The Journal of Open Source Software

MultiMin: Mixture-of-Gaussians fitting and visualization for multivariate data and single-valued functions

Jorge I. Zuluaga [1¶]

1 SEAP/FACom, Instituto de Física - FCEN, Universidad de Antioquia, Colombia ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

`MultiMin` is a Python package for fitting and analyzing *Mixtures of Gaussians* (MoGs, also known as Gaussian mixture models) in one or many dimensions. The package provides (i) a compact parameterization of multivariate Gaussian mixtures, (ii) numerical maximum-likelihood fitting tools for data samples (including partially truncated domains), and (iii) publication-ready visualizations through a unified corner-plot-like interface. In addition, `MultiMin` includes an experimental workflow to fit *single-valued functions* (e.g., time-series, spectra, or model curves) using MoG basis functions, returning interpretable diagnostics such as R^2 and goodness-of-fit measures.

Statement of need

Gaussian mixture models are a standard and versatile tool for density estimation, clustering, and probabilistic modeling because they can approximate complex continuous distributions through weighted sums of multivariate Gaussians (Bishop, 2006; McLachlan & Peel, 2000). In research workflows, however, practitioners often need more than a black-box mixture estimator: they need a parameterization that is convenient for scientific interpretation, explicit control over covariance structure, support for bounded (truncated) variables, and a visualization layer that can compare samples and fitted models consistently across dimensions.

`MultiMin` addresses these needs by providing a research-oriented API centered on three core components:

- `MixtureOfGaussians`: definition, sampling, evaluation, and serialization of MoGs.
- `FitMoG` / `FitFunctionMoG`: numerical fitting tools for (i) multivariate samples via maximum-likelihood minimization and (ii) single-valued functions (e.g., spectra) via nonlinear least squares, both exposed through a consistent, data-first interface with many optional controls (domains, bounds, optimizer options, diagnostics).
- `MultiPlot`: a consistent grid visualization system for samples, PDFs, histograms, contours, and optional marginals (i.e., a unified alternative to piecemeal corner plotting).

The package is implemented in Python and integrates with the scientific Python ecosystem (NumPy (Harris et al., 2020), SciPy (Virtanen et al., 2020), and Matplotlib (Hunter, 2007)). It is distributed as open source under the AGPLv3 license and includes extensive example notebooks that can be executed end-to-end.

State of the field

Several mature libraries provide Gaussian mixture models, most prominently `scikit-learn`'s `GaussianMixture` estimator (Pedregosa et al., 2011). Such tools are excellent general-purpose solutions for machine learning pipelines, but they typically emphasize predictive usage and integration rather than scientific interpretability and publication-centric workflows. In practice, many research users still need to build a layer around these estimators to (i) express domain-specific constraints (e.g., partially bounded variables), (ii) extract explicit parametric representations for reporting, and (iii) generate consistent multivariate diagnostic figures that combine samples and fitted densities.

`MultiMin` complements this landscape by focusing on scientific workflows:

- **Interpretability-first parameterization**: MoG components can be accessed and reported as weights, means, standard deviations, correlations, and full covariance matrices.
- **Partially truncated domains**: fitting can be performed while enforcing bounded support for selected variables (common in physical quantities constrained to intervals), with the truncation explicitly represented in the fitted model.
- **Unified diagnostic plotting**: the same high-level plotting interface can overlay data, mixture PDFs, contours, histograms, and marginals across dimensions, facilitating reproducible figures for papers.

Theoretical background

Mixture representation. A MoG with M components in k dimensions is represented as

$$f(\tilde{x}) = \sum_{i=1}^{M} w_i \, \mathcal{N}_k(\tilde{x}; \tilde{\mu}_i, \Sigma_i),$$

with weights $\sum_i w_i = 1$. The package provides helpers to construct covariance matrices from more interpretable quantities (e.g., per-dimension standard deviations and correlation/rotation parameterizations), which is useful in scientific settings where covariances are rarely specified directly.

