

EM Acceleration Methods for Gaussian Mixture Models

Computer Intensive Statistics
Adam Rokah

December 2025

1 EM Overview and Slow Convergence

Let $y \in \Omega_y$ denote incompletely observed data and $x \in \Omega_x$ the corresponding complete data. Consider a parametric model $f(\cdot \mid \theta)$ with $\theta \in \Omega_\theta \subset \mathbb{R}^p$ and define the observed- and complete-data log-likelihoods

$$\ell_o(\theta) = \log f(y \mid \theta), \quad \ell_c(\theta) = \log f(x \mid \theta).$$

Using the identity $f(x \mid y, \theta) = f(x \mid \theta) / f(y \mid \theta)$, we obtain

$$\ell_o(\theta) = \ell_c(\theta) - \log f(y \mid \theta). \quad (1)$$

Since x is unobserved, direct maximization of $\ell_o(\theta)$ is often difficult. The Expectation–Maximization (EM) algorithm instead iteratively maximizes the surrogate

$$Q(\theta \mid \theta^{(t)}) = \mathbb{E}[\ell_c(\theta) \mid y, \theta^{(t)}] = \int \log f(x \mid \theta) f(x \mid y, \theta^{(t)}) dx. \quad (2)$$

The algorithm alternates between computing $Q(\theta \mid \theta^{(t)})$ (E-step) and maximizing it with respect to θ (M-step), producing a sequence $\{\theta^{(t)}\}$ that monotonically increases $\ell_o(\theta)$.

To understand the convergence speed of EM, note that the algorithm implicitly defines a mapping $M : \Omega_\theta \rightarrow \Omega_\theta$ such that

$$\theta^{(t+1)} = M(\theta^{(t)}).$$

If $\theta^{(t)} \rightarrow \theta^*$ and M is differentiable at θ^* , then a first-order expansion yields

$$\theta^{(t+1)} - \theta^* = DM(\theta^*)(\theta^{(t)} - \theta^*) + O(\|\theta^{(t)} - \theta^*\|^2),$$

where $DM(\theta^*)$ is the Jacobian of the EM mapping at the fixed point. Let λ denote the spectral radius of $DM(\theta^*)$. Then $\|\theta^{(t)} - \theta^*\| = O(\lambda^t)$, so convergence is slow when λ is close to one. In mixture models, this commonly occurs when components overlap strongly, motivating the use of acceleration methods.

2 The SQUAREM Algorithm

The Squared Extrapolation Method (SQUAREM), introduced by Varadhan and Roland (2008), is a vector extrapolation technique for accelerating fixed-point iterations such as EM. Let $M(\cdot)$ denote the EM mapping and define the first- and second-order finite differences

$$\Delta\theta^{(t)} = M(\theta^{(t)}) - \theta^{(t)}, \quad \Delta^2\theta^{(t)} = M(M(\theta^{(t)})) - 2M(\theta^{(t)}) + \theta^{(t)}.$$

Here, $\Delta\theta^{(t)}$ denotes the EM update direction at iteration t , while $\Delta^2\theta^{(t)}$ captures the change in this update across successive iterations. Near a fixed point θ^* , EM exhibits approximately linear convergence, so the error $\theta^{(t)} - \theta^*$ is dominated by a slowly decaying component aligned with $\Delta\theta^{(t)}$. SQUAREM exploits this local linear structure by extrapolating along the EM update direction.

Specifically, SQUAREM constructs the extrapolated point

$$\psi^{(t)} = \theta^{(t)} - 2\alpha^{(t)}\Delta\theta^{(t)} + (\alpha^{(t)})^2\Delta^2\theta^{(t)},$$

which can be interpreted as applying two EM-like steps with an optimized step length. The step size $\alpha^{(t)} < 0$ is chosen to approximately minimize the dominant linear error, leading to the practical choice

$$\alpha^{(t)} = -\frac{\|\Delta\theta^{(t)}\|_2}{\|\Delta^2\theta^{(t)}\|_2}.$$

To preserve EM stability and monotonicity, the extrapolated point is mapped back through the EM operator,

$$\theta^{(t+1)} = M(\psi^{(t)}),$$

with backtracking applied if the observed-data log-likelihood fails to increase. This allows SQUAREM to achieve substantial acceleration while retaining the robust convergence properties of standard EM.

3 Numerical Experiments

3.1 Data Generation

All experiments are conducted on synthetic data generated from a one-dimensional Gaussian mixture model (see appendix for detailed explanation) with $K = 3$ components. The true data-generating distribution is given by

$$f(y) = \sum_{k=1}^3 \pi_k \mathcal{N}(y \mid \mu_k, \sigma_k^2),$$

with mixing proportions $\pi = (0.25, 0.35, 0.40)$, means $\mu = (-2.0, 0.0, 2.0)$, and variances $\sigma^2 = (0.8^2, 0.7^2, 0.9^2)$. A dataset of size $n = 10000$ is generated by

first sampling latent component labels and subsequently drawing observations from the corresponding Gaussian distributions.

This configuration yields a clearly multimodal but partially overlapping mixture, providing a nontrivial setting for mixture estimation while remaining well-identified.

3.2 Experimental Setup: Uncertainty and Efficiency Assessment

To evaluate statistical uncertainty and computational efficiency, we compare standard EM and its accelerated variant SQUAREM under identical experimental conditions. Classical EM uncertainty quantification based on the observed information matrix relies on the EM update structure and does not directly apply when acceleration alters the optimization path. Uncertainty is therefore assessed using a parametric bootstrap, while efficiency is measured in terms of iteration count and wall-clock runtime.

Bootstrap datasets are simulated from the fitted model and refit using the same estimation procedure, with percentile confidence intervals obtained from the empirical $\alpha/2$ and $1 - \alpha/2$ quantiles. Because resampling is performed after convergence, this yields valid uncertainty estimates for both EM and SQUAREM. All benchmarks use identical data, convergence tolerances, and stopping criteria to ensure a fair comparison.

