DPPy documentation

Guillaume Gautier et al.

Nov 25, 2020

CONTENTS

Determinantal point processes (DPPs) are specific probability distributions over clouds of points, which have been popular as models or computational tools across physics, probability, statistics, random matrices, and more recently machine learning. DPPs are often used to induce diversity or repulsiveness among the points of a sample.

Sampling from DPPs is more tractable than sampling generic point processes with interaction, but it remains a nontrivial matter and a research area of its own.

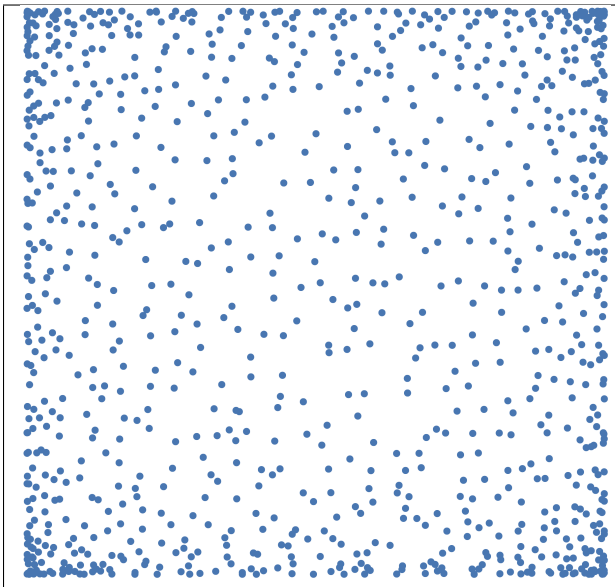
As a contraction of **DPPs and Python**, DPPy is an effort to gather:

- *exact* **and** *approximate* **samplers** for *finite DPPs*
- **random matrix models** (*full* and *banded*) for β-*Ensembles*
- *Multivariate Jacobi ensemble* used for **Monte Carlo integration**
- **exact samplers** for more *Exotic DPPs*

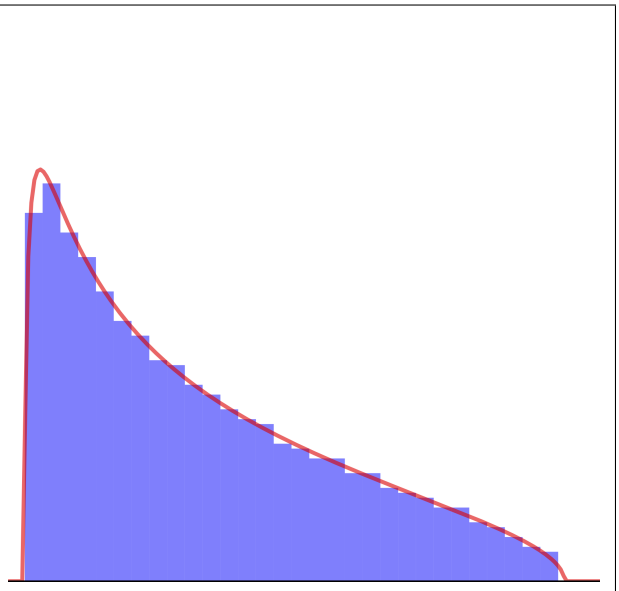
  - *uniform spanning trees*
  - *descent processes*
  - the *Poissonized Plancherel*

The purpose of this **documentation** is to both provide a **quick survey of DPPs and relate each mathematical property with its implementation in DPPy**. The documentation can thus be read in different ways:

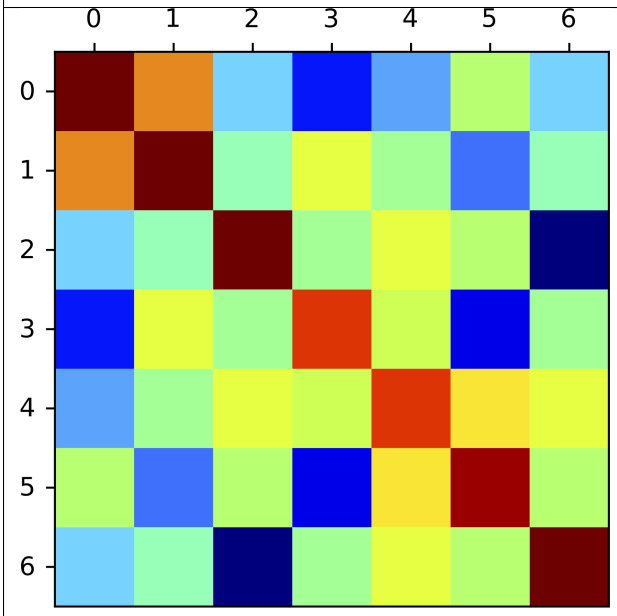
- if you read the sections in the order they appear, they will first take you through mathematical definitions and quick illustrations of how these definitions are encoded in DPPy.
- for more a traditional library documentation please refer to the corresponding API sections documenting the methods of each object, along with pointers to the mathematical definitions if needed.
- you can also directly jump to the Jupyter notebooks, which showcase the use of some DPPy objects in more detail.



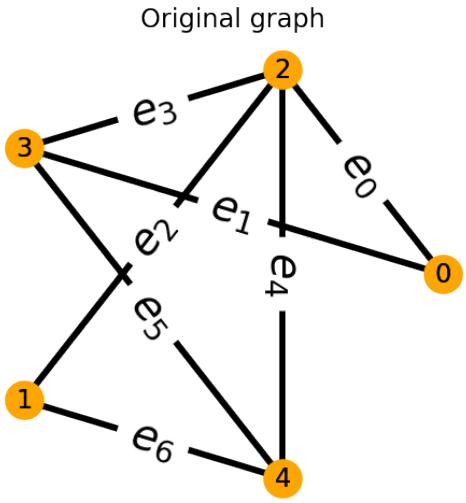
2D *Multivariate Jacobi ensemble*



$\beta = 2$-*Laguerre ensemble*



K kernel of *Uniform Spanning Tree*



Graph associated to **K**

CHAPTER

ONE

INSTALLATION INSTRUCTIONS

See the installation instructions on GitHub.

CHAPTER

TWO

HOW TO CITE THIS WORK?

We wrote a companion paper to DPPy which got accepted for publication in the MLOSS track of JMLR, see [GPBV19].

If you use this package, please consider citing it with this piece of BibTeX:

```
@article{GPBV19,
  author = {Gautier, Guillaume and Polito, Guillermo and Bardenet, R{\'{e}}mi and
↪Valko, Michal},
  journal = {Journal of Machine Learning Research - Machine Learning Open Source
↪Software (JMLR-MLOSS)},
  title = {{DPPy: DPP Sampling with Python}},
  keywords = {Computer Science - Machine Learning, Computer Science - Mathematical
↪Software, Statistics - Machine Learning},
  url = {http://jmlr.org/papers/v20/19-179.html},
  year = {2019},
  archivePrefix = {arXiv},
  arxivId = {1809.07258},
  note = {Code at http://github.com/guilgautier/DPPy/ Documentation at http://dppy.
↪readthedocs.io/}
}
```

CHAPTER

THREE

DOCUMENTATION CONTENTS

3.1 Finite DPPs

3.1.1 Definition

A finite point process $\mathcal{X}$ on $[N] \triangleq \{1, \dots, N\}$ can be understood as a random subset. It is defined either via its:

- inclusion probabilities (also called correlation functions)

$$\mathbb{P}[S \subset \mathcal{X}], \text{ for } S \subset [N],$$

- likelihood

$$\mathbb{P}[\mathcal{X} = S], \text{ for } S \subset [N].$$

Hint: The *determinantal* feature of DPPs stems from the fact that such inclusion, resp. marginal probabilities are given by the principal minors of the corresponding correlation kernel $\mathbf{K}$ (resp. likelihood kernel $\mathbf{L}$).

Inclusion probabilities

We say that $\mathcal{X} \sim \mathrm{DPP}(\mathbf{K})$ with correlation kernel a complex matrix $\mathbf{K}$ if

$$\mathbb{P}[S \subset \mathcal{X}] = \det \mathbf{K}_S, \quad \forall S \subset [N], \tag{3.1}$$

where $\mathbf{K}_S = [\mathbf{K}_{ij}]_{i,j \in S}$ i.e. the square submatrix of $\mathbf{K}$ obtained by keeping only rows and columns indexed by S.

Likelihood

We say that $\mathcal{X} \sim \mathrm{DPP}(\mathbf{L})$ with likelihood kernel a complex matrix $\mathbf{L}$ if

$$\mathbb{P}[\mathcal{X} = S] = \frac{\det \mathbf{L}_S}{\det[I + \mathbf{L}]}, \quad \forall S \subset [N]. \tag{3.2}$$

Existence

Some common sufficient conditions to guarantee existence are:

$$\mathbf{K} = \mathbf{K}^\dagger \quad \text{and} \quad 0_N \preceq \mathbf{K} \preceq I_N, \tag{3.3}$$

$$\mathbf{L} = \mathbf{L}^\dagger \quad \text{and} \quad \mathbf{L} \succeq 0_N, \tag{3.4}$$

where the dagger $\dagger$ symbol means *conjugate transpose*.

Note: In the following, unless otherwise specified:

- we work under the sufficient conditions (3.3) and (3.3),
- $(\lambda_1, \ldots, \lambda_N)$ denote the eigenvalues of $\mathbf{K}$,
- $(\gamma_1, \ldots, \gamma_N)$ denote the eigenvalues of $\mathbf{L}$.

```
# from numpy import sqrt
from numpy.random import rand, randn
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

r, N = 4, 10
e_vecs, _ = qr(randn(N, r), mode='economic')

# Inclusion K
e_vals_K = rand(r)  # in [0, 1]
dpp_K = FiniteDPP('correlation', **{'K_eig_dec': (e_vals_K, e_vecs)})
# or
# K = (e_vecs * e_vals_K).dot(e_vecs.T)
# dpp_K = FiniteDPP('correlation', **{'K': K})
dpp_K.plot_kernel()

# Marginal L
e_vals_L = e_vals_K / (1.0 - e_vals_K)
dpp_L = FiniteDPP('likelihood', **{'L_eig_dec': (e_vals_L, e_vecs)})
# or
# L = (e_vecs * e_vals_L).dot(e_vecs.T)
# dpp_L = FiniteDPP('likelihood', **{'L': K})
# Phi = (e_vecs * sqrt(e_vals_L)).T
# dpp_L = FiniteDPP('likelihood', **{'L_gram_factor': Phi})  # L = Phi.T Phi
dpp_L.plot_kernel()
```

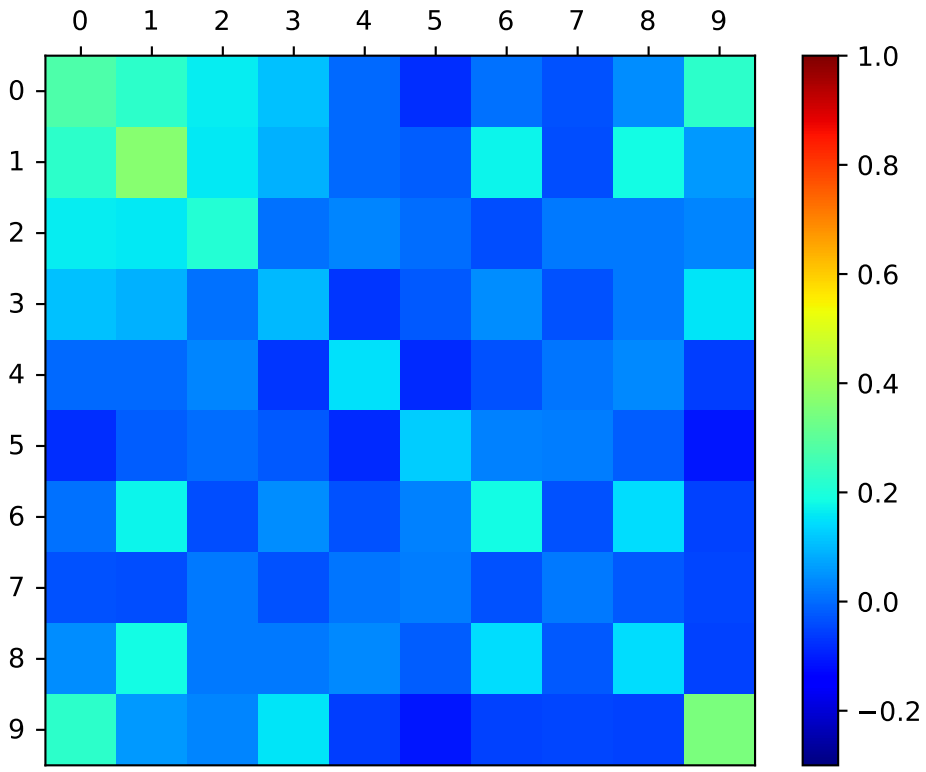


Fig. 3.1: Correlation $\mathbf{K}$ kernel

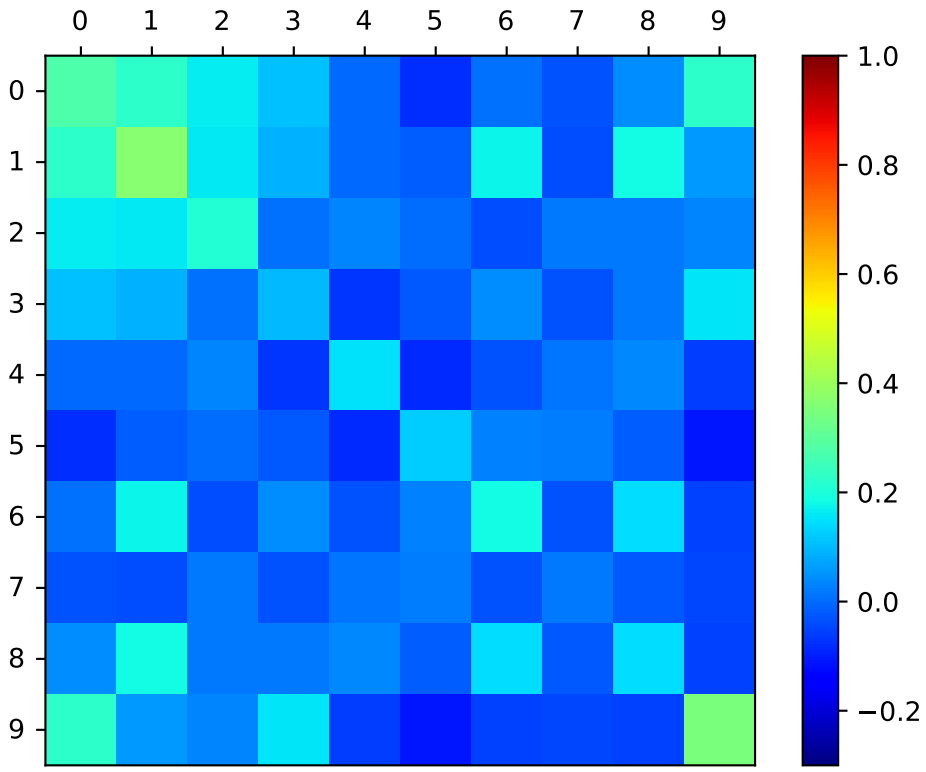


Fig. 3.2: Correlation $\mathbf{K}$ kernel

Projection DPPs

Important: $\mathrm{DPP}(\mathbf{K})$ defined by an *orthogonal projection* correlation kernel $\mathbf{K}$ are called *projection* DPPs.

Recall that orthogonal projection matrices are notably characterized by

a. $\mathbf{K}^2 = \mathbf{K}$ and $\mathbf{K}^\dagger = \mathbf{K}$,

b. or equivalently by $\mathbf{K} = UU^\dagger$ with $U^\dagger U = I_r$ where $r = \mathrm{rank}(\mathbf{K})$.

They are indeed valid kernels since they meet the above sufficient conditions: they are Hermitian with eigenvalues 0 or 1.

```
from numpy import ones
from numpy.random import randn
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

r, N = 4, 10

eig_vals = ones(r)
A = randn(r, N)
eig_vecs, _ = qr(A.T, mode='economic')

proj_DPP = FiniteDPP('correlation', projection=True,
                     **{'K_eig_dec': (eig_vals, eig_vecs)})
# or
# proj_DPP = FiniteDPP('correlation', projection=True, **{'A_zono': A})
# K = eig_vecs.dot(eig_vecs.T)
# proj_DPP = FiniteDPP('correlation', projection=True, **{'K': K})
```

k-DPPs

A k-DPP can be defined as $\mathrm{DPP}(\mathbf{L})$ (3.2) conditioned to a fixed sample size $|\mathcal{X}| = k$, we denote it k-$\mathrm{DPP}(\mathbf{L})$.

It is naturally defined through its joint probabilities

$$\mathbb{P}_{k\text{-DPP}}[\mathcal{X} = S] = \frac{1}{e_k(L)} \det \mathbf{L}_S 1_{|S|=k}, \tag{3.5}$$

where the normalizing constant $e_k(L)$ corresponds to the elementary symmetric polynomial of order k evaluated in the eigenvalues of $\mathbf{L}$,

$$e_k(\mathbf{L}) \triangleq e_k(\gamma_1, \ldots, \gamma_N) = \sum_{\substack{S \subset [N] \\ |S|=k}} \prod_{s \in S} \gamma_s = \sum_{\substack{S \subset [N] \\ |S|=k}} \det L_S.$$

Note: Obviously, one must take $k \leq \mathrm{rank}(L)$ otherwise $\det \mathbf{L}_S = 0$ for $|S| = k > \mathrm{rank}(L)$.

Warning: k-DPPs are not DPPs in general. Viewed as a DPP conditioned to a fixed sample size $|\mathcal{X}| = k$, the only case where they coincide is when the original DPP is a *projection* $\mathrm{DPP}(\mathbf{K})$, and $k = \mathrm{rank}(\mathbf{K})$, see (3.13).

See also:

- *Exact sampling of k-DPPs*
- `FiniteDPP`
- [KT12] Section 2 for DPPs
- [KT12] Section 5 for k-DPPs

3.1.2 Properties

Throughout this section, we assume $\mathbf{K}$ and $\mathbf{L}$ satisfy the sufficient conditions (3.3) and (3.4) respectively.

Relation between correlation and likelihood kernels

1. Considering the DPP defined by $\mathbf{L} \succeq 0_N$, the associated correlation kernel $\mathbf{K}$ (3.1) can be derived as

$$\mathbf{K} = \mathbf{L}(I + \mathbf{L})^{-1} = I - (I + \mathbf{L})^{-1}. \tag{3.6}$$

   See also:

   Theorem 2.2 [KT12].

2. Considering the DPP defined by $0_N \preceq \mathbf{K} \prec I_N$, the associated likelihood kernel $\mathbf{L}$ (3.2) can be derived as

$$\mathbf{L} = \mathbf{K}(I - \mathbf{K})^{-1} = -I + (I - \mathbf{K})^{-1}. \tag{3.7}$$

   See also:

   Equation 25 [KT12].

Important: Thus, except for correlation kernels $\mathbf{K}$ with some eigenvalues equal to 1, both $\mathbf{K}$ and $\mathbf{L}$ are diagonalizable in the same basis

$$\mathbf{K} = U\Lambda U^\dagger, \quad \mathbf{L} = U\Gamma U^\dagger \qquad \text{with} \qquad \lambda_n = \frac{\gamma_n}{1+\gamma_n}. \tag{3.8}$$

Note: For DPPs with *projection* correlation kernel $\mathbf{K}$, the likelihood kernel $\mathbf{L}$ cannot be computed via (3.7), since $\mathbf{K}$ has at least one eigenvalue equal to 1 ($\mathbf{K}^2 = \mathbf{K}$).

Nevertheless, if you recall that the *number of points of a projection DPP*, then its likelihood reads

$$\mathbb{P}[\mathcal{X} = S] = \det \mathbf{K}_S 1_{|S|=\mathrm{rank}(\mathbf{K})} \quad \forall S \subset [N].$$

```
from numpy.random import randn, rand
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

r, N = 4, 10
```

(continues on next page)

(continued from previous page)

```
eig_vals = rand(r)  # 0< <1
eig_vecs, _ = qr(randn(N, r), mode='economic')

DPP = FiniteDPP('correlation', **{'K_eig_dec': (eig_vals, eig_vecs)})
DPP.compute_L()

# - L (likelihood) kernel computed via:
# - eig_L = eig_K/(1-eig_K)
# - U diag(eig_L) U.T
```

See also:

- *compute_K()*
- *compute_L()*

Generic DPPs as mixtures of projection DPPs

Projection DPPs are the building blocks of the model in the sense that generic DPPs are mixtures of *projection* DPPs.

Important: Consider $\mathcal{X} \sim \mathrm{DPP}(\mathbf{K})$ and write the spectral decomposition of the corresponding kernel as

$$\mathbf{K} = \sum_{n=1}^{N} \lambda_n u_n u_n^\dagger.$$

Then, denote $\mathcal{X}^B \sim \mathrm{DPP}(\mathbf{K}^B)$ with

$$\mathbf{K}^B = \sum_{n=1}^{N} B_n u_n u_n^\dagger, \quad \text{where} \quad B_n \overset{\text{i.i.d.}}{\sim} \mathcal{B}er(\lambda_n),$$

where $\mathcal{X}^B$ is obtained by first choosing $B_1, \ldots, B_N$ independently and then sampling from $\mathrm{DPP}(\mathbf{K}^B)$ the DPP with orthogonal projection kernel $\mathbf{K}^B$.

Finally, we have $\mathcal{X} \overset{d}{=} \mathcal{X}^B$.

See also:

- Theorem 7 in [HKPVirag06]
- *Exact sampling*
- Continuous case of *Generic DPPs as mixtures of projection DPPs*

Number of points

For projection DPPs, i.e., when $\mathbf{K}$ is an orthogonal projection matrix, one can show that $|\mathcal{X}| = \mathrm{rank}(\mathbf{K}) = \mathrm{Trace}(\mathbf{K})$ almost surely (see, e.g., Lemma 17 of [HKPVirag06] or Lemma 2.7 of [KT12]).

