

# Diagrammatic Monte Carlo

## Exercises

---

Jonas Märtens

June 25, 2023

University of Bologna

# Contents

1. Task 1: Monte Carlo Basics - Estimating PI

2. Task 2: CDF Inversion

3. Task 3: Towards DMC - MCMC

4. Task 4&5: Diagrammatic Monte Carlo

5. Task 6: DMC - Green Function Estimator

## Task 1: Monte Carlo Basics - Estimating PI

---

# Estimating PI - Task

Two different methods to estimate  $\pi$ :

- Area Method (as seen in the picture)
- MC Integration Method

$$\pi/4 = \int_0^1 \sqrt{1-x^2} dx = \int_0^1 \frac{\sqrt{1-x^2}}{p(x)} p(x) dx =$$

$$\left\langle \frac{\sqrt{1-x^2}}{p(x)} \right\rangle_p \approx \frac{1}{N} \sum_{i=1}^N \frac{\sqrt{1-x_i^2}}{p(x_i)} \text{ for } p(x) = 1 \text{ and } x_i \sim U(0,1)$$

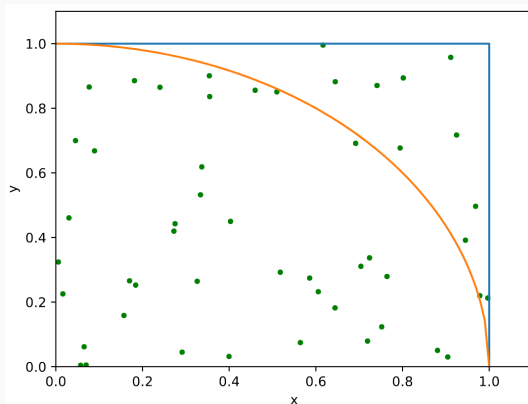


Figure 1: Visualization of the area method.[Ces22]

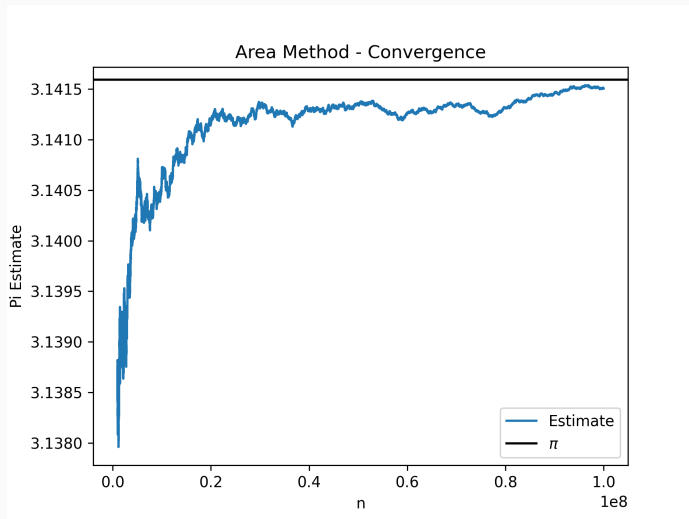
# Estimating PI: Area Method - Implementation

```
1 def area_method(n, plot=False):
2     n_in = 0
3     estimates_n = np.array([], dtype=int)
4     estimates_area = np.array([])
5     for i in range(n):
6         # generate random numbers between 0 and 1
7         x = np.random.random()
8         y = np.random.random()
9         # check if the point is in the circle
10        if x**2 + y**2 <= 1:
11            n_in += 1
12        # every 10000 points, calculate the area
13        if plot:
14            if i % (int(n / 100000)) == 0 and i > 100000:
15                estimates_n = np.append(estimates_n, i + 1)
16                estimates_area = np.append(estimates_area, 4 * n_in / (i + 1))
17    if plot:
18        # Do the plotting here
19        # ...
20    return 4 * n_in / n
```

