted m and n values, the average minimized distance calculated in Eq. 4, and the average of various characterizations of the model weight error $\vec{w}_{\mathrm{err}}$. The `solver` term specifies the type of minimization method used. There are numerous SciPy Optimize Minimize solvers, some of which support constrainted optimization (e.g. COBYLA and SLSQP).

[8]:
```
def run_single(m, n, iterate, solver, bounds_constraint, penalty, filename):
    '''
    SUMMARY:
    Calls minimize function for a single experiment
    Creates table and .ecsv file of data run

    PARAMETERS:
    m (int): number of molecular clouds
    n (int): number of emission lines
    iterate (int): # of iterations
    solver (str): type of SciPy minimize solver
    bounds_constraint (bool): determines whether bounds are used for constraint
    penalty (bool): determines whether penalty variable is used for constraint
    filename (str): .ecsv filename
    '''
    time_start = time.time()

    # initializing table and lists
    summarized_data = Table([])
    min_dist = []
    mean_error = []
    median_error = []

    for k in range(iterate):
        result = minimize(m, n, solver, bounds_constraint, penalty)
        # appending each iteration to existing lists
        min_dist.append(result["F(l)"])
        mean_error.append(getPercentError(result["input_weights"],
                                          result["output_weights"],
                                          "mean")
                          )
        median_error.append(getPercentError(result["input_weights"],
                                                          result["output_weights"],
                                                          "median")
                                  )
    # saving lists to table
    summarized_data['m'] = [m]
    summarized_data['n'] = [n]
    summarized_data['F(l)'] = [np.average(min_dist)]
    summarized_data['Percent Error Mean'] = [np.average(mean_error)]
    summarized_data['Percent Error Median'] = [np.average(median_error)]

    # saving and retrieving run in *.ecsv* file
    summarized_data.write(f"{filename}.ecsv", overwrite=True)
    summarized_data = Table.read(f"{filename}.ecsv")
    print('\nTime elapsed: {} seconds'\
          .format(round((time.time() - time_start),2)))
    return summarized_data
```

In a similar manner, we define a function which runs multiple instances of m and n through `minimize` iteratively, giving us the ability to test multiple experiements simultaneously. The `m_n_array` term is a specified array of m and n values. The function iteratively runs every possible combination of numerical pairs in `m_n_array` through `minimize`.

The output term `summarized_data` is given as a table, where each row contains the inputted m and n value, the average minimized distance calculated in Eq. 4, and the average of various characterizations of the model weight error $\vec{w}_{\text{err}}$. Another output term `data` contains the unaveraged data, where each row contains information for each iteration.

[9]:
```
def run_multiple(m_n_array, iterate, solver, bounds_constraint, penalty,␣
 ↪filename):
    '''
    SUMMARY:
    Calls minimize function for multiple experiments
    Creates table and .ecsv file of data run

    PARAMETERS:
    m_n_array (arr): m and n values
    iterate (int): # of iterations
    solver (str): type of SciPy minimize solver
    bounds_constraint (bool): determines whether bounds are used for constraint
    penalty (bool): determines whether penalty variable is used for constraint
    filename (str): .ecsv filename
    '''
    data = Table([])
    time_start = time.time()
    pbar = tqdm(desc="Progress", total = len(m_n_array)**2*iterate)
    tqdm._instances.clear()

    # running iterations for each unique m and n combination
    for num_molecular_clouds in m_n_array:
        for num_line in m_n_array:
            for k in range(iterate):
                result = minimize(num_molecular_clouds,
                                  num_line,
                                  solver,
                                  bounds_constraint,
                                  penalty
                                 )
                experiment_data = Table([])
                experiment_data["m"] = [num_molecular_clouds]
                experiment_data["n"] = [num_line]
                experiment_data["input_luminosities"] = \
                    [np.array(str({"data": np.
↪array(result["input_luminosities"])}), dtype=str)]
                experiment_data["input_weights"] = \
                    [np.array(str({"data": np.array(result["input_weights"])}),␣
↪dtype=str)]
                experiment_data["output_weights"] = \
                    [np.array(str({"data": np.
↪array(result["output_weights"])}), dtype=str)]
                experiment_data["F(l)"] = [result["F(l)"]]
                experiment_data["Percent Error Mean"] = \
                    [getPercentError(result["input_weights"],␣
↪result["output_weights"], "mean")]
                experiment_data["Percent Error Median"] = \
                    [getPercentError(result["input_weights"],␣
↪result["output_weights"], "median")]
                experiment_data["Percent Error Max"] = \
                    [getPercentError(result["input_weights"],␣
↪result["output_weights"], "max")]
                data = vstack([data, experiment_data])
                pbar.update(1)

    # averaging over iterations
    summarized_data = Table([])
    for i in range(0, len(data), iterate):
        row = Table([])
        row["m"] = [data["m"][i]]
        row["n"] = [data["n"][i]]
        row["F(l)"] = \
            [np.average(data["F(l)"][i:i+iterate].data)]
        row["Percent Error Mean"] = \
            [np.average(data["Percent Error Mean"][i:i+iterate].data)]
        row["Percent Error Median"] = \
            [np.average(data["Percent Error Median"][i:i+iterate].data)]
        row["Percent Error Max"] = \
            [np.average(data["Percent Error Max"][i:i+iterate].data)]
        summarized_data = vstack([summarized_data, row])

    # saving and retrieving run in *.ecsv* file
    summarized_data.write(f"{filename}.ecsv", overwrite=True)
    summarized_data = Table.read(f"{filename}.ecsv")
    print('\nTime elapsed: {} seconds'\
          .format(round((time.time() - time_start),2)))
    return summarized_data, data
```

Lastly, a plotting function is defined using `pcolormesh` to visualize the output of `run_multiple` in one figure. Each data point in the plot contains a row within `summarized_data`. In other words, each "pixel" represents an experiment performed with a given m and n value, averaged over a given amount of iterations.