In the general case, based on the fact that *generic DPPs are mixtures of projection DPPs*, we have

$$|\mathcal{X}| = \sum_{n=1}^{N} \mathcal{B}\mathrm{er}\left(\lambda_n\right) = \sum_{n=1}^{N} \mathcal{B}\mathrm{er}\left(\frac{\gamma_n}{1+\gamma_n}\right). \tag{3.9}$$

Note: From (3.9) it is clear that $|\mathcal{X}| \leq \mathrm{rank}(\mathbf{K}) = \mathrm{rank}(\mathbf{L})$.

Expectation

$$\mathbb{E}[|\mathcal{X}|] = \operatorname{trace}\mathbf{K} = \sum_{n=1}^{N} \lambda_n = \sum_{n=1}^{N} \frac{\gamma_n}{1+\gamma_n}. \tag{3.10}$$

The expected size of a DPP with likelihood matrix $\mathbf{L}$ is also related to the effective dimension $d_{\text{eff}}(\mathbf{L}) = \operatorname{trace}(\mathbf{L}(\mathbf{L}+\mathbf{I})^{-1}) = \operatorname{trace}\mathbf{K} = \mathbb{E}[|\mathcal{X}|]$ of $\mathbf{L}$, a quantity with many applications in randomized numerical linear algebra and statistical learning theory (see e.g., [DerezinskiCV19]).

Variance

$$\mathbb{V}\mathrm{ar}[|\mathcal{X}|] = \operatorname{trace}\mathbf{K} - \operatorname{trace}\mathbf{K}^2 = \sum_{n=1}^{N} \lambda_n(1-\lambda_n) = \sum_{n=1}^{N} \frac{\gamma_n}{(1+\gamma_n)^2}. \tag{3.11}$$

See also:

Expectation and variance of *Linear statistics*.

```
import numpy as np
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

rng = np.random.RandomState(1)

r, N = 5, 10
eig_vals = rng.rand(r) # 0< <1
eig_vecs, _ = qr(rng.randn(N, r), mode='economic')

dpp_K = FiniteDPP('correlation', projection=False,
                  **{'K_eig_dec': (eig_vals, eig_vecs)})

nb_samples = 2000
for _ in range(nb_samples):
    dpp_K.sample_exact(random_state=rng)

sizes = list(map(len, dpp_K.list_of_samples))
print('E[|X|]:\n emp={:.3f}, theo={:.3f}'
      .format(np.mean(sizes), np.sum(eig_vals)))
print('Var[|X|]:\n emp={:.3f}, theo={:.3f}'
      .format(np.var(sizes), np.sum(eig_vals*(1-eig_vals))))
```

```
E[|X|]:
 emp=1.581, theo=1.587
Var[|X|]:
 emp=0.795, theo=0.781
```

Special cases

1. When the correlation kernel $\mathbf{K}$ (3.1) of $\mathrm{DPP}(\mathbf{K})$ is an orthogonal projection kernel, i.e., $\mathrm{DPP}(\mathbf{K})$ is a *projection DPP*, we have

$$|\mathcal{X}| = \mathrm{rank}(\mathbf{K}) = \mathrm{trace}(\mathbf{K}), \quad \text{almost surely.} \tag{3.12}$$

```
import numpy as np
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

r, N = 4, 10
eig_vals = np.ones(r)
eig_vecs, _ = qr(rng.randn(N, r), mode='economic')

DPP = FiniteDPP('correlation', projection=True,
                **{'K_eig_dec': (eig_vals, eig_vecs)})

for _ in range(1000):
    DPP.sample_exact()

sizes = list(map(len, DPP.list_of_samples))
# np.array(DPP.list_of_samples).shape = (1000, 4)

assert([np.mean(sizes), np.var(sizes)] == [r, 0])
```

Important: Since $|\mathcal{X}| = \mathrm{rank}(\mathbf{K})$ points, almost surely, the likelihood of the projection $\mathrm{DPP}(\mathbf{K})$ reads

$$\mathbb{P}[\mathcal{X} = S] = \det \mathbf{K}_S 1_{|S|=\mathrm{rank}\,\mathbf{K}}. \tag{3.13}$$

In other words, the projection DPP having for **correlation** kernel the orthogonal projection matrix $\mathbf{K}$ coincides with the *k-DPP* having **likelihood** kernel $\mathbf{K}$ when $k = \mathrm{rank}(\mathbf{K})$.

---

2. When the likelihood kernel $\mathbf{L}$ of $\mathrm{DPP}(\mathbf{L})$ (3.2) is an orthogonal projection kernel we have

$$|\mathcal{X}| \sim \mathrm{Binomial}(\mathrm{rank}(\mathbf{L}), 1/2). \tag{3.14}$$

```
import numpy as np
from scipy.stats import binom, chisquare
from scipy.linalg import qr
import matplotlib.pyplot as plt
from dppy.finite_dpps import FiniteDPP


r, N = 5, 10
e_vals = np.ones(r)
e_vecs, _ = qr(np.random.randn(N, r), mode='economic')
```

(continues on next page)

(continued from previous page)

```
dpp_L = FiniteDPP('likelihood',
                  projection=True,
                  **{'L_eig_dec': (e_vals, e_vecs)})

nb_samples = 1000
dpp_L.flush_samples
for _ in range(nb_samples):
    dpp_L.sample_exact()

sizes = list(map(len, dpp_L.list_of_samples))

p = 0.5 # binomial parameter
rv = binom(r, p)

fig, ax = plt.subplots(1, 1)

x = np.arange(0, r + 1)

pdf = rv.pmf(x)
ax.plot(x, pdf,
        'ro', ms=8,
        label=r'pdf $Bin({}, {})$'.format(r, p))

hist = np.histogram(sizes, bins=np.arange(0, r + 2), density=True)[0]
ax.vlines(x, 0, hist,
          colors='b', lw=5, alpha=0.5,
          label='hist of sizes')

ax.legend(loc='best', frameon=False)

plt.title('p_value = {:.3f}'.format(chisquare(hist, pdf)[1]))
plt.show()
```

Geometrical insights

Kernels satisfying the sufficient conditions (3.3) and (3.4) can be expressed as

$$\mathbf{K}_{ij} = \langle \phi_i, \phi_j \rangle \quad \text{and} \quad \mathbf{L}_{ij} = \langle \psi_i, \psi_j \rangle,$$

where each item is represented by a feature vector ϕ_i (resp. ψ_i).

The geometrical view is then straightforward.

a. The inclusion probabilities read

$$\mathbb{P}[S \subset \mathcal{X}] = \det \mathbf{K}_S = \mathrm{Vol}^2\{\phi_s\}_{s \in S}.$$

b. The likelihood reads

$$\mathbb{P}[\mathcal{X} = S] \propto \det \mathbf{L}_S = \mathrm{Vol}^2\{\psi_s\}_{s \in S}.$$

That is to say, DPPs favor subsets S whose corresponding feature vectors span a large volume i.e. *DPPs sample softened orthogonal bases*.

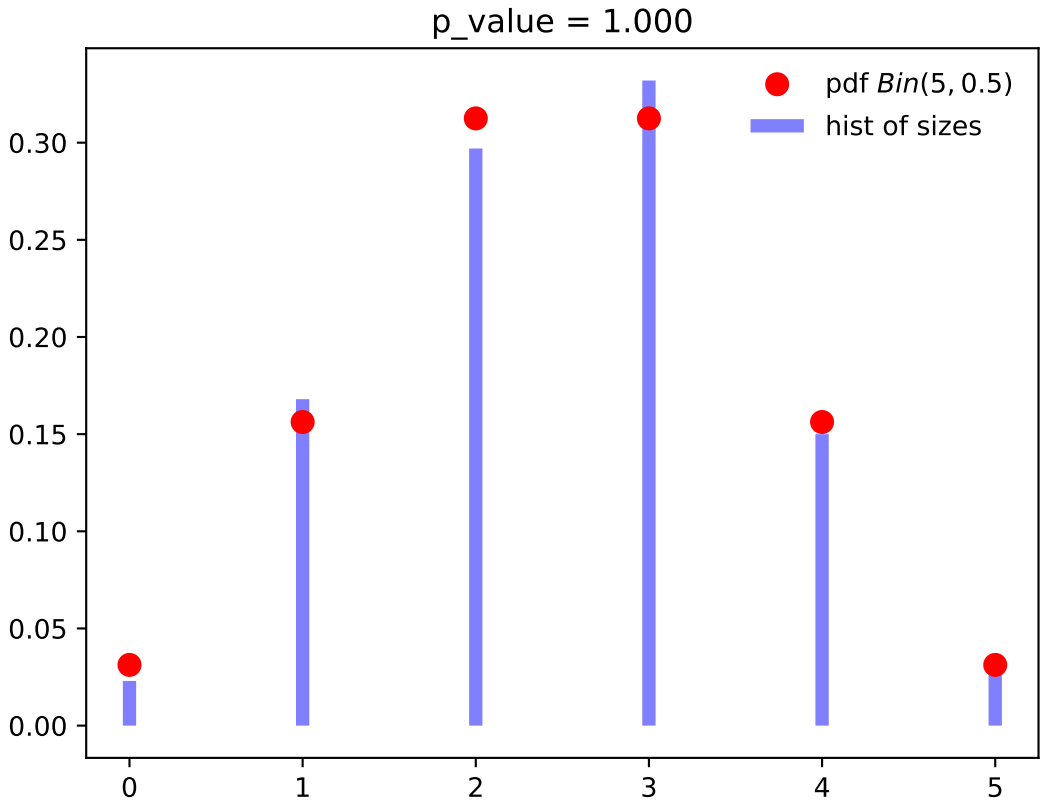


Fig. 3.3: Distribution of the numbe of points of $\mathrm{DPP}(\mathbf{L})$ with orthogonal projection kernel $\mathbf{L}$ with rank 5.

See also:

Geometric interpretation of the chain rule for projection DPPs

Diversity

The *determinantal* structure of DPPs encodes the notion of diversity. Deriving the pair inclusion probability, also called the 2-point correlation function using (3.1), we obtain

$$\begin{aligned}\mathbb{P}[\{i,j\} \subset \mathcal{X}] &= \begin{vmatrix} \mathbb{P}[i \in \mathcal{X}] & \mathbf{K}_{ij} \\ \overline{\mathbf{K}_{ij}} & \mathbb{P}[j \in \mathcal{X}] \end{vmatrix} \\ &= \mathbb{P}[i \in \mathcal{X}]\mathbb{P}[j \in \mathcal{X}] - |\mathbf{K}_{ij}|^2,\end{aligned}$$

so that, the larger $|\mathbf{K}_{ij}|$ less likely items i and j co-occur. If K_{ij} models the *similarity* between items i and j, DPPs are thus random diverse sets of elements.

Conditioning

Like many other statistics of DPPs, the conditional probabilities can be expressed my means of a determinant and involve the correlation kernel $\mathbf{K}$ (3.1).

For any disjoint subsets $S, T \subset [N]$, i.e., such that $S \cap T = \emptyset$ we have

$$\mathbb{P}[T \subset \mathcal{X} \mid S \subset \mathcal{X}] = \det\left[\mathbf{K}_T - \mathbf{K}_{TS}\mathbf{K}_S^{-1}\mathbf{K}_{ST}\right], \tag{3.15}$$

$$\mathbb{P}[T \subset \mathcal{X} \mid S \cap \mathcal{X} = \emptyset] = \det\left[\mathbf{K}_T - \mathbf{K}_{TS}(\mathbf{K}_S - I)^{-1}\mathbf{K}_{ST}\right]. \tag{3.16}$$

See also:

- Propositions 3 and 5 of [Pou19] for the proofs
- Equations (3.15) and (3.15) are key to derive the *Cholesky-based exact sampler* which makes use of the chain rule on sets.

3.1.3 Exact sampling

Consider a finite DPP defined by its correlation kernel $\mathbf{K}$ (3.1) or likelihood kernel $\mathbf{L}$ (3.2). There exist three main types of exact sampling procedures:

1. The spectral method (used by default) requires the eigendecomposition of the correlation kernel $\mathbf{K}$ or the likelihood kernel $\mathbf{L}$. It is based on the fact that *generic DPPs are mixtures of projection DPPs* together with the application of the chain rule to sample projection DPPs. It is presented in Section *Spectral method*.
2. A Cholesky-based procedure which requires the correlation kernel $\mathbf{K}$ (even non-Hermitian!). It boils down to applying the chain rule on sets; where each item in turn is decided to be excluded or included in the sample. It is presented in Section *Cholesky-based method*.
3. Recently, [DerezinskiCV19] have also proposed an alternative method to get exact samples: first sample an intermediate distribution and correct the bias by thinning the intermediate sample using a carefully designed DPP. This approach does not require a Cholesky/Eigen-decomposition of the DPP, but the runtime instead scale with the expected sample size of the DPP (see *Number of points*). It is presented in Section *Intermediate sampling method*. A more refined procedure based on this approach was introduced in [CDerezinskiV20] for k-DPP sampling.

In general, for small N (i.e. less than 1000) spectral or cholesky samplers are recommended for numerical stability. For larger N (i.e. up to millions) and moderate k (i.e. in the hundreds) intermediate sampling is recommended for scalability.

The following table summarizes the complexity of all exact samplers currently available, where the expected sample size $\mathbb{E}[|X|]$ is equal to k for k-DPPs and d_{eff} for random-sized DPPs.

	`mode=`	Time to first sample		Time to subsequent samples		Notes
		DPP	k-DPP	DPP	k-DPP	
Spectral sampler	`"GS"`, `"GS_bis"`, `"KuTa12"`	$O(N^3)$	$O(N^3)$	$O(Nd_{\text{eff}}^2)$	$O(Nk^2)$	The three variants differ slightly, and depending on the DPP they can have different numerical stability.
Cholesky sampler	`"chol"`	$O(N^3)$	$O(N^3)$	$O(N^3)$	$O(N^3)$	Also works for non-Hermitian DPPs.
Intermediate sampler	`"vfx"`	$O(Nd_{\text{eff}}^6)$	$O(Nk^{10}+k^{15})$	$O(d_{\text{eff}}^6)$	$O(k^6)$	For `"alpha"` we report worst case runtime, but depending on the DPP structure best case runtime can be much faster than `"vfx"`. For particularly ill-posed DPPs `"vfx"` can be more numerically stable.
	`"alpha"`	$O(Nd_{\text{eff}}^5)$	$O(Nk^6/d_{\text{eff}}+k^9)$	$O(d_{\text{eff}}^6)$	$O(k^6)$	

Note:

- There exist specific samplers for special DPPs, like the ones presented in Section *Exotic DPPs*.

Important: In the next section, we describe the Algorithm 18 of [HKPVirag06], based on the chain rule, which was originally designed to *sample continuous projection DPPs*. Obviously, it has found natural a application in the finite setting for sampling projection $\mathrm{DPP}(\mathbf{K})$. However, **we insist on the fact that this chain rule mechanism is specific to orthogonal projection kernels**. In particular, it cannot be applied blindly to sample general k-$\mathrm{DPP}(\mathbf{L})$ but it is valid when $\mathbf{L}$ is an orthogonal projection kernel.

This crucial point is developed in the following *Caution* section.

Projection DPPs: the chain rule

In this section, we describe the generic projection DPP sampler, originally derived by [HKPVirag06] Algorithm 18.

Recall that the *number of points of a projection* $r = \mathrm{DPP}(\mathbf{K})$ is, almost surely, equal to $\mathrm{rank}(K)$, so that the likelihood (3.13) of $S = \{s_1, \dots, s_r\}$ reads

$$\mathbb{P}[\mathcal{X} = S] = \det \mathbf{K}_S.$$

Using the invariance by permutation of the determinant and the fact that $\mathbf{K}$ is an orthogonal projection matrix, it is sufficient to apply the chain rule to sample $(s_1, \dots, s_r)$ with joint distribution

$$\mathbb{P}[(s_1, \dots, s_r)] = \frac{1}{r!}\mathbb{P}[\mathcal{X} = \{s_1, \dots, s_r\}] = \frac{1}{r!}\det \mathbf{K}_S,$$

and forget about the sequential feature of the chain rule to get a valid sample $\{s_1, \dots, s_r\} \sim \mathrm{DPP}(\mathbf{K})$.

Considering $S = \{s_1, \dots, s_r\}$ such that $\mathbb{P}[\mathcal{X} = S] = \det \mathbf{K}_S > 0$, the following generic formulation of the chain rule

$$\mathbb{P}[(s_1, \dots, s_r)] = \mathbb{P}[s_1] \prod_{i=2}^{r} \mathbb{P}[s_i | s_{1:i-1}],$$

can be expressed as a telescopic ratio of determinants

$$\mathbb{P}[(s_1, \dots, s_r)] = \frac{\mathbf{K}_{s_1,s_1}}{r} \prod_{i=2}^{r} \frac{1}{r-(i-1)} \frac{\det \mathbf{K}_{S_i}}{\det \mathbf{K}_{S_{i-1}}}, \tag{3.17}$$

where $S_{i-1} = \{s_1, \dots, s_{i-1}\}$.

Using Woodbury's formula the ratios of determinants in (3.17) can be expanded into

$$\mathbb{P}[(s_1, \dots, s_r)] = \frac{\mathbf{K}_{s_1,s_1}}{r} \prod_{i=2}^{r} \frac{\mathbf{K}_{s_i,s_i} - \mathbf{K}_{s_i,S_{i-1}} {\mathbf{K}_{S_{i-1}}}^{-1} \mathbf{K}_{S_{i-1},s_i}}{r-(i-1)}. \tag{3.18}$$

Hint: MLers will recognize in (3.18) the incremental posterior variance of the Gaussian Process (GP) associated to $\mathbf{K}$, see [RW06] Equation 2.26.

Caution: The connexion between the chain rule (3.18) and Gaussian Processes is valid in the case where the GP kernel is an **orthogonal projection kernel**, see also *Caution*.

See also:

- Algorithm 18 [HKPVirag06]
- *Projection DPPs: the chain rule* in the continuous case

Geometrical interpretation

Hint: Since $\mathbf{K}$ is an **orthogonal projection** matrix, the following Gram factorizations provide an insightful geometrical interpretation of the chain rule mechanism (3.17):

1. Using $\mathbf{K} = \mathbf{K}^2$ and $\mathbf{K}^\dagger = \mathbf{K}$, we have $\mathbf{K} = \mathbf{K}\mathbf{K}^\dagger$, so that the chain rule (3.17) becomes

$$\begin{aligned} \mathbb{P}[(s_1, \dots, s_r)] &= \frac{1}{r!} \operatorname{Volume}^2(\mathbf{K}_{s_1,:}, \dots, \mathbf{K}_{s_r,:}) \\ &= \frac{\|\mathbf{K}_{s_1,:}\|^2}{r} \prod_{i=2}^{r} \frac{\operatorname{distance}^2(\mathbf{K}_{s_i,:}, \operatorname{Span}\{\mathbf{K}_{s_1,:}, \dots, \mathbf{K}_{s_{i-1},:}\}}{r-(i-1)}. \end{aligned} \tag{3.19}$$

2. Using the eigendecomposition, we can write $\mathbf{K} = UU^\dagger$ where $U^\dagger U = I_r$, so that the chain rule (3.17) becomes

$$\begin{aligned} \mathbb{P}[(s_1, \dots, s_r)] &= \frac{1}{r!} \operatorname{Volume}^2(U_{s_1,:}, \dots, U_{s_r,:}) \\ &= \frac{\|U_{s_1,:}\|^2}{r} \prod_{i=2}^{r} \frac{\operatorname{distance}^2(U_{s_i,:}, \operatorname{Span}\{U_{s_1,:}, \dots, U_{s_{i-1},:}\}}{r-(i-1)}. \end{aligned} \tag{3.20}$$

In other words, the chain rule formulated as (3.19) and (3.20) is akin to do Gram-Schmidt orthogonalization of the *feature vectors* $\mathbf{K}_{i,:}$ or $\mathbf{U}_{i,:}$. These formulations can also be understood as an application of the base $\times$ height formula.

In the end, projection DPPs favors sets of $r = \mathrm{rank}(\mathbf{K})$ of items are associated to feature vectors that span large volumes. This is another way of understanding *diversity*.

