Rao-Blackwellization

Abde-rrahman Marght & Killian Coustoulin

Contents

1 Introduction

In this project, we explore how the Rao-Blackwell theorem can be used to improve estimates in Monte Carlo Markov Chain (MCMC) methods. This theorem is notably effective in reducing errors in statistical estimates, which significantly enhances the efficiency of computations in statistics and data science.
Our study will focus on several key areas:

- **Understanding the Theory:** We will provide a clear explanation of the Rao-Blackwell theorem, detailing when and why it improves the accuracy of estimates.
- **Presenting the Theory:** We will present methodologies from various scientific papers that demonstrate how the Rao-Blackwell theorem is used with different MCMC sampling techniques. This includes well-known methods such as the Accept-Reject algorithm, Metropolis-Hastings algorithm, and Gibbs sampling.
- **Programming the Concepts:** We will develop code in Python (and possibly R) to implement these theories. This involves translating mathematical concepts from scientific papers into practical, executable code. Special emphasis will be placed on optimizing this code to ensure maximum efficiency and effectiveness.
- **Testing and Comparing:** Through experiments that compare standard and Rao-Blackwellized estimators, we will practically demonstrate the enhancements brought about by the Rao-Blackwell theorem.

2 Rao-Blackwell Theorem

First, let us define a sufficient statistic :

Definition : Let $X_1, X_2, ..., X_n$ be a random sample from a probability distribution with unknown parameter θ. Then, the statistic :

$$T = u(X_1, X_2, ..., X_n)$$

is said to be sufficient for θ if the conditional distribution of $X_1, X_2, ..., X_n$, given T, does not depend on the parameter θ.

For instance, we can flip a coin ten times, with probability θ of hitting tail and $1 - \theta$ of hitting face. If we are told the number of times heads occurred which is equal to T, then we know everything about the random sample and in this sense, T is sufficient. (Maybe add proof ?)

Theorem (Rao-Blackwell : Let $\hat{\theta}$ be an estimator of θ with a finite second moment for all θ. Let $T(X)$ be a sufficient statistic for θ. Then for all θ,

1. $\theta_{RB} = \mathbb{E}[\hat{\theta} \mid T(X)] = \mathbb{E}[\theta]$,
2. $V(\theta_{RB}) \leq V(\hat{\theta})$.

In other words, the Rao–Blackwell theorem states that if $\hat{\theta}$ is an estimator of a parameter θ, then the conditional expectation of $\hat{\theta}$ given T(X), where T is a sufficient statistic, is typically a better estimator of θ, and is never worse.

proof: i) We have :

$$\mathbb{E}[\theta_{RB}] = \mathbb{E}[\mathbb{E}[\hat{\theta}|T(X)]] = \mathbb{E}[\theta]$$

Consequently, if $\hat{\theta}$ is unbiased, then θ_{RB} will be unbiased and vice-versa.

ii) We have :

$$\begin{aligned}
\mathbb{V}[\theta_{RB}] &= \mathbb{E}[(\theta_{RB} - \theta)^2] \\
&= \mathbb{E}[(\mathbb{E}[\hat{\theta}|T(X)] - \theta)^2] \\
&= \mathbb{E}[(\mathbb{E}[\hat{\theta} - \theta|T(X)])^2] \\
&\leq \mathbb{E}[\mathbb{E}[(\hat{\theta} - \theta)^2|T(X)]] \\
&= \mathbb{E}[(\hat{\theta} - \theta)^2] \\
&= \mathbb{V}[\hat{\theta}]
\end{aligned}$$

The inequality holds because of the Jensen's inequality since the square function is convex.

The Rao-Blackwell theorem merely states that this new estimator has variance less than or equal to the usual estimator. In practice, this difference can be enormous and we will verify that claim in this report.

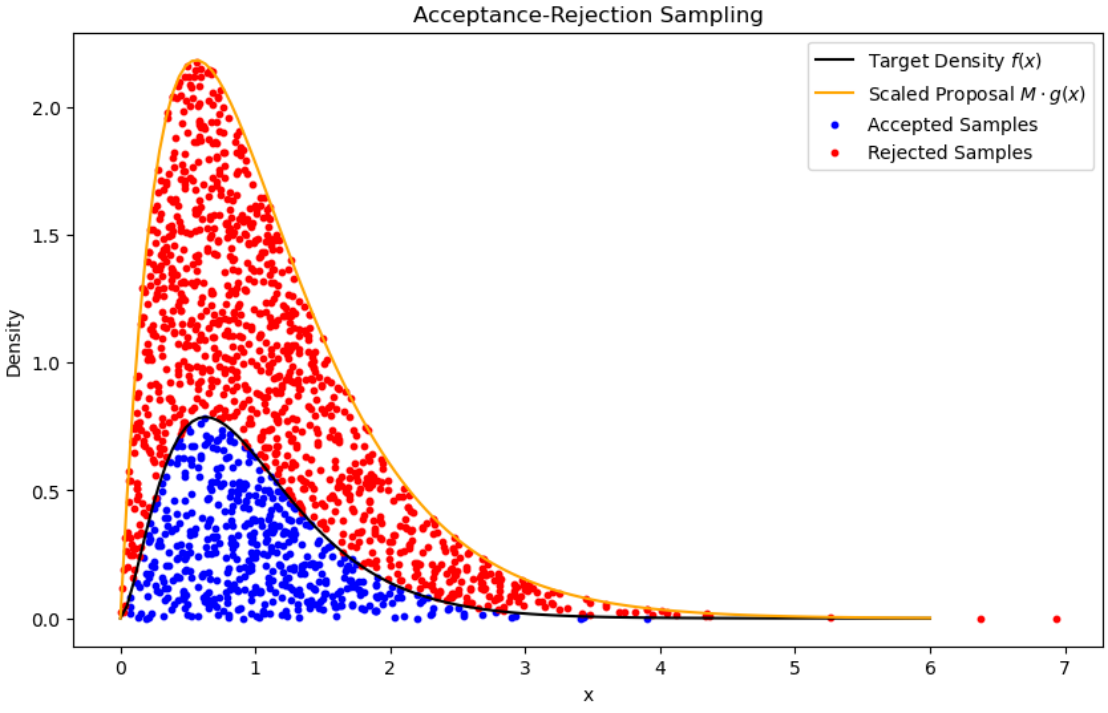
3 Accept-Reject

The python code related to this section can be found inside the RB_AR.ipynb file.

3.1 Presentation of the algorithm

Algorithm: If f and g are two densities, such that $f(x) \leq Mg(x)$ for M>0, then the random variable X provided by the following algorithm is distributed according to f :

- Obtain a sample y from distribution Y and a sample u from Unif$(0, 1)$ (the uniform distribution over the unit interval).
- Check if $u < \frac{f(y)}{Mg(y)}$.
  - If this holds, accept y as a sample drawn from f;
  - if not, reject the value of y and return



Here is the code for a simple and somewhat optimized version of Accept-Reject which returns both the accepted and proposed sample :

```
def Accept_Reject(t,M):
    sample_accept = []
    sample_proposal = []
    while len(sample_accept) < t:

        # Calculate the required batch size based on remaining needed samples
        remaining_needed = t - len(sample_accept)
        batch_size = remaining_needed

        y = g.rvs(size=batch_size)
        u = np.random.uniform(size=batch_size)

        # Calculate acceptance condition in bulk
        accepted = u < f.pdf(y) / (M * g.pdf(y))

        # Append all proposals to sample_proposal and only accepted ones to␣
↪sample_accept
        sample_proposal.extend(y)
        sample_accept.extend(y[accepted])

    return sample_accept, sample_proposal
```

This implementation takes advantages of numpy to compute the accepted samples in bulk. Note that it does not generate more random variables than strictly necessary.

Rejection sampling is mostly used when f is difficult to sample from and using an auxiliary distribution g is more efficient. The acceptance rate is equal to $\frac{1}{M}$.