[10]:
```
def plot_colormesh(m_n_array, summarized_data, evaluation_metric, fig_num,␣
 ↪description):
    '''
    SUMMARY:
    Creates a colormesh plot of data run

    PARAMETERS:
    m_n_array (arr): m and n values
    summarized_data (astropy.table): table containing data run
    evaluation_metric (str): specified statistical metric
    fig_num (str): running figure number
    description (str): displayed figure description
    '''
    plt.style.use('_mpl-gallery-nogrid')
    fig, ax = plt.subplots(figsize=(4,4))

    input_length = len(m_n_array)
    graph = ax.pcolormesh(np.array(np.reshape(summarized_data["n"],
                                              [input_length, input_length]
                                             )),
                          np.array(np.reshape(summarized_data["m"],
                                              [input_length, input_length]
                                             )),
                          np.array(np.
 ↪reshape(summarized_data[evaluation_metric],
                                              [input_length, input_length]
                                             )),
                          shading='nearest',
                          vmin=0,
                          vmax=100,
                          cmap='plasma'
```

```
                            )
    ax.set(xlim=(0, max(m_n_array)), ylim=(0, max(m_n_array)))
    ax.set_aspect('equal')
    cb = plt.colorbar(graph)
    cb.set_label(r'\rm{}Percent Error')
    x1, y1 = [0, max(m_n_array)], [0, max(m_n_array)]
    plt.plot(x1, y1, color="yellow")
    plt.xlabel(r"\rm{}Number of Molecular Emission Lines (n)", fontsize=18,␣
↪labelpad=18)
    plt.ylabel(r"\rm{}Number of Molecular Clouds (m)", fontsize=18, labelpad=18)
    # include figure caption
    fig.text(0.43, -.2, f'Fig. {fig_num} {description}', fontsize=12,␣
↪ha='center')
    plt.show()
    return fig_num + 1
```

3 Testing the model

Here, we run a single experiment with 5 molecular clouds and 15 emission lines. This experiment is averaged over 100 iterations.

[11]:
```
m = 5
n = 15
iterate = 100
solver = 'COBYLA'
bounds_constraint = True
penalty = False
filename = 'my_data'
```

[12]:
```
summarized_data = run_single(m,
                             n,
                             iterate,
                             solver,
                             bounds_constraint,
                             penalty,
                             filename
                            )
```

```
Time elapsed: 0.28 seconds
```

[13]:
```
summarized_data
```

[13]:
```
<Table length=1>
  m     n          F(l)        Percent Error Mean Percent Error Median
int64 int64      float64             float64             float64
----- ----- ------------------ ------------------ --------------------
```

```
    5    15 0.2581621583853951  94.25911089620668   25.874628581476976
```

Here, we run multiple experiments with 100 possible combinations of molecular cloud and emission line values. Each experiments is averaged over 100 iterations.

[14]:
```
m_n_array = np.arange(1,51,5)  # multiple combinations
iterate = 50                   # num of iterations over each combination
solver = 'COBYLA'
bounds_constraint = True
penalty = False
filename = 'my_data'
description = 'multi-experiment run of minimization'
print(len(m_n_array),'values for m and n: \n', m_n_array,end='\n\n')
```

```
10 values for m and n:
 [ 1  6 11 16 21 26 31 36 41 46]
```

[15]:
```
summarized_data, data  = run_multiple(m_n_array,
                                      iterate,
                                      solver,
                                      bounds_constraint,
                                      penalty,
                                      filename
                                     )
```

```
Progress: 100%|                | 5000/5000 [16:20<00:00,  5.10it/s]


Time elapsed: 980.56 seconds
```

[16]:
```
summarized_data[10:20]
```

[16]:
```
<Table length=10>
  m     n            F(l)          … Percent Error Median Percent Error Max
int64 int64        float64         …        float64             float64
----- ----- ---------------------- … -------------------- ------------------
    6     1 1.2984842983410934e-08 …    58.50075472174936 1153.7549920978406
    6     6   0.013054962256854754 …    70.50912657322493  494.6086290458383
    6    11     0.1424795021984006 …   39.558640064118535   660.097079862168
    6    16    0.24951989940237163 …   29.843910010864143  343.3090034155021
    6    21     0.4554501125753825 …    23.94717872995079  335.2891052463472
    6    26     0.5760688814743458 …    19.79142992354043 232.86348451603962
    6    31      0.769484691544954 …   19.073205100588776  287.4755745331578
    6    36     0.9296607900118322 …   14.573214042293095  494.9957705032012
    6    41     1.0299895534754735 …   13.699098016764589  425.5225278245356
    6    46     1.1644388701248318 …   13.639436560960299 155.05843309157765
```

```
[17]: fig_num = plot_colormesh(m_n_array,
                             summarized_data,
                             "Percent Error Median",
                             fig_num,
                             description
                            )
```

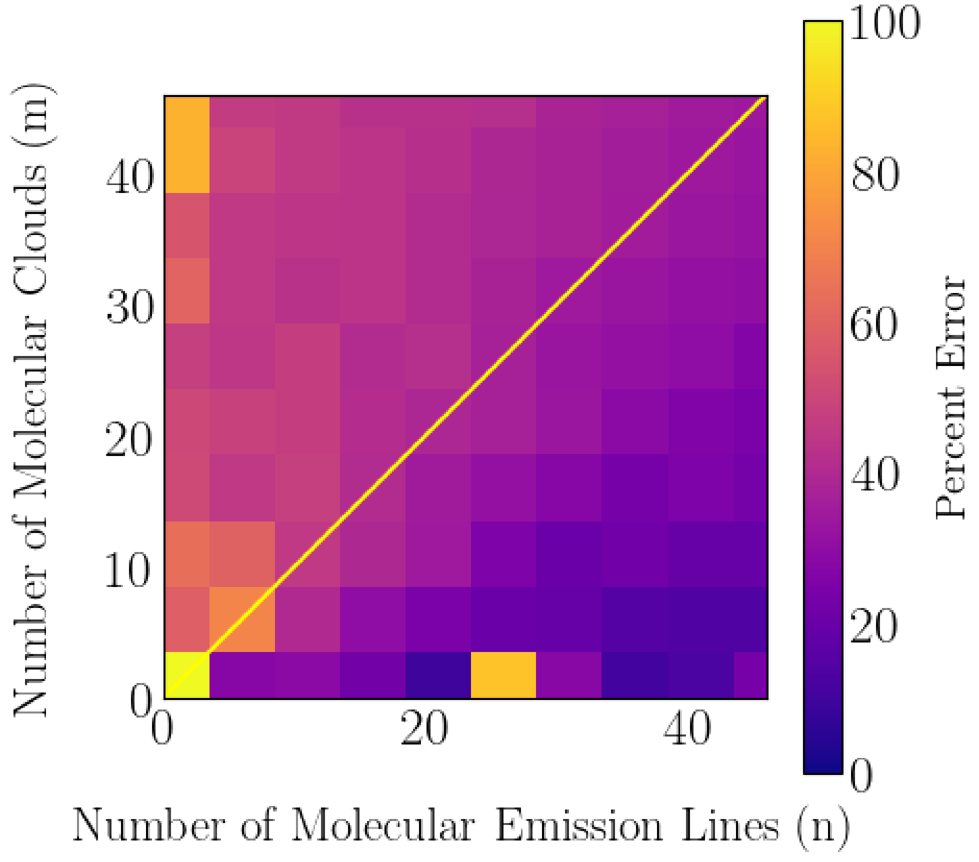


Fig. 1 multi-experiment run of minimization

In Fig. 1, the darker pixels represent instances of where $\vec{w}_{\text{model}}$ is at a farther distance from $\vec{w}_{\text{obs}}$, indicating a higher percent error in the model. There is significantly higher error in the upper triangle region. The predicted weights $\vec{w}_{\text{model}}$ have a higher uncertainty where experiments have more molecular clouds (m) than emission lines (n). As originally predicted, our results are supported by the backbone for systems of linear equations: the system is unsolvable when there are fewer constraints than unknowns.