See also:

MCMC samplers

- *zonotope sampling*
- *basis exchange*

In practice

The cost of getting one sample from a **projection** DPP is of order $\mathcal{O}(N\,\mathrm{rank}(\mathbf{K})^2)$, whenever $\mathrm{DPP}(\mathbf{K})$ is defined through

- $\mathbf{K}$ itself; sampling relies on formulations (3.19) or (3.18)

```
import numpy as np
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

seed = 0
rng = np.random.RandomState(seed)

r, N = 4, 10
eig_vals = np.ones(r)  # For projection DPP
eig_vecs, _ = qr(rng.randn(N, r), mode='economic')

DPP = FiniteDPP(kernel_type='correlation',
                projection=True,
                **{'K': (eig_vecs * eig_vals).dot(eig_vecs.T)})

for mode in ('GS', 'Schur', 'Chol'):  # default: GS

    rng = np.random.RandomState(seed)
    DPP.flush_samples()

    for _ in range(10):
        DPP.sample_exact(mode=mode, random_state=rng)

    print(DPP.sampling_mode)
    print(DPP.list_of_samples)
```

```
GS
[[5, 7, 2, 1], [4, 6, 2, 9], [9, 2, 6, 4], [5, 9, 0, 1], [0, 8, 6, 7], [9,
↪ 6, 2, 7], [0, 6, 2, 9], [5, 2, 1, 8], [5, 4, 0, 8], [5, 6, 9, 1]]
Schur
[[5, 7, 2, 1], [4, 6, 2, 9], [9, 2, 6, 4], [5, 9, 0, 1], [0, 8, 6, 7], [9,
↪ 6, 2, 7], [0, 6, 2, 9], [5, 2, 1, 8], [5, 4, 0, 8], [5, 6, 9, 1]]
Chol
[[5, 7, 6, 0], [4, 6, 5, 7], [9, 5, 0, 1], [5, 9, 2, 4], [0, 8, 1, 7], [9,
↪ 0, 5, 1], [0, 6, 5, 9], [5, 0, 1, 9], [5, 0, 2, 8], [5, 6, 9, 1]]
```

See also:

- *sample_exact()*

- [HKPVirag06] Theorem 7, Algorithm 18 and Proposition 19, for the original idea
- [Pou19] Algorithm 3, for the equivalent Cholesky-based perspective with cost of order $\mathcal{O}(N \operatorname{rank}(\mathbf{K})^2)$

- its eigenvectors U, i.e., $\mathbf{K} = UU^\dagger$ with $U^\dagger U = I_{\operatorname{rank}(\mathbf{K})}$; sampling relies on (3.20)

```
import numpy as np
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

seed = 0
rng = np.random.RandomState(seed)

r, N = 4, 10
eig_vals = np.ones(r)  # For projection DPP
eig_vecs, _ = qr(rng.randn(N, r), mode='economic')

DPP = FiniteDPP(kernel_type='correlation',
                                projection=True,
                                **{'K_eig_dec': (eig_vals, eig_vecs)})

rng = np.random.RandomState(seed)

for _ in range(10):
        # mode='GS': Gram-Schmidt (default)
        DPP.sample_exact(mode='GS', random_state=rng)

print(DPP.list_of_samples)
```

```
[[5, 7, 2, 1], [4, 6, 2, 9], [9, 2, 6, 4], [5, 9, 0, 1], [0, 8, 6, 7], [9,
↪ 6, 2, 7], [0, 6, 2, 9], [5, 2, 1, 8], [5, 4, 0, 8], [5, 6, 9, 1]]
```

See also:

- *sample_exact()*
- [HKPVirag06] Theorem 7, Algorithm 18 and Proposition 19, for the original idea
- [KT12] Algorithm 1, for a first interpretation of the spectral counterpart of [HKPVirag06] Algorithm 18 running in $\mathcal{O}(N \operatorname{rank}(\mathbf{K})^3)$
- [Gil14] Algorithm 2, for the $\mathcal{O}(N \operatorname{rank}(\mathbf{K})^2)$ implementation
- [TBA18] Algorithm 3, for a technical report on DPP sampling

Spectral method

Main idea

The procedure stems from Theorem 7 of [HKPVirag06], i.e., the fact that *generic DPPs are mixtures of projection DPPs*, suggesting the following two steps algorithm. Given the spectral decomposition of the correlation kernel $\mathbf{K}$

$$\mathbf{K} = U \Lambda U^\dagger = \sum_{n=1}^{N} \lambda_n u_n u_n^\dagger.$$

Step 1. Draw independent Bernoulli random variables $B_n \sim \mathcal{B}\mathrm{er}(\lambda_n)$ for $n = 1, \dots, N$ and collect $\mathcal{B} = \{n \, ; \, B_n = 1\}$,

Step 2. Sample from the **projection** DPP with correlation kernel $U_{:\mathcal{B}}U_{:\mathcal{B}}^\dagger = \sum_{n\in\mathcal{B}} u_n u_n^\dagger$, see *the section above*.

Note: **Step 1.** selects a component of the mixture while **Step 2.** requires sampling from the corresponding **projection** DPP, cf. *Projection DPPs: the chain rule*

In practice

- Sampling *projection* $\mathrm{DPP}(\mathbf{K})$ from the eigendecomposition of $\mathbf{K} = UU^\dagger$ with $U^\dagger U = I_{\mathrm{rank}(\mathbf{K})}$ was presented in *the section above*
- Sampling $\mathrm{DPP}(\mathbf{K})$ from $0_N \preceq \mathbf{K} \preceq I_N$ can be done by following

  Step 0. compute the eigendecomposition of $\mathbf{K} = U\Lambda U^\dagger$ in $\mathcal{O}(N^3)$.

  Step *1.* draw independent Bernoulli random variables $B_n \sim \mathcal{B}\mathrm{er}(\lambda_n)$ for $n = 1, \dots, N$ and collect $\mathcal{B} = \{n\ ;\ B_n = 1\}$

  Step *2.* sample from the **projection** DPP with correlation kernel defined by its eigenvectors $U_{:,\mathcal{B}}$

  Important: Step 0. must be performed once and for all in $\mathcal{O}(N^3)$. Then the average cost of getting one sample by applying Steps 1. and 2. is $\mathcal{O}(N\mathbb{E}\left[|\mathcal{X}|\right]^2)$, where $\mathbb{E}\left[|\mathcal{X}|\right] = \mathrm{trace}(\mathbf{K}) = \sum_{n=1}^N \lambda_n$.

```
from numpy.random import RandomState
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

rng = RandomState(0)

r, N = 4, 10
eig_vals = rng.rand(r)  # For projection DPP
eig_vecs, _ = qr(rng.randn(N, r), mode='economic')

DPP = FiniteDPP(kernel_type='correlation',
                                projection=False,
                                **{'K': (eig_vecs*eig_vals).dot(eig_vecs.
↪T)})

for _ in range(10):
        # mode='GS': Gram-Schmidt (default)
        DPP.sample_exact(mode='GS', random_state=rng)

print(DPP.list_of_samples)
```

```
[[7, 0, 1, 4], [6], [0, 9], [0, 9], [8, 5], [9], [6, 5, 9], [9], [3, 0],␣
↪[5, 1, 6]]
```

- Sampling $\mathrm{DPP}(\mathbf{L})$ from $\mathbf{L} \succeq 0_N$ can be done by following

  Step 0. compute the eigendecomposition of $\mathbf{L} = V\Gamma V^\dagger$ in $\mathcal{O}(N^3)$.

  Step *1.* is adapted to: draw independent Bernoulli random variables $B_n \sim \mathcal{B}\mathrm{er}(\frac{\gamma_n}{1+\gamma_n})$ for $n = 1, \dots, N$ and collect $\mathcal{B} = \{n\ ;\ B_n = 1\}$

  Step *2.* is adapted to: sample from the **projection** DPP with correlation kernel defined by its eigenvectors $V_{:,\mathcal{B}}$.

Important: Step 0. must be performed once and for all in $\mathcal{O}(N^3)$. Then the average cost of getting one sample by applying Steps 1. and 2. is $\mathcal{O}(N\mathbb{E}\left[|\mathcal{X}|\right]^2)$, where $\mathbb{E}\left[|\mathcal{X}|\right] = \mathrm{trace}(\mathbf{L}(\mathbf{I}+\mathbf{L})^{-1}) = \sum_{n=1}^{N} \frac{\gamma_n}{1+\gamma_n}$

```
from numpy.random import RandomState
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

rng = RandomState(0)

r, N = 4, 10
phi = rng.randn(r, N)

DPP = FiniteDPP(kernel_type='likelihood',
                                projection=False,
                                **{'L': phi.T.dot(phi)})

for _ in range(10):
        # mode='GS': Gram-Schmidt (default)
        DPP.sample_exact(mode='GS', random_state=rng)

print(DPP.list_of_samples)
```

```
[[3, 1, 0, 4], [9, 6], [4, 1, 3, 0], [7, 0, 6, 4], [5, 0, 7], [4, 0, 2],␣
↪[5, 3, 8, 4], [0, 5, 2], [7, 0, 2], [6, 0, 3]]
```

- Sampling a $\mathrm{DPP}(\mathbf{L})$ for which each item is represented by a $d \leq N$ dimensional feature vector, all stored in a *feature matrix* $\Phi \in \mathbb{R}^{d\times N}$, so that $\mathbf{L} = \Phi^\top\Phi \succeq 0_N$, can be done by following

  Step 0. compute the so-called *dual* kernel $\tilde{L} = \Phi\Phi^\dagger \in \mathbb{R}^{d\times}$, eigendecompose it $\tilde{\mathbf{L}} = W\Delta W^\top$ and recover the eigenvectors of $\mathbf{L}$ as $V = \Phi^\top W\Delta^{-\frac{1}{2}}$ This corresponds to a cost of order $\mathcal{O}(Nd^2 + d^3 + d^2 + Nd^2)$.

  Step *1.* is adapted to: draw independent Bernoulli random variables $B_i \sim \mathcal{B}\mathrm{er}(\frac{\delta_i}{1+\delta_i})$ for $i = 1, \ldots, d$ and collect $\mathcal{B} = \{i\ ;\ B_i = 1\}$

  Step *2.* is adapted to: sample from the **projection** DPP with correlation kernel defined by its eigenvectors $V_{:,\mathcal{B}} = \left[\Phi^\top W\Delta^{-1/2}\right]_{:,\mathcal{B}}$.

  Important: Step 0. must be performed once and for all in $\mathcal{O}(Nd^2 + d^3)$. Then the average cost of getting one sample by applying Steps 1. and 2. is $\mathcal{O}(N\mathbb{E}\left[|\mathcal{X}|\right]^2)$, where $\mathbb{E}\left[|\mathcal{X}|\right] = \mathrm{trace}(\tilde{\mathbf{L}}(\mathbf{I}+\tilde{\mathbf{L}})^{-1}) = \sum_{i=1}^{d} \frac{\delta_i}{1+\delta_i} \leq d$

  See also:

  For a different perspective see

  – [Gil14] Section 2.4.4 and Algorithm 3

  – [KT12] Section 3.3.3 and Algorithm 3

```
from numpy.random import RandomState
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP
```

(continues on next page)

(continued from previous page)

```
rng = RandomState(0)

r, N = 4, 10
phi = rng.randn(r, N)  # L = phi.T phi, L_dual = phi phi.T

DPP = FiniteDPP(kernel_type='likelihood',
                                projection=False,
                                **{'L_gram_factor': phi})

for _ in range(10):
        # mode='GS': Gram-Schmidt (default)
        DPP.sample_exact(mode='GS', random_state=rng)

print(DPP.list_of_samples)
```

```
L_dual = Phi Phi.T was computed: Phi (dxN) with d<N
[[9, 0, 2, 3], [0, 1, 5, 2], [7, 0, 9, 4], [2, 0, 3], [6, 4, 0, 3], [5, 0,
↪ 6, 3], [0, 6, 3, 9], [4, 0, 9], [7, 3, 9, 4], [9, 4, 3]]
```

Cholesky-based method

Main idea

This method requires access to the correlation kernel $\mathbf{K}$ to perform a bottom-up chain rule on sets: starting from the empty set, each item in turn is decided to be added or excluded from the sample. This can be summarized as the exploration of the binary probability tree displayed in Fig. 3.4.

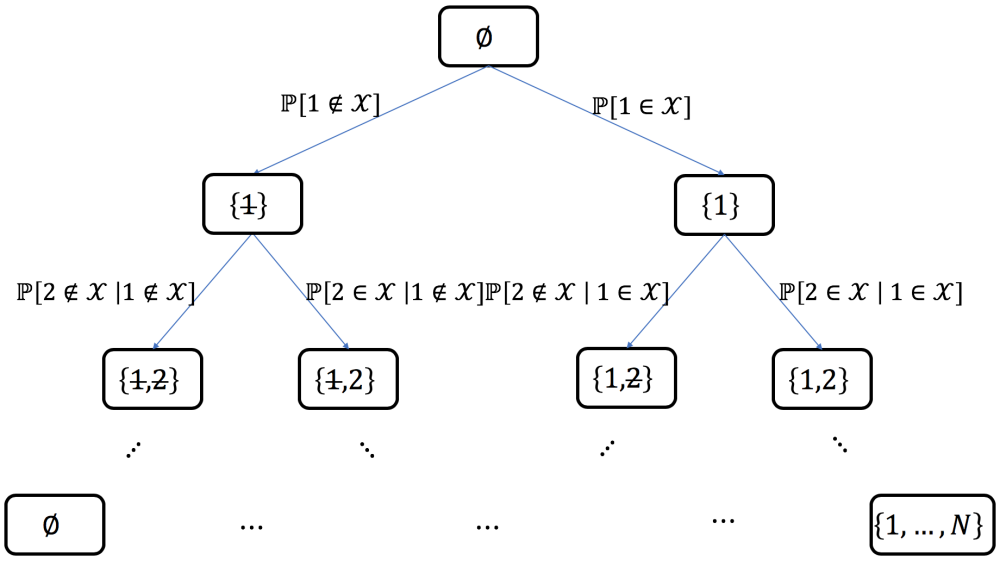


Fig. 3.4: Probability tree corresponding to the chain rule on sets

Example: for $N = 5$, if $\{1, 4\}$ was sampled, the path in the probability tree would correspond to

$$\begin{aligned}\mathbb{P}[\mathcal{X} = \{1,4\}] =& \mathbb{P}[1 \in \mathcal{X}] \\ & \times \mathbb{P}[2 \notin \mathcal{X} \mid 1 \in \mathcal{X}] \\ & \times \mathbb{P}[3 \notin \mathcal{X} \mid 1 \in \mathcal{X}, 2 \notin \mathcal{X}] \\ & \times \mathbb{P}[4 \in \mathcal{X} \mid 1 \in \mathcal{X}, \{2,3\} \cap \mathcal{X} = \emptyset] \\ & \times \mathbb{P}[5 \notin \mathcal{X} \mid \{1,4\} \subset \mathcal{X}, \{2,3\} \cap \mathcal{X} = \emptyset] \,,\end{aligned}$$

where each conditional probability can be written in closed formed using (3.15) and (3.16), namely

$$\begin{aligned}\mathbb{P}[T \subset \mathcal{X} \mid S \subset \mathcal{X}] &= \det\left[\mathbf{K}_T - \mathbf{K}_{TS}\mathbf{K}_S^{-1}\mathbf{K}_{ST}\right] \\ \mathbb{P}[T \subset \mathcal{X} \mid S \cap \mathcal{X} = \emptyset] &= \det\left[\mathbf{K}_T - \mathbf{K}_{TS}(\mathbf{K}_S - I)^{-1}\mathbf{K}_{ST}\right] .\end{aligned}$$

Important: This quantities can be computed efficiently as they appear in the computation of the Cholesky-type $LDL^\dagger$ or LU factorization of the correlation $\mathbf{K}$ kernel, in the Hermitian or non-Hermitian case, respectively. See [Pou19] for the details.

Note: The sparsity of $\mathbf{K}$ can be leveraged to derive faster samplers using the correspondence between the chain rule on sets and Cholesky-type factorizations, see e.g., [Pou19] Section 4.

In practice

Important:

- The method is fully generic since it applies to any (valid), even non-Hermitian, correlation kernel $\mathbf{K}$.
- Each sample costs $\mathcal{O}(N^3)$.
- Nevertheless, the link between the chain rule on sets and Cholesky-type factorization is nicely supported by the surprisingly simple implementation of the corresponding generic sampler.

```
# Poulson (2019, Algorithm 1) pseudo-code

sample = []
A = K.copy()

for j in range(N):

        if np.random.rand() < A[j, j]:  # Bernoulli(A_jj)
                sample.append(j)
        else:
                A[j, j] -= 1

        A[j+1:, j] /= A[j, j]
        A[j+1:, j+1:] -= np.outer(A[j+1:, j], A[j, j+1:])

return sample, A
```

```
from numpy.random import RandomState
from scipy.linalg import qr
from dppy.finite_dpps import FiniteDPP

rng = RandomState(1)

r, N = 4, 10
eig_vals = rng.rand(r)
eig_vecs, _ = qr(rng.randn(N, r), mode='economic')

DPP = FiniteDPP(kernel_type='correlation',
                                projection=False,
                                **{'K': (eig_vecs*eig_vals).dot(eig_vecs.T)})

for _ in range(10):
        DPP.sample_exact(mode='Chol', random_state=rng)

print(DPP.list_of_samples)
```

```
[[2, 9], [0], [2], [6], [4, 9], [2, 7, 9], [0], [1, 9], [0, 1, 2], [2]]
```

See also:

- [Pou19]
- [LGD18]

Intermediate sampling method

Main idea

This method is based on the concept of a **distortion-free intermediate sample**, where we draw a larger sample of points in such a way that we can then downsample to the correct DPP distribution. We assume access to the likelihood kernel $\mathbf{L}$ (although a variant of this method also exists for projection DPPs). Crucially the sampling relies on an important connection between DPPs and so-called **ridge leverage scores** (RLS, see [AM15]), which are commonly used for sampling in randomized linear algebra. Namely, the marginal probability of the i-th point in $\mathcal{X} \sim \mathrm{DPP}(\mathbf{L})$ is also the i-th ridge leverage score of $\mathbf{L}$ (with ridge parameter equal 1):

$$\mathbb{P}[i \in \mathcal{X}] = \left[\mathbf{L}(I + \mathbf{L})^{-1}\right]_{ii} = \tau_i, \quad i\text{th 1-ridge leverage score.}$$

Suppose that we draw a sample σ of t points i.i.d. proportional to ridge leverage scores, i.e., $\sigma = (\sigma_1, \sigma_2, ..., \sigma_t)$ such that $\mathbb{P}[\sigma_j = i] \propto \tau_i$. Intuitively, this sample is similar fo $\mathcal{X} \sim \mathrm{DPP}(\mathbf{L})$ because the marginals are the same, but it "ignores" all the dependencies between the points. However, if we sample sufficiently many points i.i.d. according to RLS, then a proper sample $\mathcal{X}$ will likely be contained within σ. This can be formally shown for $t = O(\mathbb{E}[|\mathcal{X}|]^2)$. When $\mathbb{E}[|\mathcal{X}|]^2 \ll N$, then this allows us to reduce the size of the DPP kernel $\mathbf{L}$ from $N \times N$ to a much smaller size $\tilde{\mathbf{L}}$ $t \times t$. Making this sampling exact requires considerably more care, because even with a large t there is always a small probability that the i.i.d. sample σ is not sufficiently diverse. We guard against this possibility by rejection sampling.

Important: Use this method for sampling $\mathcal{X} \sim \mathrm{DPP}(\mathbf{L})$ when $\mathbb{E}\left[|\mathcal{X}|\right] \ll \sqrt{N}$.

- Preprocessing costs $\mathcal{O}\big(N \cdot \mathrm{poly}(\mathbb{E}\left[|\mathcal{X}|\right])\,\mathrm{polylog}(N)\big)$.
- Each sample costs $\mathcal{O}\big(\mathbb{E}[|\mathcal{X}|]^6\big)$.

There are two implementations of intermediate sampling available in `dppy`: the `mode='vfx'` sampler and the `mode='alpha'` sampler.

In practice

```
from numpy.random import RandomState
from dppy.finite_dpps import FiniteDPP
from dppy.utils import example_eval_L_linear

rng = RandomState(1)

r, N = 4, 10

DPP = FiniteDPP('likelihood',
        **{'L_eval_X_data': (example_eval_L_linear, rng.randn(N, r))})

for _ in range(10):
    DPP.sample_exact(mode='vfx', random_state=rng, verbose=False)

print(DPP.list_of_samples)
```

```
[[5, 1, 0, 3], [9, 0, 8, 3], [6, 4, 1], [5, 1, 2], [2, 1, 3], [3, 8, 4, 0], [0, 8, 1],
↪ [7, 8], [1, 8, 2, 0], [5, 8, 3]]
```

The `verbose=False` flag is used to suppress the default progress bars when running in batch mode (e.g. when generating these docs).

Given, the RLS $\tau_1, \ldots, \tau_N$, the normalization constant $\det(I + \tilde{\mathbf{L}}_\sigma)$ and access to the likelihood kernel $\tilde{\mathbf{L}}_\sigma$, the intermediate sampling method proceeds as follows:

repeat

sample $t \sim \text{Poisson}(k^2 \, \mathrm{e}^{1/k})$, where $k = \mathbb{E}[|\mathcal{X}|]$

sample $\sigma_1, \ldots, \sigma_t \sim (\tau_1, \ldots \tau_N)$,

sample $Acc \sim \text{Bernoulli}\Big(\dfrac{\mathrm{e}^k \det(I + \tilde{\mathbf{L}}_\sigma)}{\mathrm{e}^{t/k} \det(I + \mathbf{L})}\Big)$, where $\tilde{L}_{ij} = \dfrac{1}{k\sqrt{\tau_i \tau_j}} L_{ij}$,

until $Acc = \text{true}$

return $\mathcal{X} = \{\sigma_i : i \in \tilde{\mathcal{X}}\}$ where $\tilde{\mathcal{X}} \sim \text{DPP}(\tilde{\mathbf{L}}_\sigma)$

It can be shown that $\mathcal{X}$ is distributed exactly according to $\text{DPP}(\mathbf{L})$ and the expected number of rejections is a small constant. The intermediate likelihood kernel $\tilde{\mathbf{L}}_\sigma$ forms a $t \times t$ DPP subproblem that can be solved using any other DPP sampler.

- Since the size of the intermediate sample is $t = \mathcal{O}(\mathbb{E}[\mathcal{X}]^2)$, the primary cost of the sampling is computing $\det(I + \tilde{\mathbf{L}}_\sigma)$ which takes $\mathcal{O}(t^3) = \mathcal{O}(\mathbb{E}[\mathcal{X}]^6)$ time. This is also the expected cost of sampling from $\text{DPP}(\tilde{\mathbf{L}}_\sigma)$ if we use, for example, the spectral method.
- The algorithm requires precomputing the RLS $\tau_1, \ldots, \tau_n$ and $\det(I+\mathbf{L})$. Computing them exactly takes $\mathcal{O}(N^3)$, however, surprisingly, if we use sufficiently accurate approximations then the exactness of the sampling can be retained (details in [DerezinskiCV19]). Efficient methods for approximating leverage scores (see [RCCR18]) bring the precomputing cost down to $\mathcal{O}(N\text{poly}(\mathbb{E}\,[|\mathcal{X}|])\text{polylog}(N))$.

- When $\mathbb{E}[|\mathcal{X}|]$ is sufficiently small, the entire sampling procedure only looks at a small fraction of the entries of $\mathbf{L}$. This makes the method useful when we want to avoid constructing the entire likelihood kernel.
- When the likelihood kernel is given implicitly via a matrix $\mathbf{X}$ such that $\mathbf{L} = \mathbf{X}\mathbf{X}^\top$ (dual formulation) then a version of this method is given by [Derezinski19]
- A variant of this method also exists for projection DPPs [DWH18]

See also:

- [DerezinskiCV19] (Likelihood kernel)
- [Derezinski19] (Dual formulation)
- [DWH18] (Projection DPP)

k-DPPs

Main idea

Recall from (3.5) that k-DPP$(\mathbf{L})$ can be viewed as a DPP$(\mathbf{L})$ constrained to a have fixed cardinality $k \leq \mathrm{rank}(L)$.

