Technical Implementation Report

Informal Bids MCMC Estimation

A Python-to-MATLAB Translation Guide

Austin Li

January 2026

Contents

1 Executive Summary

This report provides a comprehensive technical breakdown of the Python codebase for estimating admission cutoffs in M&A-style auctions using Markov Chain Monte Carlo (MCMC) methods. The code implements the threshold/interval framework discussed in the December 18, 2025 meeting notes.

What the code does:

- **Task A**: Estimates a single constant cutoff b^{I*} for informal bid admission
- **Task B**: Estimates type-specific cutoffs b_S^{I*} and b_F^{I*} for two bidder types

The estimation uses Gibbs sampling with data augmentation, where latent admission thresholds are sampled subject to interval constraints implied by observed admit/reject decisions. In the current simulation setup, the sample is *conditional on reaching the formal stage* (auctions with zero admitted bidders are treated as unobserved and excluded), and all-admit auctions enter via one-sided upper bounds.

Structure of this report:

1. Code architecture overview
2. Python-MATLAB translation guide
3. Module-by-module deep dive with code annotations
4. Windows installation and setup guide
5. Running the code

2 Code Architecture Overview

2.1 Module Structure

The codebase is organized as a Python package called `informal_bids` with the following modules:

2.2 Module Purposes

3 Python-MATLAB Translation Guide

This section helps MATLAB users understand Python syntax and libraries used in the codebase.

Module	Purpose	Key Dependencies
`cli.py`	Entry points for running baselines and sensitivity analyses	config, data, samplers, analysis, visualization, sensitivity
`data.py`	DGP parameters, auction data classes, simulation generators	utils
`data_io.py`	CSV loader for real auction data (optional)	data
`samplers.py`	Gibbs samplers for Task A and Task B	data, utils, numba_kernels
`analysis.py`	Bias/RMSE/CI/coverage metrics	(pure NumPy)
`visualization.py`	Diagnostics and interval plots	data
`sensitivity.py`	Sample-size sensitivity framework	data, samplers
`config.py`	Plot styling and output paths	matplotlib, seaborn
`utils.py`	Shared math utilities (truncnorm, R-hat, covariates)	scipy
`numba_kernels.py`	JIT-compiled kernels for intercept-only MCMC	numba

Module	MATLAB Equivalent	Purpose
`data.py`	Struct definitions	DGP parameters, auction data structures, data generators
`samplers.py`	Main estimation script	MCMC samplers using Gibbs sampling
`analysis.py`	Post-processing functions	Compute bias, RMSE, coverage metrics
`visualization.py`	Plotting scripts	Generate diagnostic plots
`sensitivity.py`	Batch runner	Run parameter sweeps across N, cutoffs
`cli.py`	`main.m`	Entry points for running simulations
`utils.py`	Helper functions	Truncated normal sampling, Gelman-Rubin
`numba_kernels.py`	MEX files	JIT-compiled fast loops (like MATLAB MEX)
`config.py`	Configuration file	Output paths, plotting settings

3.1 Basic Syntax Comparison

3.2 Key Libraries

3.3 Classes and Dataclasses

Python uses *classes* to bundle data and methods together. The `@dataclass` decorator automatically generates `__init__` methods:

```python
# Python dataclass (like MATLAB struct with defaults)
@dataclass
class MCMCConfig:
    n_iterations: int = 20000    # MATLAB: config.n_iterations = 20000
    burn_in: int = 10000         # MATLAB: config.burn_in = 10000
    n_chains: int = 3            # MATLAB: config.n_chains = 3
```

In MATLAB, this would be:

```
config = struct();
```

MATLAB	Python
`% comment`	`# comment`
`function y = foo(x)`	`def foo(x): return y`
`end`	Indentation (no end keyword)
`if cond ... end`	`if cond: ...`
`for i = 1:N ... end`	`for i in range(N): ...`
`a(1)` (1-indexed)	`a[0]` (0-indexed)
`a(1:3)`	`a[0:3]` or `a[:3]`
`a(end)`	`a[-1]`
`[a; b]` (vertical concat)	`np.vstack([a, b])`
`[a, b]` (horizontal concat)	`np.hstack([a, b])`
`zeros(N, M)`	`np.zeros((N, M))`
`rand(N, 1)`	`np.random.rand(N)`
`randn(N, 1)`	`np.random.randn(N)`
`A * B` (matrix multiply)	`A @ B`
`A .* B` (elementwise)	`A * B`