# Estimating $\pi$ : Area Method - Convergence

1

```
area_method(int(1e8), plot=True)
```

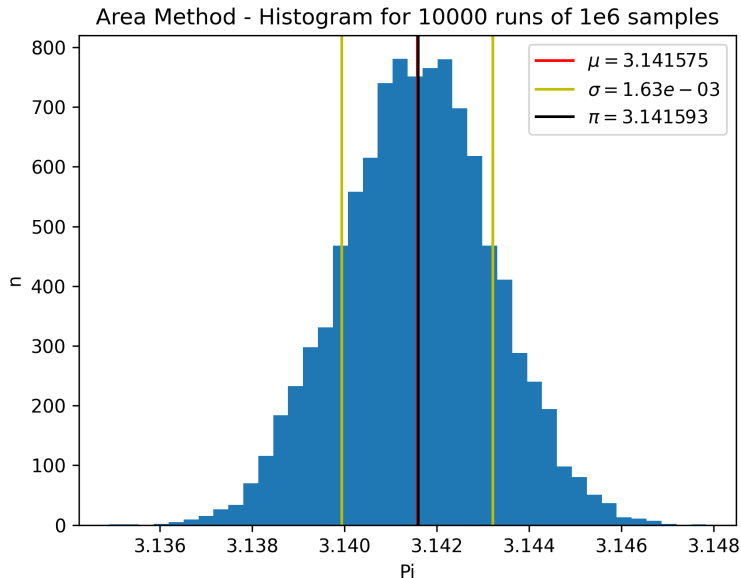


# Estimating PI: Area Method - We can do better

```
1 def area_method_fast(n):
2     x = np.random.random(n)
3     y = np.random.random(n)
4     n_in = np.sum(x**2 + y**2 <= 1)
5     return 4 * n_in / n
```

```
1 pis = np.zeros(10000)
2 for i in range(10000):
3     pis[i] = area_method_fast(int(1e6))
4 std = np.std(pis)
5 mean = np.mean(pis)
6 # Do the plotting here
7 # ...
8 print("Area Method:")
9 print(f"{mean = }")
10 print(f"{std = }")
```

# Estimating PI: Area Method - Histogram





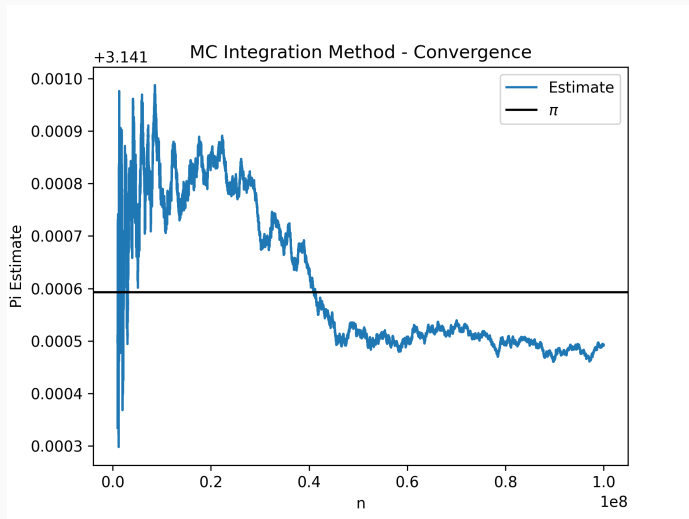
# Estimating PI: MC Integration Method - Implementation

```
1 def p(x):
2     return 1
3
4 def f(x):
5     return np.sqrt(1 - x**2)
6
7 def mc_integrate(n, plot=False):
8     sum = 0
9     estimates_n = np.array([], dtype=int)
10    estimates_area = np.array([], dtype=np.float32)
11    for i in range(n):
12        x = np.random.random()
13        sum += f(x) / p(x)
14        if plot and i % (int(n / 100000)) == 0 and i > 1000000:
15            estimates_n = np.append(estimates_n, i+1)
16            estimates_area = np.append(estimates_area, 4 * sum / (i+1))
17    if plot:
18        # Do the plotting here
19    return sum / n * 4
```