To generate a sample of k-DPP$(\mathbf{L})$, one natural solution would be to use a rejection mechanism: draw $S \sim \mathrm{DPP}(\mathbf{L})$ and keep it only if $|X| = k$. However, the rejection constant may be pretty bad depending on the choice of k regarding the distribution of the number of points (3.9) of $S \sim \mathrm{DPP}(\mathbf{L})$.

An alternative solution was found by [KT12] Section 5.2.2. The procedure relies on a slight modification of *Step 1.* of the *Spectral method* which requires the computation of the elementary symmetric polynomials.

In practice

Sampling k-DPP$(\mathbf{L})$ from $\mathbf{L} \succeq 0_N$ can be done by following

Step 0.

a) compute the eigendecomposition of $\mathbf{L} = V\Gamma V^\dagger$ in $\mathcal{O}(N^3)$

b) evaluate the elementary symmetric polynomials in the eigenvalues of $\mathbf{L}$: $E[l, n] := e_l(\gamma_1, \ldots, \gamma_n)$ for $0 \leq l \leq k$ and $0 \leq n \leq N$. These computations can done recursively in $\mathcal{O}(Nk)$ using Algorithm 7 of [KT12].

Step *1.* is replaced by Algorithm 8 of [KT12] which we illustrate with the following pseudo-code

```
# Algorithm 8 of Kulesza Taskar (2012).
# This is a pseudo-code of in particular Python indexing is not␣
↪respected everywhere

B = set({})
l = k

for n in range(N, 0, -1):

  if Unif(0,1) < gamma[n] * E[l-1, n-1] / E[l, n]:
    l -= 1
    B.union({n})

    if l == 0:
      break
```

Step 2. is adapted to: sample from the **projection** DPP with correlation kernel defined by its eigenvectors $V_{:,\mathcal{B}}$, with a cost of order $\mathcal{O}(Nk^2)$.

Important: Step 0. must be performed once and for all in $\mathcal{O}(N^3 + Nk)$. Then the cost of getting one sample by applying Steps 1. and 2. is $\mathcal{O}(Nk^2)$.

```
import numpy as np
from dppy.finite_dpps import FiniteDPP

rng = np.random.RandomState(1)

r, N = 5, 10
# Random feature vectors
Phi = rng.randn(r, N)
DPP = FiniteDPP('likelihood', **{'L': Phi.T.dot(Phi)})

k = 4
for _ in range(10):
    DPP.sample_exact_k_dpp(size=k, random_state=rng)

print(DPP.list_of_samples)
```

```
[[1, 8, 5, 7], [3, 8, 5, 9], [5, 3, 1, 8], [5, 8, 2, 9], [1, 2, 9, 6], [1, 0, 2, 3],␣
↪[7, 0, 3, 5], [8, 3, 7, 6], [0, 2, 3, 7], [1, 3, 7, 5]]
```

See also:

- *sample_exact_k_dpp()*
- Step 0. requires [KT12] Algorithm 7 for the recursive evaluation of the elementary symmetric polynomials $[e_l(\gamma_1, \dots, \gamma_n)]_{l=1,n=1}^{k,N}$ in the eigenvalues of $\mathbf{L}$
- Step 1. calls [KT12] Algorithm 8 for selecting the eigenvectors for Step 2.

Caution

Attention: Since the number of points of k-DPP$(\mathbf{L})$ is fixed, like for *projection DPPs*, it might be tempting to sample k-DPP$(\mathbf{L})$ using a chain rule in the way it was applied in (3.18) to sample projection DPPs. **However, it is incorrect: sampling sequentially**

$$s_1 \propto \mathbf{L}_{s,s}, \quad \text{then} \quad s_i \mid s_1, \dots, s_{i-1} \propto \mathbf{L}_{s,s} - \mathbf{L}_{s,S_{i-1}} \mathbf{L}_{S_{i-1}}{}^{-1} \mathbf{L}_{S_{i-1},s}, \quad \text{for } 2 \leq i \leq k, \tag{3.21}$$

where $S_{i-1} = \{s_1, \dots, s_{i-1}\}$, **and forgetting about the order** $s_1, \dots, s_k$ **were selected does not provide a subset** $\{s_1, \dots, s_k\} \sim k$-DPP$(\mathbf{L})$, **in the general case. Nevertheless, it is valid when** $\mathbf{L}$ **is an orthogonal projection kernel!**

Here are the reasons why

1. First keep in mind that, the ultimate goal is to draw a **subset** $S = \{s_1, \dots, s_k\} \sim k$-DPP$(\mathbf{L})$ with probability (3.5)

$$\mathbb{P}[\mathcal{X} = S] = \frac{1}{e_k(\mathbf{L})} \det \mathbf{L}_S 1_{|S|=k}. \tag{3.22}$$

2. Now, if we were to use the chain rule (3.21) this would correspond to sampling sequentially the items $s_1, \dots, s_k$, so that the resulting **vector** $(s_1, \dots, s_k)$ has probability

$$\begin{aligned}\mathbb{Q}[(s_1, \dots, s_k)] &= \frac{\mathbf{L}_{s_1,s_1}}{Z_1} \prod_{i=2}^{k} \frac{\mathbf{L}_{s_i,s_i} - \mathbf{L}_{s_i,S_{i-1}} \mathbf{L}_{S_{i-1}}{}^{-1} \mathbf{L}_{S_{i-1},s_i}}{Z_i(s_1, \dots, s_{i-1})} \\ &= \frac{1}{Z(s_1, \dots, s_k)} \det \mathbf{L}_S.\end{aligned} \tag{3.23}$$

Contrary to $Z_1 = \mathrm{trace}(\mathbf{L})$, the normalizations $Z_i(s_1, \dots, s_{i-1})$ of the successive conditionals depend, *a priori*, on the order $s_1, \dots, s_k$ were selected. For this reason we denote the global normalization constant $Z(s_1, \dots, s_k)$.

Warning: Equation (3.23) suggests that, the sequential feature of the chain rule matters, *a priori*; the distribution of $(s_1, \dots, s_k)$ is not exchangeable *a priori*, i.e., it is not invariant to permutations of its coordinates. This fact, would only come from the normalization $Z(s_1, \dots, s_k)$, since $\mathbf{L}_S$ is invariant by permutation.

Note: To see this, let's compute the normalization constant $Z_i(s_1, \dots, s_{i-1})$ in (3.23) for a generic $\mathbf{L} \succeq 0_N$ factored as $\mathbf{L} = VV^\dagger$, with no specific assumption on V.

$$\begin{aligned}Z_i(s_1, \dots, s_{i-1}) &= \sum_{i=1}^{N} \mathbf{L}_{ii} - \mathbf{L}_{i,S_{i-1}} \mathbf{L}_{S_{i-1}}^{-1} \mathbf{L}_{S_{i-1},i} \\ &= \mathrm{trace}(\mathbf{L} - \mathbf{L}_{:,S_{i-1}} \left[\mathbf{L}_{S_{i-1}}\right]^{-1} \mathbf{L}_{S_{i-1},:}) \\ &= \mathrm{trace}\left(\mathbf{L} - VV^\dagger{}_{:,S_{i-1}} \left[V_{S_{i-1},:} V^\dagger{}_{:,S_{i-1}}\right]^{-1} V_{S_{i-1},:} V^\dagger\right) \\ &= \mathrm{trace}\left(\mathbf{L}_{ii} - \underbrace{V_{S_{i-1},:}{}^\dagger \left[V_{S_{i-1},:} V_{S_{i-1},:}{}^\dagger\right]^{-1} V_{S_{i-1},:}}_{\Pi_{V_{S_{i-1},:}}} V^\dagger V\right) \\ &= \mathrm{trace}(\mathbf{L}) - \mathrm{trace}(\Pi_{V_{S_{i-1},:}} V^\dagger V),\end{aligned} \tag{3.24}$$

where $\Pi_{V_{S_{i-1},:}}$ denotes the orthogonal projection onto $\mathrm{Span}\{V_{s_1,:}, \dots, V_{s_i-1,:}\}$, the supspace spanned the feature vectors associated to $s_1, \dots, s_{i-1}$.

Then, summing (3.23) over the $k!$ permutations of $1, \dots, k$, yields the probability of drawing the **subset** $S = \{s_1, \dots, s_k\}$, namely

$$\mathbb{Q}[\{s_1, \dots, s_k\}] = \sum_{\sigma \in \mathfrak{S}_k} \mathbb{Q}[(s_{\sigma(1)}, \dots, s_{\sigma(k)})] = \det \mathbf{L}_S \underbrace{\sum_{\sigma \in \mathfrak{S}_k} \frac{1}{Z(s_{\sigma(1)}, \dots, s_{\sigma(k)})}}_{1/Z_S}. \tag{3.25}$$

3. For the chain rule (3.23) to be a valid procedure for sampling k-DPP$(\mathbf{L})$, we must be able to identify (3.22) and (3.25), i.e., $\mathbb{Q}[S] = \mathbb{P}[S]$ for all $|S| = k$, or equivalently $Z_S = e_k(L)$ for all $|S| = k$.

Important: A sufficient condition (very likely to be necessary) is that the joint distribution of $(s_1, \dots, s_k)$, generated by the chain rule mechanism (3.23) is exchangeable (invariant to permutations of the coordinates). In that case, the normalization in (3.23) would then be constant $Z(s_1, \dots, s_k) = Z$. So that Z_S would in fact play the role of the normalization constant of (3.25), since it would be constant as well and equal to $Z_S = \frac{Z}{k!}$. Finally, $Z_S = e_k(L)$ by identification of (3.22) and (3.25).

This is what we can prove in the particular case where L is an orthogonal projection matrix.

To do this, denote $r = \mathrm{rank}(\mathbf{L})$ and recall that in this case $\mathbf{L}$ satisfies $\mathbf{L}^2 = \mathbf{L}$ and $\mathbf{L}^\dagger = \mathbf{L}$, so that it can be factored as $\mathbf{L} = \Pi_{\mathbf{L}} = \mathbf{L}^\dagger\mathbf{L} = \mathbf{L}\mathbf{L}^\dagger$

Finally, we can plug $V = \mathbf{L}$ in (3.24) to obtain

$$\begin{aligned} Z_i(s_1, \ldots, s_{i-1}) &= \mathrm{trace}(\mathbf{L}) - \mathrm{trace}(\Pi_{\mathbf{L}_{S_{i-1},:}}\mathbf{L}^\dagger\mathbf{L}) \\ &= \mathrm{trace}(\Pi_{\mathbf{L}}) - \mathrm{trace}(\Pi_{\mathbf{L}_{S_{i-1},:}}\Pi_{\mathbf{L}}) \\ &= \mathrm{trace}(\Pi_{\mathbf{L}}) - \mathrm{trace}(\Pi_{\mathbf{L}_{S_{i-1},:}}) \\ &= \mathrm{rank}(\Pi_{\mathbf{L}}) - \mathrm{rank}(\Pi_{\mathbf{L}_{S_{i-1},:}}) \\ &= r - (i-1) := Z_i. \end{aligned}$$

Thus, the normalization $Z(s_1, \ldots, s_k)$ in (3.24) is constant as well equal to

$$Z(s_1, \ldots, s_k) = \prod_{i=1}^{k} Z_i = \prod_{i=1}^{k} r - (i-1) = \frac{r!}{(r-k)!} = k!\binom{r}{k} = k!e_k(\mathbf{L}) := Z,$$

where the last equality is a simple computation of the elementary symmetric polynomial

$$e_k(\mathbf{L}) = e_k(\gamma_{1:r} = 1, \gamma_{r+1:N} = 0) = \sum_{\substack{S \subset [N] \\ |S|=k}} \prod_{s \in S} \gamma_s = \binom{r}{k}$$

Important: This shows that, when $\mathbf{L}$ is an orthogonal projection matrix, the order the items $s_1, \ldots, s_r$ were selected by the chain rule (3.23) can be forgotten, so that $\{s_1, \ldots, s_r\}$ can be considered as valid sample of k-DPP($\mathbf{L}$).

```
# For our toy example, this sub-optimized implementation is enough
# to illustrate that the chain rule applied to sample k-DPP(L)
# draws s_1, ..., s_k sequentially, with joint probability
# P[(s_1, ..., s_k)] = det L_S / Z(s_1, ..., s_k)
#
# 1. is exchangeable when L is an orthogonal projection matrix
#    P[(s1, s2)] = P[(s_2, s_1)]
# 2. is a priori NOT exchangeable for a generic L >= 0
#    P[(s1, s2)] /= P[(s_2, s_1)]

import numpy as np
import scipy.linalg as LA
from itertools import combinations, permutations

k, N = 2, 4
potential_samples = list(combinations(range(N), k))

rank_L = 3

rng = np.random.RandomState(1)

eig_vecs, _ = LA.qr(rng.randn(N, rank_L), mode='economic')

for projection in [True, False]:

    eig_vals = 1.0 + (0.0 if projection else 2 * rng.rand(rank_L))
    L = (eig_vecs * eig_vals).dot(eig_vecs.T)
```

(continues on next page)

(continued from previous page)

```
    proba = np.zeros((N, N))
    Z_1 = np.trace(L)

    for S in potential_samples:

        for s in permutations(S):

            proba[s] = LA.det(L[np.ix_(s, s)])

            Z_2_s0 = np.trace(L - L[:, s[:1]].dot(LA.inv(L[np.ix_(s[:1], s[:1])])).
↪dot(L[s[:1], :]))

            proba[s] /= Z_1 * Z_2_s0

    print('L is {}projection'.format('' if projection else 'NOT '))

    print('P[s0, s1]', proba, sep='\n')
    print('P[s0]', proba.sum(axis=0), sep='\n')
    print('P[s1]', proba.sum(axis=1), sep='\n')

    print(proba.sum(), '\n' if projection else '')
```

```
L is projection
P[s0, s1]
[[0.         0.09085976 0.01298634 0.10338529]
 [0.09085976 0.         0.06328138 0.15368033]
 [0.01298634 0.06328138 0.         0.07580691]
 [0.10338529 0.15368033 0.07580691 0.        ]]
P[s0]
[0.20723139 0.30782147 0.15207463 0.33287252]
P[s1]
[0.20723139 0.30782147 0.15207463 0.33287252]
1.0000000000000002

L is NOT projection
P[s0, s1]
[[0.         0.09986722 0.01463696 0.08942385]
 [0.11660371 0.         0.08062998 0.20535251]
 [0.01222959 0.05769901 0.         0.04170435]
 [0.07995922 0.15726273 0.04463087 0.        ]]
P[s0]
[0.20879253 0.31482896 0.13989781 0.33648071]
P[s1]
[0.20392803 0.4025862  0.11163295 0.28185282]
1.0
```

3.1.4 MCMC sampling

Add/exchange/delete

[AGR16], [LJS16c] and [LJS16d] derived variants of a Metropolis sampler having for stationary distribution $\mathrm{DPP}(\mathbf{L})$ (3.2). The proposal mechanism works as follows.

At state $S \subset [N]$, propose S' different from S by at most 2 elements by picking

$$s \sim \mathcal{U}_S \quad \text{and} \quad t \sim \mathcal{U}_{[N]\setminus S}.$$

Then perform

Exchange

Pure exchange moves

$$S' \leftrightarrow S \setminus s \cup t.$$

Add-Delete

Pure addition/deletion moves

- Add $S' \leftrightarrow S \cup t$
- Delete $S' \leftrightarrow S \setminus s$

Add-Exchange-Delete

Mix of exchange and add-delete moves

- Delete $S' \leftrightarrow S \setminus s$
- Exchange $S' \leftrightarrow S \setminus s \cup t$
- Add $S' \leftrightarrow S \cup t$

Hint: Because moves are allowed between subsets having at most 2 different elements, transitions are very local inducing correlation, however *fast* mixing was proved.

```
import numpy as np
from dppy.finite_dpps import FiniteDPP

rng = np.random.RandomState(413121)

r, N = 4, 10

# Random feature vectors
Phi = rng.randn(r, N)
L = Phi.T.dot(Phi)
DPP = FiniteDPP('likelihood', **{'L': L})

DPP.sample_mcmc('AED', random_state=rng)  # AED, AD, E
print(DPP.list_of_samples)  # list of trajectories, here there's only one
```

```
[[[0, 2, 3, 6], [0, 2, 3, 6], [0, 2, 3, 6], [0, 2, 3, 6], [0, 2, 3, 6], [0, 2, 3, 6],␣
↪[0, 2, 6, 9], [0, 2, 6, 9], [2, 6, 9], [2, 6, 9]]]
```

See also:

- *sample_mcmc()*
- [AGR16], [LJS16c] and [LJS16d]
- *Exact samplers for DPPs*

Zonotope

[GBV17] target a *projection* $\mathrm{DPP}(\mathbf{K})$ with

$$\mathbf{K} = \Phi^\top [\Phi\Phi^\top]^{-1}\Phi,$$

where Φ, is the underlying $r \times N$ feature matrix satisfying $\mathrm{rank}(\Phi) = \mathrm{rank}(\mathbf{K}) = r$.

In this setting the *Number of points* is almost surely equal to r and we have

$$\mathbb{P}[\mathcal{X} = S] = \det \mathbf{K}_S 1_{|S|=r} = \frac{\det^2 \Phi_{:S}}{\det \Phi\Phi^\top} 1_{|S|=r} = \frac{\mathrm{Vol}^2\{\phi_s\}_{s\in S}}{\det \Phi\Phi^\top} 1_{|S|=r}. \tag{3.26}$$

The original finite ground set is embedded into a continuous domain called a zonotope. The hit-and-run procedure is used to move across this polytope and visit the different tiles. To recover the finite DPP samples one needs to identify the tile in which the successive points lie, this is done by solving linear programs (LPs).

Hint: Sampling from a *projection* DPP boils down to solving randomized linear programs (LPs).

Important: For its LPs solving needs DPPy uses the `cvxopt` library, but `cvxopt` is not installed by default when installing DPPy. Please refer to the installation instructions on GitHub for more details on how to install the necessary dependencies.

```
from numpy.random import RandomState
from dppy.finite_dpps import FiniteDPP

rng = RandomState(413121)

r, N = 4, 10
A = rng.randn(r, N)

DPP = FiniteDPP('correlation', projection=True, **{'A_zono': A})

DPP.sample_mcmc('zonotope', random_state=rng)
print(DPP.list_of_samples)  # list of trajectories, here there's only one
```

```
[[[2, 4, 5, 7], [2, 4, 5, 7], [2, 4, 5, 7], [1, 4, 5, 7], [1, 4, 5, 7], [1, 4, 5, 7],␣
↪[0, 4, 7, 8], [0, 2, 7, 9], [0, 2, 7, 9], [2, 4, 5, 7]]]
```

Note: On the one hand, the *Zonotope* perspective on sampling *projection* DPPs yields a better exploration of the state space. Using hit-and-run, moving to any other state is possible but at the cost of solving LPs at each step. On the other hand, the *Add/exchange/delete* view allows to perform cheap but local moves.

See also:

- *sample_mcmc()*
- [GBV17]

k-DPPs

To preserve the size k of the samples of k-DPP($\mathbf{L}$), only *Exchange* moves can be performed.

Caution: k must satisfy $k \leq \text{rank}(L)$

```
from numpy.random import RandomState
from dppy.finite_dpps import FiniteDPP

rng = RandomState(123)

r, N = 5, 10

# Random feature vectors
Phi = rng.randn(r, N)
L = Phi.T.dot(Phi)
DPP = FiniteDPP('likelihood', **{'L': L})

k = 3
DPP.sample_mcmc_k_dpp(size=k, random_state=rng)
print(DPP.list_of_samples)  # list of trajectories, here there's only one
```

```
[[[7, 2, 5], [7, 2, 5], [7, 2, 9], [7, 8, 9], [7, 8, 9], [7, 8, 2], [7, 8, 2], [6, 8,␣
↪2], [1, 8, 2], [1, 8, 2]]]
```

See also:

- *sample_mcmc_k_dpp()*
- [LJS16a] for a core-set perspective
- *Exact sampling of k-DPPs*

3.1.5 Approximate sampling

[LJS16b]

Todo: In a near future this section will include approximation of the kernel through random projections.

3.1.6 API

Implementation of *FiniteDPP* object which has 6 main methods:

- *sample_exact()*, see also *sampling DPPs exactly*
- *sample_exact_k_dpp()*, see also *sampling k-DPPs exactly*
- *sample_mcmc()*, see also *sampling DPPs with MCMC*
- *sample_mcmc_k_dpp()*, see also *sampling k-DPPs with MCMC*
- *compute_K()*, to compute the correlation K kernel from initial parametrization
- *compute_L()*, to compute the likelihood L kernel from initial parametrization

class dppy.finite_dpps.**FiniteDPP**(*kernel_type*, *projection=False*, ***params*)
Bases: object

Finite DPP object parametrized by

Parameters

- **kernel_type** (*string*) –
  - 'correlation' $\mathbf{K}$ kernel
  - 'likelihood' $\mathbf{L}$ kernel
- **projection** (bool, default False) – Indicate whether the provided kernel is of projection type. This may be useful when the *FiniteDPP* object is defined through its correlation kernel $\mathbf{K}$.
- **params** (*dict*) – Dictionary containing the parametrization of the underlying
  - correlation kernel
    - {'K': K}, with $0 \preceq \mathbf{K} \preceq I$
    - {'K_eig_dec': (eig_vals, eig_vecs)}, with $0 \leq eigvals \leq 1$
    - {'A_zono': A}, with $A(d \times N)$ and $\text{rank}(A) = d$
  - likelihood kernel
    - {'L': L}, with $\mathbf{L} \succeq 0$
    - {'L_eig_dec': (eig_vals, eig_vecs)}, with $eigvals \geq 0$
    - {'L_gram_factor': Phi}, with $\mathbf{L} = \Phi^\top \Phi$
    - {'L_eval_X_data': (eval_L, X_data)}, with $\mathbf{X}_{data}(N \times d)$ and $eval\_L$ a likelihood function such that $\mathbf{L} = eval\_L(\mathbf{X}_{data}, \{X_{data}\})$. For a full description of the requirements imposed on *eval_L*'s interface, see the documentation dppy.vfx_sampling.vfx_sampling_precompute_constants(). For an example, see the implementation of any of the kernels provided by scikit-learn (e.g. sklearn.gaussian_process.kernels.PairwiseKernel).