Proof: If we denote $\mathbb{P}(U \leq f(Y)/Mg(Y))$ the probability of accepting Y, then we have :

$$\begin{aligned}
\mathbb{P}(U \leq f(y)/Mg(y)) &= \mathbb{E}[\mathbf{1}_{U \leq f(Y)/Mg(Y)}] \\
&= \mathbb{E}[\mathbb{E}[\mathbf{1}_{U \leq f(Y)/Mg(Y)}|Y]] \\
&= \mathbb{E}[\mathbb{P}(U \leq f(Y)/Mg(Y)|Y)] \\
&= \mathbb{E}[f(Y)/Mg(Y)] \\
&= \int \frac{f(y)}{Mg(y)} g(y)dy \\
&= \frac{1}{M}
\end{aligned}$$

3.2 Rao-Blackwellization

Let $X_1, X_2, .., X_t$ the iid accepted random variable generated using the Accept-Reject algorithm and the $Y_1, Y_2, ..., Y_N$ iid random variables generated during this process. We denote t the size of the sample of accepted random variables and N the amount of random variable Y generated to attain t. An estimator of $\mathbb{E}[h(X)]$ is given by :

$$\hat{\theta}_1 = \frac{1}{t} \sum_{i=1}^{t} h(X_i)$$

By defining $w_i = f(Y_i)/Mg(Y_i)$, $\hat{\theta}_1$ can be written :

$$\hat{\theta}_1 = \frac{1}{t}\sum_{i=1}^{N} \mathbf{1}_{U_i \leq w_i} h(Y_i)$$

Thus, we can define the Rao-Blackwellized version of $\hat{\theta}_1$:

$$\begin{aligned}\hat{\theta}_1^{(RB)} &= \frac{1}{t}\mathbb{E}[\sum_{i=1}^{N} \mathbf{1}_{U_i \leq w_i} h(Y_i) | N, Y_1, Y_2, .., Y_n] \\ &= \frac{1}{t}\sum_{1=1}^{N} \rho_i h(Y_i)\end{aligned}$$

With, for $i < n$:

$$\rho_i = w_i \sum_{(i_1,\ldots,i_{t-2})=1} \prod_{j=1}^{t-2} w_{i_j} \prod_{j=t-1}^{N-2} (1 - w_{i_j}) \; / \sum_{(i_1,\ldots,i_{t-1})=1} \prod_{j=1}^{t-1} w_{i_j} \prod_{j=t}^{N-1} (1 - w_{i_j})$$

and p_N=1. The numerator sum is over all the subsets of {1,...,N-1}/{i} of size t-2, and the denominator is over all the subsets of {1,...,N-1} of size t-1.
Unfortunately, this function is not computationally tractable. For instance, if we suppose that we want to calculate $\hat{\theta}_1^{(RB)}$ for t=10 and N=20, it would involves calculating the numerator products $19 * \binom{8}{19} = 1436058$ times.

Instead, we will compute :

$$S_k(m) = \sum_{(i_1,\ldots,i_k)} \prod_{j=1}^{k} w_{i_j} \prod_{j=k+1}^{m} (1 - w_{i_j}) \quad \text{with } (k \leq m \leq n)$$

With the sum being over all the subset of {1,..m} of size k. When we calculate S_k^i, the sum will be over all the subset of {1,..m}/{i} instead.
We can recursively calculate :

$$S_k(m) = w_m S_{k-1}(m-1) + (1 - w_m) S_k(m-1) \tag{1}$$

Where the base cases will be :

- $k = m$: then, $S_m(m) = \prod_{j=1}^{m} w_j$
- $k = 0$: then, $S_0(m) = \prod_{j=1}^{m} (1 - w_j)$

Finally, the weight ρ_i is equal to :

$$\rho_i = w_i \frac{S_{t-2}^i(n-1)}{S_{t-1}(n-1)} \quad \text{with } (i < n).$$

To compute this, a first idea would be to create a recursive function such as this :

```
@lru_cache(maxsize=None)
def S(k, m, sample_hash):
    # Convert hash back to array for processing, if passed as a tuple
    sample = np.array(sample_hash) if isinstance(sample_hash, tuple) else sample_hash

    if k == 0:
        temp = 1
```

```
        for j in range(m):
            temp *= (1 - w_i(sample[j]))
        return temp

    if k == m:
        temp = 1
        for j in range(m):
            temp *= w_i(sample[j])
        return temp

    return (w_i(sample[m-1]) * S(k-1, m-1, tuple(sample)) +
            (1 - w_i(sample[m-1])) * S(k, m-1, tuple(sample)))
```

@lru_cache indicate to python that it should stores the value of S(k,m) to avoid redundant calculations.

While this recursive function is much faster than using ρ_i with permutations, we can do even better by taking advantage of numpy vectorization and the Just In Time library to compute ρ_i even more effectively. Instead of recursively calling S(k,m), we can compute the whole sequence inside a matrix :

```
@jit(nopython=True)
def iterative_S(n, t, weights):
    S_matrix = np.zeros((t+1, n))
    S_matrix[0, :] = 1  # Base case for k=0

    for m in range(1, n+1):
        weight_m1 = weights[m-1]

        #Multiply last columns by (1-w_m) to get new columns
        S_matrix[:, m] = S_matrix[:, m-1] * (1 - weight_m1)

        #add the product of the last column, shifted by 1 toward the bottom, by 1-w_m␣
↪to the new columns
        S_matrix[1:t+1, m] += weight_m1 * S_matrix[:t, m-1]

    return S_matrix
```

The desired result of the computation is stored in the bottom-right element of the matrix.

The recursive versions are much faster than the other :

Estimator	Average time (s)
Non-Recursive ρ_i	2.67
Recursive ρ_i	0.010
Matrix ρ_i	0.002

All measurements taken with $t = 6$, acceptance rate of 30%, and averaged over 100 runs.

If we increase t further, it is not even worth it to measure the non-recursive version of ρ_i, it would takes hours to compute. While the difference between recursive ρ_i and matrix rho_i seems slim, by increasing t we can see that the latter one is much faster :

Estimator	Average time (s)
Recursive ρ_i	3.952
Matrix ρ_i	0.0024

All measurements taken with $t = 50$, acceptance rate of 30%, and averaged over 100 runs.

3.3 Comparative Analysis

To compare those two estimators, we will computing them over different sample size = t and comparing their mean squared error over n = 10000 runs given by :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{\theta} - \mu)^2$$

Where μ is the true value that the estimator is trying to estimate. For this simulation, we used a $\Gamma(2.70, 2.70)$ density as the target whose expected value is equal to 1, and a $\Gamma(2, 1.78)$. We will use an acceptance rate of 80% and 30%.

Sample size	AR estimate	RB estimate	AR MSE	RB MSE	Decrease in MSE (%)
5	0.9934	0.9921	0.0736	0.0580	21.18
10	0.9979	0.9958	0.0362	0.0259	28.36
15	0.9999	1.0001	0.0243	0.0169	30.44
20	0.9986	0.9939	0.0184	0.0126	31.60
30	0.9990	0.9988	0.0121	0.0081	33.07
40	0.9995	1.0006	0.0091	0.0062	32.29
50	0.9986	0.9989	0.0073	0.0048	34.31
100	1.0002	1.0001	0.0036	0.0024	32.81

Estimation of a gamma mean = 1, using Accept-Reject with acceptance rate 0.8 based on 10000 simulations.

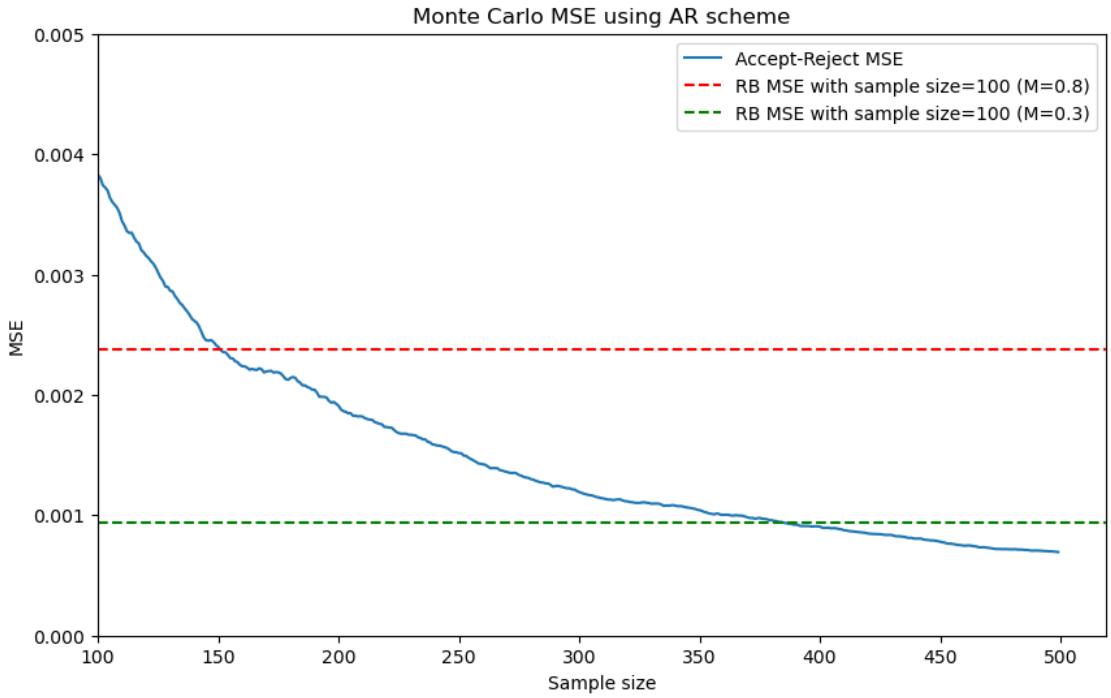
Sample size	AR estimate	RB estimate	AR MSE	RB MSE	Decrease in MSE (%)
5	1.0022	1.0018	0.0746	0.0290	61.07
10	0.9955	0.9988	0.0371	0.0113	69.32
15	0.9995	0.9994	0.0236	0.0067	71.42
20	1.0009	1.0006	0.0187	0.0050	73.26
30	0.9978	1.0001	0.0119	0.0031	74.00
40	1.0001	1.0002	0.0092	0.0022	75.48
50	0.9991	0.9994	0.0072	0.0018	74.96
100	1.0006	1.0003	0.0038	0.0008	76.73