Python Library	MATLAB Equivalent	Used For
`numpy` (np)	Built-in arrays	Arrays, linear algebra
`scipy.stats`	Statistics Toolbox	Distributions (invgamma, truncnorm)
`matplotlib`	Built-in plotting	Figures, subplots
`pandas`	Tables	Data frames, CSV I/O
`numba`	MEX/Coder	JIT compilation for speed

```
config.n_iterations = 20000;
config.burn_in = 10000;
config.n_chains = 3;
```

4 Module Deep Dive: `data.py`

This module defines the data structures for simulation parameters and auction data.

4.1 DGP Parameters (Task A)

```python
@dataclass
class TaskADGPParameters:
    """Data Generating Process for Task A (single cutoff).

    MATLAB equivalent:
        dgp.N = 100;          % Number of observed (formal-stage) auctions
        dgp.J = 3;            % Bidders per auction
        dgp.mu_v = 1.3;       % Mean valuation
        dgp.sigma_v = 0.2;    % Valuation std dev
        dgp.b_star = 1.4;     % True cutoff
    """
    N: int              # Number of observed (formal-stage) auctions
    J: int              # Bidders per auction
```

```
    mu_v: float          # Mean valuation
    sigma_v: float       # Std dev of valuation shock
    b_star: float        # True admission cutoff
```

4.2 Auction Data Structure

Each auction is represented as a `TaskAAuctionData` object containing:

```
@dataclass
class TaskAAuctionData:
    auction_id: int        # Auction index i
    X_i: np.ndarray        # Covariates (for extensions)
    bids: np.ndarray       # All informal bids in auction
    admitted: np.ndarray   # Boolean: True if admitted
    L_i: float             # Lower bound = max(rejected bids)
    U_i: float             # Upper bound = min(admitted bids)
    is_complete: bool      # True if L_i < U_i (both bounds finite)
```

Key insight: The interval $[L_i, U_i]$ contains the true cutoff b^{I*} for complete auctions.

4.3 Data Generation

The `TaskADataGenerator` class simulates synthetic auction data:

```
def generate_auction_data(self):
    """
    Generate N observed auctions (conditional on reaching the formal stage):
      while len(auctions) < N:
        1. Draw valuations: v_ij = mu_v + epsilon_ij,  epsilon ~ N(0, sigma_v
    ^2)
        2. Set bids = valuations (truth-telling at informal stage)
        3. Draw covariates X_i and cutoff: b*_i = X_i' beta (+ optional shock)
        4. Apply admission: j in A_i iff b_ij >= b*_i
        5. If no one admitted: skip (unobserved all-reject)
        6. Else compute bounds:
           - mixed outcomes: L_i = max(rejected), U_i = min(admitted) (two-
    sided)
           - all admitted:   L_i = -inf,        U_i = min(bids)      (one-sided
    )
    """
    while len(auctions) < self.params.N:
        # Draw valuation shocks
        epsilon = np.random.normal(0, self.params.sigma_v, self.params.J)
        valuations = self.params.mu_v + epsilon
        bids = valuations.copy()

        # Admission rule (with optional covariates)
        X_i = self._draw_covariates()
        b_star_i = X_i @ self.params.beta
        admitted = bids >= b_star_i

        # Compute interval bounds
        if admitted.sum() == 0:
            continue  # unobserved all-reject
```

```
        if admitted.sum() > 0 and (~admitted).sum() > 0:
            L_i = np.max(bids[~admitted])  # Max rejected
            U_i = np.min(bids[admitted])   # Min admitted
            is_complete = True
        else:
            L_i = -np.inf
            U_i = np.min(bids)
            is_complete = False
```

5 Module Deep Dive: `samplers.py`

This is the core estimation module implementing Gibbs sampling with data augmentation.