Caution: For now we only consider real valued matrices $\mathbf{K}, \mathbf{L}, A, \Phi, \mathbf{X}_{data}$.

See also:

- *Definition*

- *Exact sampling*

Todo: add `.kernel_rank` attribute

compute_K(*msg=False*)
Compute the correlation kernel $\mathbf{K}$ from the original parametrization of the *FiniteDPP* object.

The kernel is stored in the `K` attribute.

See also:

Relation between correlation and likelihood kernels

compute_L(*msg=False*)
Compute the likelihood kernel $\mathbf{L}$ from the original parametrization of the *FiniteDPP* object.

The kernel is stored in the `L` attribute.

See also:

Relation between correlation and likelihood kernels

flush_samples()
Empty the `list_of_samples` attribute.

info()
Display infos about the *FiniteDPP* object

plot_kernel(*kernel_type='correlation'*, *save_path=''*)
Display a heatmap of the kernel used to define the *FiniteDPP* object (correlation kernel $\mathbf{K}$ or likelihood kernel $\mathbf{L}$)

Parameters

- **kernel_type** (*str*) – Type of kernel (`'correlation'` or `'likelihood'`), default `'correlation'`
- **save_path** (*str*) – Path to save plot, if empty (default) the plot is not saved.

sample_exact(*mode='GS'*, ***params*)
Sample exactly from the corresponding *FiniteDPP* object.

Parameters

- **mode** (string, default `'GS'`) –

  – projection=True:

  * `'GS'` (default): Gram-Schmidt on the rows of $\mathbf{K}$.
  * `'Chol'` [Pou19] Algorithm 3
  * `'Schur'`: when DPP defined from correlation kernel `K`, use Schur complement to compute conditionals

  – projection=False:

  * `'GS'` (default): Gram-Schmidt on the rows of the eigenvectors of $\mathbf{K}$ selected in Phase 1.
  * `'GS_bis'`: Slight modification of `'GS'`
  * `'Chol'` [Pou19] Algorithm 1
  * `'KuTa12'`: Algorithm 1 in [KT12]

* `'vfx'`: the dpp-vfx rejection sampler in [DerezinskiCV19]
* `'alpha'`: the alpha-dpp rejection sampler in [CDerezinskiV20]

- **params** (*dict*) – Dictionary containing the parameters for exact samplers with keys
  - `'random_state'` (default None)
  - If `mode='vfx'`

    See `dpp_vfx_sampler()` for a full list of all parameters accepted by 'vfx' sampling. We report here the most impactful

    * `'rls_oversample_dppvfx'` (default 4.0) Oversampling parameter used to construct dppvfx's internal Nystrom approximation. This makes each rejection round slower and more memory intensive, but reduces variance and the number of rounds of rejections.
    * `'rls_oversample_bless'` (default 4.0) Oversampling parameter used during bless's internal Nystrom approximation. This makes the one-time preprocessing slower and more memory intensive, but reduces variance and the number of rounds of rejections

    Empirically, a small factor [2,10] seems to work for both parameters. It is suggested to start with a small number and increase if the algorithm fails to terminate.
  - If `mode='alpha'`

    See `alpha_k_dpp_sampler()` for a full list of all parameters accepted by 'alpha' sampling. We report here the most impactful

    * `'rls_oversample_alphadpp'` (default 4.0) Oversampling parameter used to construct alpha-dpp's internal Nystrom approximation. This makes each rejection round slower and more memory intensive, but reduces variance and the number of rounds of rejections.
    * `'rls_oversample_bless'` (default 4.0) Oversampling parameter used during bless's internal Nystrom approximation. This makes the one-time preprocessing slower and more memory intensive, but reduces variance and the number of rounds of rejections

    Empirically, a small factor [2,10] seems to work for both parameters. It is suggested to start with a small number and increase if the algorithm fails to terminate.

Returns Returns a sample from the corresponding *FiniteDPP* object. In any case, the sample is appended to the `list_of_samples` attribute as a list.

Return type list

Note: Each time you call this method, the sample is appended to the `list_of_samples` attribute as a list.

The `list_of_samples` attribute can be emptied using *flush_samples()*

Caution: The underlying kernel $\mathbf{K}$, resp. $\mathbf{L}$ must be real valued for now.

See also:

- *Exact sampling*

- *flush_samples()*
- *sample_mcmc()*

sample_exact_k_dpp (*size*, *mode='GS'*, ***params*)

Sample exactly from k-DPP. A priori the *FiniteDPP* object was instanciated by its likelihood $\mathbf{L}$ kernel so that

$$\mathbb{P}_{\text{k-DPP}}(\mathcal{X} = S) \propto \det \mathbf{L}_S \, 1_{|S|=k}$$

Parameters

- **size** (*int*) – size k of the k-DPP
- **mode** (string, default `'GS'`) –
  - **projection=True:**
    - `'GS'` (default): Gram-Schmidt on the rows of $\mathbf{K}$.
    - `'Schur'`: Use Schur complement to compute conditionals.
  - **projection=False:**
    - `'GS'` (default): Gram-Schmidt on the rows of the eigenvectors of $\mathbf{K}$ selected in Phase 1.
    - `'GS_bis'`: Slight modification of `'GS'`
    - `'KuTa12'`: Algorithm 1 in [KT12]
    - `'vfx'`: the dpp-vfx rejection sampler in [DerezinskiCV19]
    - `'alpha'`: the alpha-dpp rejection sampler in [CDerezinskiV20]
- **params** (*dict*) – Dictionary containing the parameters for exact samplers with keys

  `'random_state'` (default None)

  - If `mode='vfx'`

    See `k_dpp_vfx_sampler()` for a full list of all parameters accepted by 'vfx' sampling. We report here the most impactful

    - `'rls_oversample_dppvfx'` (default 4.0) Oversampling parameter used to construct dppvfx's internal Nystrom approximation. This makes each rejection round slower and more memory intensive, but reduces variance and the number of rounds of rejections.
    - `'rls_oversample_bless'` (default 4.0) Oversampling parameter used during bless's internal Nystrom approximation. This makes the one-time pre-processing slower and more memory intensive, but reduces variance and the number of rounds of rejections

    Empirically, a small factor [2,10] seems to work for both parameters. It is suggested to start with a small number and increase if the algorithm fails to terminate.

  - **If `mode='alpha'`** See `alpha_k_dpp_sampler()` for a full list of all parameters accepted by 'alpha' sampling. We report here the most impactful
    - `'rls_oversample_alphadpp'` (default 4.0) Oversampling parameter used to construct alpha-dpp's internal Nystrom approximation. This

makes each rejection round slower and more memory intensive, but reduces variance and the number of rounds of rejections.

* `'rls_oversample_bless'` (default 4.0) Oversampling parameter used during bless's internal Nystrom approximation. This makes the one-time pre-processing slower and more memory intensive, but reduces variance and the number of rounds of rejections

* `'early_stop'` (default False) Wheter to return as soon as a k-DPP sample is drawn, or to continue with alpha-dpp internal binary search to make subsequent sampling faster.

Empirically, a small factor [2,10] seems to work for both parameters. It is suggested to start with a small number and increase if the algorithm fails to terminate.

Returns

A sample from the corresponding k-DPP.

In any case, the sample is appended to the `list_of_samples` attribute as a list.

Return type list

Note: Each time you call this method, the sample is appended to the `list_of_samples` attribute as a list.

The `list_of_samples` attribute can be emptied using *`flush_samples()`*

Caution: The underlying kernel $\mathbf{K}$, resp. $\mathbf{L}$ must be real valued for now.

See also:

- *`sample_exact()`*
- *`sample_mcmc_k_dpp()`*

`sample_mcmc`(*mode*, ***params*)

Run a MCMC with stationary distribution the corresponding *`FiniteDPP`* object.

Parameters

- **`mode`** (*`string`*) –
  - `'AED'` Add-Exchange-Delete
  - `'AD'` Add-Delete
  - `'E'` Exchange
  - `'zonotope'` Zonotope sampling
- **`params`** (*`dict`*) – Dictionary containing the parameters for MCMC samplers with keys

  `'random_state'` (default None)

  - If `mode='AED','AD','E'`
    * `'s_init'` (default None) Starting state of the Markov chain

* `'nb_iter'` (default 10) Number of iterations of the chain
* `'T_max'` (default None) Time horizon
* `'size'` (default None) Size of the initial sample for `mode='AD'/'E'`
  - $\mathrm{rank}(\mathbf{K}) = \mathrm{trace}(\mathbf{K})$ for projection $\mathbf{K}$ (correlation) kernel and `mode='E'`

– If `mode='zonotope'`:

* `'lin_obj'` linear objective in main optimization problem (default np.random.randn(N))
* `'x_0'` initial point in zonotope (default A*u, u~U[0,1]^n)
* `'nb_iter'` (default 10) Number of iterations of the chain
* `'T_max'` (default None) Time horizon

Returns

The last sample of the trajectory of Markov chain.

In any case, the full trajectory of the Markov chain, made of `params['nb_iter']` samples, is appended to the `list_of_samples` attribute as a list of lists.

Return type list

Note: Each time you call this method, the full trajectory of the Markov chain, made of `params['nb_iter']` samples, is appended to the `list_of_samples` attribute as a list of lists.

The `list_of_samples` attribute can be emptied using *`flush_samples()`*

See also:

- *MCMC sampling*
- *`sample_exact()`*
- *`flush_samples()`*

`sample_mcmc_k_dpp`(*size*, *mode='E'*, ***params*)

Calls *`sample_mcmc()`* with `mode='E'` and `params['size'] = size`

See also:

- *MCMC sampling*
- *`sample_mcmc()`*
- *`sample_exact_k_dpp()`*
- *`flush_samples()`*

3.2 Continuous DPPs

3.2.1 Definition

Point Process

Let $\mathbb{X} = \mathbb{R}^d, \mathbb{C}^d$ or $\mathbb{S}^{d-1}$ be the ambient space, we endow it with the corresponding Borel σ-algebra $\mathcal{B}(\mathbb{X})$ together with a reference measure μ.

For our purpose, we consider point processes as locally finite random subsets $\mathcal{X} \subset \mathbb{X}$ i.e.

$$\forall C \subset \mathbb{X} \text{ compact}, \quad \#(\mathcal{X} \cap C) < \infty.$$

Hint: A point process is a random subset of points $\mathcal{X} \triangleq \{X_1, \dots, X_N\} \subset \mathbb{X}$ with N being random.

See also:

More formal definitions can be found in [MollerW04] Section 2 and [Joh06] Section 2 and bibliography therein.

To understand the interaction between the points of a point process, one focuses on the interaction of each cloud of k points (for all k). The corresponding k-correlation functions characterize the underlying point process.

Correlation functions

For $k \geq 0$, the k-correlation function ρ_k is defined by:

$\forall f : \mathbb{X}^k \to \mathbb{C}$ bounded measurable

$$\mathbb{E}\left[\sum_{\substack{(X_1,\dots,X_k) \\ X_1 \neq \cdots \neq X_k \in \mathcal{X}}} f(X_1, \dots, X_k) \right] = \int_{\mathbb{X}^k} f(x_1, \dots, x_k) \rho_k(x_1, \dots, x_k) \prod_{i=1}^{k} \mu(dx_i).$$

Hint:

The k-correlation function does not always exists, but but when they do, they have a meaningful interpretation.

$$\text{"}\rho_k(x_1, \dots, x_k)\mu(dx_1), \dots, \mu(dx_N) = \mathbb{P}\left[\begin{array}{c} \text{there is 1 point in each} \\ B(x_1, dx_1), \dots, B(x_n, dx_n) \end{array} \right]\text{"},$$

where $B(x, dx)$ denotes the ball centered at x with radius dx.

A Determinant Point Process (DPP) is a point process on $(\mathbb{X}, \mathcal{B}(\mathbb{X}), \mu)$ parametrized by a kernel K associated to the reference measure μ. The k-correlation functions read

$$\forall k \geq 1, \quad \rho_k(x_1, \dots, x_k) = \det[K(x_i, x_j)]_{i,j=1}^{k}.$$

See also:

[Mac75] [Sos00] [Joh06] [HKPVirag06]

Existence

One can view K as an integral operator on $L^2(\mu)$

$$\forall x \in \mathbb{X}, Kf(x) = \int_{\mathbb{X}} K(x,y)f(y)\mu(dy).$$

To access spectral properties of the kernel, it is common practice to assume K

1. Hilbert Schmidt

$$\iint_{\mathbb{X}\times\mathbb{X}} |K(x,y)|^2 \mu(dx)\mu(dy) < \infty,$$

2. Self-adjoint equiv. Hermitian

$$K(x,y) = \overline{K(y,x)},$$

3. Locally trace class

$$\forall B \subset \mathbb{X} \text{ compact}, \quad \int_B K(x,x)\mu(dx) < \infty.$$

Hint:

- 1. implies K to be a continuous compact operator.
- 2. with 1. allows to apply the spectral theorem, providing

$$K(x,y) = \sum_{n=0}^{\infty} \lambda_n \phi_n(x)\phi_n(y), \quad \text{where } K\phi_n = \lambda_n \phi_n.$$

- 3. makes sure there is no accumulation of points: $|\mathcal{X} \cap B| = \int_B K(x,x)\mu(dx) \leq \infty$, see also *Number of points*

Warning: These are only sufficient conditions, there indeed exist DPPs with non symmetric kernels, see, e.g., *Carries process*.

Important: Under assumptions 1, 2, and 3

$$\text{DPP}(K) \text{ exists} \iff 0 \leq \lambda_n \leq 1, \quad \forall n \in \mathbb{N}$$

See also:

- Remarks 1-2 and Theorem 3 [Sos00]
- Theorem 22 [HKPVirag06]

Projection DPPs

$\mathrm{DPP}(K)$ is said to be a **projection** DPP with reference measure μ when $K : \mathbb{X} \times \mathbb{X} \to \mathbb{C}$ is a orthogonal projection kernel, that is

$$K(x,y) = \overline{K(y,x)} \quad \text{and} \quad \int_{\mathbb{X}} K(x,z)K(z,y)\mu(dz) = K(x,y)$$

Construction

A canonical way to construct DPPs generating configurations of at most N points is the following.

Consider N orthonormal functions $\phi_0, ..., \phi_{N1} \in L^2(\mu)$

$$\int \phi_k(x)\phi_l(x)\mu(dx) = \delta_{kl},$$

and attach $[0,1]$-valued coefficients λ_n such that

$$K(x,y) = \sum_{n=0}^{N-1} \lambda_n \phi_n(x)\phi_n(y).$$

The special case where $\lambda_0 = \cdots = \lambda_{N-1} = 1$ corresponds to the construction of a projection DPP with N points.

See also:

- *Number of points*
- Lemma 21 [HKPVirag06]
- Proposition 2.11 [Joh06] biorthogonal families

3.2.2 Properties

Generic DPPs as mixtures of projection DPPs

Projection DPPs are the building blocks of the model in the sense that generic DPPs are mixtures of *projection* DPPs.

Consider $\mathcal{X} \sim \mathrm{DPP}(K)$ and write the spectral decomposition of the corresponding kernel as

$$K = \sum_{n=1}^{\infty} \lambda_n \phi(x)\overline{\phi(y)}.$$

Then, denote $\mathcal{X}^B \sim \mathrm{DPP}(K^B)$ with

$$K = \sum_{n=1}^{\infty} B_n \phi(x)\overline{\phi(y)}, \quad \text{where} \quad B_n \sim \mathcal{B}er(\lambda_n) \text{ are independent,}$$

$\mathcal{X}^B$ is obtained by first sampling $B_1, \ldots, B_N$ independently and then sampling conditionally from $\mathrm{DPP}(K^B)$, the DPP with orthogonal projection kernel K^B.

Finally, we have $\mathcal{X} \overset{d}{=} \mathcal{X}^B$.

See also:

- Theorem 7 in [HKPVirag06]
- Finite case of *Generic DPPs as mixtures of projection DPPs*
- *Sampling*

Linear statistics

Expectation

$$\mathbb{E}\left[\sum_{X\in\mathcal{X}} f(X)\right] = \int f(x)K(x,x)\mu(dx) = \mathrm{trace}(Kf) = \mathrm{trace}(fK).$$

Variance

$$\begin{aligned}
\mathbb{V}\mathrm{ar}\left[\sum_{X\in\mathcal{X}} f(X)\right] &= \mathbb{E}\left[\sum_{X\neq Y\in\mathcal{X}} f(X)f(Y) + \sum_{X\in\mathcal{X}} f(X)^2\right] - \mathbb{E}\left[\sum_{X\in\mathcal{X}} f(X)\right]^2 \\
&= \iint f(x)f(y)[K(x,x)K(y,y) - K(x,y)K(y,x)]\mu(dx)\mu(dy) \\
&\quad + \int f(x)^2 K(x,x)\mu(dx) - \left[\int f(x)K(x,x)\mu(dx)\right]^2 \\
&= \int f(x)^2 K(x,x)\mu(dx) - \iint f(x)f(y)K(x,y)K(y,x)\mu(dx)\mu(dy) \\
&= \mathrm{trace}(f^2K) - \mathrm{trace}(fKfK).
\end{aligned}$$

a. Hermitian kernel i.e. $K(x,y) = \overline{K(y,x)}$

$$\mathbb{V}\mathrm{ar}\left[\sum_{X\in\mathcal{X}} f(X)\right] = \int f(x)^2 K(x,x)\mu(dx) - \iint f(x)f(y)|K(x,y)|^2\mu(dx)\mu(dy).$$

b. Orthogonal projection case i.e. $K^2 = K = K^*$

Using $K(x,x) = \int K(x,y)K(y,x)\mu(dy) = \int |K(x,y)|^2\mu(dy)$,

$$\mathbb{V}\mathrm{ar}\left[\sum_{X\in\mathcal{X}} f(X)\right] = \frac{1}{2}\iint [f(x)-f(y)]^2|K(x,y)|^2\mu(dy)\mu(dx).$$

Number of points

For projection DPPs, i.e., when K is the kernel associated to an orthogonal projector, one can show that $|\mathcal{X}| = \mathrm{rank}(K) = \mathrm{Trace}(K)$ almost surely (see, e.g., [HKPVirag06] Lemma 17).

In the general case, based on the fact that *generic DPPs are mixtures of projection DPPs*, we have

$$|\mathcal{X}| = \sum_{i=1}^{\infty} \mathcal{B}\mathrm{er}(\lambda_i).$$

Note:

- For any Borel set B, instantiating $f = 1_B$ yields nice expressions for the expectation and variance of the number of points falling in B.

See also:

Number of points in the finite case

Thinning

Important: The class of DPPs is closed under independent thinning.

Let $\lambda > 1$. The configuration of points $\mathcal{X}^\lambda$ obtained after subsampling the points of a configuration $\mathcal{X} \sim \mathrm{DPP}(K)$ with i.i.d. $\mathcal{B}\mathrm{er}\left(\frac{1}{\lambda}\right)$ is still a DPP with kernel $\frac{1}{\lambda}K$. To see this, let's compute the correlation functions of the thinned process

$$\begin{aligned}
\mathbb{E}\left[\sum_{\substack{(x_1,\dots,x_k)\\ x_i\neq x_j\in\mathcal{X}^\lambda}} f(x_1,\dots,x_k)\right] &= \mathbb{E}\left[\mathbb{E}\left[\sum_{\substack{(x_1,\dots,x_k)\\ x_i\neq x_j\in\mathcal{X}}} f(x_1,\dots,x_k)\prod_{i=1}^k 1_{\{x_i\in\mathcal{X}^\lambda\}}\middle|\mathcal{X}\right]\right] \\
&= \mathbb{E}\left[\sum_{\substack{(x_1,\dots,x_k)\\ x_i\neq x_j\in\mathcal{X}}} f(x_1,\dots,x_k)\mathbb{E}\left[\prod_{i=1}^k B_i\middle|\mathcal{X}\right]\right] \\
&= \mathbb{E}\left[\sum_{\substack{(x_1,\dots,x_k)\\ x_i\neq x_j\in\mathcal{X}}} f(x_1,\dots,x_k)\frac{1}{\lambda^k}\right] \\
&= \int f(x_1,\dots,x_k)\det\left[\frac{1}{\lambda}K(x_i,x_j)\right]_{1\le i,j\le k}\mu^{\otimes k}(dx).
\end{aligned}$$

3.2.3 Sampling

In contrast to the *finite case* where the ML community has put efforts in improving the efficiency and tractability of the sampling routine, much less has been done in the continuous setting.

Exact sampling

In this section, we describe the main techniques for sampling exactly continuous DPPs.

As for *finite DPPs* the most prominent one relies on the fact that *generic DPPs are mixtures of projection DPPs*.

Projection DPPs: the chain rule

Let's focus on sampling from projection $\mathrm{DPP}(K)$ with a real-valued orthogonal projection kernel $K : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ and reference measure μ, that is

$$K(x, y) = K(y, x) \quad \text{and} \quad \int_{\mathbb{X}} K(x, z)K(z, y)\mu(dz) = K(x, y)$$

In this setting, recall that the *number of points* is μ-almost surely equal to $r = \mathrm{rank}(K)$.

To generate a valid sample $X = \{x_1, \dots, x_r\} \sim \mathrm{DPP}(K)$, [HKPVirag06] Proposition 19 showed that it is sufficient to apply the chain rule to sample $(x_1, \dots, x_r)$ with joint distribution

$$\mathbb{P}[(x_1, \dots, x_r)] = \frac{1}{r!} \det[K(x_p, x_q)]_{p,q=1}^{r} \mu^{\otimes r}(dx_{1:r}) \tag{3.27}$$

and forget the order the points were selected.