Estimation of a gamma mean = 1, using Accept-Reject with acceptance rate 0.3 based on 10000 simulations.

We observe that the Rao-Blackwellized estimator offers a substantial improvement over the classic Monte-Carlo estimator. An improvement of 30% is given when using an acceptance rate of 80%, meanwhile the improvement almost reaches 77% when using an acceptance rate of 30%.

The reason is quite intuitive : Since the Rao-Blackwellized estimator makes use of the proposed sample instead of the accepted one, a lower acceptance rate means that the proposed sample must be bigger to achieve the same accepted sample size. Thus, $\hat{\theta}_1^{(RB)}$ has access to a much bigger sample and improves the MSE by much more.

While $\hat{\theta}_1^{(RB)}$ improves on $\hat{\theta}_1$ given the same accepted sample size, if we set $t' > t$, by how much do we have to increase t' for the Monte-Carlo estimator using t' to reach the same MSE as the Rao-Blackwellized estimator using t ?



As we can see, for an acceptance rate of 80%, increasing the sample size by only 50 is enough to reach the same MSE, while for an acceptance rate of 30%, increasing the sample size by 300 is needed.

To get a sense of wether or not it is computationally viable to use the Rao-Blackwellized estimator, let's compare the computation time of both of these estimators :

Estimator	Size	Average time (s)	MSE
$\hat{\theta}_1$	150	0.0004	0.0023
$\hat{\theta}_1^{(RB)}$	100	0.0062	0.0025

All measurements taken with an acceptance rate of 80%, and averaged over 1000 runs.

Estimator	Size	Average time (s)	MSE
$\hat{\theta}_1$	400	0.002	0.0008
$\hat{\theta}_1^{(RB)}$	100	0.039	0.0008

All measurements taken with an acceptance rate of 30%, and averaged over 1000 runs.

We can see that computing the Rao-Blackwell estimator does have a non-negligible computation time, taking 15 to 20 times the amount of time of the computation time of the Monte-Carlo estimator to achieve the same MSE. Computationally, it would be more interesting to compute a bigger sample size, however, this approach may not always be possible. Furthermore, relying on the simulation of random variables is statistical deficiency fixed by the Rao-Blackwell estimator.

4 Independent Metropolis Hasting

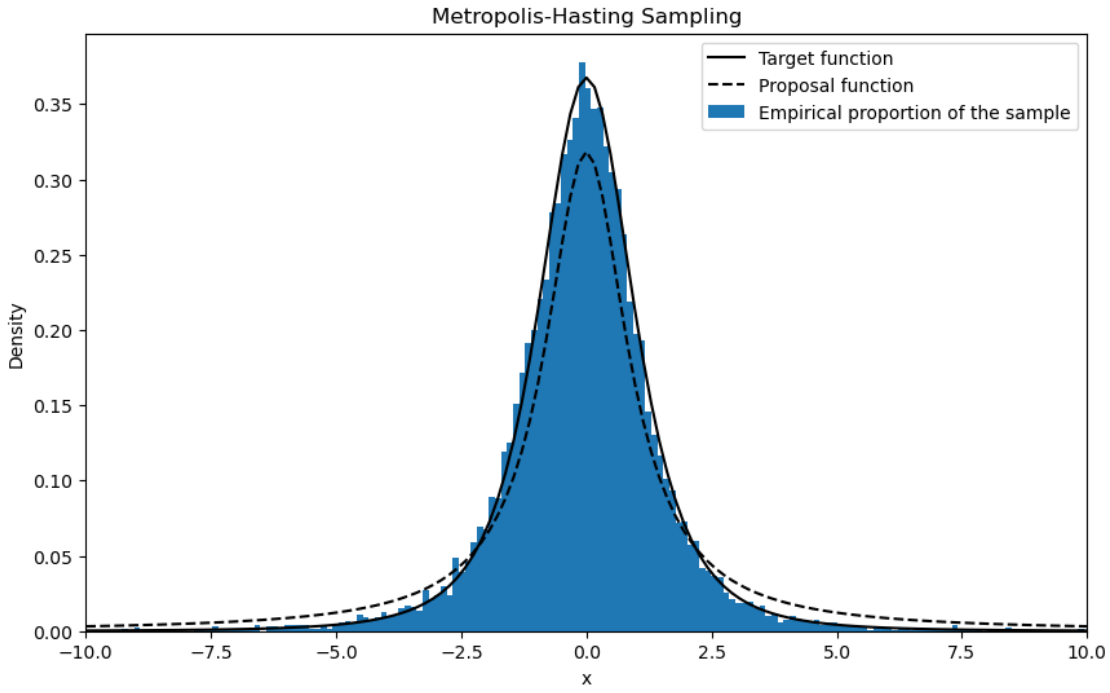
The python code related to this section can be found inside the RB_Independent_MH.ipynb file.

4.1 Presentation of the Algorithm

Algorithm: Let f be the target distribution and g the proposal distribution.

1. **Initialization:** Sample Z_0 from f or g.
2. **For each iteration i:**
   - Propose a candidate Y_{i+1} sampled from g.
   - Compute ρ_{i+1}:
   $$\rho_{i+1} = \min\left(1, \frac{f(Y_{i+1})g(Z_i)}{f(Z_i)g(Y_{i+1})}\right)$$
   - Set Z_{i+1}:
   $$Z_{i+1} = \begin{cases} Z_i & \text{with probability } 1-\rho_{i+1}, \\ Y_{i+1} \sim g(y) & \text{with probability } \rho_{i+1}. \end{cases}$$

The resulting accepted sequence $Z_0, Z_1, ...Z_n$ can be used to approximate the target distribution f. Usually, the convergence only happens after a "burn-in" period. Note that the proposed $Y_0, Y_1, ...Y_n$ is also of size $n+1$ and that we will mostly be sampling Z_0 from the target distribution directly.



This is a first naïve implementation of this algorithm where the whole process is contained in a loop :

```
n = 20000
samples = []
current_sample = f.rvs()
```

```
for _ in range(n):
    proposed_sample = g.rvs()
    acceptance_ratio = min((f.pdf(proposed_sample) * g.pdf(current_sample)) / (f.
 ↪pdf(current_sample) * g.pdf(proposed_sample)),1)

    if np.random.rand() < acceptance_ratio:
            current_sample = proposed_sample

    samples.append(current_sample)
```

This version is quite slow and not suitable to generate large amount of random variables.

Here is a somewhat optimized implementation of this algorithm. Instead of simulating the proposal one at a time, we simulate in bulk the whole proposal sample and pre-calculate $f(Y_i)$ and $g(Y_i)$ for $i \in \{0, ..., n\}$. This operation had to be done outside of the main loop because the Just In Time module does not work with scipy.stats functions.