# Estimating PI: MC Integration Method - Convergence

1

```
mc_integrate(int(1e8), plot=True)
```

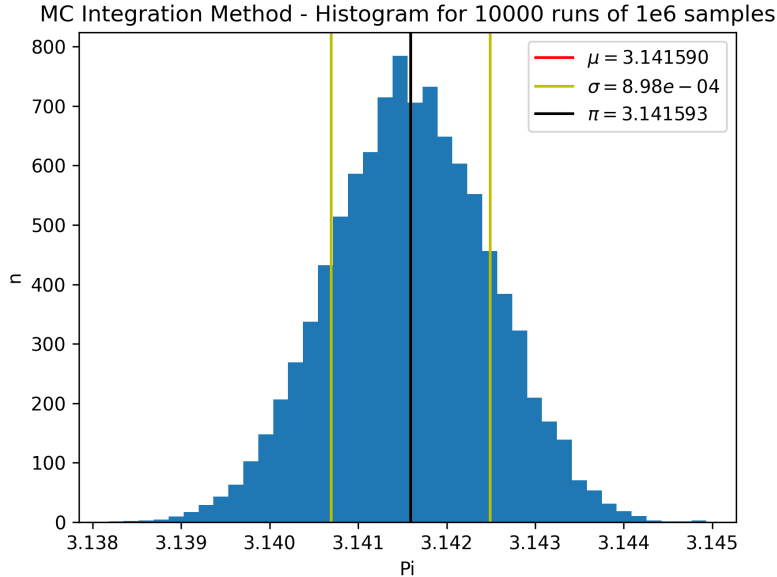


# Estimating PI: MC Integration Method - We can do better

```
1 def mc_integrate_fast(n):
2     x = np.random.random(n)
3     return np.sum(f(x)) / n * 4
```

```
1 pis = np.zeros(10000)
2 for i in range(10000):
3     pis[i] = mc_integrate_fast(int(1e6))
4 std = np.std(pis)
5 mean = np.mean(pis)
6 # Do the plotting here
7 # ...
8 print("MC Integration Method:")
9 print(f"{mean = }")
10 print(f"{std = }")
```

# Estimating PI: Area Method - Histogram



## Task 2: CDF Inversion

---

# CDF Inversion - Task

Let  $Q(\tau, \alpha) = \exp(-\alpha\tau)$ . and  $F(\tau) = \frac{1 - \exp(-\alpha\tau)}{\alpha}$  the CDF. Sample from  $Q$  by inverting:

$$r = \frac{\int_{\tau_{\min}}^{\tau} Q(\tau') d\tau'}{\int_{\tau_{\min}}^{\tau_{\max}} Q(\tau') d\tau'} = \frac{F(\tau) - F(\tau_{\min})}{F(\tau_{\max}) - F(\tau_{\min})}$$

with  $r \sim U(0, 1)$ . Setting  $\tau_{\min} = 0$  and  $\tau_{\max} = 5$  and solving:

$$\tau = -\frac{\log(1 - \alpha \cdot r \cdot F(\tau_{\max}))}{\alpha}$$

We can now

- Sample from  $Q$  and plot the histogram
- Calculate  $I_1 = \int_0^5 \tau Q(\tau, \alpha) d\tau = \langle \tau \rangle \cdot \text{Norm} = \langle \tau \rangle \cdot F(\tau_{\max})$
- Calculate  $I_2 = \int_0^5 \tau^2 Q(\tau, \alpha) d\tau = \langle \tau^2 \rangle \cdot F(\tau_{\max})$
- Calculate  $\sigma(I_1)$  and  $\sigma(I_2)$ :  $\sigma(\bar{\tau}) = \frac{\sigma_{\text{Samples}}}{\sqrt{n}}$  [GKW16, p.47]

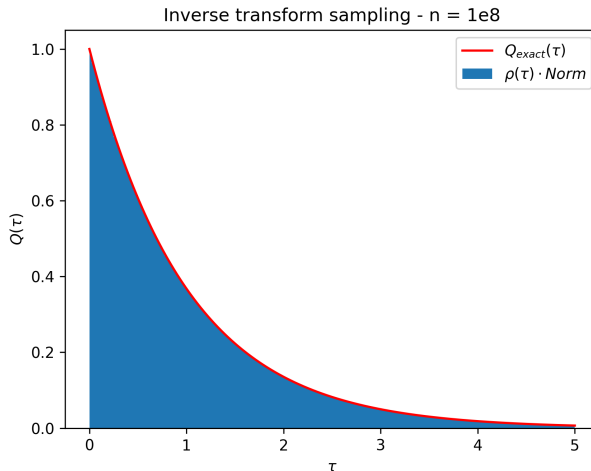
# CDF Inversion - Implementation

```
1 alpha = 1
2 x_max = 5
3 n = int(1e8)
4
5 def F(x):
6     return (1 - np.exp(-alpha*x))/alpha
7
8 F_max = F(x_max)
9
10 def F_inv(r):
11     return -np.log(1 - alpha*r*F_max)/alpha
12
13 #generate all samples at once using numpy
14 r = np.random.random(n)
15 x = F_inv(r)
16 # Do the plotting here
17 # Use np.histogram or plt.hist
18 print("I1 exact: 1-6*exp(-5) = 0.9595723")
19 print(f"I1 mean: { np.mean(x)*F_max:.6f}")
20 print(f"I1 std: {np.std(x)*F_max/np.sqrt(n):.3e}")
21 print("I2 exact: 2-37*exp(-5) = 1.750696")
22 print(f"I2 mean: {np.mean(x**2)*F_max:.6f}")
23 print(f"I2 std: {np.std(x**2)*F_max/np.sqrt(n):.3e}")
```