4 Implementation Notes

While testing this minimization routine with a different optimization solver, we noticed an odd feature in the pcolormesh plot. A slope of significantly high error resides between our upper and lower triangle regions.

```
[18]: solver = 'BFGS'
      bounds_constraint = False
      penalty = False
      iterate = 10
      description = 'unconstrained run of minimization'
      summarized_data, data  = run_multiple(m_n_array,
                                            iterate, solver,
                                            bounds_constraint,
                                            penalty,
                                            filename
                                           )
```

```
Progress: 100%|                | 1000/1000 [00:45<00:00, 22.12it/s]


Time elapsed: 45.21 seconds
```

```
[19]: fig_num = plot_colormesh(m_n_array,
                               summarized_data,
                               "Percent Error Median",
                               fig_num,
                               description
                              )
```

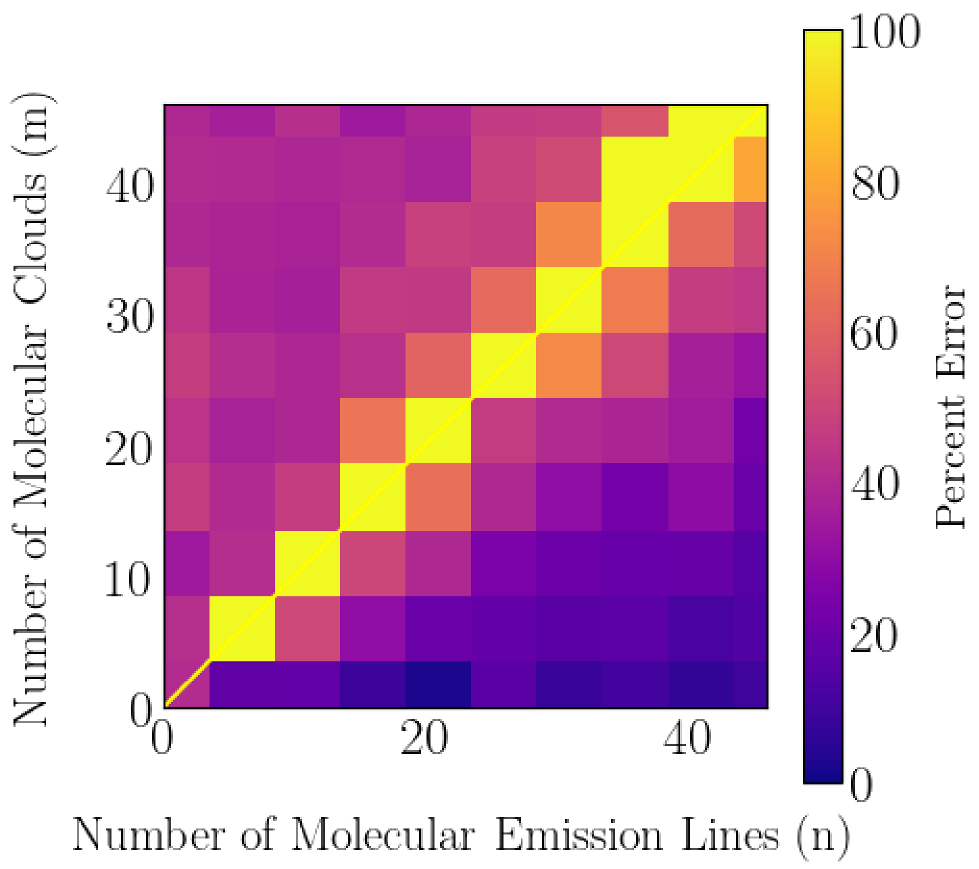


Fig. 2 unconstrained run of minimization

Upon inspection of the `data` output below, we noticed negative weights in the `output_weights` column were being predicted by the minimization routine. This is an issue since a model galaxy with negative weights is unphysical.

[20]:
```
data[110:120:30]
```

[20]:
```
<Table length=1>
  m     n    … Percent Error Median Percent Error Max
int64 int64 …       float64              float64
----- ----- … -------------------- -----------------
    6     6 …   18.329002720177602 605.7715285522087
```

One solution to this is to use a contrained solver and add a penalty term (Eq. 5) within the function we are minimizing (Eq. 4)

[21]:
```
solver = 'COBYLA'
bounds_constraint = False
penalty = True
```

```
description = 'penalty run of minimization'
summarized_data, data = run_multiple(m_n_array,
                                     iterate,
                                     solver,
                                     bounds_constraint,
                                     penalty,
                                     filename
                                    )
```

```
Progress: 100%|                | 1000/1000 [00:54<00:00, 18.46it/s]


Time elapsed: 54.17 seconds
```

```
[22]: fig_num = plot_colormesh(m_n_array,
                              summarized_data,
                              "Percent Error Median",
                              fig_num,
                              description
                             )
```