4 Results

4.1 Efficiency and Accuracy Results

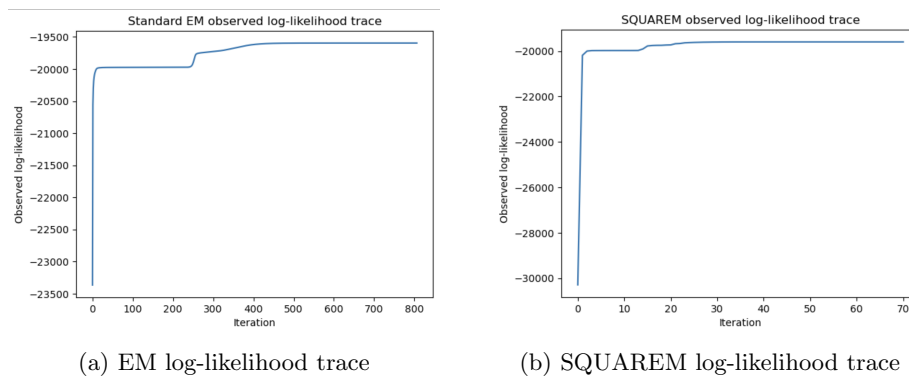


Figure 1: Observed-data log-likelihood traces for standard EM and SQUAREM

Figure 1 shows that both EM and SQUAREM produce monotone increases in the observed-data log-likelihood, but that SQUAREM converges in substantially

fewer iterations. This illustrates how vector extrapolation accelerates the EM fixed-point iteration without compromising numerical stability.

To compare computational efficiency and estimation accuracy, we report the number of iterations, wall-clock runtime, and root-mean-square (RMS) parameter errors across 20 runs (each with randomly initialized parameter values).

Method	Iterations (mean \pm sd)	Time (s) (mean \pm sd)
Standard EM	607.0 \pm 193.9	0.8648 \pm 0.2689
SQUAREM + EM	54.5 \pm 16.3	0.2309 \pm 0.0749
Speedup (EM / SQUAREM)	11.14 \times	3.75 \times
Time saved		73.3%

Method	RMS error in π	RMS error in μ	RMS error in σ^2
Standard EM	0.0221 \pm 0.0005	0.0621 \pm 0.0007	0.0553 \pm 0.0010
SQUAREM + EM	0.0221 \pm 0.0001	0.0622 \pm 0.0002	0.0554 \pm 0.0003

Benchmark: repeats=20. Randomized init per repeat. tol=1e-5, max_iter=2000. Errors computed after sorting components by mean.

Figure 2: Efficiency statistics and RMS errors (mean \pm SD)

Figure 2 shows that SQUAREM reduces the number of iterations by more than an order of magnitude and achieves a 3.75 \times speedup in wall-clock time compared to standard EM. At the same time, the root-mean-square errors of the estimated parameters (π, μ, σ^2) are virtually identical for the two methods, indicating no loss in statistical accuracy. Thus, SQUAREM substantially improves computational efficiency while preserving the quality of the final maximum likelihood estimates.

4.2 Parametric Bootstrap and Percentile Confidence Intervals

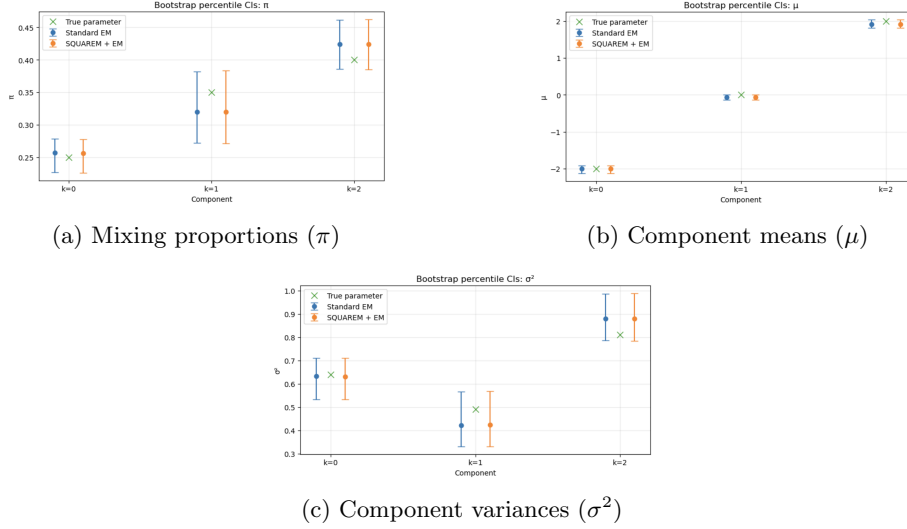


Figure 3: Percentile bootstrap confidence intervals compared to true GMM parameters

Figure 3 summarizes the parametric bootstrap uncertainty results for standard EM and SQUAREM. Across all mixture weights, component means, and variances, the percentile confidence intervals from the two methods are nearly identical and cover the true generating parameters. Thus, although SQUAREM substantially accelerates convergence, it does not affect the statistical uncertainty of the final maximum likelihood estimates, confirming that EM acceleration preserves inferential validity (see the corresponding numerical values in Appendix Figure 6).

5 Conclusion

This project showed that SQUAREM substantially accelerates EM for Gaussian mixture models while preserving monotone likelihood ascent, final maximum likelihood estimates, and bootstrap-based uncertainty quantification. Thus, SQUAREM yields significant computational gains without loss of statistical accuracy, at the cost of increased algorithmic complexity.

References

Kuroda, Y. and Geng, J. (2022). *Acceleration of the EM Algorithm*. Statistical Science, 37(1), 123–145.

6 Appendix

6.1 Gaussian Mixture Model and EM Updates

We consider a univariate Gaussian mixture model (GMM) with K components. The observed data y_1, \dots, y_n are assumed to follow the density

$$f(y | \theta) = \sum_{k=1}^K \pi_k \mathcal{N}(y | \mu_k, \sigma_k^2),$$

where $\pi_k \geq 0$, $\sum_k \pi_k = 1$. Introduce latent variables $Z_i \in \{1, \dots, K\}$ indicating component membership.

E-Step

Given $\theta^{(t)}$, the E-step computes the responsibilities

$$\tau_{ik}^{(t)} = \mathbb{P}(Z_i = k | y_i, \theta^{(t)}) = \frac{\pi_k^{(t)} \mathcal{N}(y_i | \mu_k^{(t)}, \sigma_k^{2(t)})}{\sum_{j=1}^K \pi_j^{(t)} \mathcal{N}(y_i | \mu_j^{(t)}, \sigma_j^{2(t)})}.$$

M-Step

The M-step updates the parameters as

$$\begin{aligned} \pi_k^{(t+1)} &= \frac{1}{n} \sum_{i=1}^n \tau_{ik}^{(t)}, & \mu_k^{(t+1)} &= \frac{\sum_i \tau_{ik}^{(t)} y_i}{\sum_i \tau_{ik}^{(t)}}, \\ \sigma_k^{2(t+1)} &= \frac{\sum_i \tau_{ik}^{(t)} (y_i - \mu_k^{(t+1)})^2}{\sum_i \tau_{ik}^{(t)}}. \end{aligned}$$

These updates are iterated until convergence, typically monitored via $\ell_o(\theta)$.