5.1 The Statistical Model

For Task A, the model is:

$$b_i^{I*} = \mu + \nu_i \tag{1}$$

$$\nu_i \sim \mathcal{N}(0, \sigma^2) \tag{2}$$

$$\nu_i \in [L_i - \mu,\ U_i - \mu] \quad \text{(interval constraint)} \tag{3}$$

The MCMC alternates between:

1. **Step 1**: Sample latent ν_i from truncated normal, given current (μ, σ)
2. **Step 2**: Update μ (or β with covariates) via conjugate normal update
3. **Step 3**: Update σ^2 via conjugate inverse-gamma update

5.2 Gibbs Sampler: Line-by-Line Annotation

```
# Gibbs sampling main loop
for t in range(self.config.n_iterations):
    # ============================================================
    # STEP 1: Sample latent nu_i for each auction
    # ============================================================
    # This is data augmentation: we sample the "missing data" (nu_i)
    # subject to the interval constraint from observed admits/rejects

    nu_samples = np.zeros(self.N)      # Storage for N auctions
    b_star_samples = np.zeros(self.N)  # Implied cutoffs

    for i, auction in enumerate(self.working_auctions):
        xb = float(auction.X_i @ beta)  # X'beta (linear predictor)
        lower = auction.L_i - xb        # Transform bounds
        upper = auction.U_i - xb        # to nu_i scale

        # Sample nu_i from TruncatedNormal(0, sigma, [lower, upper])
        # MATLAB: nu_i = trandn((lower-0)/sigma, (upper-0)/sigma) * sigma
        nu_i = sample_truncated_normal(0, sigma, lower, upper)
```

```
        nu_samples[i] = nu_i
        b_star_samples[i] = xb + nu_i    # Reconstruct b*_i

    # ============================================================
    # STEP 2: Update beta (conjugate normal-normal update)
    # ============================================================
    # Prior: beta ~ N(mu_0, V_0)
    # Likelihood: b*_i | beta ~ N(X_i'beta, sigma^2)
    # Posterior: beta | {b*_i} ~ N(mu_post, V_post)

    XtX = self.X.T @ self.X                # X'X matrix
    V_post = np.linalg.inv(                # Posterior variance
        V0_inv + XtX / (sigma ** 2)
    )
    beta_post = V_post @ (                 # Posterior mean
        V0_inv @ beta_prior_mean +
        (self.X.T @ b_star_samples) / (sigma ** 2)
    )
    beta = np.random.multivariate_normal(beta_post, V_post)

    # ============================================================
    # STEP 3: Update sigma^2 (conjugate inverse-gamma)
    # ============================================================
    # Prior: sigma^2 ~ InvGamma(a, b)
    # Likelihood: sum of squared residuals
    # Posterior: sigma^2 ~ InvGamma(a_post, b_post)

    a_post = a_prior + self.N / 2
    b_post = b_prior + np.sum(nu_samples ** 2) / 2
    sigma_sq = invgamma.rvs(a_post, scale=b_post)
    sigma = np.sqrt(sigma_sq)

    # Store draws
    beta_chain[t] = beta
    sigma_chain[t] = sigma
```

5.3 MATLAB Pseudocode Equivalent

For readers more comfortable with MATLAB, here is equivalent pseudocode:

```
% Gibbs sampler for Task A
for t = 1:n_iterations
    % Step 1: Sample nu_i from truncated normal
    for i = 1:N
        xb = X(i,:) * beta;
        lower = L(i) - xb;
        upper = U(i) - xb;
        nu(i) = sample_truncnorm(0, sigma, lower, upper);
        b_star(i) = xb + nu(i);
    end
```

```
    % Step 2: Update beta (normal-normal conjugacy)
    V_post = inv(V0_inv + (X'*X) / sigma^2);
    mu_post = V_post * (V0_inv * mu_prior + (X'*b_star) / sigma^2);
    beta = mvnrnd(mu_post, V_post);

    % Step 3: Update sigma^2 (inverse-gamma conjugacy)
    a_post = a_prior + N/2;
    b_post = b_prior + sum(nu.^2)/2;
    sigma_sq = 1 / gamrnd(a_post, 1/b_post);  % Inverse-gamma
    sigma = sqrt(sigma_sq);

    beta_chain(t,:) = beta;
    sigma_chain(t) = sigma;
end
```