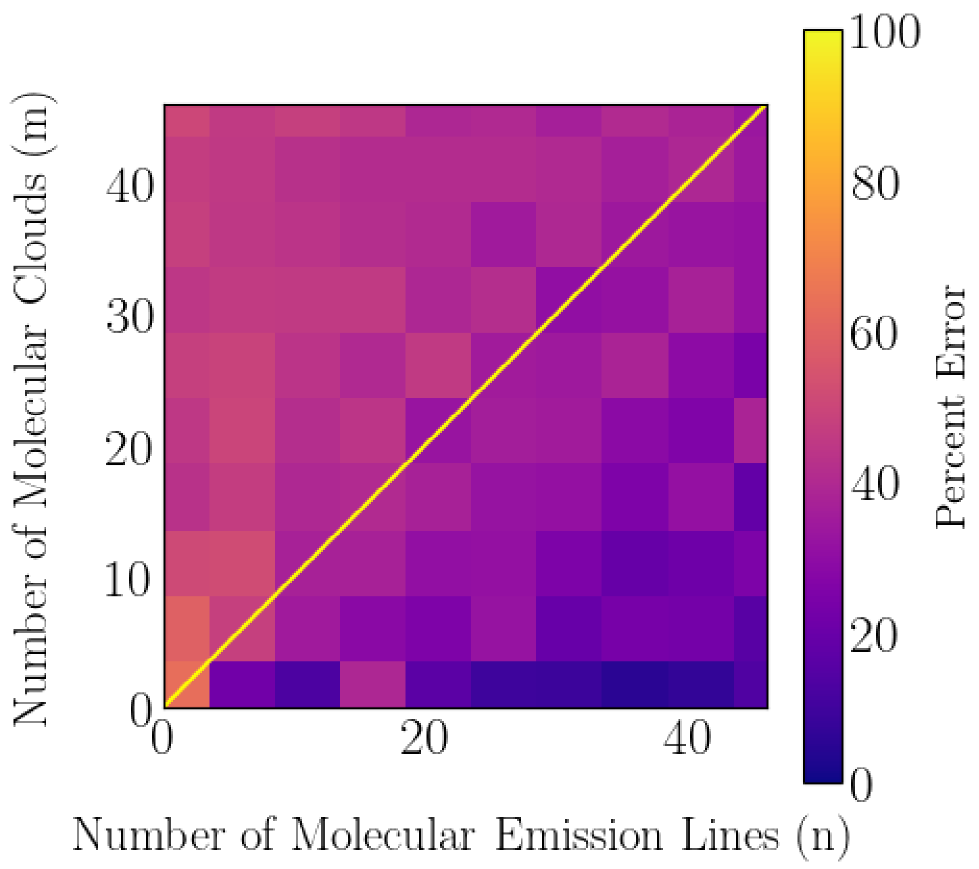


Fig. 3 penalty run of minimization

Another solution is to use a constrained solver and adjust the bounds directly within the `scipy.optimize.minimize` function.

[23]:
```
bounds_constraint = True
penalty = False
description = 'bounds constraint run of minimization'
summarized_data, data  = run_multiple(m_n_array,
                                      iterate,
                                      solver,
                                      bounds_constraint,
                                      penalty,
                                      filename
                                     )
```

```
Progress: 100%|                 | 1000/1000 [00:59<00:00, 16.78it/s]


Time elapsed: 59.59 seconds
```

[24]:
```
fig_num = plot_colormesh(m_n_array,
                         summarized_data,
                         "Percent Error Median",
                         fig_num,
                         description
                        )
```

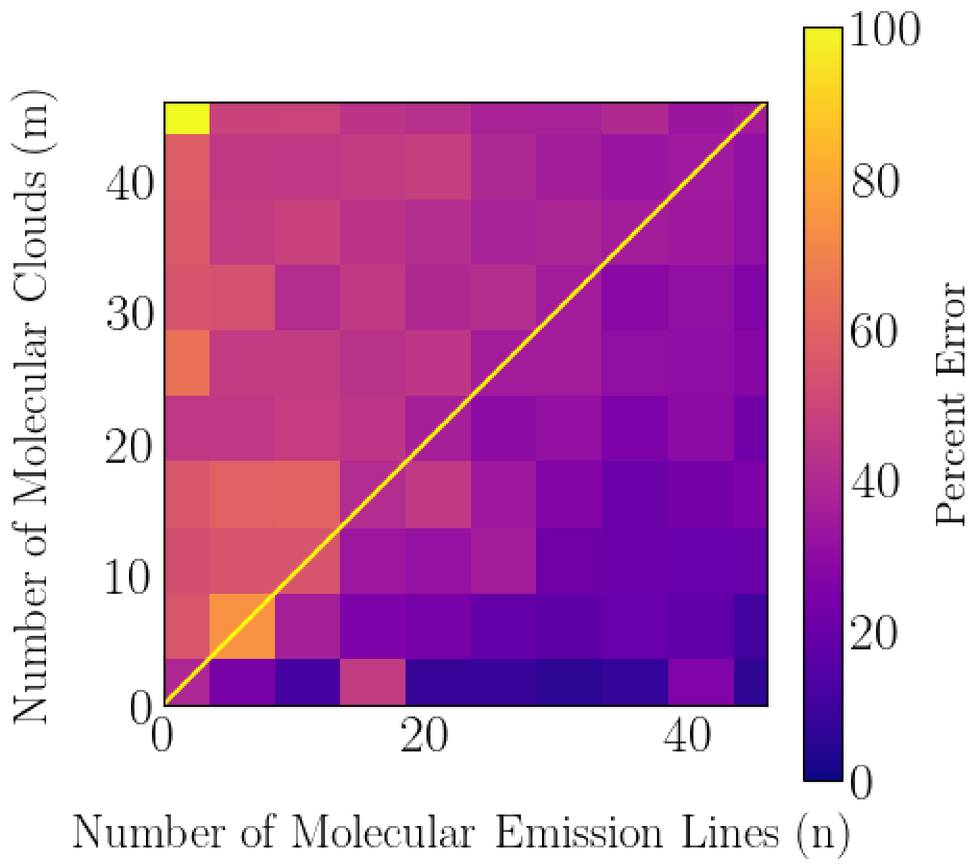


Fig. 4 bounds constraint run of minimization

Comparing these two solutions shown in Fig. 3 and Fig. 4, both are able to remove the unwanted negative weights. However, the bounds method is more desirable due to the penalty term having the potential to cause issues within the optimization function's derivative computations over Eq. 4.

B. CLOUD POPULATION SYNTHESIS: IMPLEMENTATION VIA PROBABILISTIC PROGRAMMING

The relative weighting between cloud populations can be derived via a Bayesian analysis of probabilities, as implemented via Markov Chain Monte Carlo (MCMC) calculations. This appendix documents an implementation of this analysis strategy in form of a Jupyter notebook.

Appendix_B

August 2, 2023

Appendix B

MCMC Theoretical Toy Model

The following two cells enable automatic equation numbering for this document using MathJax .