6.2 Additional Figures

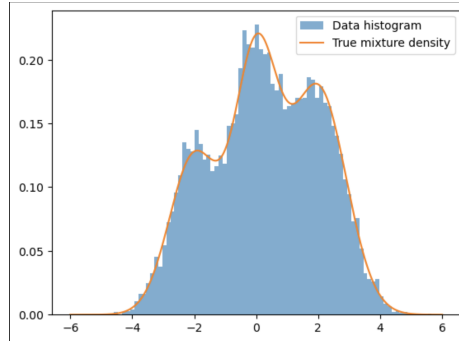


Figure 4: Data Generation + True GMM Density

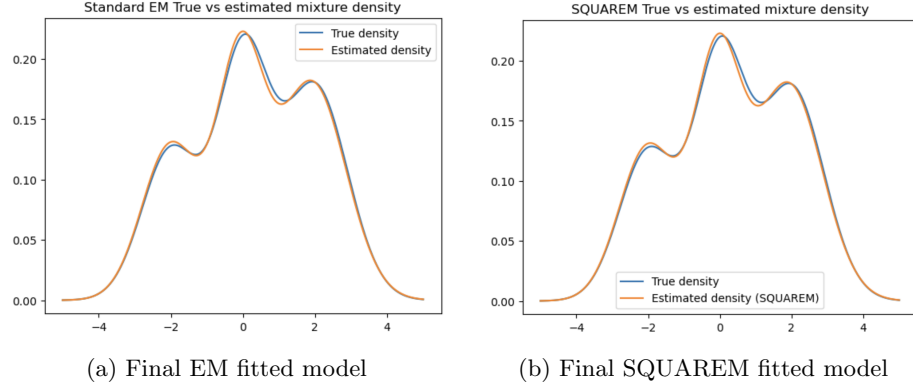


Figure 5: Final Gaussian mixture model fits obtained via EM and SQUAREM

Figure 5 demonstrates that both methods converge to essentially the same fitted mixture density, confirming that SQUAREM alters the optimization path but not the final maximum likelihood estimate.

Component k	True π_k	EM $\hat{\pi}_k$	EM 95% CI	SQUAREM $\hat{\pi}_k$	SQUAREM 95% CI
0	0.2500	0.2567	[0.2269, 0.2785]	0.2565	[0.2262, 0.2781]
1	0.3500	0.3195	[0.2721, 0.3819]	0.3200	[0.2713, 0.3835]
2	0.4000	0.4238	[0.3858, 0.4614]	0.4235	[0.3850, 0.4618]

(a) Mixing proportion (π) confidence intervals

Component k	True μ_k	EM $\hat{\mu}_k$	EM 95% CI	SQUAREM $\hat{\mu}_k$	SQUAREM 95% CI
0	-2.0000	-2.0044	[-2.1237, -1.9116]	-2.0051	[-2.1252, -1.9107]
1	0.0000	-0.0677	[-0.1352, 0.0069]	-0.0674	[-0.1355, 0.0070]
2	2.0000	1.9145	[1.8075, 2.0337]	1.9155	[1.8075, 2.0334]

(b) Component mean (μ) confidence intervals

Component k	True σ_k^2	EM $\hat{\sigma}_k^2$	EM 95% CI	SQUAREM $\hat{\sigma}_k^2$	SQUAREM 95% CI
0	0.6400	0.6315	[0.5330, 0.7092]	0.6309	[0.5314, 0.7093]
1	0.4900	0.4220	[0.3304, 0.5656]	0.4230	[0.3303, 0.5687]
2	0.8100	0.8799	[0.7859, 0.9865]	0.8791	[0.7832, 0.9877]

(c) Component variance (σ^2) confidence intervals

Figure 6: Parametric bootstrap confidence intervals for GMM parameters

6.3 Implementation Details

Listing 1: Python implementation of EM and SQUAREM + Uncertainty Estimation for Gaussian mixture models

```

1
2 # Imports + global config
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from scipy.stats import norm
6 from scipy.stats import binom
7 import time

```

```

8
9 # for numerical stability
10 from scipy.special import logsumexp
11
12 # Reproducibility
13 np.random.seed(12345)
14
15
16 def Density(y):
17     return (
18         0.25 * norm.pdf(y, loc=-2.0, scale=0.8) +
19         0.35 * norm.pdf(y, loc=0.0, scale=0.7) +
20         0.40 * norm.pdf(y, loc=2.0, scale=0.9)
21     )
22
23 n = 10000
24
25 pi_true = np.array([0.25, 0.35, 0.40])
26 mu_true = np.array([-2.0, 0.0, 2.0])
27 sigma_true = np.array([0.8, 0.7, 0.9])
28
29 # Sample component labels
30 Z = np.random.choice(len(pi_true), size=n, p=pi_true)
31
32 # Generate observations
33 Y = np.zeros(n)
34 for i in range(n):
35     k = Z[i]
36     Y[i] = norm.rvs(loc=mu_true[k], scale=sigma_true[k])
37
38 # Plot histogram and true density
39 xgrid = np.linspace(-6, 6, 2000)
40 plt.hist(Y, bins=80, density=True, alpha=0.6, label='Data histogram')
41 plt.plot(xgrid, Density(xgrid), label='True mixture density')
42 plt.legend()
43 plt.show()
44
45
46
47 def obs_log_like(y, mu, var, prop):
48     n = len(y) #number of data points
49     K = len(mu) #number of components
50
51     # mixture density for each data point
52     dens = np.zeros((n, K))
53     for k in range(K):
54         dens[:, k] = prop[k] * norm.pdf(y, loc=mu[k], scale=np.sqrt(var[k]))
55
56     # sum over components, then log and sum over observations
57     return np.sum(np.log(np.sum(dens, axis=1)))
58
59
60 def obs_log_like_stable(y, mu, var, prop):
61     y = np.asarray(y)
62     mu = np.asarray(mu)
63     var = np.asarray(var)
64     prop = np.asarray(prop)