# CDF Inversion - Histogram

Console Output:

```
1 I1 exact: 0.9595723
2 I1 mean: 0.959501
3 I1 std: 9.045e-05
4 I2 exact: 1.750696
5 I2 mean: 1.750502
6 I2 std: 3.205e-04
```





## Task 3: Towards DMC - MCMC

---

# Markov Chain Monte Carlo - Task

Basic principle behind DMC is Markov Chain Monte Carlo (MCMC). We again sample from a distribution  $f$ , this time using the **Metropolis-Hastings algorithm**:

- Propose a random configuration  $x'$  from a proposal distribution  $p$
- Accept the configuration with probability  $A(x, x') = \min \left( 1, \frac{f(x')p(x)}{f(x)p(x')} \right)$
- $p$  can be optimized to increase acceptance rate (similarly to CDF inversion, see also [FSW08, Chapter 12.3.1.1])
- This **Markov Chain** converges to desired distribution  $f$ , guarantees ergodicity
- Samples now **correlated** → **Blocking Analysis** to estimate  $\sigma$

# Markov Chain Monte Carlo - Implementation

```
1  alpha = 1
2  n_equ = int(1e4)
3  n_sam = int(1e6)
4  n_acc = 0
5  n_rej = 0
6  rng = np.random.default_rng()
7
8  #function to sample from
9  def f(x):
10     return np.exp(-alpha*x)
11
12  #uniform distribution
13  uni = rng.uniform
14
15  #acceptance ratio for Metropolis-Hastings
16  def acc_ratio(x, x_new):
17     return f(x_new)/f(x)
18
19  #initial sample
20  curr = uni(0,5)
21
22  #array to store samples
23  samples = np.zeros(n_sam)
```

# Markov Chain Monte Carlo - Implementation

```
1  #equilibration
2  for i in range(n_equ):
3      prop = uni(0,5)
4      if acc_ratio(curr, prop) > np.random.random():
5          curr = prop
6          n_acc += 1
7      else:
8          n_rej += 1
9
10 #sampling
11 for i in range(n_sam):
12     prop = uni(0,5)
13     if acc_ratio(curr, prop) > np.random.random():
14         curr = prop
15         n_acc += 1
16     else:
17         n_rej += 1
18     samples[i] = curr
19
20 # Plotting, printing, etc.
21 # ...
```

# Markov Chain Monte Carlo - Blocking Analysis

```
1 def blocking(self, samples, min_blocks: int = 32) -> tuple[float, float]:
2     means = np.copy(samples) # Copy the samples to avoid changing the original array
3     mean = np.mean(means).astype(float) # Calculate the mean of the samples
4     n = np.log2(len(means) // min_blocks).astype(int) # Calculate the iteration steps
5     block_sizes = np.logspace(0, n, n + 1, base=2) # Minimum block size is 1, max is 2**n
6     vars = np.zeros(n + 1, dtype=float) # Initialize the array for the variances
7     vars[0] = 1 / (len(means) - 1) * (np.mean(means**2) - mean**2) # Naive variance
8     Rx = np.zeros(n + 1) # Initialize Rx array
9     Rx[0] = 1
10    for i in range(1, n + 1): # Perform blocking
11        if means.size % 2 != 0: # Make sure the number of blocks is even in order to divide by 2
12            means = means[:-1]
13        means = 0.5 * (means[::2] + means[1::2]) # Double the block size
14        n_blocks = means.size
15        varXBlock = n_blocks / (n_blocks - 1) * (np.mean(means**2) - mean**2)
16        vars[i] = varXBlock / n_blocks # Calculate the variance
17        Rx[i] = block_sizes[i] * varXBlock / vars[0] / samples.size
18    plateau_index = np.argmax(np.abs(Rx[1:] - Rx[:-1])[3:]) + 4 # Find plateau in Rx
19    # Plotting...
20    return mean, vars[plateau_index]
```