Furthermore, since the acceptance ratio depends on the previous value of the chain, we can not compute it in bulk like we did for Accept-Reject. However, we do try to avoid redundant computation as much as possible.

```
@jit(nopython=True)
def loop(num_samples,current_sample,proposed_samples,random_uniforms,F,G):

    accepted_samples = np.empty(num_samples + 1)  # Preallocate memory
    accepted_samples[0] = current_sample

    current_f_pdf = F[0]
    current_g_pdf = G[0]

    for i in range(1, num_samples + 1):
        proposed_f_pdf = F[i]
        proposed_g_pdf = G[i]

        # Calculate the acceptance ratio
        acceptance_ratio = min(1, (proposed_f_pdf / proposed_g_pdf) /
                               (current_f_pdf / current_g_pdf))

        # Decide whether to accept the proposal
        if random_uniforms[i] < acceptance_ratio:
            current_sample = proposed_samples[i]
            current_f_pdf = proposed_f_pdf
            current_g_pdf = proposed_g_pdf

        accepted_samples[i] = current_sample
    return accepted_samples, proposed_samples

def metropolis_hastings(num_samples,target_density_start=True):
    current_sample = f.rvs() if target_density_start else g.rvs()
    proposed_samples = g.rvs(size=num_samples + 1)
    proposed_samples[0] = current_sample
    random_uniforms = np.random.rand(num_samples + 1)
```

```
    F = f.pdf(proposed_samples)
    G = g.pdf(proposed_samples)
    return loop(num_samples,current_sample,proposed_samples,random_uniforms,F,G)
```

The optimized version is much faster than the old one :

Function	Total time (s)
Naïve Metropolis-Hasting	12.71
Optimized Metropolis-Hasting	0.32

Measurements taken with n=50000

Metropolis-Hasting is used when it is difficult to sample from f directly. Unlike the Accept-Reject algorithm, the ratio between the target and the proposal does not need to be bounded to work, but boundedness is desirable to achieve a faster convergence.

4.2 Rao-Blackwellization

First, we denote the classic Monte-Carlo estimation of $\mathbb{E}[h(Z)]$:

$$\hat{\tau}_2 = \frac{1}{n+1}\sum_{i=0}^{n} h(Z_i)$$

This estimator can also be written in the form :

$$\begin{aligned}\hat{\tau}_2 &= \frac{1}{n+1}\left[h(Z_0) + \sum_{i=1}^{n}(\mathbf{1}_{Z_i=Y_i}h(Y_i) + \mathbf{1}_{Z_i=Z_{i-1}}h(Z_{i-1}))\right] \\ &= \frac{1}{n+1}\sum_{i=0}^{n} h(Y_i)\sum_{j=i}^{n}\mathbf{1}_{Z_j=Y_i}\end{aligned}$$

If we define the following intermediary values :

$$w_i = f(Y_i)g(Y_i), \quad \rho_{ij} = \frac{w_i}{w_j} \wedge 1,$$

$$\xi_{ii} = 1, \quad \xi_{ij} = \prod_{k=i+1}^{j}(1-\rho_{ik}) \quad (i<j)$$

$$\delta_i = \mathbb{P}(Z_i = Y_i \mid Y_1, \ldots, Y_i) = \sum_{j=0}^{i-1}\delta_j\xi_{j(i-1)}\rho_{ji} \quad (i>0),$$

Then, the Rao-Blackwellized version of $\hat{\tau}_2$ is :

$$\begin{aligned}\hat{\tau}_2^{(RB)} &= \frac{1}{n+1}\mathbb{E}\left[\sum_{i=0}^{n} h(Y_i)\sum_{j=i}^{n}\mathbf{1}_{Z_j=Y_i}|Y_1, Y_2, \ldots Y_n\right] \\ &= \frac{1}{n+1}\sum_{i=0}^{n} h(Y_i)\mathbb{E}\left[\sum_{j=i}^{n}\mathbf{1}_{Z_j=Y_i}|Y_1, Y_2, \ldots Y_n\right] \\ &= \frac{1}{n+1}\sum_{i=0}^{n} h(Y_i)\delta_i\sum_{j=i}^{n}\xi_{ij} \\ &= \frac{1}{n+1}\sum_{i=0}^{n}\varphi_i h(Y_i) \quad \text{with } \varphi_i = \delta_i\sum_{j=i}^{n}\xi_{ij}\end{aligned}$$

φ_i is the expected number of times that $\{Z_j = Y_i\}$ occurs.

The python implementation for the intermediary values is pretty straightforward. Once again, to maximize efficiency, we pre-compute all the intermediary value while taking advantage of numpy vectorization and JIT as much as possible. To use JIT, we need to compute the pdf values of f and g with respect to the sample before calling the estimator function.

```
def compute_pdf_values(Y, f, g):
    f_pdf_values = f.pdf(Y)
    g_pdf_values = g.pdf(Y)
    return f_pdf_values, g_pdf_values

@jit(nopython=True)
def compute_rho(f_pdf_values, g_pdf_values):
    w_values = f_pdf_values / g_pdf_values
    rho = np.minimum(1, w_values / w_values[:, np.newaxis])
    return rho

@jit(nopython=True)
def compute_zeta(Y, rho):
    n = len(Y)
    zeta = np.eye(n)  # Start with an identity matrix
    for i in range(n):
        zeta[i, i+1:] = np.cumprod(1 - rho[i, i+1:])
    return zeta

@jit(nopython=True)
def compute_deltas(zeta, rho, n):
    deltas = np.zeros(n + 1)
    deltas[0] = 1  # By convention, delta_0 = 1
    for i in range(1, n + 1):
        sum_term = 0
        for j in range(i):
            sum_term += deltas[j] * zeta[j, i - 1] * rho[j, i]
        deltas[i] = sum_term
    return deltas

@jit(nopython=True)
def compute_rb_estimator(Y, f_pdf_values, g_pdf_values):
    n = len(Y) - 1
    rho = compute_rho(f_pdf_values, g_pdf_values)
    zeta = compute_zeta(Y, rho)
    deltas = compute_deltas(zeta, rho, n)
    est = 0.0
    for i in range(n + 1):
        est += deltas[i] * np.sum(zeta[i, i:]) * Y[i] / (n + 1)
    return est
```

We also define the importance sampling estimator that we will use in the comparative analysis :

$$\hat{\tau}_2^{(IS)} = \frac{\sum_{i=1}^{n} \frac{f(Y_i)}{g(Y_i)} h(Y_i)}{\sum_{i=1}^{n} \frac{f(Y_i)}{g(Y_i)}}$$

```python
@jit(nopython=True)
def compute_importance_sampling_estimator(Y, f_pdf_values, g_pdf_values):
    weights = f_pdf_values / g_pdf_values
    numerator = np.sum(weights * Y)
    denominator = np.sum(weights)
    return numerator / denominator
```

4.3 Comparative analysis

In this analysis, we try to estimate the mean of student distribution of degree 3 using the Cauchy distribution as the proposal. The true mean is equal to 0. Earlier, we have supposed that Z_0 was sampled from the target distribution. We will also analyze the situation where Z_0 is sampled from the proposal distribution.

Sample Size	MH Estimate	RB Estimate	IS Estimate	MC MSE	Decrease in MSE RB over MH	Decrease in MSE IS over MH
5	0.0013	-0.0022	0.0007	0.5722	39.69	47.91
10	-0.0028	-0.0055	-0.0060	0.3235	41.77	50.08
25	-0.0000	0.0004	0.0005	0.1408	47.30	53.19
50	-0.0017	0.0003	0.0000	0.0749	48.54	54.27
100	-0.0026	-0.0019	-0.0019	0.0372	47.57	53.15
200	0.0010	0.0010	0.0010	0.0181	47.67	52.95
500	-0.0005	-0.0001	-0.0000	0.0083	53.43	58.33
1000	-0.0002	-0.0004	-0.0003	0.0037	48.76	53.56

Estimation of the mean of a T(3) distribution with MH starting from the target density over 20000 runs