```

```

65     # log N(y_i | mu_k, var_k) ---- VECTORIZED
66     log_pdf = (
67         -0.5 * np.log(2 * np.pi * var)
68         - 0.5 * (y[:, None] - mu[None, :])**2 / var[None, :]
69     )
70
71     # log of the mixture density
72     log_weighted = log_pdf + np.log(prop)
73
74     #logsumexp sums over components, then we sum over observations
75     return np.sum(logsumexp(log_weighted, axis=1))
76
77 def e_step_resp(y, mu, var, prop):
78     n = len(y)
79     K = len(mu)
80
81     resp = np.zeros((n, K))
82
83     # resp[:, k] prop[k] * N(y | mu[k], var[k])
84     for k in range(K):
85         resp[:, k] = norm.pdf(y, loc=mu[k], scale=np.sqrt(var[k])) * prop[k]
86
87     # Normalize across components so each row sums to 1 (denominator in Bayes'
88     # rule)
89     row_sums = np.sum(resp, axis=1, keepdims=True) # (n, 1)
90     resp = resp / row_sums
91
92     return resp
93
94 def m_step_update(y, resp):
95     n = len(y)
96     K = resp.shape[1]
97
98     #Nk := the number of data points assigned to each component (membership count
99     #      per component)
100     Nk = np.sum(resp, axis=0) # (K,)
101
102     # Update proportions
103     prop_new = Nk / n # (K,)
104
105     # Update means
106     mu_new = np.zeros(K)
107     for k in range(K):
108         mu_new[k] = np.sum(resp[:, k] * y) / Nk[k]
109
110     # Update variances
111     var_new = np.zeros(K)
112     for k in range(K):
113         var_new[k] = np.sum(resp[:, k] * (y - mu_new[k])**2) / Nk[k]
114
115     return prop_new, mu_new, var_new
116
117 def em_gmm_1d(y, K, mu_init, var_init, prop_init, max_iter=2000, tol=1e-6):
118     n = len(y) #number of data points
119

```

```

120 # traces (store every iterate so you can plot later)
121 mu_trace = [mu_init.copy()]
122 var_trace = [var_init.copy()]
123 prop_trace = [prop_init.copy()]
124 ll_trace = [obs_log_like_stable(y=y, mu=mu_init, var=var_init, prop=prop_init
    )]
125
126 # main EM iterations
127 for t in range(max_iter):
128     # Current parameters
129     mu_t = mu_trace[-1]
130     var_t = var_trace[-1]
131     prop_t = prop_trace[-1]
132
133     # ---- E-step ----
134     resp = e_step_resp(y=y, mu=mu_t, var=var_t, prop=prop_t) # (n, K)
135
136     # ---- M-step ----
137     prop_new, mu_new, var_new = m_step_update(y=y, resp=resp)
138
139     # Store new parameters
140     prop_trace.append(prop_new.copy())
141     mu_trace.append(mu_new.copy())
142     var_trace.append(var_new.copy())
143
144     # Compute observed log-likelihood at new params (monitoring only)
145     ll_new = obs_log_like(y=y, mu=mu_new, var=var_new, prop=prop_new)
146     ll_trace.append(ll_new)
147
148     # Check convergence
149     increase = abs(ll_trace[-1] - ll_trace[-2])
150     if increase < tol:
151         break
152
153 # Final outputs
154 return {
155     means: mu_trace[-1],
156     vars: var_trace[-1],
157     prop: prop_trace[-1],
158     resp: resp, # from last E-step
159     iter: len(ll_trace) - 1,
160     tol: increase,
161     trace: {
162         means: mu_trace,
163         vars: var_trace,
164         prop: prop_trace,
165         obs_loglike: ll_trace,
166     },
167 }
168
169
170 # Fit the 2-component mixture
171 start = time.perf_counter()
172 fit = em_gmm_1d(
173     y=Y,
174     K=3,
175     mu_init=np.array([0, 0.1, 1]),

```

```

176     var_init=np.array([0.1, 0.2, 1]),
177     prop_init=np.array([0.5, 0.5, 1]),
178     tol=1e-5,
179     max_iter=2000
180 )
181 end = time.perf_counter()
182
183 # Quick sanity prints
184 print(Iterations:, fit[iter])
185 print(Final tol (abs ll increase):, fit[tol])
186 print(Final means:, fit[means])
187 print(Final vars:, fit[vars])
188 print(Final props:, fit[prop])
189 print(fElapsed time: {end - start} seconds)
190
191 # 1) Plot log-likelihood trace
192 plt.figure()
193 plt.plot(fit[trace][obs_loglike])
194 plt.xlabel(Iteration)
195 plt.ylabel(Observed log-likelihood)
196 plt.title(Standard EM observed log-likelihood trace)
197 plt.show()
198
199 # Plot fitted density vs true density
200 def est_density_grid(xgrid, fit_dict):
201     mu = fit_dict[means]
202     var = fit_dict[vars]
203     prop = fit_dict[prop]
204     # mixture density on a grid
205     out = np.zeros_like(xgrid, dtype=float)
206     #we loop over mixture components and sum the densities
207     #out(x)=1* N(x; 1,1^2) + 2*N(x; 2,2^2)
208     for k in range(len(mu)):
209         out += prop[k] * norm.pdf(xgrid, loc=mu[k], scale=np.sqrt(var[k]))
210     return out
211
212 plt.figure()
213 plt.plot(yvalues, Density(yvalues), label=True density)
214 plt.plot(yvalues, est_density_grid(yvalues, fit), label=Estimated density)
215 plt.legend()
216 plt.title(Standard EM True vs estimated mixture density)
217 plt.show()
218
219
220
221 def squarem_gmm_1d(y, K, mu_init, var_init, prop_init, max_iter=2000, tol=1e-6,
222     max_backtrack=20):
223
224     # --- helpers to pack/unpack parameters into a single vector ---
225     # we need this since SQUAREM math requires a single vector of parameters
226     # whereas EM steps require separate vectors for mu, var, prop
227     def pack_theta(mu, var, prop):
228         return np.concatenate([mu, var, prop]) # shape (3K,)
229
230     def unpack_theta(theta):
231         mu = theta[0:K]
232         var = theta[K:2*K]