For the maths behind the blocking analysis, see [GKW16, Chapter 3.4].

# Markov Chain Monte Carlo - Results

The results are very similar to the CDF inversion method, as we sample from the same distribution in a different way.

We will take a closer look at the blocking analysis after the next task.

## Task 4&5: Diagrammatic Monte Carlo

---

# Diagrammatic Monte Carlo - Task

We are interested in the following function:

$$Q(\tau, \alpha, V) = \exp(-\alpha\tau) + \sum_{\beta} \int_0^{\tau} d\tau_2 \int_0^{\tau_2} d\tau_1 e^{-\alpha\tau_1} V e^{-\beta(\tau_2 - \tau_1)} V e^{-\alpha(\tau - \tau_2)}$$

which can be represented as zero and second order **Feynman Diagrams** of the following form:

$$\overline{0 \quad \alpha \quad \tau_j} = D_0^{\xi_0}(\tau_j, \alpha)$$

$$\overline{0 \quad \alpha \quad \tau_1 \quad \beta \quad \tau_2 \quad \alpha \quad \tau} = D_2^{\xi_2}(\tau, \alpha, V; \tau_1, \tau_2, \beta) = e^{-\alpha\tau_1} V e^{-\beta(\tau_2 - \tau_1)} V e^{-\alpha(\tau - \tau_2)}$$

How can we sample from this function?



# Diagrammatic Monte Carlo - Task

We use the **Metropolis-Hastings algorithm** again, using the diagrams as weights. For ergodicity, we need to implement the following updates:

- Change of  $\tau$
- Increasing the order of the diagram
- Decreasing the order of the diagram
- (Change of  $\alpha$ )

Mind the context factors of the updates when changing the order of the diagram<sup>[FSW08, ch. 12.3.1.2]</sup>.

$$A_{0 \rightarrow 2} = \min \left\{ 1, \frac{p_{\text{rem}}}{p_{\text{add}}} \frac{D_2^{\xi_2}(\tau, \alpha, V; \tau_1, \tau_2, \beta) d\tau_1 d\tau_2}{D_0^{\xi_0}(\tau_0, \alpha_0) p_1(\tau_1) p_2(\tau_2) p_3(\beta) d\tau_1 d\tau_2} \right\}$$

We implement a Python class to sample, analyse and plot the results of the DMC algorithm. We will first take a look at some of the methods of this class, then have a look at the whole class **DMC.py** and finally look at the results.

# Diagrammatic Monte Carlo Implementation - Initialization

```
1 def __init__(...) -> None:
2     # check parameters
3     if tau_max <= tau_min:
4         raise ValueError("tau_max must be greater than tau_min")
5     # And so on...
6     self.rng = np.random.default_rng() # set up random number generator
7     self.uni = lambda a, b: self.rng.random() * (b - a) + a
8     self.n_equ, n_sam, tau_min; tau_max, V, use_change_alpha = ... # Set up config parameters
9     self.tau, alpha, beta, tau1, tau2 = ... # Set up state variables
10    self.n_tau_acc, n_tau_rej, n_alpha_acc, n_alpha_rej... = 0 # Counters
11    self.samples = np.zeros((n_sam, 2) if use_change_alpha else n_sam) # Set up sample array
12    # Set up updater methods
13    self.updaters = [self.change_tau]
14    if use_change_alpha:
15        self.updaters.append(self.change_alpha)
16    # And so on...
17
```

# Diagrammatic Monte Carlo Implementation - Weight Calculation

```
1 def weight(self, tau=None, alpha=None, beta=None, tau1=None, tau2=None) -> float:
2     if tau is None:
3         tau = self.tau
4     # Same for alpha, beta, tau1 and tau2
5     # ...
6     # If the diagram is of zero order, the weight is simple
7     if beta == 0:
8         return np.exp(-alpha * tau)
9     # If the diagram is of second order, we need to include
10    # two interaction vertices and the beta propagator
11    return (
12        np.exp(-alpha * tau1)
13        * self.V
14        * np.exp(-beta * (tau2 - tau1))
15        * self.V
16        * np.exp(-alpha * (tau - tau2))
17    )
```

As we will see, this method is used in every update to calculate the acceptance ratio.