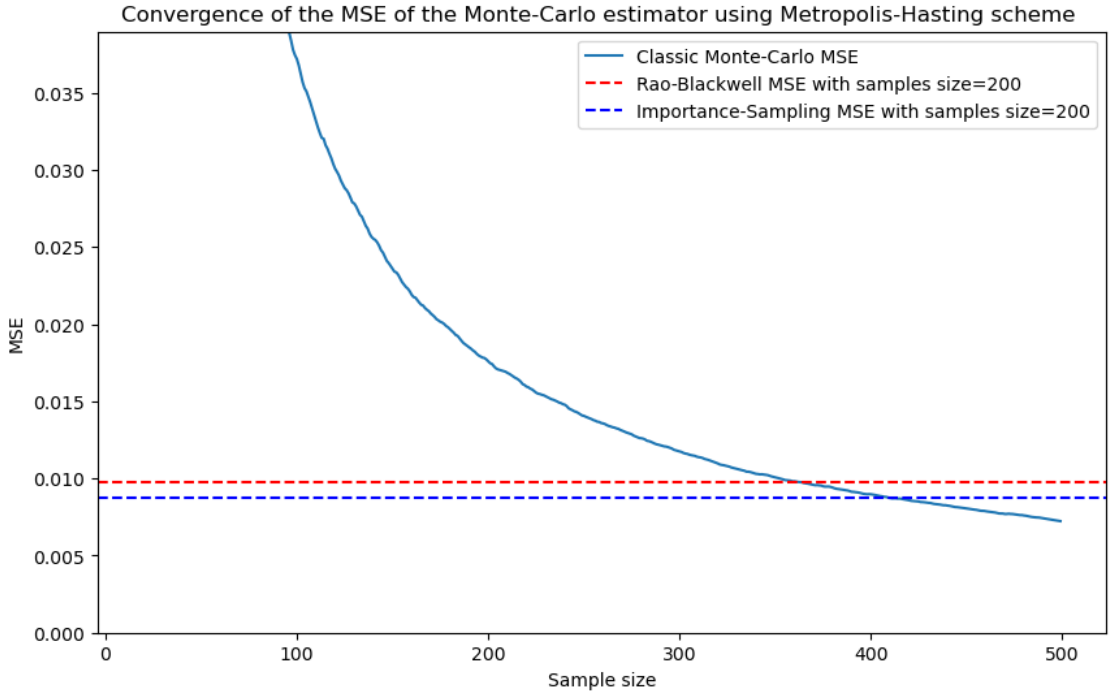
Sample Size	MH Estimate	RB Estimate	IS Estimate	MC MSE	Decrease in MSE RB over MH	Decrease in MSE IS over MH
5	0.0497	0.0526	-0.0007	369.51	0.03	99.91
10	0.2505	0.2497	0.0001	296.70	0.07	99.94
25	0.0209	0.0205	-0.0000	457.70	0.16	99.98
50	0.0171	0.0176	-0.0002	3.3868	1.22	98.98
100	0.0259	0.0272	-0.0005	7.3758	0.10	99.76
200	-0.0151	-0.0151	0.0000	1.7175	0.58	99.49
500	0.0007	0.0014	0.0006	0.0357	10.4	90.42
1000	-0.0026	-0.0025	0.0002	0.0631	2.85	97.28

Estimation of the mean of a T(3) distribution with MH starting from the proposal density over 20000 runs

When we sample the first value from the target density, we can see that both the Rao-Blackwell and the importance sampling estimator brings more or less the same improvement, with the importance sampling estimator being a bit better. The Rao-Blackwellized estimator decrease the MSE of 40-50% over the Monte-Carlo estimator, and the importance sampling estimator brings an improvement of 48-54%.
Now, on the second table, where the chain does not start from the target distribution but the proposal's, we can see that the MSE of the Monte-Carlo estimator is very high, meaning that the chain is having a hard time converging to the target distribution. Furthermore, the Rao-Blackwellized estimator barely bring any improvement.

For the rest of this analysis, we will focus on the case where Z_0 is sampled from the target distribution.

While $\hat{\theta}_2^{(RB)}$ and $\hat{\theta}_2^{(IS)}$ improves on $\hat{\theta}_2$ given the same sample size, if we set $n' > n$, by how much do we have to increase n' for the Monte-Carlo estimator using n' to reach the same MSE as the Rao-Blackwellized or the Importance sampling estimators using t ?



It seems that $\hat{\theta}_2$ reaches the same MSE as $\hat{\theta}_2^{(RB)}$ and $\hat{\theta}_2^{(IS)}$ when n' is set to roughly 400. Let's take a look at the computation time in this case :

Estimator	Size	Average time (s)	MSE
Monte-Carlo	400	0.00022	0.0093
Rao-Blackwell	200	0.00041	0.0096
Importance Sampling	200	0.00032	0.0086

Average measurements taken over 10000 runs

Unlike the Accept-Reject algorithm, these measurement are quite close to each-other. However, it seems that the Importance Sampling estimator would be the best choice here, with a computation time which is very close to the Monte-Carlo estimator while using less sample. Furthermore, the comparison is not even close when considering the case where Z_0 does not start from the target density.

5 General Metropolis Hasting and Rao-Blackwellized Importance Sampling

The code related to this section can be found in RB_MH_IS.ipynb

5.1 Presentation of the algorithm

Algorithm: Let f be the target distribution and g the proposal distribution.

1. **Initialization:** Sample Z_0 from f
2. **For each iteration i:**
   - Propose a candidate Y_{i+1} sampled from g.
   - Compute ρ_{i+1}:
   $$\rho_{i+1} = \min\left(1, \frac{f(Y_{i+1})g(Z_i|Y_{i+1})}{f(Z_i)g(Y_{i+1}|Z_n)}\right)$$
   - Set Z_{i+1}:
   $$Z_{i+1} = \begin{cases} Z_i & \text{with probability } 1-\rho_{i+1}, \\ Y_{i+1} \sim g(y_{i+1}|Z_i) & \text{with probability } \rho_{i+1}. \end{cases}$$

The resulting accepted sequence $Z_0, Z_1, ...Z_n$ can be used to approximate the target distribution f. Usually, the convergence only happens after a "burn-in" period. Note that the proposed $Y_0, Y_1, ...Y_n$ is also of size $n+1$ and that $Z_0 \sim f$

The naïve implementation works the same way as the independent Matropolis-Hasting case, so we will focus on the optimized version instead :

```
@jit(nopython=True)
def metropolis_hastings(numb_runs, num_samples, sigma, print_acceptance_rate=False):
    proposed_samples = np.zeros((numb_runs, num_samples + 1))
    accepted_samples = np.zeros((numb_runs, num_samples + 1))
    random_uniforms = np.random.rand(numb_runs, num_samples + 1)
    compteur_accept = 0


    for j in range(numb_runs):
        current_sample = f_sampling()  # Starting sample from the target
        accepted_samples[j, 0] = current_sample
        proposed_samples[j, 0] = current_sample
        f_current = f_target(current_sample)

        for i in range(1, num_samples + 1):
            proposal = g_sampling(current_sample, sigma)
            g_current_to_proposal = g_proposal(current_sample, proposal, sigma)
            g_proposal_to_current = g_proposal(proposal, current_sample, sigma)


            f_proposed = f_target(proposal)

            # Calculate the acceptance ratio
            acceptance_ratio = (f_proposed / f_current) * (g_current_to_proposal /␣
↪g_proposal_to_current)
            acceptance_ratio = min(1, acceptance_ratio)

            # Accept or reject the proposal
            if random_uniforms[j, i - 1] < acceptance_ratio:
                current_sample = proposal
                f_current = f_proposed
                compteur_accept += 1

            proposed_samples[j, i] = proposal
            accepted_samples[j, i] = current_sample
```

```
    if print_acceptance_rate:
        acceptance_rate = 100 * compteur_accept / (numb_runs * num_samples)
        print("Acceptance Rate with sigma =", sigma, "is", acceptance_rate, "%")
    return accepted_samples, proposed_samples
```