Partially truncated multivariate Gaussians. Many scientific variables are naturally bounded (e.g., ratios, angles, physical parameters constrained to an interval). `MultiMin` supports *partially truncated* multivariate Gaussians by explicitly representing the truncation domain and using it consistently in evaluation, sampling, and fitting. Starting from the unbounded multivariate normal,

$$\mathcal{N}_k(\tilde{x}; \tilde{\mu}, \Sigma) = \frac{1}{\sqrt{(2\pi)^k \det}} \exp\left[-\frac{1}{2}(\tilde{x}-\tilde{\mu})^\top \Sigma^{-1} (\tilde{x}-\tilde{\mu})\right],$$

let $T \subset \{l, \ldots, m\}$ be the set of truncated coordinates and let $a_i < b_i$ be the bounds for each $i \in T$. The truncation region is

$$A_T = \{\tilde{x} \in \mathbb{R}^k : \; a_i \le \tilde{x}_i \le b_i \;\; \forall i \in T\},$$

with the remaining coordinates $i \notin T$ unbounded. The partially truncated multivariate normal is then defined as

$$\mathcal{T}\mathcal{N}_T(\tilde{x}; \tilde{\mu}, , \mathbf{a}_T, \mathbf{b}_T) = \frac{\mathcal{N}_k(\tilde{x}; \tilde{\mu},) \, 1_{A_T}(\tilde{x})}{Z_T(\tilde{\mu}, , \mathbf{a}_T, \mathbf{b}_T)},$$

where $\mathbf{1}_{A_T}$ is the indicator function of A_T and the normalization constant is

$$Z_T(\tilde{\mu}, , \mathbf{a}_T, \mathbf{b}_T) = \int_{A_T} \mathcal{N}_k(\tilde{\mathbf{V}}; \tilde{\mu},) \, d\tilde{\mathbf{V}} = \mathbb{P}_{\tilde{x} \sim \mathcal{N}_k(\tilde{\mu},)} \left(\tilde{x} \in A_T\right).$$

`MultiMin` encodes bounds through a per-dimension `domain` argument (e.g., `domain=[[a_1,b_1], ..., [a_k,b_k]]`, with `None` for unbounded coordinates). A *truncated MoG* is obtained by mixing truncated components with the same truncation region,

$$f_T(\tilde{x}) = \sum_{i=1}^{M} w_i \, \mathcal{T}\mathcal{N}_T(\tilde{x}; \tilde{\mu}_i, {}_i, \mathbf{a}_T, \mathbf{b}_T).$$

Maximum-likelihood fitting. For a dataset $\{\tilde{x}_n\}_{n=1}^{N}$ and a parametric MoG density $f(\tilde{x} \,|\, \theta)$ (where θ is a serialized list of parameters containing the components of $\tilde{\mu}_i, \Sigma_i$), `MultiMin` estimates parameters by classical maximum-likelihood. The likelihood and (negative) log-likelihood objectives are

$$\mathcal{L}(\theta) = \prod_{n=1}^{N} f(\tilde{x}_n \,|\, \theta),$$

$$\mathrm{NLL}(\theta) = -\sum_{n=1}^{N} \log f(\tilde{x}_n \,|\, \theta),$$

and the package minimizes NLL (or its normalized variant) with `scipy.optimize.minimize` (Virtanen et al., 2020). We do not use an expectation–maximization (EM) clustering formulation as a primary strategy because the typical use case here is not unsupervised cluster discovery: instead, users select a predetermined number of components (e.g., for a controlled parametric approximation or a physically motivated mixture), and the goal is a direct parametric fit under constraints such as partial truncation.

Software design

`MultiMin` is organized as a compact, research-oriented library where the main public objects are imported from the top-level namespace (`import multimin as mn`). Internally, functionality is split into a few focused modules that mirror the conceptual workflow "define model → fit → diagnose/visualize → export":

- **Modeling (`mog`)**: `MixtureOfGaussians` implements parameterized multivariate mixtures, including partially truncated domains via a per-dimension `domain` specification. The class supports sampling, evaluation (PDF/log-PDF), parameter transformations convenient for scientific reporting (e.g., standard-deviation/correlation forms and covariance constructors in `Stats`), and serialization of a fitted model.
- **Fitting (`fitting`)**: `FitMoG` performs classical maximum-likelihood estimation by minimizing the negative log-likelihood with `scipy.optimize.minimize`, using a flattened parameter vector that is mapped to physical parameters (weights, means, covariance structure) and optionally constrained through bounds. Truncated variables are handled consistently through the same `domain` definition used by `MixtureOfGaussians`. For single-valued functions, `FitFunctionMoG` fits a MoG basis to $(X, F(X))$ pairs via nonlinear least squares, using an analytically updated global normalization at each objective evaluation and offering higher-level modes (e.g., adaptive routines guided by peak finding and smoothing utilities from SciPy).
- **Visualization (`plotting`)**: `MultiPlot` provides a consistent grid layout for scatter/histogram/PDF/contour overlays, with a single "properties" specification controlling labels and ranges across panels. The plotting API is intentionally "thin" and Pythonic: most methods accept standard Matplotlib-style keyword arguments (plus structured dictionaries such as `sargs`, `hargs`, `dargs`, and `margs`) so users can tune aesthetics without rewriting plotting logic. Optional marginal panels can be enabled when needed.
- **Utilities (`util`, `base`)**: small helpers provide robust I/O (e.g., `Util.get_data` for packaged example datasets), numerical/statistical helpers (`Stats`), and lightweight base functionality shared by the main classes.