# Diagrammatic Monte Carlo Implementation - Change- $\tau$ & $\alpha$

```
1 def change_tau(self) -> bool:
2     prop_tau = self.uni(self.tau2, self.tau_max)
3     M = self.weight(tau=prop_tau) / self.weight()
4     if M > self.rng.random():
5         self.tau = prop_tau
6         self.n_tau_acc += 1
7         return True
8     else:
9         self.n_tau_rej += 1
10        return False
11
12 def change_alpha(self) -> bool:
13     prop_alpha = [0.5, 1][int(self.rng.random() * 2)]
14     M = self.weight(alpha=prop_alpha) / self.weight()
15     if M > self.rng.random():
16         self.alpha = prop_alpha
17         self.n_alpha_acc += 1
18         return True
19     else:
20         self.n_alpha_rej += 1
21        return False
```

# Diagrammatic Monte Carlo Implementation - Change-Order

```
1 def add_beta(self) -> bool:
2     if self.beta != 0:
3         self.n_beta_rej += 1
4         return False
5     prop_tau2 = self.uni(self.tau_min, self.tau)
6     prop_tau1 = self.uni(self.tau_min, prop_tau2)
7     prop_beta = [0.25, 0.75][int(self.rng.random() * 2)]
8     M = self.weight(beta=prop_beta, tau1=prop_tau1, tau2=prop_tau2) / (
9         self.weight() / (self.tau - self.tau_min) / (prop_tau2 - self.tau_min) * 0.5
10    )
11    if M > self.rng.random():
12        self.beta = prop_beta
13        self.tau1 = prop_tau1
14        self.tau2 = prop_tau2
15        self.n_beta_acc += 1
16        return True
17    else:
18        self.n_beta_rej += 1
19        return False
```

The remove- $\beta$  update works the same way. The Metropolis ratio is the inverse of the add- $\beta$  ratio.

# Diagrammatic Monte Carlo Implementation - Updating, Sampling

```
1 def update(self) -> bool:
2     updater = self.updaters[int(self.rng.random() * len(self.updaters))]
3     return updater()
4
5 def sample(self) -> None:
6     for i in range(self.n_sam):
7         self.update()
8         if self.use_change_alpha:
9             self.samples[i, 0] = self.tau
10            self.samples[i, 1] = self.alpha
11            if self.beta == 0:
12                self.n_zero_order[self.alpha] = self.n_zero_order.get(self.alpha, 0) + 1
13        else:
14            self.samples[i] = self.tau
15            if self.beta == 0:
16                self.n_zero_order += 1
17
18 # Logging ...
```

Equilibration works the same way, but without sampling.

# Diagrammatic Monte Carlo Implementation - Normalization

The normalization constant could still be calculated analytically, but normally this is not possible.

**Solution:** Use normalization constant  $C_0$  of zero order diagram and number of zero order diagrams  $N_0$  to estimate  $C$  [RFT20, ch. 3.3]:

$$C \approx C_0 \frac{N}{N_0}$$

This is implemented as an alternative to the analytical normalization. Furthermore, helper methods for plotting and analysis are implemented.

Let's take a look at the whole class **DMC.py**, before we come back to the results.