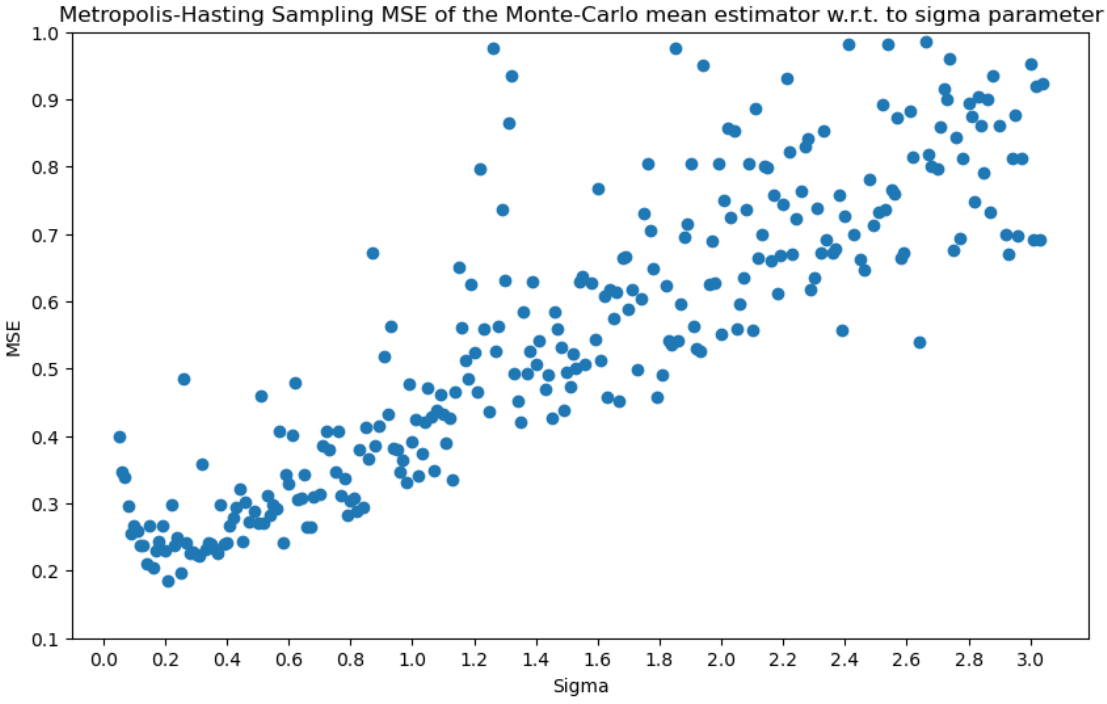
This time, we could not use pre-computation of the proposed sample, since said proposed sample depends on the last accepted candidate. This means that we could not initally use JIT to speed up the function since JIT does not work when scipy function are presents in the function.
Instead, we stopped using scipy and defined the target and distribution and a way to sample from them directly. This way, we could use JIT and speed up this function by a factor of 30 or so. Coincidentally, it turns out that the pdf method from scipy is quite slow too.

Function	Total time (s)
General Metropolis-Hasting	5.17
General Metropolis-Hasting optimized	0.18

Measurements taken with n=10000

For the next graph, we use a Student distribution target and a Cauchy distribution centered on the previous accepted candidate.

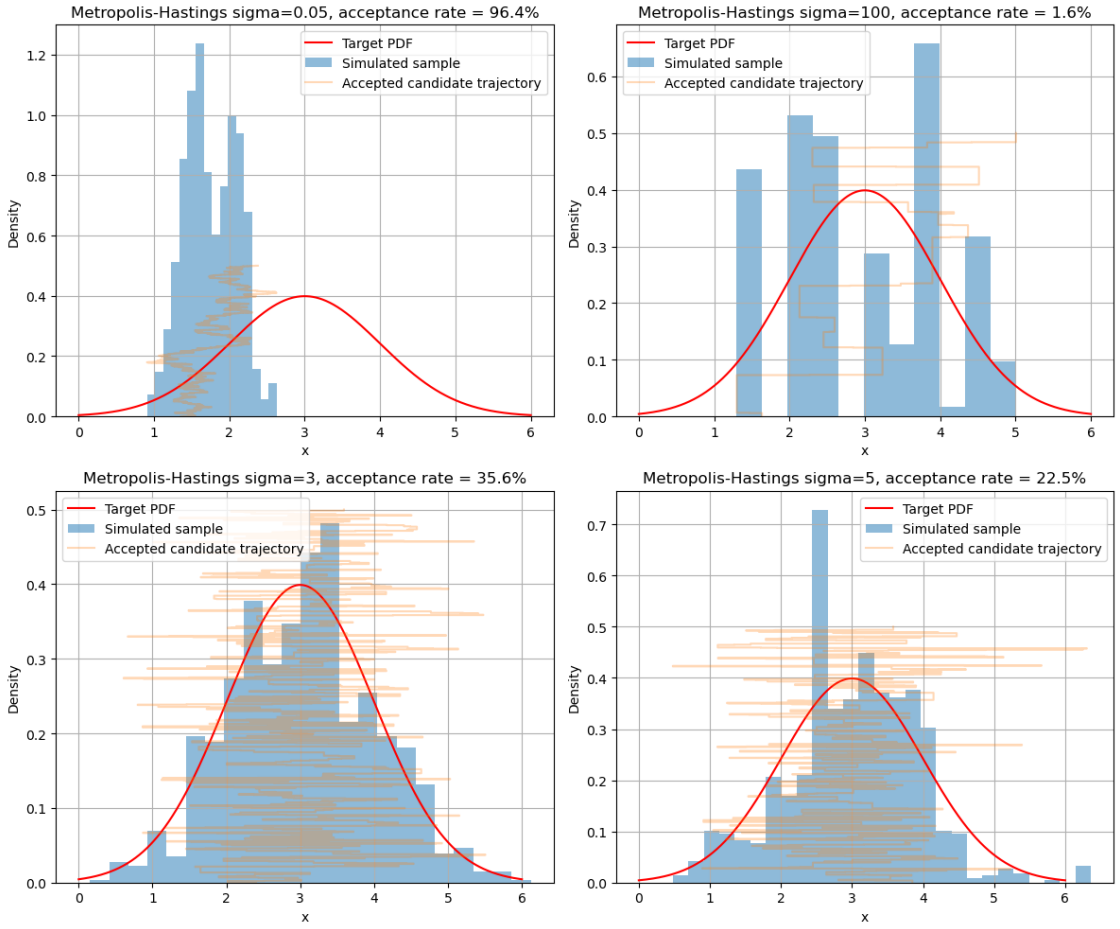
When choosing the proposal, we can scale it using the σ parameter. The choice of scaling of the proposal distribution is very important to ensure that the Metropolis Hasting chain explores the space properly. Changing the σ parameter influence the acceptance rate, which in turn determines how well Metropolis-Hasting converges to the target distribution :



We can see here that the optimal value of sigma seems to be around 0.2 to 0.3, which yield an acceptance rate of 22-30%, which aligns with the findings of Roberts and Rosenthal in their paper on optimal scaling

for Metropolis-Hastings algorithms [2].

On the next graph, we are using a Gaussian target and proposal distribution to see the trajectory of the chain of size 1000 generated by Metropolis-Hasting :



We can observe three distinct cases when analyzing the chain trajectory for different acceptance rates in the Metropolis-Hastings algorithm:

- High acceptance rate: The chain remains mostly stationary, making small leaps between states.
- Low acceptance rate: The chain moves infrequently, but when it does, it makes large leaps.
- Balanced acceptance rate: The chain effectively explores the space of the target distribution, providing good convergence properties.

5.2 Rao-Blackwellization of the general Metropolis algorithm

First, we denote the classic Monte-Carlo estimation of $\mathbb{E}[h(Z)]$:

$$\hat{\tau}_3 = \frac{1}{n+1} \sum_{i=0}^{n} h(Z_i)$$

If we define the following intermediary values :

$$\rho_{ij} = \frac{f(Y_j)g(Y_j|Y_i)}{f(Y_i)g(Y_i|Y_j)} \wedge 1,$$
$$\rho^*_{ij} = \rho_{ij} g(Y_{j+1}|Y_j), \rho_{*ij} = (1-\rho_{ij})g(Y_{j+1}|Y_i) \quad (i<j<n)$$
$$\xi_{jj} = 1, \quad \xi_{jt} = \prod_{i=j+1}^{t} \rho_{*ji} \quad (j<t<n)$$
$$\delta_0 = 1, \ \delta_j = \sum_{t=0}^{j-1} \delta_t \xi_{t(j-1)} \rho^*_{tj}, \ \delta_n = \sum_{t=0}^{n-1} \delta_t \xi_{t(n-1)} \rho_{tn} \quad (j<n),$$
$$\omega^j_n = 1, \quad \omega^j_i = \rho^*_{ji}\omega^i_{i+1} + \rho_{*ji}\omega^j_{i+1} \quad (0 \le j < i < n).$$

Then, the Rao-Blackwellized version of $\hat{\tau}_3$ is :

$$\hat{\tau}_3^{RB} = \frac{1}{n+1}\mathbb{E}\left[\sum_{i=0}^{n} h(Y_i) \sum_{j=i}^{n} \mathbf{1}_{Z_j=Y_i} | Y_1, Y_2, ...Y_n\right]$$
$$= \frac{1}{n+1} \frac{\sum_{i=0}^{n} \varphi_i h(Y_i)}{\sum_{i=0}^{n-1} \delta_i \xi_{i(n-1)}}$$

where $\varphi_n = \delta_n$ and, for $i < n$:

$$\varphi_i = \delta_i \left(\sum_{j=i}^{n-1} \xi_{ij}\omega^i_{j+1} + \xi_{i(n-1)}(1-\rho_{in}) \right)$$

The python implementation of this estimator is quite long and can be found inside the ipynb file. Once again, it is important to pre-compute the intermediary values to avoid redundant calculations. The only notable part of this implementation would be the ω matrix :

```
@jit(nopython=True)
def compute_omega(Y,rho_hat_star, rho_star):
    n = len(Y)
    omegas = np.zeros((n, n))
    for i in range(n):
        omegas[n-1,i]=1
    for i in range(n - 2, 0, -1):
        for j in range(i):
            omegas[i, j] = rho_hat_star[j,i] * omegas[i + 1, i] + rho_star[j, i] *␣
↪omegas[i + 1, j]
    return omegas
```

Here, we used the same method of computing the recursive function inside a Matrix instead of using nested functions.