```

```

232     prop = theta[2*K:3*K]
233     return mu, var, prop
234
235 def normalize_params(mu, var, prop):
236     # keep variances positive and proportions valid
237     # it doesn't guarantee that the parameters are valid in GMM context
238     # so we need to normalize them and ensure they are valid
239     # note that the normalization step is a safeguard, rather than the final
estimator
240     # we still do theta_next = M(tilde_theta) ie run EM on the extrapolated
vector
241     # ensures that we have a valid EM step.
242     # parameter space, a normalization step is required to ensure that
variances remain
243     # positive and mixing proportions remain valid probabilities before
applying the EM mapping.
244     var = np.maximum(var, 1e-12)
245     prop = np.maximum(prop, 1e-15)
246     prop = prop / np.sum(prop)
247     return mu, var, prop
248
249 # --- the EM mapping M(theta): one E-step + one M-step ---
250 # this is the function that takes a single vector of parameters,
251 # performs one EM iteration, and returns a new vector of parameters.
252 # it is the last step of the SQUAREM iteration, where we map back
253 # through M(.) to preserve stability.
254 def em_map(theta_vec):
255     mu, var, prop = unpack_theta(theta_vec)
256     # normalize parameters to ensure they are valid (since theta_vec is
unconstrained)
257     mu, var, prop = normalize_params(mu, var, prop)
258     # perform one EM iteration
259     resp = e_step_resp(y=y, mu=mu, var=var, prop=prop)
260     prop_new, mu_new, var_new = m_step_update(y=y, resp=resp)
261     # note: this second normalization is not theoretically required, but it
is a safeguard
262     # to ensure that the parameters are valid in case of numerical rounding
errors.
263     mu_new, var_new, prop_new = normalize_params(mu_new, var_new, prop_new)
264     return pack_theta(mu_new, var_new, prop_new)
265
266 # --- initialize ---
267 mu0 = np.asarray(mu_init, dtype=float).copy()
268 var0 = np.asarray(var_init, dtype=float).copy()
269 prop0 = np.asarray(prop_init, dtype=float).copy()
270 # normalize to insure initial parameters are valid
271 mu0, var0, prop0 = normalize_params(mu0, var0, prop0)
272 # pack initial parameters into a single vector for SQUAREM extrapolation
273 theta = pack_theta(mu0, var0, prop0)
274
275 # traces (store every accepted iterate so you can plot later)
276 mu_trace = [mu0.copy()]
277 var_trace = [var0.copy()]
278 prop_trace = [prop0.copy()]
279 ll_trace = [obs_log_like_stable(y=y, mu=mu0, var=var0, prop=prop0)]
280
281 # main SQUAREM iterations

```

```

282     for t in range(max_iter):
283         # (article): psi0 = theta, psi1 = M(psi0), psi2 = M(psi1)
284         psi0 = theta
285         psi1 = em_map(psi0)
286         psi2 = em_map(psi1)
287
288         # psi0 (difference between iterations) and ^2psi0 (difference between
differences)
289         d1 = psi1 - psi0
290         d2 = psi2 - 2.0 * psi1 + psi0
291
292         # choose alpha (Eq. (40) in the article)
293         norm_d1 = np.linalg.norm(d1)
294         norm_d2 = np.linalg.norm(d2)
295
296         # if d2 is ^0, fall back to a regular EM step since then the
extrapolation is not reliable
297         # ie alpha is not well-defined since alpha = - norm_d1 / norm_d2
298         if norm_d2 < 1e-14 or norm_d1 < 1e-14:
299             theta_next = psi1 # regular EM
300         else:
301             alpha = - norm_d1 / norm_d2 # negative step length (Eq. 40)
302
303             # modify alpha (article safeguard)
304             # and that the extrapolation is not too aggressive (FROM ARTICLE)
305             if alpha > -1.0:
306                 alpha = -1.0
307
308             # extrapolated point epsi(alpha) = psi0 - 2a*d1 + a^2*d2
309             # We backtrack on alpha until log-likelihood improves (relative to
psi0).
310             ll_psi0 = ll_trace[-1]
311             accepted = False
312
313             for _ in range(max_backtrack):
314                 # compute the extrapolated point
315                 epsi = psi0 - 2.0 * alpha * d1 + (alpha ** 2) * d2
316                 # shape (3K,) since this extrapolation is done in the ambient
parameter space
317                 # unpack the extrapolated point and normalize it
318                 mu_e, var_e, prop_e = unpack_theta(epsi)
319                 mu_e, var_e, prop_e = normalize_params(mu_e, var_e, prop_e)
320                 # compute the observed log-likelihood of the extrapolated point
321                 ll_eps = obs_log_like_stable(y=y, mu=mu_e, var=var_e, prop=prop_e
)
322
323                 # accept if extrapolated point observed log-likelihood is not
worse than current
324                 if ll_eps >= ll_psi0:
325                     accepted = True
326                     break
327
328                 # backtracking rule in the article: alpha <- (alpha - 1)/2
329                 # if the observed log-likelihood of the extrapolated point is
worse than current,
330

```

```

331         # then we backtrack and try again. We do this max_backtrack (20)
times, and if we still
332         # haven't found a good alpha, then we revert to regular EM.
333         alpha = (alpha - 1.0) / 2.0
334
335         if not accepted:
336             # if we failed to find a good alpha, revert to regular EM
337             epsi = psi0
338
339             # map back through EM: theta_{t+1} = M(eps_i)
340             # note epsi is the extrapolated point if accepted, otherwise it is
the current point
341             theta_next = em_map(eps_i)
342
343             # unpack + log-likelihood + stopping
344             mu_next, var_next, prop_next = unpack_theta(theta_next)
345             mu_next, var_next, prop_next = normalize_params(mu_next, var_next,
prop_next)
346
347             ll_next = obs_log_like_stable(y=y, mu=mu_next, var=var_next, prop=
prop_next)
348
349             mu_trace.append(mu_next.copy())
350             var_trace.append(var_next.copy())
351             prop_trace.append(prop_next.copy())
352             ll_trace.append(ll_next)
353
354             increase = abs(ll_trace[-1] - ll_trace[-2])
355             theta = theta_next
356
357             if increase < tol:
358                 break
359
360             # final E-step responsibilities (for consistency with em_gmm_1d output)
361             resp_final = e_step_resp(y=y, mu=mu_trace[-1], var=var_trace[-1], prop=
prop_trace[-1])
362
363             return {
364                 means: mu_trace[-1],
365                 vars: var_trace[-1],
366                 prop: prop_trace[-1],
367                 resp: resp_final,
368                 iter: len(ll_trace) - 1,
369                 tol: increase,
370                 trace: {
371                     means: mu_trace,
372                     vars: var_trace,
373                     prop: prop_trace,
374                     obs_loglike: ll_trace,
375                 },
376             }
377
378 start = time.perf_counter()
379 fit_sq = squarem_gmm_1d(
380     y=Y,
381     K=3,