# Switching to VS Code



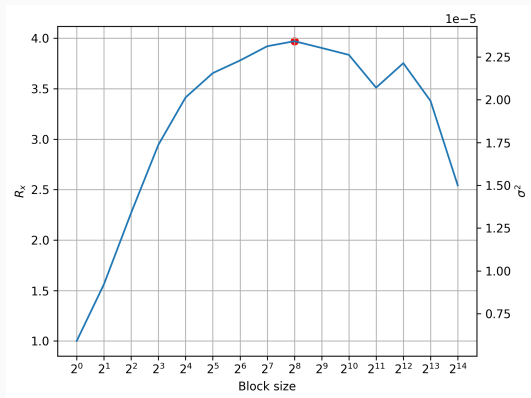
# Diagrammatic Monte Carlo - Results

```
1 from DMC import DMC
2 dmc = DMC(
3     n_equ=int(1e4),
4     n_sam=int(5e6),
5     tau_min=0,
6     tau_max=5,
7     alpha_0=1,
8     tau_0=1,
9     use_change_alpha=True,
10    use_change_beta=False,
11    use_analytical=True,
12 )
13 dmc.equilibrate()
14 dmc.sample()
15 dmc.print_I1(0.5)
16 dmc.print_I1(1)
17 dmc.print_I2(0.5)
18 dmc.print_I2(1)
19 dmc.plot_hist(1)
20 dmc.plot_hist(0.5)
```

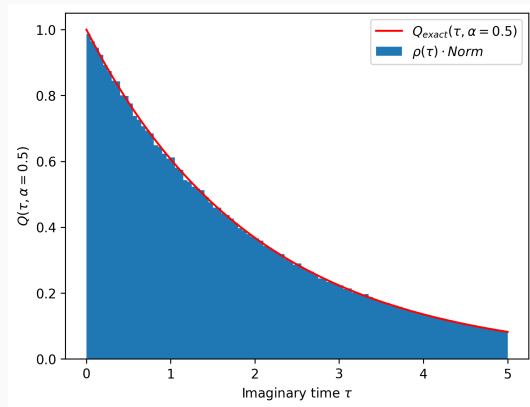
```
1 Warming up: 10000 steps
2 Warmup complete:  $\tau$  acc. ratio of 0.54,  $\alpha$  acc. ratio of 0.86
3 Sampling: 5000000 steps
4 100%|██████████| 5000000/5000000 [00:25<00:00, 194393.91it/s]
5 Sampling complete:  $\tau$  acc. ratio of 0.54,  $\alpha$  acc. ratio of 0.85
6 I1( $\alpha=0.5$ ) mean: 2.85430, var: 9.53e-06
7 I1( $\alpha=1$ ) mean: 0.96035, var: 2.40e-06
8 I2( $\alpha=0.5$ ) mean: 7.31758, var: 1.63e-04
9 I2( $\alpha=1$ ) mean: 1.75439, var: 2.34e-05
```

Plots are shown on the next slide.

# Diagrammatic Monte Carlo - Results



One of the blocking analysis plots

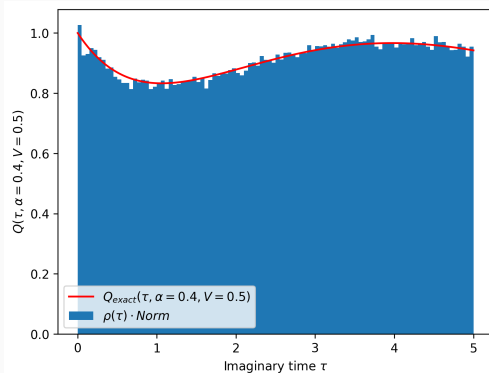


Histogram for  $\alpha = 0.5$

# Diagrammatic Monte Carlo - Results

```
1 from DMC import DMC
2
3 dmc = DMC(
4     n_equ=int(1e4),
5     n_sam=int(3e6),
6     tau_min=0,
7     tau_max=5,
8     alpha_0=0.4,
9     tau_0=1,
10    V=0.5,
11    use_change_alpha=False,
12    use_change_beta=True,
13    use_analytical=True,
14 )
15 dmc.equilibrate()
16 dmc.sample()
17 dmc.plot_hist()
```

```
1 Warming up: 10000 steps
2 Warmup complete:  $\tau$  acc. ratio of 0.76,  $\beta$  acc. ratio of 0.22
3 Sampling: 3000000 steps
4 100%|██████████| 3000000/3000000 [00:14<00:00, 205720.34it/s]
5 Sampling complete:  $\tau$  acc. ratio of 0.76,  $\beta$  acc. ratio of 0.22
```