The original projection DPP sampler of [HKPVirag06] Algorithm 18, was given in an abstract form, which can be implemented using the following strategy. Write the determinant in (3.27) as a telescopic product of ratios of determinants and use Schur complements to get

$$\begin{aligned} \mathbb{P}[(x_1, \dots, x_r)] &= \frac{K(x_1, x_1)}{r}\mu(dx_1) \prod_{i=2}^{r} \frac{1}{r-(i-1)} \frac{\det \mathbf{K}_i}{\det \mathbf{K}_{i-1}} \mu(dx_i) \\ &= \frac{K(x_1, x_1)}{r}\mu(dx_1) \prod_{i=2}^{r} \frac{K(x_i, x_i) - \mathbf{K}_{i-1}(x_i)^\top \mathbf{K}_{i-1}^{-1} \mathbf{K}_{i-1}(x_i)}{r-(i-1)} \mu(dx_i), \end{aligned} \tag{3.28}$$

where $\mathbf{K}_{i-1} = [K(x_p, x_q)]_{p,q=1}^{i-1}$ and $\mathbf{K}_{i-1}(x) = (K(x, x_1), \dots, K(x, x_{i-1}))^\top$.

Important:

a) The expression (3.27) indeed defines a probability distribution, with normalization constant $r!$. In particular this distribution is exchangeable, i.e., invariant by permutation of the coordinates.

b) The successive ratios that appear in (3.28) are the normalized conditional densities (w.r.t. μ) that drive the chain rule. The associated normalizing constants $r - (i - 1)$ are independent of the previous points.

c) Sampling **projection** DPPs does not require the eigendecomposition of the kernel!

Hint: MLers will recognize (3.28) as the incremental posterior variance of a noise-free Gaussian Process (GP) model with kernel K, see [RW06] Equation 2.26.

Caution: The connexion between the chain rule (3.28) and Gaussian Processes is valid in the case where the GP kernel is an **orthogonal projection kernel**, see also *Caution*.

Geometrical interpretation

When the eigendecomposition of the kernel is available, the chain rule can be interpreted and implemented from a geometrical perspective, see, e.g., [LMollerR12] Algorithm 1.

Writing the Gram formulation of the kernel as

$$K(x,y) = \sum_{i=1}^{r} \phi_i(x)\phi_i(y) = \Phi(x)^\top \Phi(y),$$

where $\Phi(x) \triangleq (\phi_1(x), \dots, \phi_r(x))$ denotes the *feature vector* associated to $x \in \mathbb{X}$.

The joint distribution (3.27) reads

$$\begin{aligned} \mathbb{P}[(x_1, \dots, x_r)] &= \frac{1}{r!} \det[\Phi(x_p)^\top \Phi(x_q))]_{p,q=1}^{r} \mu^{\otimes r}(dx_{1:r}) \\ &= \frac{1}{r!} \operatorname{Volume}^2\{\Phi(x_1), \dots, \Phi(x_r)\} \mu^{\otimes r}(dx_{1:r}), \end{aligned} \tag{3.29}$$

Hint: The joint distribution (3.29) characterizes the fact that **projection** $\mathrm{DPP}(K)$ favor sets of $r = \operatorname{rank}(\mathbf{K})$ points $(x_1, \dots, x_r)$ whose feature vectors $\Phi(x_1), \dots \Phi(x_r)$ span a large volume. This is another way of understanding the repulsive/diversity feature of DPPs.

Then, the previous telescopic product of ratios of determinants in (3.28) can be understood as the base $\times$ height formula applied to compute $\operatorname{Volume}^2\{\Phi(x_1), \dots, \Phi(x_r)\}$, so that

$$\begin{aligned} \mathbb{P}[(x_1, \dots, x_r)] &= \frac{\langle \Phi(x_1)^\top \Phi(x_1) \rangle}{r} \mu(dx_1) \prod_{i=2}^{r} \frac{1}{r-(i-1)} \frac{\det \mathbf{K}_i}{\det \mathbf{K}_{i-1}} \mu(dx_i) \\ &= \frac{\|\Phi(x_1)\|^2}{r} \mu(dx_1) \prod_{i=2}^{r} \frac{\operatorname{distance}^2\left(\Phi(x_i), \operatorname{Span}\{\Phi(x_1), \dots, \Phi(x_{i-1})\}\right)}{r-(i-1)} \mu(dx_i), \end{aligned} \tag{3.30}$$

where $\mathbf{K}_{i-1} = \left[\langle \Phi(x_p)^\top \Phi(x_q) \rangle\right]_{p,q=1}^{i-1}$.

Hint: The overall procedure is akin to a sequential Gram-Schmidt orthogonalization of $\Phi(x_1), \dots, \Phi(x_N)$.

Attention: In contrast to the *finite case* where the conditionals are simply probability vectors, the chain rule formulations (3.28) and (3.30) require sampling from a continuous distribution. This can be done using a rejection sampling mechanism but finding a good proposal density with tight rejection bounds is a challenging problem [LMollerR12] Section 2.4. But it is achievable in some specific cases, see, e.g., *Multivariate Jacobi Ensemble*.

See also:

- Algorithm 18 [HKPVirag06] for the original abstract **projection** DPP sampler
- *Projection DPPs: the chain rule* in the finite case
- Some *Orthogonal Polynomial Ensembles* (specific instances of projection DPPs) can be *sampled* in $\mathcal{O}(r^2)$ by computing the eigenvalues of properly randomised tridiagonal matrices.
- The *multivariate Jacobi ensemble* whose `sample()` method relies on the chain rule described by (3.30).

Generic DPPs: the spectral method

The procedure stems from the fact that *generic DPPs are mixtures of projection DPPs*, suggesting the following two steps algorithm. Given the spectral decomposition of the kernel

$$K(x,y) = \sum_{i=1}^{\infty} \lambda_i \phi_i(x)\overline{\phi_i(y)}, \tag{3.31}$$

Step 1. Draw $B_i \sim \mathcal{B}\mathrm{er}(\lambda_i)$ independently and note $\{i_1, \dots, i_N\} = \{i\ ;\ B_i = 1\}$,

Step 2. Sample from the *projection* DPP with kernel $\tilde{K}(x,y) = \sum_{n=1}^{N} \phi_{i_n}(x)\overline{\phi_{i_n}(y)}$.

Important:

- Step *1.* selects a component of the mixture, see [LMollerR12] Section 2.4.1
- Step *2.* generates a sample from the projection $\mathrm{DPP}(\tilde{K})$, cf. *previous section*.

Attention: Contrary to projection DPPs, the general case requires the eigendecomposition of the kernel (3.31).

See also:

Spectral method for sampling finite DPPs.

Perfect sampling

[DFL13] uses Coupling From The Past (CFTP).

Approximate sampling

See also:

- Approximation of $K(x,y) = K(x-y)$ by Fourier series [LMollerR12] Section 4

3.2.4 β-Ensembles

Definition

Let $\beta > 0$, the joint distribution of the β-Ensemble associated to the reference measure μ writes

$$(x_1, \dots, x_N) \sim \frac{1}{Z_{N,\beta}} \left|\Delta(x_1, \dots, x_N)\right|^{\beta} \prod_{i=1}^{N} \mu(dx_i). \tag{3.32}$$

Hint:

- $|\Delta(x_1, \dots, x_N)| = \prod_{i<j} |x_i - x_j|$ is the absolute value of the determinant of the Vandermonde matrix,

$$\Delta(x_1, \dots, x_N) = \det \begin{bmatrix} 1 & \dots & 1 \\ x_1 & \dots & x_N \\ \vdots & & \vdots \\ x_1^{N-1} & & x_N^{N-1} \end{bmatrix}, \tag{3.33}$$

encoding the repulsive interaction. The *closer* the points are the lower the density.

- β is the inverse temperature parameter quantifying the strength of the repulsion between the points.

Important: For Gaussian, Gamma and Beta reference measures, the $\beta = 1, 2$ and 4 cases received a very special attention in the random matrix literature, e.g. [DE02].

The associated ensembles actually correspond to the eigenvalues of random matrices whose distribution is invariant to the action of the orthogonal ($\beta = 1$), unitary ($\beta = 2$) and symplectic ($\beta = 4$) group respectively.

μ	$\mathcal{N}$	Γ	Beta
Ensemble name	Hermite	Laguerre	Jacobi
support	$\mathbb{R}$	$\mathbb{R}^+$	$[0, 1]$

Note: The study of the distribution of the eigenvalues of random orthogonal, unitary and symplectic matrices lying on the unit circle is also very thorough [KN04].

Orthogonal Polynomial Ensembles

The case $\beta = 2$ corresponds a specific type of *projection* DPPs also called Orthogonal Polynomial Ensembles (OPEs) [Konig04] with associated kernel

$$K_N(x, y) = \sum_{n=0}^{N-1} P_n(x) P_n(y),$$

where (P_n) are the orthonormal polynomials w.r.t. μ i.e. $\deg(P_n) = n$ and $\langle P_k, P_l \rangle_{L^2(\mu)} = \delta_{kl}$.

Note: OPEs (with N points) correspond to *projection* DPPs onto $\mathrm{Span}\{P_n\}_{n=0}^{N-1} = \mathbb{R}^{N-1}[X]$

Hint: First, linear combinations of the rows of $\Delta(x_1, \dots, x_N)$ allow to make appear the orthonormal polynomials (P_n) so that

$$|\Delta(x_1, \dots, x_N)| \propto \begin{vmatrix} P_0(x_1) & \dots & P_0(x_N) \\ P_1(x_1) & \dots & P_1(x_N) \\ \vdots & & \vdots \\ P_{N-1}(x_1) & & P_{N-1}(x_N) \end{vmatrix}.$$

Then,

$$|\Delta|^2 = |\Delta^\top \Delta| \propto \det \left[K_N(x_i, x_j) \right]_{i,j=1}^N.$$

Finally, the joint distribution of $(x_1, \dots, x_N)$ reads

$$(x_1, \dots, x_N) \sim \frac{1}{N!} \det \left[K_N(x_i, x_j)\right]_{i,j=1}^N \prod_{i=1}^N \mu(dx_i). \tag{3.34}$$

See also:

[Konig04], [Joh06]

Sampling

Full matrix models

For specific reference measures the $\beta = 1, 2, 4$ cases are very singular in the sense that the corresponding ensembles coincide with the eigenvalues of random matrices.

This is a highway for sampling exactly such ensembles in $\mathcal{O}(N^3)$!

Hermite Ensemble

Take for reference measure $\mu = \mathcal{N}(0, 2)$, the pdf of the corresponding β-Ensemble reads

$$(x_1, \dots, x_N) \sim |\Delta(x_1, \dots, x_N)|^\beta \prod_{i=1}^N e^{-\frac{1}{2}\frac{x_i^2}{2}}\, dx_i,$$

$where from the definition in : eq : `eq : abs_v and er monde_d et` we have : math : ` |\Delta(x_1, \dots, x_N)| = \prod_{i<j} |x_i - x_j|`.

Hint: The Hermite ensemble (whose name comes from the fact that Hermite polynomials are orthogonal w.r.t the Gaussian distribution) refers to the eigenvalue distribution of random matrices formed by i.i.d. Gaussian vectors.

- $\beta = 1$

$$X \sim \mathcal{N}_{N,N}(0, 1) \qquad A = \frac{X + X^\top}{\sqrt{2}}.$$

- $\beta = 2$

$$X \sim \mathcal{N}_{N,N}(0, 1) + i\, \mathcal{N}_{N,N}(0, 1) \qquad A = \frac{X + X^\dagger}{\sqrt{2}}.$$

- $\beta = 4$

$$\begin{cases} X \sim \mathcal{N}_{N,M}(0, 1) + i\, \mathcal{N}_{N,M}(0, 1) \\ Y \sim \mathcal{N}_{N,M}(0, 1) + i\, \mathcal{N}_{N,M}(0, 1) \end{cases} \qquad A = \begin{bmatrix} X & Y \\ -Y^* & X^* \end{bmatrix} \quad A = \frac{X + X^\dagger}{\sqrt{2}}.$$

Normalization $\sqrt{\beta N}$ to concentrate as the semi-circle law.

$$\frac{\sqrt{4 - x^2}}{2\pi} 1_{[-2,2]} dx.$$

```
from dppy.beta_ensembles import HermiteEnsemble


hermite = HermiteEnsemble(beta=4)  # beta in {0,1,2,4}, default beta=2
hermite.sample_full_model(size_N=500)
# hermite.plot(normalization=True)
hermite.hist(normalization=True)

# To compare with the sampling speed of the tridiagonal model simply use
# hermite.sample_banded_model(size_N=500)
```

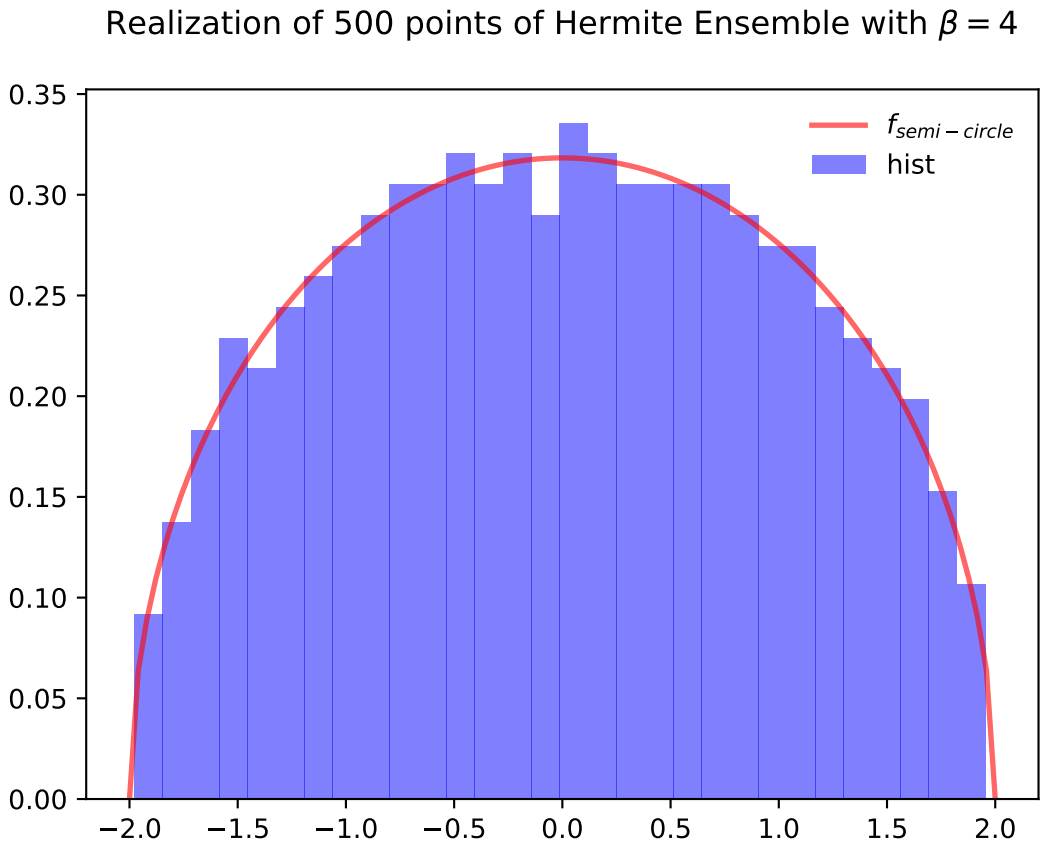


Fig. 3.5: Full matrix model for the Hermite ensemble

See also:

- *Banded matrix model* for Hermite ensemble
- `HermiteEnsemble` in API

Laguerre Ensemble

Take for reference measure $\mu = \Gamma\left(\frac{\beta}{2}(M-N+1), 2\right) = \chi^2_{\beta(M-N+1)}$, the pdf of the corresponding β-Ensemble reads

$$(x_1, \dots, x_N) \sim |\Delta(x_1, \dots, x_N)|^\beta \prod_{i=1}^N x_i^{\frac{\beta}{2}(M-N+1)-1} e^{-\frac{1}{2}x_i}\, dx_i,$$

$where from the definition in : eq : `eq : abs_v and er monde_d et` we have : math : ` |\Delta(x_1, \dots, x_N)| = \prod_{i<j} |x_i - x_j|`.

Hint: The Laguerre ensemble (whose name comes from the fact that Laguerre polynomials are orthogonal w.r.t the Gamma distribution) refers to the eigenvalue distribution of empirical covariance matrices of i.i.d. Gaussian vectors.

- $\beta = 1$

$$X \sim \mathcal{N}_{N,M}(0,1) \qquad A = XX^\top.$$

- $\beta = 2$

$$X \sim \mathcal{N}_{N,M}(0,1) + i\,\mathcal{N}_{N,M}(0,1) \qquad A = XX^\dagger.$$

- $\beta = 4$

$$\begin{cases} X \sim \mathcal{N}_{N,M}(0,1) + i\,\mathcal{N}_{N,M}(0,1) \\ Y \sim \mathcal{N}_{N,M}(0,1) + i\,\mathcal{N}_{N,M}(0,1) \end{cases} \qquad A = \begin{bmatrix} X & Y \\ -Y^* & X^* \end{bmatrix} \quad A = AA^\dagger.$$

Normalization βM to concentrate as Marcenko-Pastur law.

$$\frac{1}{2\pi} \frac{\sqrt{(\lambda_+ - x)(x - \lambda_-)}}{cx} 1_{[\lambda_-, \lambda_+]} dx,$$

where

$$c = \frac{M}{N} \quad \text{and} \quad \lambda_\pm = (1 \pm \sqrt{c})^2.$$

```
from dppy.beta_ensembles import LaguerreEnsemble


laguerre = LaguerreEnsemble(beta=1)  # beta in {0,1,2,4}, default beta=2
laguerre.sample_full_model(size_N=500, size_M=800)  # M >= N
# laguerre.plot(normalization=True)
laguerre.hist(normalization=True)

# To compare with the sampling speed of the tridiagonal model simply use
# laguerre.sample_banded_model(size_N=500, size_M=800)
```

See also:

- *Banded matrix model* for Laguerre ensemble
- `LaguerreEnsemble` in API

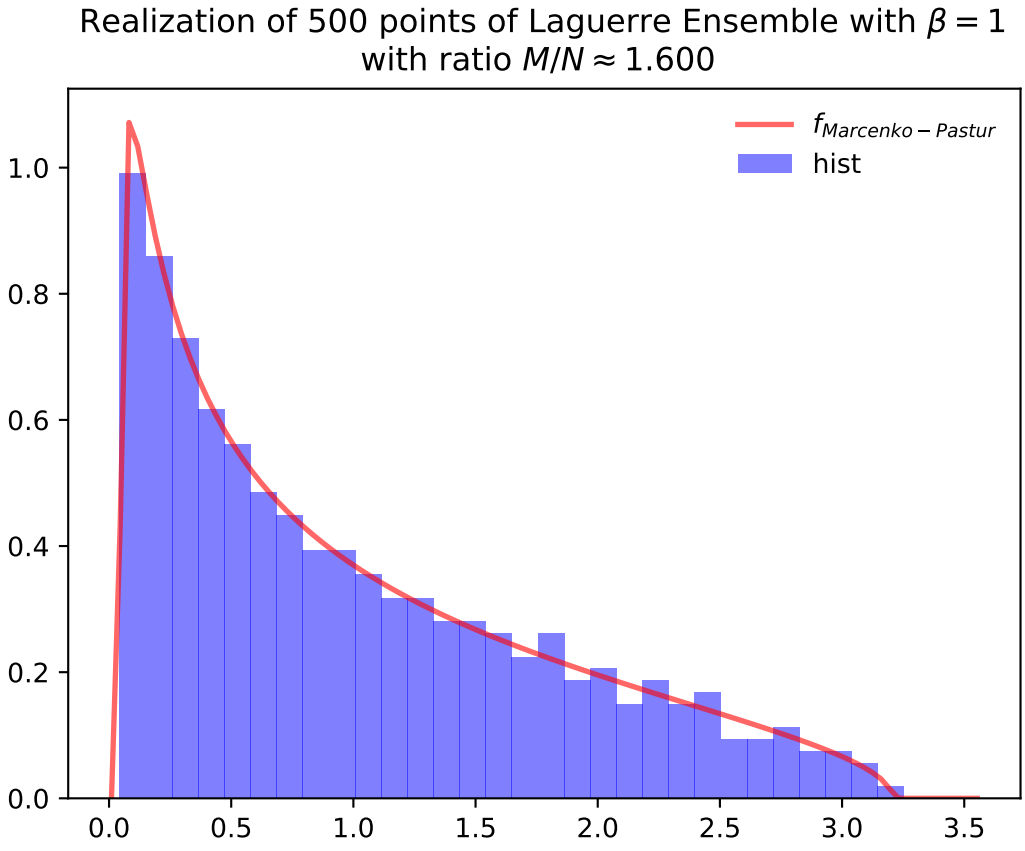


Fig. 3.6: Full matrix model for the Laguerre ensemble

Jacobi Ensemble

Take for reference measure $\mu = \text{Beta}\left(\frac{\beta}{2}(M_1 - N + 1), \frac{\beta}{2}(M_2 - N + 1)\right)$, the pdf of the corresponding β-Ensemble reads

$$(x_1, \ldots, x_N) \sim |\Delta(x_1, \ldots, x_N)|^\beta \prod_{i=1}^{N} x_i^{\frac{\beta}{2}(M_1 - N + 1) - 1} (1 - x_i)^{\frac{\beta}{2}(M_2 - N + 1) - 1} \, dx_i,$$

$where from the definition in : eq : `eq : abs_v and er monde_d et` we have : math : ` |\Delta(x_1, \ldots, x_N)| = \prod_{i<j} |x_i - x_j|`.

Hint: The Jacobi ensemble (whose name comes from the fact that Jacobi polynomials are orthogonal w.r.t the Beta distribution) is associated with the multivariate analysis of variance (MANOVA) model.