Finaly, we can implement the estimator :

```
@jit(nopython=True)
def rao_blackwell_estimator(Y,sigma,h):

    #Pre-computing
    n = len(Y) - 1
```

```
    F = compute_F(Y)
    G = compute_G(Y,sigma)
    rho = compute_rho(Y, F, G)
    rho_hat_star = compute_rho_hat_star(Y, rho, G)
    rho_star = compute_rho_star(Y, rho, G)
    zeta = compute_zeta(Y, rho_star)
    deltas = compute_deltas(Y, zeta, rho_hat_star, rho)
    omegas = compute_omega(Y, rho_hat_star, rho_star)

    #Denominator
    den=0
    for i in range(n):
        den += deltas[i] * zeta[i, n-1]

    #Numerator
    num=0
    for i in range(n + 1):
        phi_i = compute_phi(i, Y, rho, zeta, deltas, omegas)
        num += phi_i * h(Y[i])

    return (num / den) / (n+1)
```

5.3 Rao-Blackwellization of the Importance Sampling estimator

We have already used an importance estimator earlier, but let's define it properly now.

We denote f and g the target and proposal distribution as usual. Then, we can write :

$$\mathbb{E}_f[h(y)] = \int h(y)f(y)\,dy = \int h(y)\frac{f(y)}{g(y)}g(y)\,dy = \mathbb{E}_g[h(y)\frac{f(y)}{g(y)}] \approx \frac{1}{n}\sum_{i=1}^{n} h(y_i)\frac{f(y_i)}{g(y_i)}$$

where $y_1, y_2, ...y_n$ are sampled from g.

Importance sampling is a variance reduction technique that can be used to enhance the Monte Carlo estimator. The importance weights $\frac{f(x)}{g(x)}$ serve to adjust for differences in the distribution from which samples are drawn ($g(x)$) versus the target distribution ($f(x)$):

- Higher weights for undersampled areas: If a region is undersampled by $g(x)$ relative to its importance under $f(x)$, the ratio $\frac{f(x)}{g(x)}$ increases. This gives higher weights to samples from these regions, compensating for their underrepresentation in the sample set.
- Lower weights for oversampled areas: Conversely, if $g(x)$ oversamples a region compared to $f(x)$, the ratio $\frac{f(x)}{g(x)}$ decreases. This reduction in the weight reduces the influence of these samples, mitigating the effect of their oversampling.

Now, we can introduce importance sampling estimator based on the Metropolis algorithm :

$$\hat{\tau}_3^{IS} = \frac{1}{n+1}\sum_{j=0}^{n}\omega_j h(Y_j) \quad \text{with } \omega_j = \frac{f(Y_i)}{g(Y_i|Z_{i-1})}$$

However, since the distribution Y_i depends on the value of Z_{i-1}, this expression still relies on the ancillary uniform random variables. By using the intermediary values implemented in the last sub-section, we can define :

$$\hat{\tau}_3^{IS-RB} = \frac{1}{n+1}\left(h(Z_0) + \frac{f(Y_1)}{g(Y_1 \mid Y_0)}h(Y_1) + \sum_{t=2}^{n}\left(\sum_{j=0}^{t-1} f(Y_t)\delta_j\xi_{j(t-2)}(1-\rho_{j(t-1)})\right)h(Y_t)\right)\left(\sum_{i=0}^{n-1}\delta_i\xi_{i(n-1)}\right)^{-1}$$

The formula may seems complex, but it is not hard to implement, given that we have taken care of the intermediary values earlier.

```
@jit(nopython=True)
def rao_blackwell_importance_sampling_estimator(Y,sigma,h):

    #Pre-computing
    n = len(Y)-1
    F = compute_F(Y)
    G = compute_G(Y,sigma)
    rho = compute_rho(Y, F, G)
    rho_hat_star = compute_rho_hat_star(Y, rho, G)
    rho_star = compute_rho_star(Y, rho, G)
    zeta = compute_zeta(Y, rho_star)
    deltas = compute_deltas(Y, zeta, rho_hat_star, rho)
    omegas = compute_omega(Y, rho_hat_star, rho_star)

    #Denominator
    den = 0
    for i in range(n):
        den+= deltas[i]*zeta[i,n-1]

    #Numerator
    num = 0
    for i in range(2,n+1):
        for j in range(0,i):
            num+= F[i]*deltas[j]*zeta[j,i-2]*omegas[i,j]*h(Y[i])

    return (h(Y[0]) + h(Y[1])*F[1]/G[1,0]+(num/den)) / (n+1)
```

5.4 Remarks concerning the implementation

Both $\hat{\tau}_3^{IS-RB}$ and $\hat{\tau}_3^{RB}$ have some inconvenience related to their implementation. In particular, calculating the denominator of both these estimators can lead to some computational issues.
The denominator is defined as :

$$\sum_{i=0}^{n-1} \delta_i \xi_{i(n-1)}$$

We need ξ and δ to compute the denominator of both $\hat{\tau}_3^{IS-RB}$ and $\hat{\tau}_3^{RB}$. This is the formula of ξ and δ as a reminder :

$$\xi_{jj} = 1, \quad \xi_{jt} = \prod_{i=j+1}^{t} \rho_{*ji} \quad (j < t < n)$$

$$\delta_0 = 1, \ \delta_j = \sum_{t=0}^{j-1} \delta_t \xi_{t(j-1)} \rho_{tj}^*, \ \delta_n = \sum_{t=0}^{n-1} \delta_t \xi_{t(n-1)} \rho_{tn} \quad (j < n)$$

Since ρ_{*ji} is often equal to 0, the matrix ξ is sparse, in particular the last column is made of 0 and small numbers. On the other hand, since the matrix ξ is sparse, $\delta[i]$ can get so low that that python will round it to 0. Thus, the sum of the denominator will stay at 0, which will promptly send a "ZeroDivisionError: division by zero" error. For instance, using an acceptance rate of 33% with the Student target and Cauchy proposition, we would get this problem when the chain length gets larger than ≈ 180.

5.5 Comparative Analysis

In this analysis, we try to estimate the mean and the tail probability of student distribution of degree 3 using the Cauchy distribution centered on the previous accepted variable as the proposal. The proposal distribution will be scaled by σ to obtain acceptance rates of 33% and 75% The true mean is equal to 0. Earlier. We will be comparing $\hat{\tau}_3$, $\hat{\tau}_3^{RB}$, $\hat{\tau}_3^{IS}$ and $\hat{\tau}_3^{IS-RB}$

Sample Size	MC Metrop. Estimate	RB Estimate	IS Estimate	IS-RB Estimate	MC MSE	Decrease in MSE RB over MH	Decrease in MSE IS over MH	Decrease in MSE IS-RB over MH
5	0.0083	0.0066	0.0006	0.0021	2.024	10.15	85.67	81.82
10	0.0110	0.0096	0.0009	0.0014	1.651	10.20	89.32	88.31
25	0.0020	0.0012	-0.0006	-0.0007	0.910	9.42	91.44	91.40
50	0.0042	0.0035	0.0005	0.0001	0.521	8.74	92.00	92.26
100	0.0005	0.0024	0.0011	0.0003	0.374	5.74	94.41	94.72
150	-0.0003	-0.0005	0.0002	-0.0000	0.191	8.24	92.73	93.20