- **Optional acceleration (cmog)**: when the compiled library is available, MixtureOfGaussians.get_func cmog=True) generates callables that route evaluation through ctypes wrappers around optimized batch evaluators, enabling fast likelihood/PDF evaluation on large grids without changing the high-level user code.

Across the API, defaults are chosen to keep common workflows short (a few lines from raw arrays to a fitted model and a figure), while exposing many optional parameters to support controlled scientific use cases (fixed domains, constrained covariances, custom initialization and bounds, detailed optimizer options, and fully customizable plotting). A complete documentation site with narrative examples and the full API reference is provided at https://multimin.readthedocs.io.

Code examples

Visualization as a first-class feature. Multivariate visual diagnostics are handled by MultiPlot, which uses a consistent grid layout to render scatter plots, histograms, PDF heatmaps, and contour overlays. This design is conceptually similar to corner-plot workflows (Foreman-Mackey, 2016), but MultiMin couples the visualization directly to MoG objects and fitted results, ensuring that "data vs model" comparisons reuse the same ranges, labels, and aesthetics.

The typical end-to-end workflow (data generation, visualization, fitting, and plotting) is a few lines of code, mirroring the tutorial notebooks shipped with the project:

```
import numpy as np
import multimin as mn

deg = np.pi / 180

# Synthetic sample from a 3D, 2-component MoG
weights = [0.5, 0.5]
mus = [[1.0, 0.5, -0.5], [1.0, -0.5, +0.5]]
sigmas = [[1, 1.2, 2.3], [0.8, 0.2, 3.3]]
angles = [[10 * deg, 30 * deg, 20 * deg], [-20 * deg, 0.0, 30 * deg]]
Sigmas = mn.Stats.calc_covariance_from_rotation(sigmas, angles)
mog_true = mn.MixtureOfGaussians(mus=mus, weights=weights, Sigmas=Sigmas)

np.random.seed(1)
data = mog_true.rvs(5000)

# Fit and plot
F = mn.FitMoG(data=data, ngauss=2)
F.fit_data()
G=F.plot_fit(
    properties=["x","y","z"],
    pargs=dict(cmap='Spectral_r'),
    sargs=dict(s=0.2,edgecolor='None',color='w',nbins=30),
    cargs=dict(levels=50,zorder=+300,alpha=0.3),
    figsize=3,
    marginals=True
)
```