```

```

383     mu_init=np.array([0, 0.1, 1]),
384     var_init=np.array([0.1, 0.2, 1]),
385     prop_init=np.array([0.5, 0.5, 1]),
386     tol=1e-5,
387     max_iter=2000
388 )
389 end = time.perf_counter()
390
391 print(SQUAREM Iterations:, fit_sq[iter])
392 print(SQUAREM Final tol (abs ll increase):, fit_sq[tol])
393 print(SQUAREM Final means:, fit_sq[means]) #trace of means
394 print(SQUAREM Final vars:, fit_sq[vars]) #trace of variances
395 print(SQUAREM Final props:, fit_sq[prop]) #trace of mixing proportions
396 print(fSQUAREM Elapsed time: {end - start} seconds)
397
398 plt.figure()
399 plt.plot(fit_sq[trace][obs_loglike])
400 plt.xlabel(Iteration)
401 plt.ylabel(Observed log-likelihood)
402 plt.title(SQUAREM observed log-likelihood trace)
403 plt.show()
404
405 plt.figure()
406 plt.plot(yvalues, Density(yvalues), label=True density)
407 plt.plot(yvalues, est_density_grid(yvalues, fit_sq), label=Estimated density (
    SQUAREM))
408 plt.legend()
409 plt.title(SQUAREM True vs estimated mixture density)
410 plt.show()
411
412
413
414 def _fmt(x, d=3):
415     return NA if np.isnan(x) else f'{x:.{d}f}'
416
417 def make_efficiency_and_accuracy_tables(results, repeats,
418                                         digits_time=4,
419                                         digits_iter=1,
420                                         digits_err=4):
421
422     s = results[summary]
423
424     eff_col_labels = [
425         Method,
426         Iterations (mean sd),
427         Time (s) (mean sd),
428     ]
429
430     eff_cell_text = [
431         [
432             Standard EM,
433             f{_fmt(s['EM']['iter_mean'], digits_iter)} {_fmt(s['EM']['iter_sd'],
434             digits_iter)},
435             f{_fmt(s['EM']['time_mean'], digits_time)} {_fmt(s['EM']['time_sd'],
436             digits_time)},
437         ],
438         [
439             SQUAREM + EM,

```

```

437         f{fmt(s['SQUAREM']['iter_mean'], digits_iter)} {fmt(s['SQUAREM']['
iter_sd'], digits_iter)},
438         f{fmt(s['SQUAREM']['time_mean'], digits_time)} {fmt(s['SQUAREM']['
time_sd'], digits_time)},
439     ],
440     [
441         Speedup (EM / SQUAREM),
442         fmt(s[speedup][iter_factor], 2) + x,
443         fmt(s[speedup][time_factor], 2) + x,
444     ],
445     [
446         Time saved,
447         ,
448         fmt(s[speedup][time_saved_pct], 1) + %,
449     ],
450 ]
451
452 acc_col_labels = [
453     Method,
454     RMS error in  $\pi$ ,
455     RMS error in  $\mu$ ,
456     RMS error in  $\sigma^2$ ,
457 ]
458
459 acc_cell_text = [
460     [
461         Standard EM,
462         f{fmt(s['EM']['rms_pi_mean'], digits_err)} {fmt(s['EM']['rms_pi_sd
'], digits_err)},
463         f{fmt(s['EM']['rms_mu_mean'], digits_err)} {fmt(s['EM']['rms_mu_sd
'], digits_err)},
464         f{fmt(s['EM']['rms_var_mean'], digits_err)} {fmt(s['EM']['r
rms_var_sd'], digits_err)},
465     ],
466     [
467         SQUAREM + EM,
468         f{fmt(s['SQUAREM']['rms_pi_mean'], digits_err)} {fmt(s['SQUAREM
']['rms_pi_sd'], digits_err)},
469         f{fmt(s['SQUAREM']['rms_mu_mean'], digits_err)} {fmt(s['SQUAREM
']['rms_mu_sd'], digits_err)},
470         f{fmt(s['SQUAREM']['rms_var_mean'], digits_err)} {fmt(s['SQUAREM
']['rms_var_sd'], digits_err)},
471     ],
472 ]
473
474 fig, axes = plt.subplots(
475     nrows=2, ncols=1,
476     figsize=(15, 5.2),
477     gridspec_kw={height_ratios: [1.1, 0.9]}
478 )
479
480 for ax in axes:
481     ax.axis(off)
482
483 # Efficiency table
484 table_eff = axes[0].table(
485     cellText=eff_cell_text,

```

```

486         colLabels=eff_col_labels,
487         cellLoc=center,
488         loc=center,
489     )
490     table_eff.auto_set_font_size(False)
491     table_eff.set_fontsize(11)
492     table_eff.scale(1, 1.4)
493
494     # Accuracy table
495     table_acc = axes[1].table(
496         cellText=acc_cell_text,
497         colLabels=acc_col_labels,
498         cellLoc=center,
499         loc=center,
500     )
501     table_acc.auto_set_font_size(False)
502     table_acc.set_fontsize(11)
503     table_acc.scale(1, 1.4)
504
505     fig.text(
506         0.01, 0.02,
507         f'Benchmark: repeats={repeats}. Randomized init per repeat.
508         ftol=1e-5, max_iter=2000.
509         fErrors computed after sorting components by mean.,
510         fontsize=10
511     )
512
513     plt.tight_layout(rect=[0, 0.05, 1, 1])
514     plt.show()
515
516 make_efficiency_and_accuracy_tables(bench_rand, repeats=repeats)
517
518
519 def simulate_from_fitted_gmm(n, pi_hat, mu_hat, var_hat, rng):
520     K = len(pi_hat)
521     z = rng.choice(K, size=n, p=pi_hat)
522     y = rng.normal(loc=mu_hat[z], scale=np.sqrt(var_hat[z]), size=n)
523     return y
524
525 def align_components_by_mean(pi, mu, var):
526     order = np.argsort(mu)
527     return pi[order], mu[order], var[order]
528
529 def percentile_ci(samples, alpha=0.05):
530     L = np.quantile(samples, alpha / 2.0)
531     U = np.quantile(samples, 1.0 - alpha / 2.0)
532     return float(L), float(U)
533
534 def fmt_ci(ci):
535     return f'[{ci[0]:.4f}, {ci[1]:.4f}]'
536
537 # ---- wrappers around existing solvers ----
538 def fit_em(y):
539     K = 3
540     mu_init = np.array([-1.0, 0.0, 1.0])
541     var_init = np.array([1.0, 1.0, 1.0])
542     prop_init = np.array([1/3, 1/3, 1/3])