## Task 6: DMC - Green Function Estimator

---

# Diagrammatic Monte Carlo - Greens Function Estimator

An exact MC estimator for the Greens function can be derived [MPSS00, ch. III.C]:

$$\langle q_{\tau_0} \rangle_{MC} = C * Q(\tau_0, \alpha, V)$$
$$q_{\tau_0}(\nu, \tau) = \begin{cases} q(\nu) \mathcal{D}_\nu(\tau_0) / \mathcal{D}_\nu(\tau) = |\Gamma_0|^{-1} e^{-\alpha(\tau_0 - \tau)}, & \text{if } \tau \in \Gamma_0 \text{ and } \mathcal{D}_\nu(\tau) \neq 0 \\ 0, & \text{otherwise.} \end{cases}$$

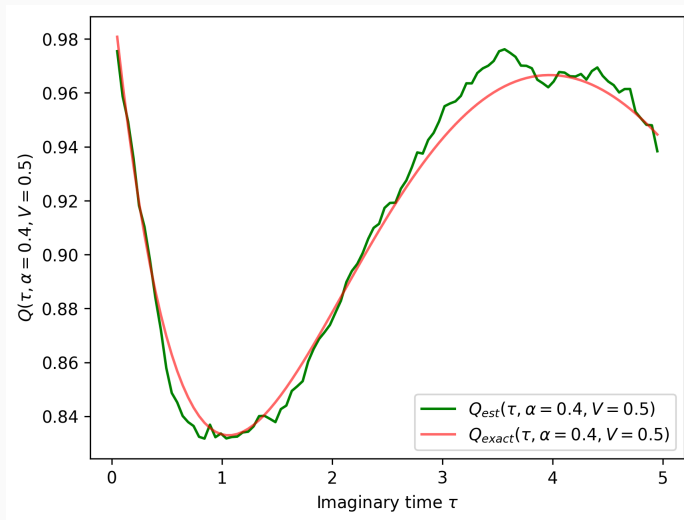
Where i choose  $\Gamma_0 = [\tau_0 - a, \tau_0 + a]$  with  $a = \min(0.2, \tau_0, \tau_{\max} - \tau_0)$

This eliminates the discretization error of the histogram method.

```
1 def green_estimator(self, tau0: float, alpha: float | None = None) -> float:
2     if alpha is None:
3         alpha = self.alpha
4     tau = self.get_samples(alpha)
5     diff = np.subtract.outer(tau0, tau)
6     tau0_stacked = np.repeat(tau0, tau.size).reshape(diff.shape)
7     mins = np.minimum(np.minimum(tau0_stacked, self.tau_max - tau0_stacked), 0.2)
8     res = np.exp(-alpha * (diff)) / (2 * mins)
9     res[np.abs(diff) > mins] = 0
10    meaned = np.mean(res, axis=1) * self.norm(alpha, analytical=self.use_analytical)
11    return meaned
```

# Diagrammatic Monte Carlo - Greens Function Estimator - Results

Plot of the Greens function estimator for  $n = 3 \cdot 10^6$  samples:



- [Ces22] Cesare Franchini.  
**Basic diagrammatic monte carlo - exercise sheet.**  
[https://virtuale.unibo.it/pluginfile.php/1542120/mod\\_resource/content/1/project\\_basic\\_dmc.pdf](https://virtuale.unibo.it/pluginfile.php/1542120/mod_resource/content/1/project_basic_dmc.pdf), 2022.  
[Online, private; accessed June 24, 2023].
- [FSW08] H. Fehske, R. Schneider, and A. Weiße, editors.  
***Computational Many-Particle Physics.***  
Springer Berlin Heidelberg, 2008.
- [GKW16] James Gubernatis, Naoki Kawashima, and Philipp Werner.  
***Quantum Monte Carlo Methods: Algorithms for Lattice Models.***  
Cambridge University Press, 2016.
- [MPSS00] A. Mishchenko, N. Prokof'ev, A. Sakamoto, and B. Svistunov.  
**Diagrammatic quantum monte carlo study of the fröhlich polaron.**  
*Physical Review B*, 62(10):6317–6336, September 2000.

- [RFT20] S. Ragni, C. Franchini, and Hahn T.  
**Diagrammatic monte carlo study of the holstein polaron.**  
Master's thesis, Alma Mater Studiorum - University of Bologna, 2020.  
[Online; accessed June 24, 2023].