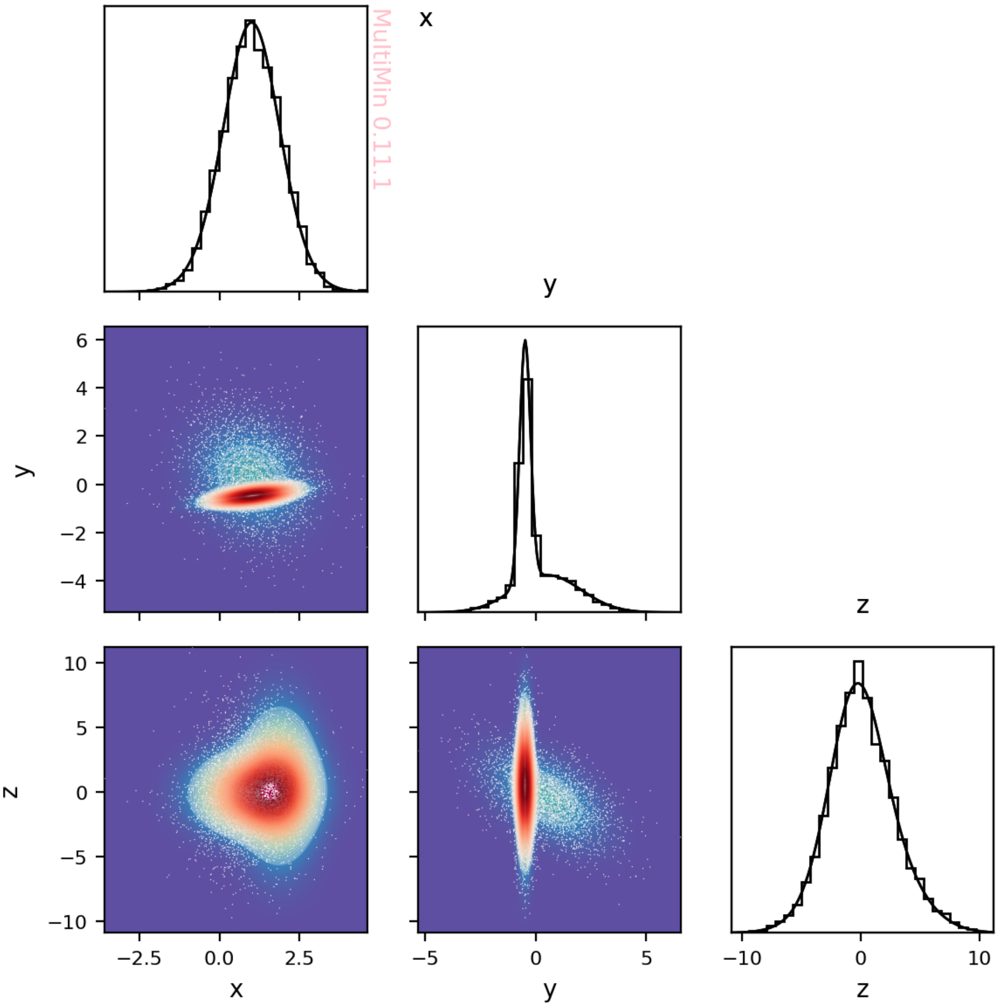

Figure 1: Example of a fitted multivariate MoG visualized with `MultiPlot`.

Fit quality diagnostics for multivariate fits are available via a Kolmogorov–Smirnov (K–S) distance computed on $S = -2\log p(\tilde{x})$ (observed vs. synthetic samples) and an optional Q–Q plot:

```
stats = F.quality_of_fit(data=data, n_sim=5000, plot_qq=True, figsize=5)
ks_dist = stats["ks_dist"]
r2_identity = stats["r2_identity"]
```

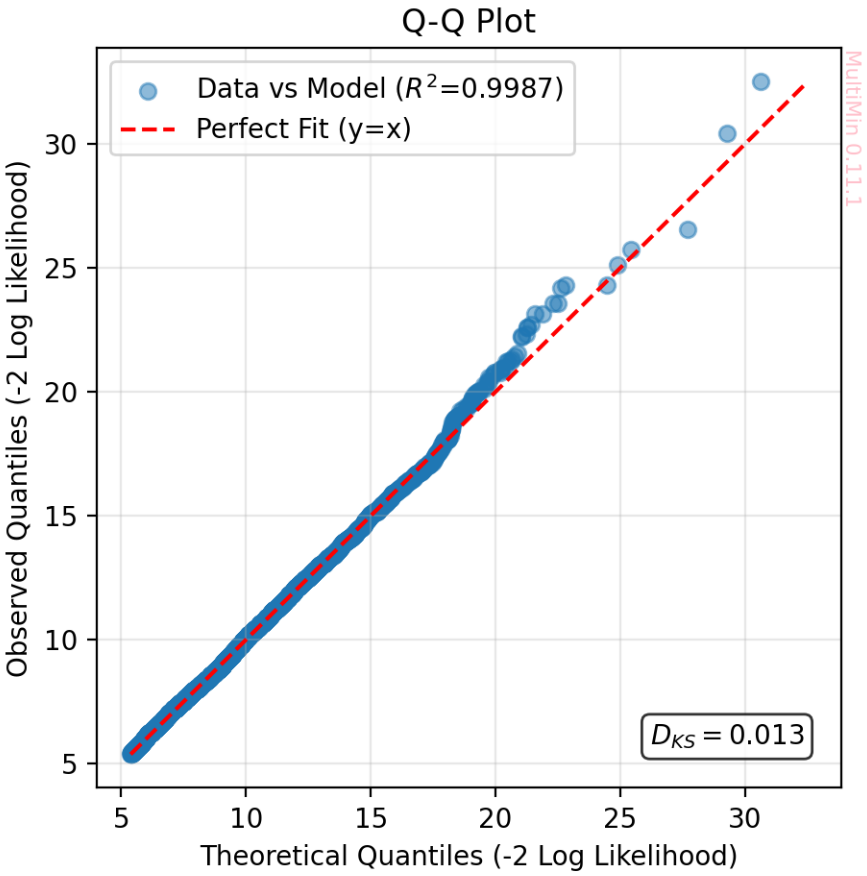

Figure 2: Q–Q plot and K–S distance for a multivariate MoG fit.