- $\beta = 1$

$$\begin{cases} X \sim \mathcal{N}_{N,M_1}(0,1) \\ Y \sim \mathcal{N}_{N,M_2}(0,1) \end{cases} \qquad A = XX^\top \left(XX^\top + YY^\top\right)^{-1}.$$

- $\beta = 2$

$$\begin{cases} X \sim \mathcal{N}_{N,M_1}(0,1) + i\,\mathcal{N}_{N,M_1}(0,1) \\ Y \sim \mathcal{N}_{N,M_2}(0,1) + i\,\mathcal{N}_{N,M_2}(0,1) \end{cases} \qquad A = XX^\dagger \left(XX^\dagger + YY^\dagger\right)^{-1}.$$

- $\beta = 4$

$$\begin{cases} X_1 \sim \mathcal{N}_{N,M_1}(0,1) + i\,\mathcal{N}_{N,M_1}(0,1) \\ X_2 \sim \mathcal{N}_{N,M_1}(0,1) + i\,\mathcal{N}_{N,M_1}(0,1) \\ Y_1 \sim \mathcal{N}_{N,M_2}(0,1) + i\,\mathcal{N}_{N,M_2}(0,1) \\ Y_2 \sim \mathcal{N}_{N,M_2}(0,1) + i\,\mathcal{N}_{N,M_2}(0,1) \end{cases} \qquad \begin{cases} X = \begin{bmatrix} X_1 & X_2 \\ -X_2^* & X_1^* \end{bmatrix} \\ Y = \begin{bmatrix} Y_1 & Y_2 \\ -Y_2^* & Y_1^* \end{bmatrix} \end{cases} \qquad A = XX^\dagger \left(XX^\dagger + YY^\dagger\right)^{-1}.$$

Concentrates as Wachter law

$$\frac{(a+b)\sqrt{(\sigma_+ - x)(x - \sigma_-)}}{2\pi x(1-x)} dx,$$

where

$$a = \frac{M_1}{N}, b = \frac{M_2}{N} \quad \text{and} \quad \sigma_\pm = \left(\frac{\sqrt{a(a+b-1)} \pm \sqrt{b}}{a+b}\right)^2,$$

itself tending to the arcsine law in the limit.

```
from dppy.beta_ensembles import JacobiEnsemble


jacobi = JacobiEnsemble(beta=2)  # beta must be in {0,1,2,4}, default beta=2
jacobi.sample_full_model(size_N=400, size_M1=500, size_M2=600)  # M_1, M_2 >= N
# jacobi.plot(normalization=True)
jacobi.hist(normalization=True)

# To compare with the sampling speed of the triadiagonal model simply use
# jacobi.sample_banded_model(size_N=400, size_M1=500, size_M2=600)
```

See also:

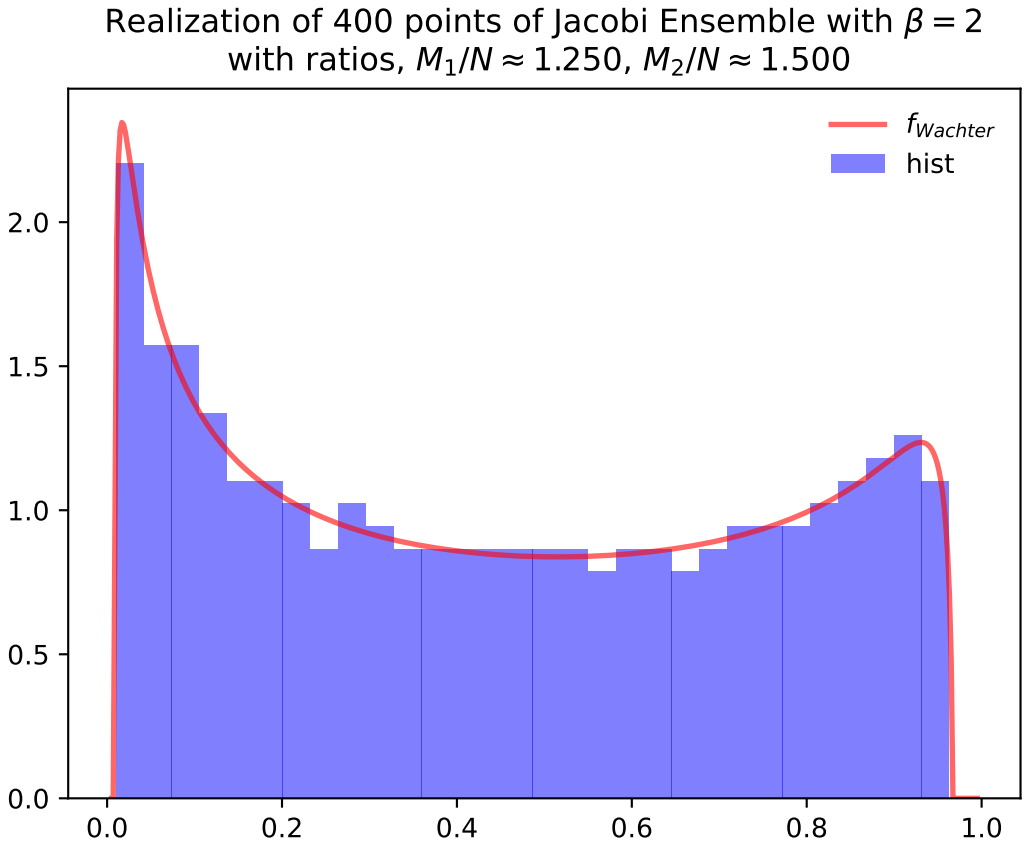


Fig. 3.7: Full matrix model for the Jacobi ensemble

- *Banded matrix model* for Jacobi ensemble
- `JacobiEnsemble` in API
- *Multivariate Jacobi ensemble*
- `MultivariateJacobiOPE` in API

Circular Ensemble

$$\left|\Delta(e^{i\theta_1},\dots,e^{i\theta_N})\right|^\beta \prod_{j=1}^{N} \frac{1}{2\pi}\mathbf{1}_{[0,2\pi]}(\theta_j)d\theta_j,$$

$where from the definition in : eq : `eq : abs_vandermonde_et` we have : math : ` |\Delta(x_1,\dots,x_N)| = \prod_{i<j}|x_i - x_j|`.$

Hint: Eigenvalues of orthogonal (resp. unitary and self-dual unitary) matrices drawn uniformly i.e. Haar measure on the respective groups. The eigenvalues lie on the unit circle i.e. $\lambda_n = e^{i\theta_n}$. The distribution of the angles θ_n converges to the uniform measure on $[0, 2\pi[$ as N grows.

- $\beta = 1$

  Uniform measure i.e. Haar measure on orthogonal matrices $\mathbb{O}_N$: $U^\top U = I_N$

  1. Via QR algorithm, see [Mez06] Section 5

```
import numpy as np
from numpy.random import randn
import scipy.linalg as la

A = randn(N, N)
Q, R = la.qr(A)
d = np.diagonal(R)
U = np.multiply(Q, d/np.abs(d), Q)
la.eigvals(U)
```

  2. The Hermite way

$$X \sim \mathcal{N}_{N,N}(0,1)$$
$$A = X + X^\top = U^\top \Lambda U$$
$$eigvals(U).$$

- $\beta = 2$

  Uniform measure i.e. Haar measure on unitary matrices $\mathbb{U}_N$: $U^\dagger U = I_N$

  1. Via QR algorithm, see [Mez06] Section 5

```
import numpy as np
from numpy.random import randn
import scipy.linalg as la

A = randn(N, N) + 1j*randn(N, N)
Q, R = la.qr(A)
```

(continues on next page)

(continued from previous page)

```
d = np.diagonal(R)
U = np.multiply(Q, d / np.abs(d), Q)
la.eigvals(U)
```

```
from dppy.beta_ensembles import CircularEnsemble

circular = CircularEnsemble(beta=2)  # beta in {0,1,2,4}, default beta=2

# 1. Plot the eigenvalues, they lie on the unit circle
circular.sample_full_model(size_N=30, haar_mode='QR')
circular.plot()

# 2. Histogram of the angle of more points, should look uniform on [0,
↪2pi]
circular.flush_samples()  # Flush previous sample

circular.sample_full_model(size_N=1000, haar_mode='QR')
circular.hist()
```

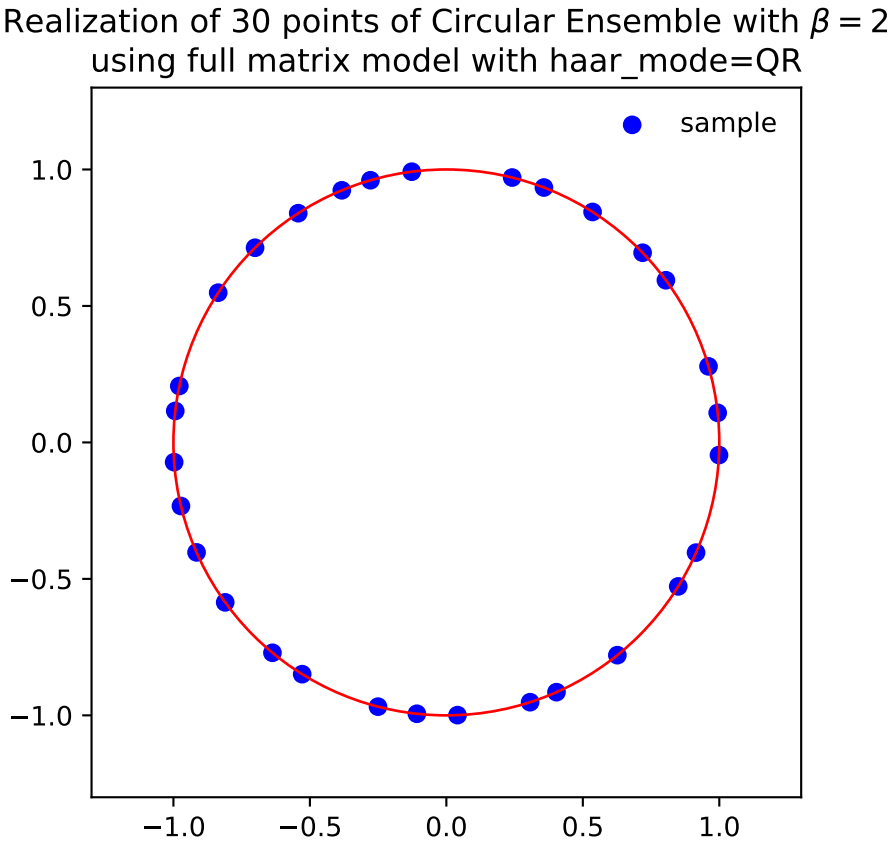


Fig. 3.8: Full matrix model for the Circular ensemble from QR on random Gaussian matrix

2. The Hermite way

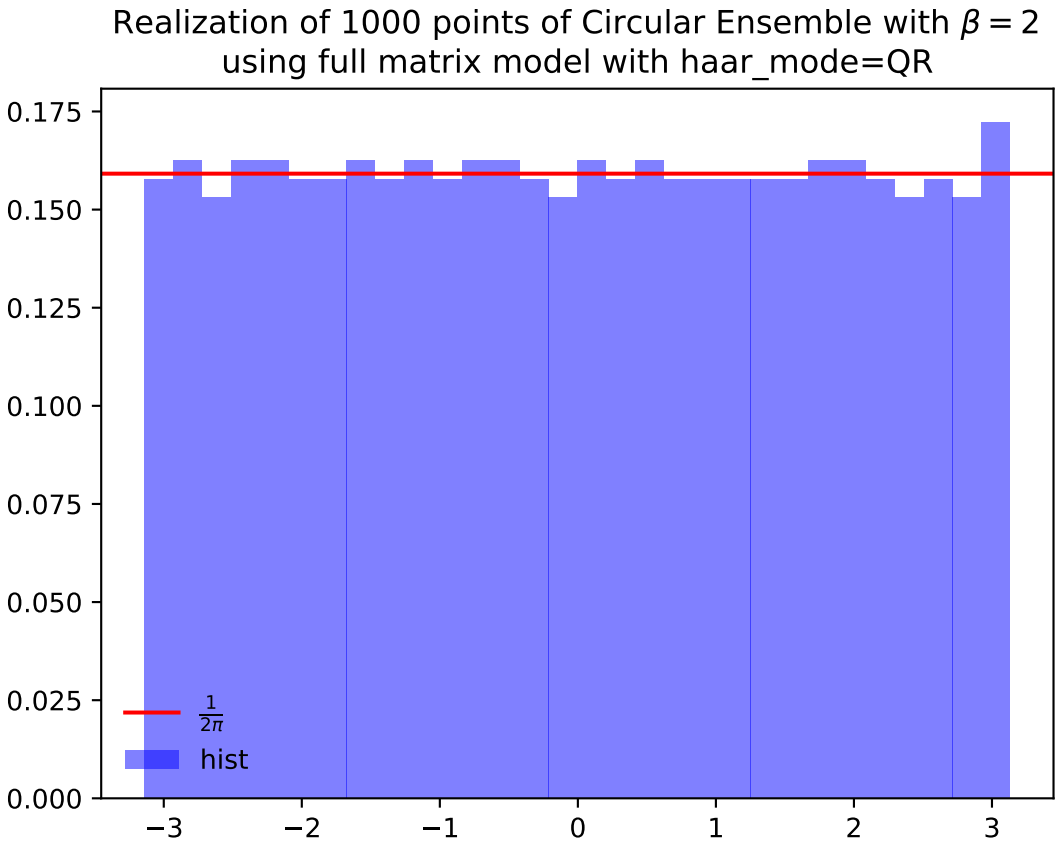


Fig. 3.9: Full matrix model for the Circular ensemble from QR on random Gaussian matrix

$$X \sim \mathcal{N}_{N,N}(0,1) + i\,\mathcal{N}_{N,N}(0,1)$$
$$A = X + X^\dagger = U^\dagger \Lambda U$$
$$eigvals(U).$$

```
from dppy.beta_ensembles import CircularEnsemble

circular = CircularEnsemble(beta=2)  # beta in {0,1,2,4}, default beta=2

# 1. Plot the eigenvalues, they lie on the unit circle
circular.sample_full_model(size_N=30, haar_mode='Hermite')
circular.plot()

# 2. Histogram of the angle of more points, should look uniform on [0,
↪2pi]
circular.flush_samples()  # Flush previous sample

circular.sample_full_model(size_N=1000, haar_mode='Hermite')
circular.hist()
```

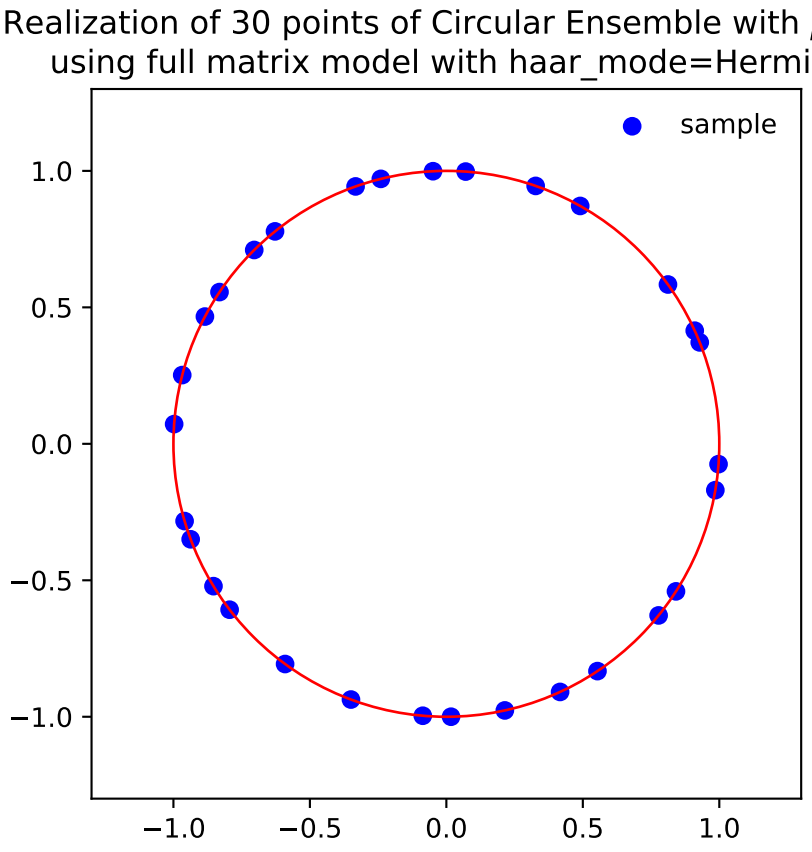


Fig. 3.10: Full matrix model for the Circular ensemble from Hermite matrix

- $\beta = 4$

  Uniform measure i.e. Haar measure on self-dual unitary matrices $\mathbb{U}\,\mathrm{Sp}_{2N}$: $U^\dagger U = I_{2N}$

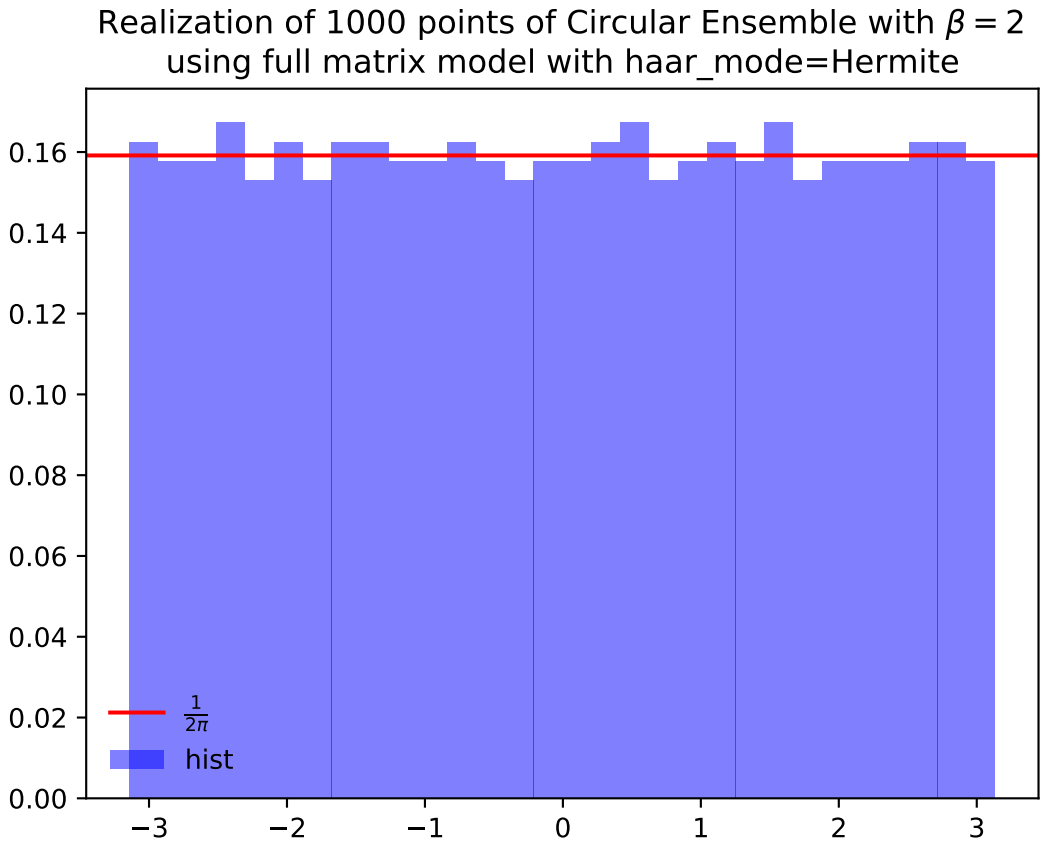


Fig. 3.11: Full matrix model for the Circular ensemble from Hermite matrix

$$\begin{cases} X \sim \mathcal{N}_{N,M}(0,1) + i\,\mathcal{N}_{N,M}(0,1) \\ Y \sim \mathcal{N}_{N,M}(0,1) + i\,\mathcal{N}_{N,M}(0,1) \end{cases}$$

$$A = \begin{bmatrix} X & Y \\ -Y^* & X^* \end{bmatrix} \quad A = X + X^\dagger = U^\dagger \Lambda U$$

$$eigvals(U).$$

See also:

- *Banded matrix model* for Circular ensemble
- `CircularEnsemble` in API

Ginibre Ensemble

$$|\Delta(z_1, \ldots, z_N)|^2 \prod_{i=1}^{N} e^{-\frac{1}{2}|z_i|^2} dz_i,$$

where from the definition in (3.33) we have $|\Delta(x_1, \ldots, x_N)| = \prod_{i<j} |x_i - x_j|$.

$$A \sim \frac{1}{\sqrt{2}} \left(\mathcal{N}_{N,N}(0,1) + i\,\mathcal{N}_{N,N}(0,1)\right).$$

Nomalization $\sqrt{N}$ to concentrate in the unit circle.

```
from dppy.beta_ensembles import GinibreEnsemble


ginibre = GinibreEnsemble()  # beta must be 2 (default)

ginibre.sample_full_model(size_N=40)
ginibre.plot(normalization=True)

ginibre.sample_full_model(size_N=1000)
ginibre.hist(normalization=True)
```

See also:

- `GinibreEnsemble` in API

Banded matrix models

Computing the eigenvalues of a full $N \times N$ random matrix is $\mathcal{O}(N^3)$, and can thus become prohibitive for large N. A way to circumvent the problem is to adopt the equivalent banded models i.e. diagonalize banded matrices.

The first tridiagonal models for the *Hermite Ensemble* and *Laguerre Ensemble* were revealed by [DE02], who left the *Jacobi Ensemble* as an open question, addressed by [KN04]. Such tridiagonal formulations made sampling possible at cost $\mathcal{O}(N^2)$ but also unlocked sampling for generic $\beta > 0$!

Note that [KN04] also derived a quindiagonal model for the *Circular Ensemble*.