[1]:
```
%%javascript
MathJax.Hub.Config({
    TeX: { equationNumbers: { autoNumber: "AMS" } }
});
```

```
<IPython.core.display.Javascript object>
```

[2]:
```
%%javascript
MathJax.Hub.Queue(
  ["resetEquationNumbers", MathJax.InputJax.TeX],
  ["PreProcess", MathJax.Hub],
  ["Reprocess", MathJax.Hub]
);
```

```
<IPython.core.display.Javascript object>
```

[3]:
```
fig_num = 1
```

1 Fundamental problem

Observing a galaxy's line luminosity with a single-dish telescope, we can represent this value as a weighted sum over several molecular clouds. If we say these clouds fall into m cloud categories and we are observing n molecular emission lines, the $(m \times n)$-dimensional cloud luminosity can be represented by $\hat{L}_{\mathrm{cloud}}$ and the n-dimensional galaxy luminosity can be represented as $\vec{L}_{\mathrm{gal}}$. Thus, the galaxy's line luminosity in emission line j can be expressed as

$$L_{\mathrm{gal},\,j} = w_{\mathrm{a}} \cdot L_{\mathrm{a},\,j} + ... + w_m \cdot L_{m,\,j} \quad , \tag{1}$$

or as

$$\vec{L}_{\text{gal}} = \sum_{i}^{m} w_i \cdot \vec{L}_i \tag{2}$$
$$= \hat{L}_{\text{cloud}} \cdot \vec{w} \quad , \tag{3}$$

where $\vec{w}$ is an m-dimensional vector representing the weight for each molecular cloud category. Thus, our question is *if the galaxy luminosity $\vec{L}_{\text{gal}}$ and molecular cloud luminosity $\hat{L}_{cloud}$ are known, can we determine the weight w_i that each cloud category has on the combined signal?*

Here, we approach this question as a theoretical Bayesian linear regression problem, where we rely on the Markov Chain Monte Carlo (MCMC) technique with `pymc3`. Using this method, the weight parameters to which the model eventually converges ($\vec{w}_{\text{model}}$) should approximate the synthetic weights we generate ($\vec{w}_{\text{obs}}$).

While this problem can be approached deterministically with numerical optimization (see Appendix A), it is not a favorable method. Since we are dealing with a high-dimensional parameter space, it is likely for deterministic methods to get stuck in local minima or maxima. By treating the problem as a sampling task, MCMC methods can more efficiently explore a parameter space, reach convergence to $\vec{w}_{\text{model}} \approx \vec{w}_{\text{obs}}$, and provide uncertainties for $\vec{w}_{\text{model}}$.

2 Setting up the model

This document utilizes NumPy, math, Matplotlib, PyMC3, ArviZ, and corner.py libraries.

[4]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import pymc3 as pm
import arviz as az
import corner
import warnings # removing warnings
warnings.filterwarnings('ignore')

# plotting specifications
from cycler import cycler
plt.rc('axes',prop_cycle=(cycler('color', ['k','b','g','r','c','m','y'])))
plt.rcParams['text.usetex']= True
plt.rcParams['mathtext.fontset']= 'custom'
plt.rcParams['mathtext.default']= 'rm'
plt.rcParams['axes.formatter.use_mathtext']=False
#-----------------------------------------
plt.rcParams['font.size']= 15.0
plt.rcParams['axes.labelsize']= 16.0
plt.rcParams['axes.unicode_minus']= False
plt.rcParams['xtick.major.size']= 6
plt.rcParams['xtick.minor.size']= 3
plt.rcParams['xtick.major.width']= 1.5
plt.rcParams['xtick.minor.width']= 1.0
plt.rcParams['axes.titlesize']= 20
plt.rcParams['xtick.labelsize'] = 20
plt.rcParams['ytick.labelsize']= 20
plt.rcParams['ytick.major.width']= 2.0
plt.rcParams['ytick.minor.width']= 1.0

%matplotlib inline
```

First, we generate synthetic observations satisfying Eq. 3. Sampling w_i and $L_{i,j}$ from random instances within [0,1], we produce $L_{\mathrm{gal},j}$. Measurement uncertainties $\delta L_{i,j}$ and $\delta L_{\mathrm{gal},j}$ are sampled from random instances of a normal distribution with a standard deviation of 0.1, corrupting $L_{i,j}$ and $L_{\mathrm{gal},j}$. These corruptions are added as

$$\hat{L}_{\text{cloud, obs}} = \hat{L}_{\text{cloud}} + \delta\hat{L}_{\text{cloud}} \quad \text{and} \tag{4}$$

$$\vec{L}_{\text{gal, obs}} = \vec{L}_{\text{gal}} + \delta\vec{L}_{\text{gal}} \quad . \tag{5}$$

Values for m and n are realistically defined for a given observation (5 cloud categories would reasonably be found within a ~100 pc region of a galaxy). However, the synthetic observations are not necessarily realistic, as luminosity signals normally vary within a broader range of values than [0,1]. In this work, we use small values to serve as a toy model approach to the theoretical problem.

```
[5]: np.random.seed(55) # reproducible results
m = 5               # number of molecular cloud categories
n = 15              # number of emission lines

# true values
weights_true = np.random.uniform(0, 1, size=m)
cloud_luminosity = np.random.uniform(0, 1, size=(m,n))
gal_luminosity = np.dot(weights_true, cloud_luminosity)

# known uncertainty
cloud_sd = 0.1
gal_sd = 0.1
cloud_err = np.random.normal(0, cloud_sd, size=(m,n))
gal_err = np.random.normal(0, gal_sd, size=n)

# observations
cloud_obs = cloud_luminosity + cloud_err
gal_obs = np.dot(weights_true, cloud_obs) + gal_err
```

Using the `pymc3` package, we create an instance of the `pm.Model()` object, which is assigned to a variable called `model`.

```
[6]: model = pm.Model()
```

Modifying our `model` instance, a prior distribution is created with assumptions about the unobserved parameters. Our model should encode a hypothesis about how our observations are generated, which in this case is identical to the generation of our synthetic data. Thus, $\vec{w}_{\text{model}}$ and the cloud luminosity $\hat{L}_{\text{cloud}}$ are initialized in the same manner (uniformly within the range [0,1]) as our observations.

```
[7]: with model:
         # ----PRIORS----
         weights = pm.Uniform('weights',
                              lower=0,
                              upper=1,
                              shape=m
                             )
         cloud_luminosity_mu = pm.Uniform('cloud_luminosity_mu',
                                          lower=0,
                                          upper=1,
                                          shape=(m,n)
                                         )
```

Next, two likelihood functions (Eq. 4 and Eq. 5) are defined within our model, in which Eq. 5 calls the output of Eq. 4. This ensures our measurement uncertainties $\delta\hat{L}_{\text{cloud}}$ and $\delta\vec{L}_{\text{gal}}$ are both accounted for in the determination of $\vec{w}_{\text{model}}$.