In addition to fitting samples, `MultiMin` includes an experimental workflow to fit *single-valued functions* (univariate or multivariate) by matching a MoG-based model to $(X, F(X))$ pairs. In the univariate function-fitting setting this is formulated as a nonlinear least-squares problem (minimizing squared residuals between the observed function values and the model prediction on the mesh), and the package reports diagnostics such as R^2 that are useful for model selection and automated fitting loops.

Univariate spectral-line fitting

A practical example of the function-fitting workflow is the decomposition of a complex univariate spectral line profile into a small number of Gaussian components. The spectrum used in this example is shipped with `MultiMin` (`complex-line.txt`) and is based on the multi-component synthetic spectra used in the context of GaussPy+ (Riener et al., 2019). It can be loaded via `mn.Util.get_data` and `numpy.loadtxt` and then fitted as:

```python
import numpy as np
import multimin as mn

path = mn.Util.get_data("complex-line.txt")
chan, spectrum = np.loadtxt(path, unpack=True, comments="#")

Ff = mn.FitFunctionMoG(data=(chan, spectrum), ngauss=3)
Ff.fit_data(mode="adaptive", advance=10)
fig = Ff.plot_fit(dargs=dict())
```

```
stats_f = Ff.quality_of_fit()
```

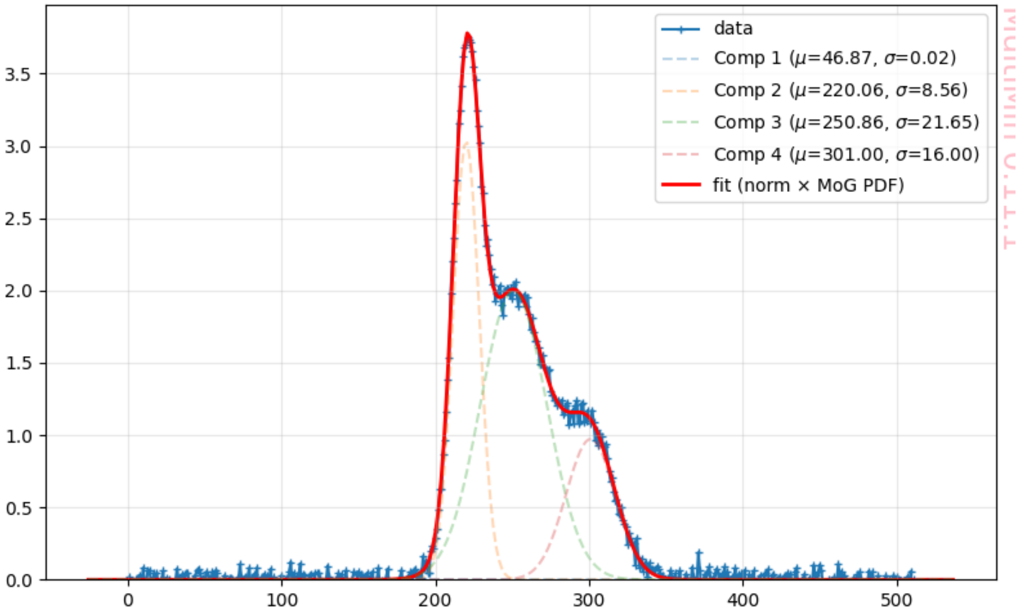

Figure 3: Example of a complex spectral line and a MoG-based fit using `FitFunctionMoG`.