Estimation of the mean of a T(3) distribution with acceptance rate 33% over 50000 runs

Sample Size	MC Metrop. Estimate	RB Estimate	IS Estimate	IS-RB Estimate	MC MSE	Decrease in MSE RB over MH	Decrease in MSE IS over MH	Decrease in MSE IS-RB over MH
5	-0.0014	-0.0014	0.0012	-0.0008	2.713	0.61	69.17	63.24
10	-0.0003	0.0004	0.0041	0.0020	2.199	0.33	77.19	73.08
25	-0.0006	-0.0009	-0.0006	0.0004	1.686	0.16	85.62	83.08
50	0.0078	0.0078	0.0007	0.0014	1.243	0.14	89.88	88.11
100	-0.0033	-0.0032	0.0010	0.0010	0.832	0.11	92.55	91.22
150	-0.0065	-0.0066	0.0007	0.0011	1.071	0.05	96.03	95.09

Estimation of the mean of a T(3) distribution with acceptance rate 75% over 50000 runs

Sample Size	MC Metrop. Estimate	RB Estimate	IS Estimate	IS-RB Estimate	MC MSE	Decrease in MSE RB over MH	Decrease in MSE IS over MH	Decrease in MSE IS-RB over MH
5	0.0501	0.0498	0.0417	0.0501	0.028	21.56	90.21	85.77
10	0.0500	0.0496	0.0453	0.0498	0.021	25.15	92.19	90.72
25	0.0501	0.0499	0.0480	0.0500	0.011	24.44	93.24	92.87
50	0.0498	0.0498	0.0491	0.0500	0.006	24.85	93.66	93.64
100	0.0500	0.0499	0.0495	0.0500	0.003	23.57	94.11	94.20
150	0.0502	0.0502	0.0497	0.0500	0.002	24.38	94.08	94.18

Estimation of the tail of a T(3) distribution chosen to be 0.05 distribution with acceptance rate 33% over 50000 runs

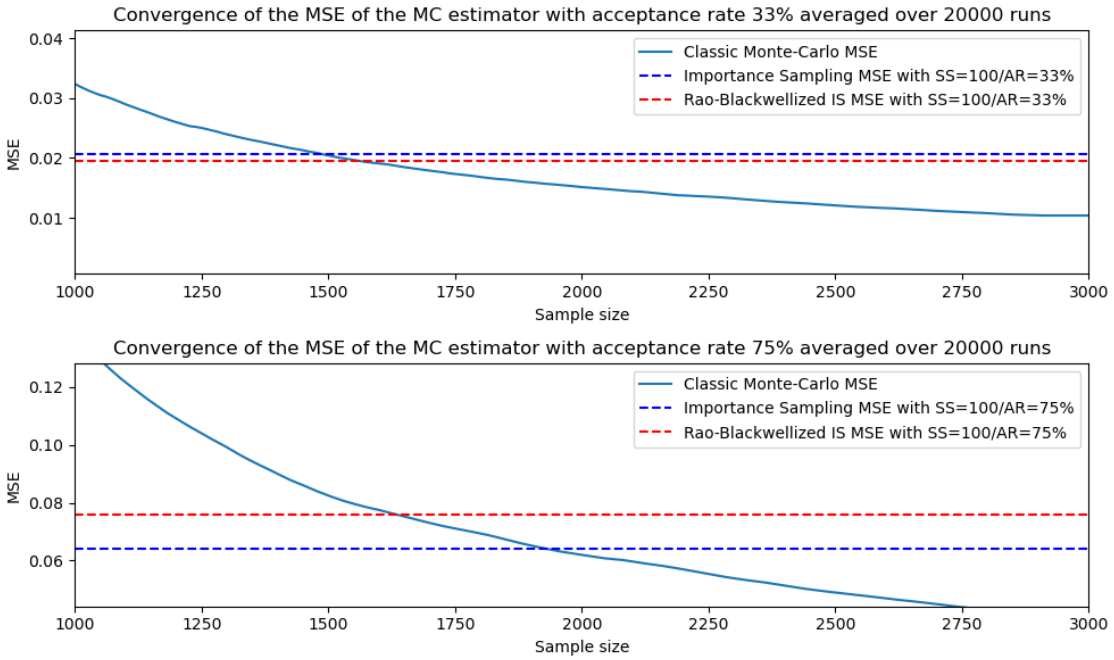
Sample Size	MC Metrop. Estimate	RB Estimate	IS Estimate	IS-RB Estimate	MC MSE	Decrease in MSE RB over MH	Decrease in MSE IS over MH	Decrease in MSE IS-RB over MH
5	0.0482	0.0482	0.0408	0.0492	0.034	2.34	65.49	52.87
10	0.0500	0.0500	0.0457	0.0502	0.030	1.59	76.36	65.71
25	0.0498	0.0497	0.0481	0.0502	0.021	0.92	84.41	75.20
50	0.0500	0.0500	0.0491	0.0501	0.014	0.58	86.60	80.97
100	0.0504	0.0504	0.0494	0.0498	0.008	0.58	89.42	84.17
150	0.0497	0.0497	0.0496	0.0499	0.006	0.60	89.79	84.10

Estimation of the tail of a T(3) distribution chosen to be 0.05 with acceptance rate 75% over 50000 runs

We can see that $\hat{\tau}_3^{RB}$ does not bring a signficant improvement over $\hat{\tau}_3$. With an acceptance rate of 33%, the improvement is about 8-10% while with an acceptance rate of 75% it barely brings any improvement at all for the estimation of the mean. For the tail probability however, the improvement brought reaches 25% using an acceptance rate of 33%. Over all, it is a bit lackluster compared to the improvement brought on for the previous sampling schemes.

However, both the importance sampling estimators bring a substantial improvement, ranging from 70 to 90% for the fours different scenarios. On the one hand, we could say that $\hat{\tau}_3^{IS}$ is better since it is easier and faster to implement, while bringing similar improvement as $\hat{\tau}_3^{IS-RB}$. On the other hand, $\hat{\tau}_3^{IS-RB}$ does seems to be less biased than $\hat{\tau}_3^{IS}$. For instance, when it comes to calculating the tail probability, we can see that $\hat{\tau}_3^{IS}$ is not very accurate below a sample size of 50, meanwhile $\hat{\tau}_3^{IS-RB}$ is always pretty accurate.

Lastly, we aim to determine the additional sample size required for the Monte Carlo estimator to achieve MSE reductions comparable to those of the importance sampling estimators. Given the superior performance of importance sampling and the negligible improvement from the Rao-Blackwellized Monte Carlo estimator, this analysis will focus solely on the importance sampling methods.



We can make the following observations :

- Acceptance Rate 33%: Both importance sampling estimator are very close to each other in terms of MSE. It takes around a sample size for 1500 for the Monte Carlo estimator to reach the same MSE.
- Acceptance Rate 75%: In this case, the importance sampling estimator is slightly better than the Rao-Blackwellized importance sampling estimator, and it should takes a sample size of 2000 for the Monte Carlo estimator to reach a similar MSE.

Let us perform some time measurement to conclude this section :

Estimator	Size	Average time (s)	MSE
Monte-Carlo	1400	0.00011	0.021
Importance Sampling	100	0.00001	0.020
Rao-Blackwellized Imp. Sampl.	100	0.00009	0.019

Average measurements taken with acceptance rate of 33% over 500 runs

Estimator	Size	Average time (s)	MSE
Monte-Carlo	2000	0.00015	0.072
Importance Sampling	100	0.00001	0.070
Rao-Blackwellized Imp. Sampl.	100	0.00009	0.069

Average measurements taken with acceptance rate of 75% over 500 runs

In this case, we can see that both importance sampling estimators should be favored compared to the Monte Carlo estimator. They simultaneously need much less sample to reach a low MSE and their computing time is faster than actually simulating the required surplus of sample for Monte-Carlo to reach their MSE.

Furthermore, the Rao-Blackwellized importance sampling estimator should be pushed aside since the computation time of the classic importance sampling estimator is about 9 times faster. Plus, this estimator is very easy to implement compared to the Rao-Blackwell version.

References

[1] Casella, George, and Robert, Christian P. (1996). Rao-Blackwellisation of sampling schemes. *Biometrika*, 83(1), 81–94.

[2] Roberts, Gareth O., and Rosenthal, Jeffrey S. (2001). Optimal scaling for various Metropolis–Hastings algorithms. *Statistical Science*, 16(4), 351–367.