```
[8]: with model:
         # ----LIKELIHOOD FUNCTIONS----
         cloud_luminosity_obs = pm.Normal('cloud_luminosity_obs',
                                          mu=cloud_luminosity_mu,
                                          sd=cloud_sd,
                                          shape=(m,n),
                                          observed=cloud_obs
                                         )
         gal_luminosity_obs = pm.Normal('gal_luminosity_obs',
                                        mu=pm.math.dot(weights,␣
      ↪cloud_luminosity_obs),
                                        sd=gal_sd,
                                        shape=n,
                                        observed=gal_obs
                                       )
```

To analyze the accuracy of $\vec{w}_{\text{model}}$, we calculate the percent error between the observed weight vector $\vec{w}_{\text{obs}}$ and model weight vector $\vec{w}_{\text{model}}$ as a vector $\vec{w}_{\text{err}}$, where

$$w_{\text{err},\ i} = \frac{|w_{\text{model},\ i} - w_{\text{obs},\ i}|}{w_{\text{obs},\ i}} \quad . \tag{6}$$

An evaluation metric (e.g. mean, median, or maximum) term `eval_metric` is applied to the error vector $\vec{w}_{\text{err}}$ outputted by Eq. 6, resulting in a single representative error value.

```
[9]: def getPercentError(original, test, eval_metric):
         '''
         SUMMARY:
         Applies specified percent error metric to weights

         PARAMETERS:
         original (arr): initial weight vector
         test (arr): new weight vector
         evaluation_metric (str): specified statistical metric
         '''
         absolute = np.absolute(np.subtract(test, original))
         divided_absolute = np.divide(absolute, original) * 100

         if eval_metric == "mean":
             averaged_divided_absolute = np.average(divided_absolute)
         elif eval_metric == "max":
             averaged_divided_absolute = np.amax(divided_absolute)
         else:
             averaged_divided_absolute = np.median(divided_absolute)
         return averaged_divided_absolute
```

3 Testing the model

To test our model, MCMC samples from a random distribution and eventually converges to the stationary (model) distribution. Since we do not want to collect samples while the model is still reaching convergence, `tune` and `draws` terms are defined. By setting `tune` to a number, we require the Markov chain to converge to the stationary distribution for that given number of iterations before collecting any `draws`. Thus, only samples taken during the `draws` period are kept. However, sometimes the Markov chain does not converge within the given number of samples (`tune` + `draws`). To check for convergence, we run multiple `chains` simultaneously to ensure the parameter space is well explored. Each chain starts from a different initial state and independently explores different regions of the parameter space.

An MCMC algorithm is specified to collect these samples. However, it's worth noting that not all algorithms use the same method of exploration. For instance, the Metropolis-Hastings algorithm utilizes "random walk" behavior, where a proposed exploration of a new point within a given parameter space is made by perturbing the current point randomly. This method is effective for exploring low-dimensional parameter spaces.

In this work, the No-U-Turn Sampler (NUTS) method is used. This is a more sophisticated algorithm which avoids random walk. It uses Hamiltonian dynamics to propose explorations, by adaptively tuning the step size and direction of motion. It also detects if the future trajectory will turn back on itself, effectively avoiding over-exploration of areas in the parameter space.

Below, we set the model to burn through 5,000 `tune` samples, collect 10,000 `draws`, and run 3 `chains` using the NUTS sampler.

[10]:
```
with model:
    # ----SAMPLING----
    trace = pm.sample(draws=10000,
                      tune=5000,
                      chains=3,
                      return_inferencedata=True)
```

```
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (3 chains in 4 jobs)
NUTS: [cloud_luminosity_mu, weights]

<IPython.core.display.HTML object>

Sampling 3 chains for 5_000 tune and 10_000 draw iterations (15_000 + 30_000
draws total) took 19 seconds.
```

Here, we save our results from the posterior as variables, display the summary of the trace for each weight $w_{\text{model, } i}$, and calculate the mean value of our error vector $\vec{w}_{\text{err}}$.

[11]:
```
weights = trace.posterior.weights
weight_vals = trace.posterior.weights.values

weights_predicted = []
weight_predicted_err = []
for i in range(m):
    weights_predicted.append(np.mean(weight_vals[:,:,i]))
    weight_predicted_err.append(np.std(weight_vals[:,:,i]))
```

[12]:
```
az.summary(trace)
```

[12]:
```
                         mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  \
weights[0]              0.162  0.072   0.022    0.289      0.000      0.0
weights[1]              0.914  0.057   0.815    1.000      0.000      0.0
weights[2]              0.420  0.069   0.290    0.550      0.000      0.0
weights[3]              0.245  0.086   0.084    0.407      0.000      0.0
weights[4]              0.535  0.067   0.411    0.664      0.000      0.0
…                         …      …       …        …          …        …
cloud_luminosity_mu[4,10]  0.732  0.099   0.549    0.921      0.001      0.0
cloud_luminosity_mu[4,11]  0.799  0.093   0.636    0.981      0.001      0.0
cloud_luminosity_mu[4,12]  0.091  0.066   0.000    0.206      0.000      0.0
cloud_luminosity_mu[4,13]  0.869  0.081   0.732    1.000      0.000      0.0
cloud_luminosity_mu[4,14]  0.578  0.099   0.390    0.762      0.000      0.0

                         ess_bulk  ess_tail  r_hat
weights[0]                28026.0   13182.0    1.0
weights[1]                29504.0   16344.0    1.0
weights[2]                39685.0   20255.0    1.0
weights[3]                34814.0   14766.0    1.0
```

```
weights[4]                  43606.0   20955.0    1.0
…                             …         …     …
cloud_luminosity_mu[4,10]   26019.0   10922.0    1.0
cloud_luminosity_mu[4,11]   28076.0   12738.0    1.0
cloud_luminosity_mu[4,12]   27935.0   15146.0    1.0
cloud_luminosity_mu[4,13]   24971.0   11887.0    1.0
cloud_luminosity_mu[4,14]   49732.0   18253.0    1.0