5.4 Task B: Type-Specific Cutoffs

Task B extends the model to two bidder types (S and F) with separate cutoffs. The sampler runs independent Gibbs updates for each type:

```python
class TaskBMCMCSampler:
    """Estimates (mu_S, sigma_S) and (mu_F, sigma_F) separately.

    Uses observed auctions (reach the formal stage), then for each type keeps
    auctions that provide any information (one- or two-sided bounds):
    - S_auctions: auctions where (L_S, U_S) are not both infinite
    - F_auctions: auctions where (L_F, U_F) are not both infinite
    """

    def run_chain(self, chain_id):
        for t in range(n_iterations):
            # Update Type S parameters
            for i in S_auctions:
                nu_S[i] = sample_truncnorm(0, sigma_S, L_S[i]-mu_S, U_S[i]-mu_S
    )
            # ... conjugate update for (mu_S, sigma_S)

            # Update Type F parameters
            for i in F_auctions:
                nu_F[i] = sample_truncnorm(0, sigma_F, L_F[i]-mu_F, U_F[i]-mu_F
    )
            # ... conjugate update for (mu_F, sigma_F)
```

6 Module Deep Dive: `analysis.py` and `visualization.py`

6.1 Computing Performance Metrics

The `TaskAResultsAnalyzer` class computes:

```python
def compute_metrics(self):
    mu_samples = self.results['mu_samples']

    mu_hat = np.mean(mu_samples)           # Posterior mean
    mu_std = np.std(mu_samples)            # Posterior std

    # Bias and RMSE
    bias = mu_hat - self.true_mu           # Point estimate - truth
    rmse = np.sqrt(np.mean((mu_samples - self.true_mu)**2))

    # 95% Credible Interval
    ci_lower = np.percentile(mu_samples, 2.5)
    ci_upper = np.percentile(mu_samples, 97.5)

    # Coverage: does CI contain true value?
    coverage = 1 if ci_lower <= self.true_mu <= ci_upper else 0

    return {'mu_hat': mu_hat, 'bias': bias, 'rmse': rmse,
            'ci': (ci_lower, ci_upper), 'coverage': coverage}
```

6.2 Diagnostic Plots

The visualization module generates:

- **Trace plots**: Show MCMC chain evolution (check for convergence)
- **Posterior histograms**: Distribution of parameter estimates
- **Interval plots**: Visualize $[L_i, U_i]$ bounds across auctions

7 Module Deep Dive: `numba_kernels.py`

Numba is a Just-In-Time (JIT) compiler that converts Python loops to fast machine code (like MATLAB's MEX files or Coder).

```python
from numba import njit

@njit  # Compile this function to machine code
def task_a_run_chain_intercept(L, U, n_iter, mu_prior, sigma_prior,
                               a_prior, b_prior, mu_init, sigma_init, seed):
    """Fast MCMC chain for intercept-only model.

    This is ~10-100x faster than pure Python loops.
    """
    np.random.seed(seed)
    mu_chain = np.zeros(n_iter)
    sigma_chain = np.zeros(n_iter)

    mu = mu_init
    sigma = sigma_init
    N = len(L)

    for t in range(n_iter):
```

```
        # Fast Gibbs updates in compiled code
        nu_sum_sq = 0.0
        b_star_sum = 0.0

        for i in range(N):
            # Sample truncated normal (vectorized in compiled code)
            lower = (L[i] - mu) / sigma
            upper = (U[i] - mu) / sigma
            nu_i = sample_truncnorm_scalar(lower, upper) * sigma
            nu_sum_sq += nu_i * nu_i
            b_star_sum += mu + nu_i

        # ... rest of Gibbs updates

    return mu_chain, sigma_chain
```

Note: The first time you run the code, Numba compiles the kernels which takes 10-30 seconds. Subsequent runs are fast.

8 Windows Installation Guide

This section provides step-by-step instructions for setting up the code on a fresh Windows machine.