```

```

543     out = em_gmm_1d(
544         y=y, K=K,
545         mu_init=mu_init, var_init=var_init, prop_init=prop_init,
546         tol=1e-5, max_iter=2000
547     )
548     return {prop: out[prop], means: out[means], vars: out[vars]}
549
550
551 def fit_squarem_em(y):
552     K = 3
553     mu_init = np.array([-1.0, 0.0, 1.0])
554     var_init = np.array([1.0, 1.0, 1.0])
555     prop_init = np.array([1/3, 1/3, 1/3])
556
557     out = squarem_gmm_1d(
558         y=y, K=K,
559         mu_init=mu_init, var_init=var_init, prop_init=prop_init,
560         tol=1e-5, max_iter=2000
561     )
562     return {prop: out[prop], means: out[means], vars: out[vars]}
563
564 def _plot_cis(point_est, ci_list, title, y_label):
565     K = len(point_est)
566     x = np.arange(1, K + 1)
567     L = np.array([ci[0] for ci in ci_list])
568     U = np.array([ci[1] for ci in ci_list])
569
570     # asymmetric error bars: lower = point - L, upper = U - point
571     yerr = np.vstack([point_est - L, U - point_est])
572
573     plt.figure()
574     plt.errorbar(x, point_est, yerr=yerr, fmt='o', capsize=5)
575     plt.xticks(x, [fk={k} for k in range(K)])
576     plt.xlabel(Component)
577     plt.ylabel(y_label)
578     plt.title(title)
579     plt.grid(True, alpha=0.3)
580     plt.show()
581
582 def parametric_bootstrap_gmm(
583     fit_fn,
584     theta_hat,
585     B=200,
586     alpha=0.05,
587     rng=None,
588     n=None,
589     verbose=False,
590     make_plots=True,
591     method_name=None,
592 ):
593     if rng is None:
594         rng = np.random.default_rng(2025)
595     if n is None:
596         n = len(Y)
597
598     # pull out and align theta_hat
599     pi_hat = np.asarray(theta_hat[prop], dtype=float)

```

```

600 mu_hat = np.asarray(theta_hat[means], dtype=float)
601 var_hat = np.asarray(theta_hat[vars], dtype=float)
602
603 pi_hat, mu_hat, var_hat = align_components_by_mean(pi_hat, mu_hat, var_hat)
604 K = len(pi_hat)
605
606 boot_pi = np.zeros((B, K))
607 boot_mu = np.zeros((B, K))
608 boot_var = np.zeros((B, K))
609
610 for b in range(B):
611     if verbose and (b % 20 == 0):
612         print(f'Bootstrap {b}/{B}')
613
614     Y_star = simulate_from_fitted_gmm(n, pi_hat, mu_hat, var_hat, rng)
615     theta_star = fit_fn(Y_star)
616
617     pi_s = np.asarray(theta_star[prop], dtype=float)
618     mu_s = np.asarray(theta_star[means], dtype=float)
619     var_s = np.asarray(theta_star[vars], dtype=float)
620
621     pi_s, mu_s, var_s = align_components_by_mean(pi_s, mu_s, var_s)
622
623     boot_pi[b] = pi_s
624     boot_mu[b] = mu_s
625     boot_var[b] = var_s
626
627 ci_pi = [percentile_ci(boot_pi[:, k], alpha=alpha) for k in range(K)]
628 ci_mu = [percentile_ci(boot_mu[:, k], alpha=alpha) for k in range(K)]
629 ci_var = [percentile_ci(boot_var[:, k], alpha=alpha) for k in range(K)]
630
631 if method_name is None:
632     method_name = EM if (fit_fn is fit_em) else SQUAREM+EM
633
634 level = 1.0 - alpha
635 print(f'\n{method_name} percentile bootstrap CIs (level={level:.0%}, B={B})')
636
637 print('\nMixing proportions _k:')
638 for k in range(K):
639     print(f' k={k}: pi_hat={pi_hat[k]:.4f}, CI={fmt_ci(ci_pi[k])}')
640
641 print('\nMeans _k:')
642 for k in range(K):
643     print(f' k={k}: mu_hat={mu_hat[k]:.4f}, CI={fmt_ci(ci_mu[k])}')
644
645 print('\nVariances _k^2:')
646 for k in range(K):
647     print(f' k={k}: var_hat={var_hat[k]:.4f}, CI={fmt_ci(ci_var[k])}')
648
649 if make_plots:
650     _plot_cis(pi_hat, ci_pi, f'{method_name}: Bootstrap percentile CIs for ,
651 )
652     _plot_cis(mu_hat, ci_mu, f'{method_name}: Bootstrap percentile CIs for ,
653 )
654     _plot_cis(var_hat, ci_var, f'{method_name}: Bootstrap percentile CIs for ,
655 )

```

```

654     return {
655         method: method_name,
656         B: B,
657         alpha: alpha,
658         theta_hat_aligned: {prop: pi_hat, means: mu_hat, vars: var_hat},
659         boot: {pi: boot_pi, mu: boot_mu, var: boot_var},
660         ci: {pi: ci_pi, mu: ci_mu, var: ci_var},
661     }
662
663     # -----
664     # -----
665     theta_hat_em = {prop: fit[prop], means: fit[means], vars: fit[vars]}
666
667     boot_em = parametric_bootstrap_gmm(
668         fit_fn=fit_em,
669         theta_hat=theta_hat_em,
670         B=200,
671         alpha=0.05,
672         rng=np.random.default_rng(2025),
673         n=len(Y),
674         verbose=True,
675         make_plots=False,
676         method_name=Standard EM,
677     )
678
679
680     theta_hat_sq = {prop: fit_sq[prop], means: fit_sq[means], vars: fit_sq[vars]}
681
682     boot_sq = parametric_bootstrap_gmm(
683         fit_fn=fit_squarem_em,
684         theta_hat=theta_hat_sq,
685         B=200,
686         alpha=0.05,
687         rng=np.random.default_rng(2025),
688         n=len(Y),
689         verbose=True,
690         make_plots=False,
691         method_name=SQUAREM + EM,
692     )
693
694     def _extract_point_and_ci(boot_res, param_key):
695         point = np.asarray(boot_res[theta_hat_aligned][{pi: prop, mu: means, var: vars}[
696             param_key]])
697         ci_list = boot_res[ci][param_key]
698         L = np.array([ci[0] for ci in ci_list], dtype=float)
699         U = np.array([ci[1] for ci in ci_list], dtype=float)
700         return point, L, U
701
702     def _aligned_true_params(param_key, true_params):
703         pi_t = np.asarray(true_params[pi], dtype=float).copy()
704         mu_t = np.asarray(true_params[mu], dtype=float).copy()
705         var_t = np.asarray(true_params[var], dtype=float).copy()
706
707         order = np.argsort(mu_t)
708         pi_t, mu_t, var_t = pi_t[order], mu_t[order], var_t[order]
709
710         return {pi: pi_t, mu: mu_t, var: var_t}[param_key]