[80 rows x 9 columns]
```

[13]:
```python
print('Mean Error:',
      round(getPercentError(weights_true, weights_predicted, 'mean'),2),'%')
print('Median Error:',
      round(getPercentError(weights_true, weights_predicted, 'median'),2),'%')
```

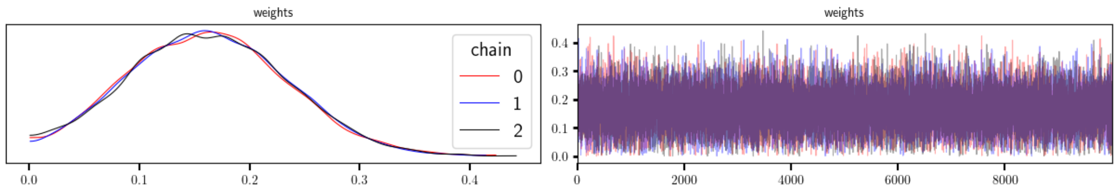
```
Mean Error: 19.07 %
Median Error: 5.96 %
```

For this model, the mean error as calculated in Eq. 6 is ~19% and the median error is ~6%. For comparison, numerical optimization performed in Appendix A under similar conditions (m = 5, n = 15, and standard deviations of 0.1) yielded a consistent median error of ~30%.

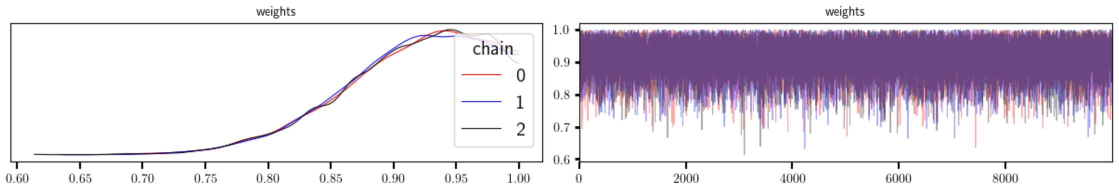
4 Posterior analysis

[14]:
```python
print(f'Fig. {fig_num}: weight traces',end='\n\n')
for i in range(weight_vals.shape[2]):
    print('weight',i+1,':',np.round(weights_predicted[i],4))
    az.plot_trace(weights[:,:,i],
                  chain_prop={'color': ['C3','C1','C7']},
                  legend=True
                 )
    plt.show()
fig_num += 1
```

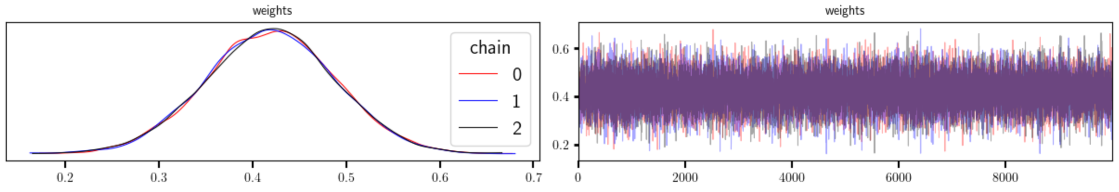
```
Fig. 1: weight traces

weight 1 : 0.1623
```



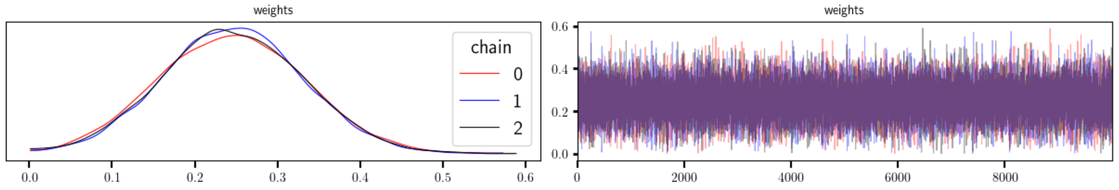
```
weight 2 : 0.9138
```



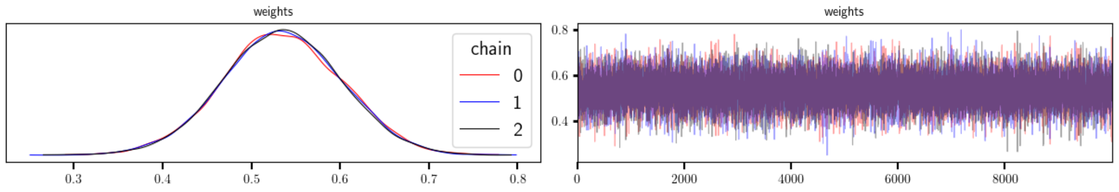
```
weight 3 : 0.4195
```