8.1 Step 1: Install Python

1. Go to https://www.python.org/downloads/
2. Download Python 3.9 or later (3.11 recommended)
3. Run the installer
4. **IMPORTANT**: Check "Add Python to PATH" at the bottom of the first screen
5. Click "Install Now"
6. Verify installation: Open Command Prompt and type:

   `python --version`

   You should see something like `Python 3.11.5`

8.2 Step 2: Download the Code

Copy the `Directory` folder to your computer, e.g., to `C:\Users\YourName\Documents\informal_bids`

8.3 Step 3: Create Virtual Environment

Open Command Prompt (or PowerShell) and run:

```
cd C:\Users\YourName\Documents\informal_bids

# Create virtual environment
python -m venv .venv

# Activate it
.venv\Scripts\activate

# You should see (.venv) at the start of your prompt
```

8.4 Step 4: Install Dependencies

With the virtual environment activated:

```
pip install -r requirements.txt
pip install -e .
```

This installs: numpy, scipy, matplotlib, pandas, numba, seaborn.

8.5 Step 5: Run the Simulations

```
# Run Task A baseline
task-a-baseline

# Run Task B baseline
task-b-baseline

# Run sensitivity analyses
task-a-sensitivity
task-b-sensitivity
```

Outputs are saved to the `outputs/` folder.

9 VS Code IDE Setup (Recommended)

Visual Studio Code provides a user-friendly development environment similar to MATLAB's editor.

9.1 Install VS Code

1. Download from https://code.visualstudio.com/
2. Run installer with default options

9.2 Install Python Extension

1. Open VS Code
2. Press `Ctrl+Shift+X` to open Extensions
3. Search for "Python" and install the Microsoft Python extension

9.3 Open the Project

1. File → Open Folder
2. Select the `informal_bids` folder
3. VS Code will detect the virtual environment

9.4 Select Python Interpreter

1. Press `Ctrl+Shift+P`
2. Type "Python: Select Interpreter"
3. Choose the one in `.venv` folder

9.5 Run Scripts

- Open a Python file
- Click the green play button (top right) or press `F5`
- Or use the integrated terminal: `Ctrl+`` `

10 Command-Line Usage

10.1 PowerShell Basics

```
# Navigate to project folder
cd C:\Users\YourName\Documents\informal_bids

# Activate virtual environment
.venv\Scripts\activate

# Run baseline simulations
task-a-baseline
task-b-baseline

# Run sensitivity analyses
task-a-sensitivity
task-b-sensitivity
```

```
# Deactivate when done
deactivate
```

10.2 Alternative: Run via Python Module

If the CLI commands don't work:

```
# Set Python path and run module
set PYTHONPATH=src
python -m informal_bids.cli
```

Or run specific functions:

```
python -c "from informal_bids.cli import task_a_baseline; task_a_baseline()"
```

11 Troubleshooting

11.1 "python" is not recognized

- Python wasn't added to PATH during installation
- Reinstall Python and check "Add Python to PATH"
- Or add manually: System Properties → Environment Variables → Path

11.2 "pip" is not recognized

Use `python -m pip` instead of `pip`:

```
python -m pip install -r requirements.txt
```

11.3 Import errors

- Make sure virtual environment is activated (`(.venv)` in prompt)
- Install the package: `pip install -e .`

11.4 Numba compilation takes forever

- First run compiles JIT kernels (10-30 seconds)
- Subsequent runs use cached compiled code
- If stuck, set `NUMBA_DISABLE_JIT=1` to use pure Python (slower)

11.5 Plots don't appear

- Plots are saved to `outputs/` folder, not displayed
- Check `outputs/baseline/task_a/` for PNG files

11.6 Out of memory on large runs

- Reduce `n_iterations` in `MCMCConfig`
- Increase `thinning` parameter

12 Summary

This report documented the Python codebase for MCMC estimation of informal bid admission cutoffs:

- **Core algorithm**: Gibbs sampling with data augmentation for interval-censored observations
- **Key modules**: `data.py` (structures), `samplers.py` (MCMC), `cli.py` (entry points)
- **MATLAB equivalents**: Python code maps closely to MATLAB with numpy arrays and scipy distributions
- **Windows setup**: Install Python, create virtual environment, install dependencies, run CLI commands

For questions about specific code sections, refer to the inline comments in the source files or the module deep-dives in Sections 4–6.