```

```

710
711 def plot_ci_comparison_overlay_with_truth(
712     boot_em,
713     boot_sq,
714     param_key,
715     y_label,
716     true_params,
717     title_prefix=,
718 ):
719     p_em, L_em, U_em = _extract_point_and_ci(boot_em, param_key)
720     p_sq, L_sq, U_sq = _extract_point_and_ci(boot_sq, param_key)
721
722     K = len(p_em)
723     if len(p_sq) != K:
724         raise ValueError(K mismatch between methods. Ensure both fits use the
725                             same K and alignment.)
726
727     p_true = _aligned_true_params(param_key, true_params)
728     if len(p_true) != K:
729         raise ValueError(K mismatch between true_params and bootstrap results.)
730
731     x = np.arange(1, K + 1)
732
733     yerr_em = np.vstack([p_em - L_em, U_em - p_em])
734     yerr_sq = np.vstack([p_sq - L_sq, U_sq - p_sq])
735
736     # horizontal jitter so error bars don't overlap perfectly
737     dx = 0.10
738
739     plt.figure(figsize=(7.6, 4.2))
740     plt.errorbar(x - dx, p_em, yerr=yerr_em, fmt='o', capsize=5, label=Standard
741                 EM)
742     plt.errorbar(x + dx, p_sq, yerr=yerr_sq, fmt='o', capsize=5, label=SQUAREM +
743                 EM)
744
745     # TRUE markers (no error bars)
746     plt.plot(x, p_true, marker='x', linestyle='None', markersize=8, label=True
747             parameter)
748
749     plt.xticks(x, [fk={k} for k in range(K)])
750     plt.xlabel(Component)
751     plt.ylabel(y_label)
752     plt.title(f{title_prefix}Bootstrap percentile CIs: {y_label}.strip())
753     plt.grid(True, alpha=0.3)
754     plt.legend()
755     plt.tight_layout()
756     plt.show()
757
758 #true param dict
759 true_params = {
760     pi: pi_true,
761     mu: mu_true,
762     var: sigma_true**2,
763 }
764
765 #create plots

```

```

763 plot_ci_comparison_overlay_with_truth(boot_em, boot_sq, param_key=pi, y_label=,
      true_params=true_params)
764 plot_ci_comparison_overlay_with_truth(boot_em, boot_sq, param_key=mu, y_label=,
      true_params=true_params)
765 plot_ci_comparison_overlay_with_truth(boot_em, boot_sq, param_key=var, y_label=,
      true_params=true_params)
766
767 def _extract_for_table(boot_res, param_key):
768     key_map = {pi: prop, mu: means, var: vars}
769     point = np.asarray(boot_res[theta_hat_aligned][key_map[param_key]])
770     ci = boot_res[ci][param_key]
771     return point, ci
772
773 def _fmt(x, d=4):
774     return f'{x:.{d}f}'
775
776 def make_ci_table_with_truth(
777     param_key,
778     param_symbol,
779     boot_em,
780     boot_sq,
781     true_params,
782     digits=4,
783 ):
784     p_em, ci_em = _extract_for_table(boot_em, param_key)
785     p_sq, ci_sq = _extract_for_table(boot_sq, param_key)
786
787     K = len(p_em)
788     if len(p_sq) != K:
789         raise ValueError(K mismatch between boot_em and boot_sq.)
790
791     # --- align the true params in the same way (sort by true means) ---
792     pi_t = np.asarray(true_params[pi], dtype=float).copy()
793     mu_t = np.asarray(true_params[mu], dtype=float).copy()
794     var_t = np.asarray(true_params[var], dtype=float).copy()
795
796     order = np.argsort(mu_t)
797     pi_t, mu_t, var_t = pi_t[order], mu_t[order], var_t[order]
798
799     true_map = {pi: pi_t, mu: mu_t, var: var_t}
800     p_true = true_map[param_key]
801     if len(p_true) != K:
802         raise ValueError(K mismatch between true_params and bootstrap results.)
803
804     col_labels = [
805         Component $k$,
806         fTrue ${param_symbol}$,
807         fEM $\hat{{{param_symbol}}}$,
808         EM 95% CI,
809         fSQUAREM $\hat{{{param_symbol}}}$,
810         SQUAREM 95% CI,
811     ]
812
813     cell_text = []
814     for k in range(K):
815         cell_text.append([
816             f{k},

```

```

817         _fmt(p_true[k], digits),
818         _fmt(p_em[k], digits),
819         f[_fmt(ci_em[k][0], digits)}, {_fmt(ci_em[k][1], digits)}],
820         _fmt(p_sq[k], digits),
821         f[_fmt(ci_sq[k][0], digits)}, {_fmt(ci_sq[k][1], digits)}],
822     ])
823
824     fig, ax = plt.subplots(figsize=(13, 2.2 + 0.4 * K))
825     ax.axis(off)
826
827     table = ax.table(
828         cellText=cell_text,
829         colLabels=col_labels,
830         cellLoc=center,
831         loc=center,
832     )
833
834     table.auto_set_font_size(False)
835     table.set_fontsize(11)
836     table.scale(1, 1.4)
837
838     plt.tight_layout()
839     plt.show()
840
841
842     true_params = {
843         pi: pi_true,
844         mu: mu_true,
845         var: sigma_true**2, # convert std devs to variances to match your tables
846     }
847
848     make_ci_table_with_truth(pi, r'\pi_k', boot_em, boot_sq, true_params)
849     make_ci_table_with_truth(mu, r'\mu_k', boot_em, boot_sq, true_params)
850     make_ci_table_with_truth(var, r'\sigma_k^2', boot_em, boot_sq, true_params)

```