Semi-analytical descriptions and accelerated evaluators

Beyond fitting and plotting, `MultiMin` can export fitted distributions as semi-analytical artifacts suitable for papers and high-throughput evaluation. In particular, fitted MoGs can be rendered as LaTeX (explicit parameters and matrices) and as self-contained Python callables. For example, the LaTeX export can be generated as follows:

```python
latex_str, _ = F.mog.get_function(type="latex", print_code=False, decimals=4)
print(latex_str)
```

and produces LaTeX directly (excerpt):

```
$$f(\mathbf{x}) = w_1 \, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \mathbf{\Sigma}_1)

where

$$w_1 = 0.4909$$
$$\boldsymbol{\mu}_1 = \left( \begin{array}{c} 0.9577 \\ 0.5176 \\ -0.4634 \end{array}\r
$$\mathbf{\Sigma}_1 = \left( \begin{array}{ccc} 1.0805 & -0.3353 & 0.2666 \\ -0.3353 & 2

$$w_2 = 0.5091$$
$$\boldsymbol{\mu}_2 = \left( \begin{array}{c} 1.0192 \\ -0.481 \\ 0.6188 \end{array}\ri
$$\mathbf{\Sigma}_2 = \left( \begin{array}{ccc} 0.6319 & 0.1054 & -0.0236 \\ 0.1054 & 0.

Here the normal distribution is defined as:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^{{k
```

This is the way as it is rendered:

$$f(\mathbf{x}) = w_1 \, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, _1) + w_2 \, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_2, _2)$$

where

$$w_1 = 0.4909$$

$$\boldsymbol{\mu}_1 = \begin{pmatrix} 0.9577 \\ 0.5176 \\ -0.4634 \end{pmatrix}$$

$${}_1 = \begin{pmatrix} 1.0805 & -0.3353 & 0.2666 \\ -0.3353 & 2.3655 & -1.716 \\ 0.2666 & -1.716 & 4.4798 \end{pmatrix}$$

$$w_2 = 0.5091$$

$$\boldsymbol{\mu}_2 = \begin{pmatrix} 1.0192 \\ -0.481 \\ 0.6188 \end{pmatrix}$$

$${}_2 = \begin{pmatrix} 0.6319 & 0.1054 & -0.0236 \\ 0.1054 & 0.0604 & -0.0145 \\ -0.0236 & -0.0145 & 11.0725 \end{pmatrix}$$

Here the normal distribution is defined as:

$$\mathcal{N}(\mathbf{x};\boldsymbol{\mu},) = \frac{1}{\sqrt{(2\pi)^k \det}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\top -1}(\mathbf{x}-\boldsymbol{\mu})\right]$$

These expressions can be incorporated into scientific publications to provide a semi-analytical representation of the fitted distribution.

Accelerated evaluators

Optionally, the generated Python code can target C-optimized batch evaluators exposed through `ctypes` for fast evaluation on large grids:

```python
code_str_c, f_fast_c = F.mog.get_function(type="python", cmog=True, print_code=False)
```

When the optional C backend is available, `cmog=True` routes evaluation through `ctypes` wrappers around compiled batch evaluators. In `examples/multimin_cmog.ipynb`, a representative micro-benchmark shows a speedup for point-wise evaluation (same fitted MoG, same machine/kernel):

- **Single point**: Python callable $\sim 60\,\mu$s per loop vs `cmog=True` $12\,\mu$s per loop. Speed-up $\sim \times 5$.
- **Batch (10,000 points)**: Python callable $445\,\mu$s per loop vs `cmog=True` $280\,\mu$s per loop. Speed-up $\sim \times 2$.

Comparison with other similar tools

`MultiMin` overlaps with existing Gaussian-mixture tooling, but targets a different point in the design space: it prioritizes scientific interpretability, bounded (truncated) domains, and reproducible diagnostics/figures as first-class outputs.

scikit-learn. `scikit-learn` provides a widely used `GaussianMixture` implementation oriented toward machine-learning workflows (e.g., clustering and density estimation pipelines) (Pedregosa et al., 2011). `MultiMin` can be used for similar density-estimation tasks, but emphasizes an explicit, report-friendly parameterization (weights, means, standard deviations/correlations/covariances), publication-ready multivariate visualization via `MultiPlot`, and domain-aware (partially truncated) likelihoods. The repository includes a notebook-level comparison between `MultiMin` and `scikit-learn` GMM (see Figure 4).