```
weight 4 : 0.2453
```



```
weight 5 : 0.5345
```



In Fig. 1, the left column consists of the marginal posterior densities for each parameter and the right column consists of our sample collection. A marginal posterior density represents the probability distribution of a single parameter, where "marginal" refers to integrating out all other parameters to focus on the distribution of interest.

[15]:
```python
fig = corner.corner(trace,
                    var_names=['weights'],
                    labels=[r'weight 1', r'weight 2',r'weight 3',r'weight␣
 ↪4',r'weight 5'],
                    truths=dict(weights=weights_true),
                    color='sienna',
                    truth_color='black',
                    show_titles=True,
                    reverse=True)
# labelsize of x,y-ticks:
for ax in fig.get_axes():
      ax.tick_params(axis='both', labelsize=9)
# include figure caption
fig_name = f'Fig. {fig_num} ' + r'corner plot of $\vec{w}_{\mathrm{model}}$.␣
 ↪True weight values $\vec{w}_{\mathrm{obs}}$ are indicated by overlaid black␣
 ↪lines.'
fig.text(.5, -.08, fig_name, fontsize=15, ha='center')
plt.show()
fig_num += 1
```

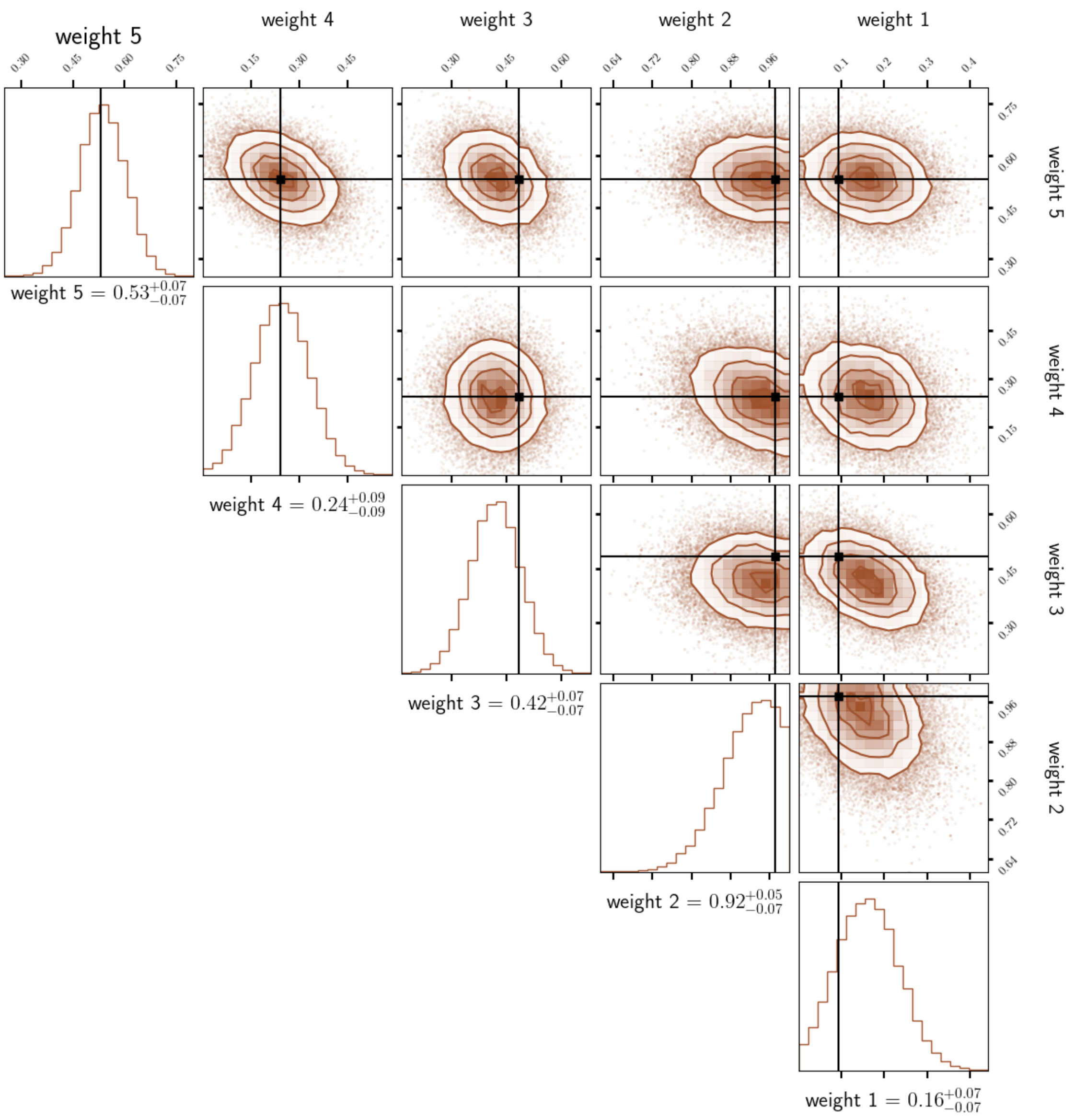


Fig. 2 corner plot of $\vec{w}_{\mathrm{model}}$. True weight values $\vec{w}_{\mathrm{obs}}$ are indicated by overlaid black lines.

In Fig. 2, the histograms on the diagonal show the marginalized posterior densities for each parameter. The contour plots show the correlation among parameters. The contour levels are defined as (0.5, 1, 1.5, 2)-sigma equivalent, containing 11.8%, 39.3%, 67.5%, 86.4% of the samples. This is explained in more detail in the `corner.corner()` sigmas documentation.

```
[19]: x = np.linspace(0,1,2)
      fig = plt.figure(figsize=(4,4))
      for cloud in range(m):
          plt.errorbar(weights_true[cloud],
                  weights_predicted[cloud],
                  yerr=weight_predicted_err[cloud],
                  marker='o',
                  capsize=3,
                  color='black'
                 )
plt.plot(x, x, color='grey')
plt.xlabel(r'$w_{\mathrm{obs}}$',fontsize=18, labelpad=18)
plt.ylabel(r'$w_{\mathrm{model}}$',fontsize=18, labelpad=18)
# include figure caption
fig_name = f'Fig. {fig_num}: model performance.'
fig.text(.55, -.2, fig_name, fontsize=12, ha='center')
plt.show()
fig_num += 1
```

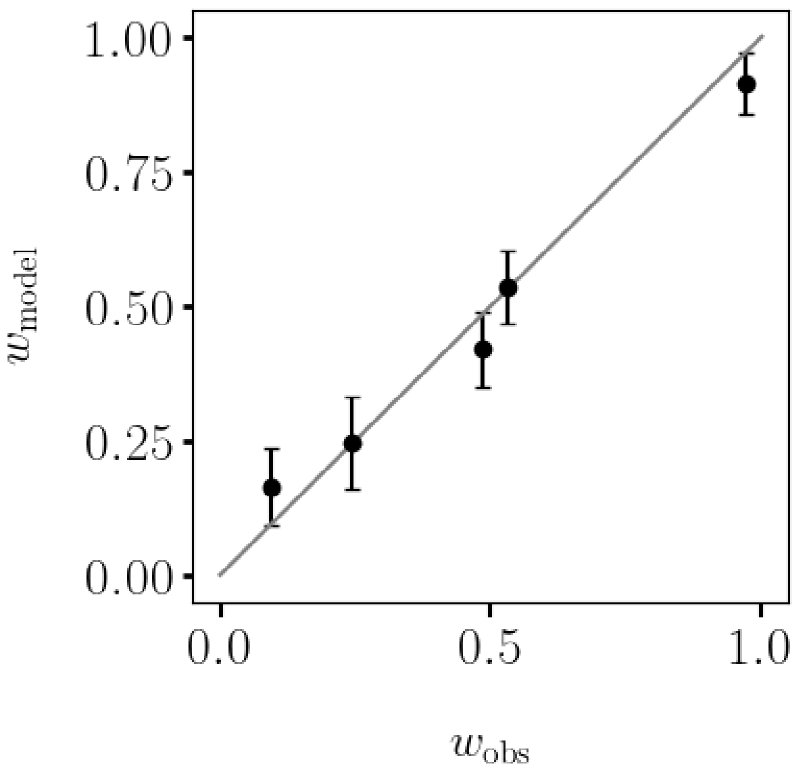


Fig. 4: model performance.

Fig. 3 demonstrates our model's performance by directly comparing $\vec{w}_{\mathrm{obs}}$ and $\vec{w}_{\mathrm{model}}$. The expected slope value for the scatter is shown by the grey solid line. The standard deviation for each weight probability is represented by error bars. When accounting for uncertainties outputted by MCMC, each data point falls within the expected slope.