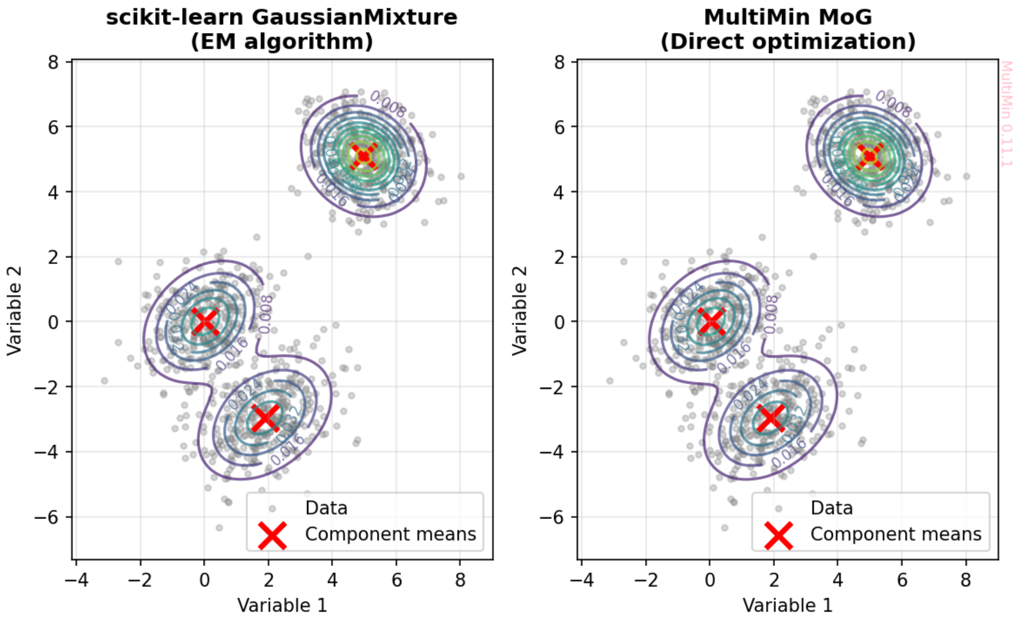

Figure 4: Comparison figure produced by the `MultiMin` notebooks, contrasting mixture-model workflows.

GaussPy+. GaussPy+ is a specialized automated Gaussian decomposition package for emission-line spectra, with substantial algorithmic machinery for noise estimation, quality control, and (optionally) spatially coherent refitting in large surveys (Riener et al., 2019). `MultiMin` is not a drop-in replacement for that end-to-end pipeline; however, for the core task of fitting a spectrum with multiple Gaussian components, the `FitFunctionMoG` workflow produces comparable decompositions in representative cases (e.g., the complex-line example included with this package and shown in Figure 3), while providing a unified API that also extends to multivariate MoG fitting with shared diagnostics (e.g., K–S / Q–Q tests).

Research impact statement

`MultiMin` was developed to support research workflows where multivariate distributions must be fitted, interpreted, and communicated with consistent diagnostics. The repository includes reproducible notebooks demonstrating applications such as Near-Earth asteroid datasets and controlled comparisons against `scikit-learn`'s GMM implementation, along with gallery figures that can be directly reused in publications and reports. By integrating fitting, diagnostics, and visualization in a single package, `MultiMin` reduces the glue code typically required to turn a fitted mixture model into reproducible, publication-quality results.

AI usage disclosure

Generative AI tools were used during the development and documentation of `MultiMin`, including the preparation of this manuscript. Specifically, AI assistance was used for drafting text, refactoring portions of code, and scaffolding tests and examples. All AI-assisted outputs were reviewed, edited, and validated by the human author(s), who made the core technical and design decisions and verified correctness via automated tests and executable notebooks.

Acknowledgements

The author thanks the SEAP/FACom community at Universidad de Antioquia for feedback during development, and Juanita Agudelo-Quiceno (Universidad de Antioquia) for beta-testing and for using the library in the context of Near-Earth Asteroid (NEA) research. The author also acknowledges the broader scientific Python ecosystem for foundational libraries used by this project.

References

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Foreman-Mackey, D. (2016). Corner.py: Scatterplot matrices in python. *The Journal of Open Source Software*, *1*(2), 24. https://doi.org/10.21105/joss.00024

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

McLachlan, G., & Peel, D. (2000). *Finite mixture models*. Wiley.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*, 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

Riener, M., Kainulainen, J., Henshaw, J. D., Orkisz, J. H., Murray, C. E., & Beuther, H. (2019). GaußPy+: A fully automated Gaussian decomposition package for emission line spectra. *Astronomy & Astrophysics*, *628*, A78. https://doi.org/10.1051/0004-6361/201935519

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., Walt, S. J. van der, Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … Mulbregt, P. van. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2