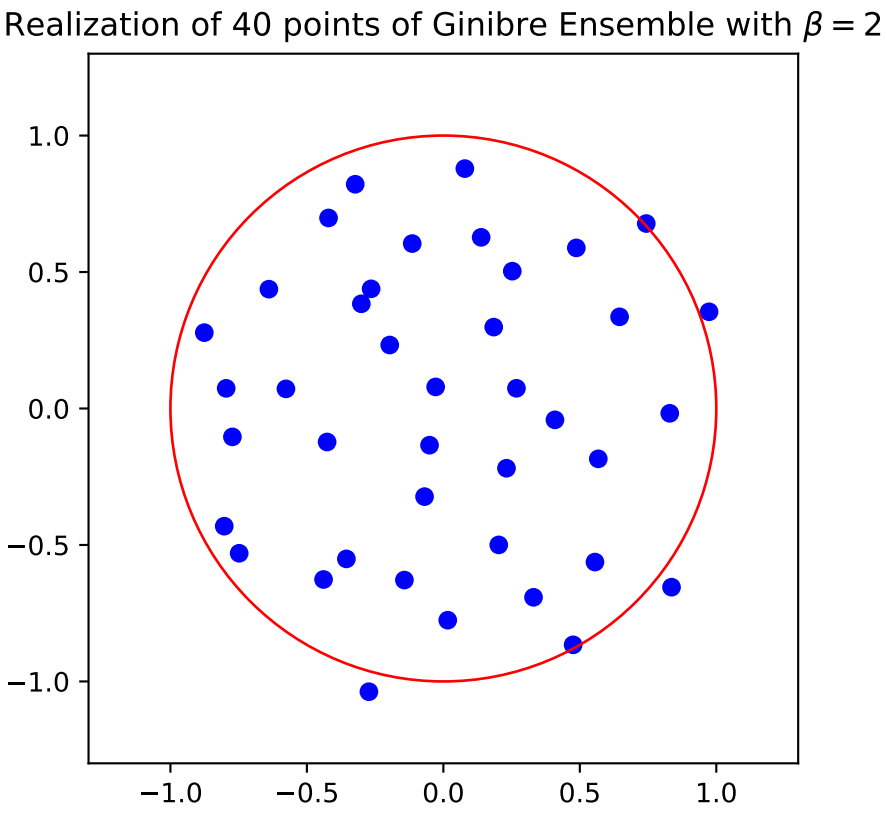


Fig. 3.12: Full matrix model for the Ginibre ensemble

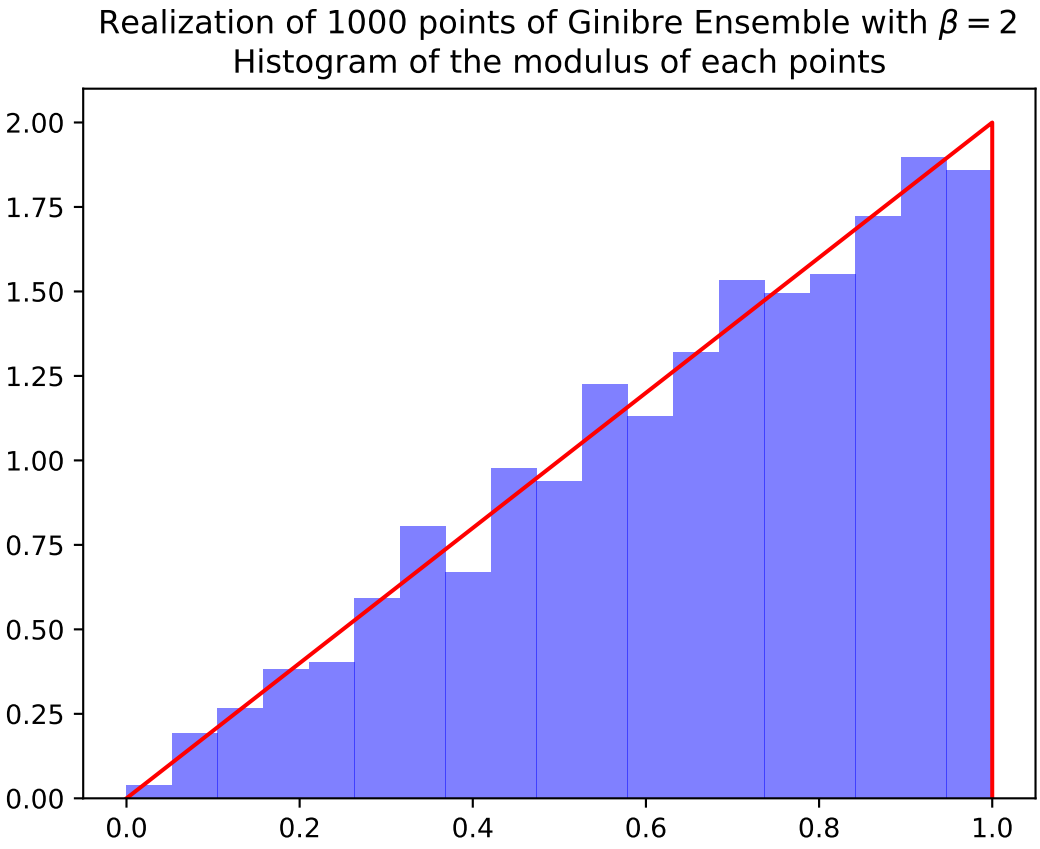


Fig. 3.13: Full matrix model for the Ginibre ensemble

Hermite Ensemble

Take for reference measure $\mu = \mathcal{N}(\mu, \sigma)$

$$(x_1, \dots, x_N) \sim |\Delta(x_1, \dots, x_N)|^\beta \prod_{i=1}^N e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\, dx_i.$$

Note: Recall that from the definition in (3.33)

$$|\Delta(x_1, \dots, x_N)| = \prod_{i<j} |x_i - x_j|.$$

The equivalent tridiagonal model reads

$$\begin{bmatrix} \alpha_1 & \sqrt{\beta_2} & 0 & 0 & 0 \\ \sqrt{\beta_2} & \alpha_2 & \sqrt{\beta_3} & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \sqrt{\beta_{N-1}} & \alpha_{N-1} & \sqrt{\beta_N} \\ 0 & 0 & 0 & \sqrt{\beta_N} & \alpha_N \end{bmatrix},$$

with

$$\alpha_i \sim \mathcal{N}(\mu, \sigma^2) \quad \text{and} \quad \beta_{i+1} \sim \Gamma\left(\frac{\beta}{2}(N-i), \sigma^2\right).$$

To recover the full matrix model for *Hermite Ensemble*, recall that $\Gamma(\frac{k}{2}, 2) \equiv \chi^2_k$ and take

$$\mu = 0 \quad \text{and} \quad \sigma^2 = 2.$$

That is to say,

$$\alpha_i \sim \mathcal{N}(0, 2) \quad \text{and} \quad \beta_{i+1} \sim \chi^2_{\beta(N-i)}.$$

```
from dppy.beta_ensembles import HermiteEnsemble


hermite = HermiteEnsemble(beta=5.43)  # beta can be >=0, default beta=2
# Reference measure is N(mu, sigma^2)
hermite.sample_banded_model(loc=0.0, scale=1.0, size_N=500)
# hermite.plot(normalization=True)
hermite.hist(normalization=True)
```

See also:

- [DE02] II-C
- *Full matrix model* for Hermite ensemble
- `HermiteEnsemble` in API

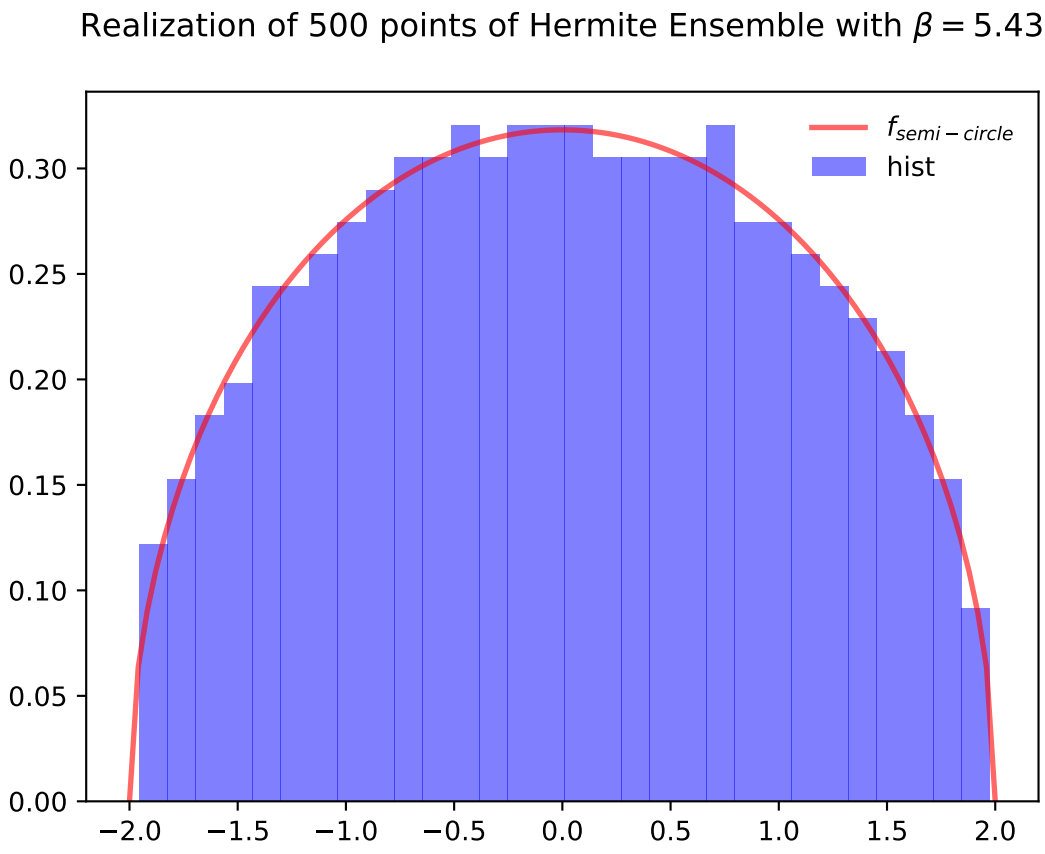


Fig. 3.14: Tridiagonal matrix model for the Hermite ensemble

Laguerre Ensemble

Take for reference measure $\mu = \Gamma(k, \theta)$

$$(x_1, \dots, x_N) \sim |\Delta(x_1, \dots, x_N)|^\beta \prod_{i=1}^N x_i^{k-1} e^{-\frac{x_i}{\theta}} \, dx_i.$$

Note: Recall that from the definition in (3.33)

$$|\Delta(x_1, \dots, x_N)| = \prod_{i<j} |x_i - x_j|.$$

The equivalent tridiagonal model reads

$$\begin{bmatrix} \alpha_1 & \sqrt{\beta_2} & 0 & 0 & 0 \\ \sqrt{\beta_2} & \alpha_2 & \sqrt{\beta_3} & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \sqrt{\beta_{N-1}} & \alpha_{N-1} & \sqrt{\beta_N} \\ 0 & 0 & 0 & \sqrt{\beta_N} & \alpha_N \end{bmatrix} = \begin{bmatrix} \sqrt{\xi_1} & & & & \\ \sqrt{\xi_2} & \sqrt{\xi_3} & & & \\ & \ddots & \ddots & & \\ & & \sqrt{\xi_{2N-2}} & \sqrt{\xi_{2N-1}} & \end{bmatrix} \begin{bmatrix} \sqrt{\xi_1} & \sqrt{\xi_2} & & \\ & \sqrt{\xi_3} & \ddots & \\ & & \ddots & \sqrt{\xi_{2N-2}} \\ & & & \sqrt{\xi_{2N-1}} \end{bmatrix},$$

with

$$\xi_{2i-1} \sim \Gamma\left(\frac{\beta}{2}(N-i) + k, \theta\right) \quad \text{and} \quad \xi_{2i} \sim \Gamma\left(\frac{\beta}{2}(N-i), \theta\right).$$

To recover the full matrix model for *Laguerre Ensemble*, recall that $\Gamma(\frac{k}{2}, 2) \equiv \chi^2_k$ and take

$$k = \frac{\beta}{2}(M - N + 1) \quad \text{and} \quad \theta = 2.$$

That is to say,

$$\xi_{2i-1} \sim \chi^2_{\beta(M-i+1)} \quad \text{and} \quad \xi_{2i} \sim \chi^2_{\beta(N-i)}.$$

```
from dppy.beta_ensembles import LaguerreEnsemble


laguerre = LaguerreEnsemble(beta=2.98)  # beta can be >=0, default beta=2
# Reference measure is Gamma(k, theta)
laguerre.sample_banded_model(shape=600, scale=2.0, size_N=400)
# laguerre.plot(normalization=True)
laguerre.hist(normalization=True)
```

See also:

- [DE02] III-B
- *Full matrix model* for Laguerre ensemble
- `LaguerreEnsemble` in API

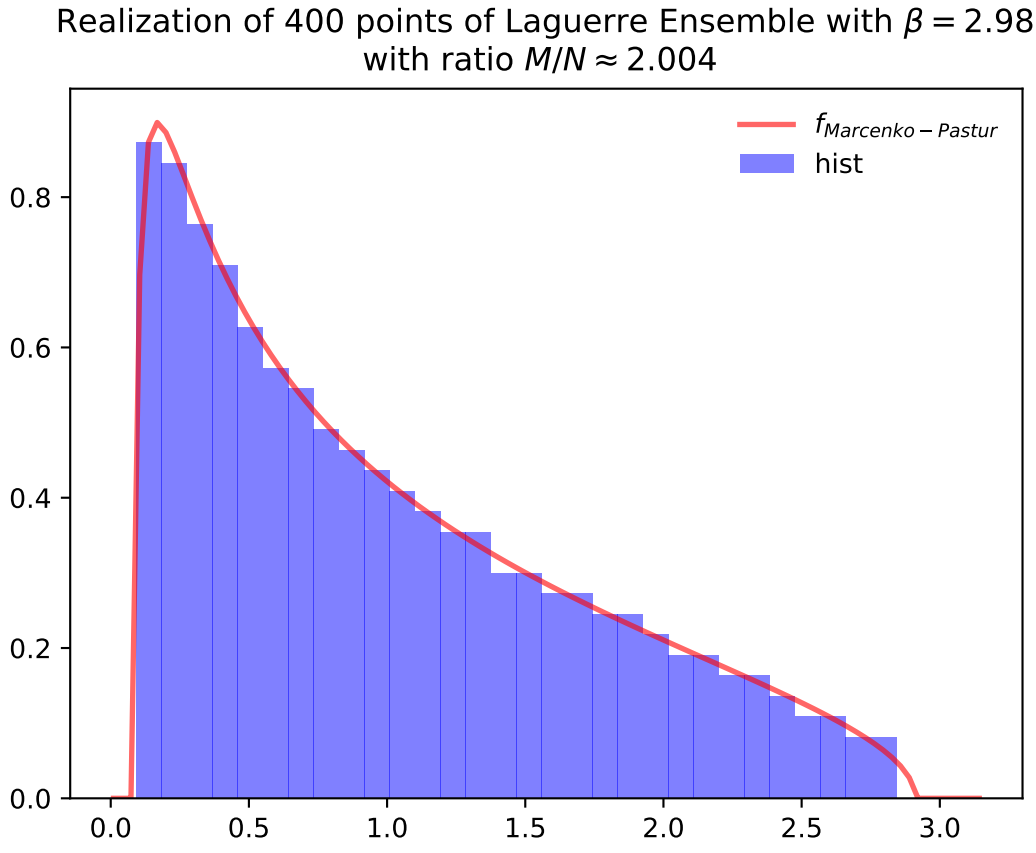


Fig. 3.15: Tridiagonal matrix model for the Laguerre ensemble

Jacobi Ensemble

Take for reference measure $\mu = \mathrm{Beta}(a, b)$

$$(x_1, \dots, x_N) \sim |\Delta(x_1, \dots, x_N)|^\beta \prod_{i=1}^N x_i^{a-1}(1-x_i)^{b-1}\, dx_i.$$

Note: Recall that from the definition in (3.33)

$$|\Delta(x_1, \dots, x_N)| = \prod_{i<j} |x_i - x_j|.$$

The equivalent tridiagonal model reads

$$\begin{bmatrix} \alpha_1 & \sqrt{\beta_2} & 0 & 0 & 0 \\ \sqrt{\beta_2} & \alpha_2 & \sqrt{\beta_3} & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \sqrt{\beta_{N-1}} & \alpha_{N-1} & \sqrt{\beta_N} \\ 0 & 0 & 0 & \sqrt{\beta_N} & \alpha_N \end{bmatrix}.$$

$$\begin{aligned} \alpha_1 &= \xi_1 & & \\ \alpha_k &= \xi_{2k-2} + \xi_{2k-1} & \beta_{k+1} &= \xi_{2k-1}\xi_{2k} \\ \xi_1 &= c_1 & \gamma_1 &= 1 - c_1 \\ \xi_k &= (1 - c_{k-1})c_k & \gamma_k &= c_{k-1}(1 - c_k) \end{aligned},$$

with

$$c_{2i-1} \sim \mathrm{Beta}\left(\frac{\beta}{2}(N-i) + a, \frac{\beta}{2}(N-i) + b\right) \quad \text{and} \quad c_{2i} \sim \mathrm{Beta}\left(\frac{\beta}{2}(N-i), \frac{\beta}{2}(N-i-1) + a + b\right).$$

To recover the full matrix model for *Laguerre Ensemble*, recall that $\Gamma(\frac{k}{2}, 2) \equiv \chi^2_k$ and take

$$a = \frac{\beta}{2}(M_1 - N + 1) \quad \text{and} \quad b = \frac{\beta}{2}(M_2 - N + 1).$$

That is to say,

$$c_{2i-1} \sim \mathrm{Beta}\left(\frac{\beta}{2}(M_1 - i + 1), \frac{\beta}{2}(M_2 - i + 1)\right) \quad \text{and} \quad c_{2i} \sim \mathrm{Beta}\left(\frac{\beta}{2}(N-i), \frac{\beta}{2}(M_1 + M_2 - N - i + 1)\right).$$

```
from dppy.beta_ensembles import JacobiEnsemble


jacobi = JacobiEnsemble(beta=3.14)  # beta can be >=0, default beta=2
# Reference measure is Beta(a,b)
jacobi.sample_banded_model(a=500, b=300, size_N=400)
# jacobi.plot(normalization=True)
jacobi.hist(normalization=True)
```

See also:

- [KN04] Theorem 2

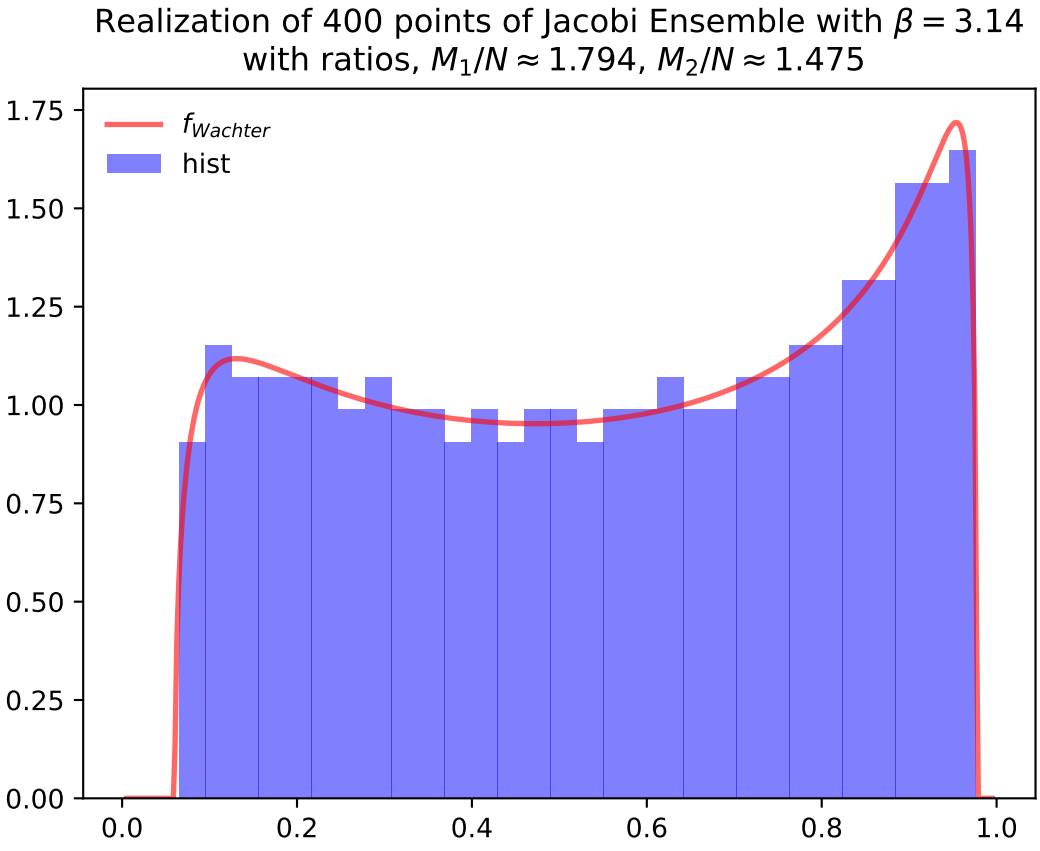


Fig. 3.16: Tridiagonal matrix model for the Jacobi ensemble

- *Full matrix model* for Jacobi ensemble
- `JacobiEnsemble` in API
- *Multivariate Jacobi ensemble*
- `MultivariateJacobiOPE` in API

Circular Ensemble

$$\left|\Delta(e^{i\theta_1},\dots,e^{i\theta_N})\right|^\beta \prod_{j=1}^N \frac{1}{2\pi}\mathbf{1}_{[0,2\pi]}(\theta_j)d\theta_j.$$

Note: Recall that from the definition in (3.33)

$$|\Delta(x_1,\dots,x_N)| = \prod_{i<j}|x_i - x_j|.$$

Important: Consider the distribution Θ_ν that for integers $\nu \geq 2$ is defined as follows:

Draw v uniformly at random from the unit sphere $\mathbb{S}^\nu \in \mathbb{R}^{\nu+1}$, then $v_1 + iv_2 \sim \Theta_\nu$

Now, given $\beta \in \mathbb{N}^*$, let

- $\alpha_k \sim \Theta_{\beta(N-k-1)+1}$ independent variables
- for $0 \leq k \leq N-1$ set $\rho_k = \sqrt{1-|\alpha_k|^2}$.

Then, the equivalent quindiagonal model corresponds to the eigenvalues of either LM or ML with

$$L = \mathrm{diag}[\Xi_0, \Xi_2, \dots] \quad \text{and} \quad M = \mathrm{diag}[\Xi_{-1}, \Xi_1, \Xi_3 \dots],$$

and where

$$\Xi_k = \begin{bmatrix} \overline{\alpha}_k & \rho_k \\ \rho_k & -\alpha_k \end{bmatrix}, \quad 0 \leq k \leq N-2, \quad \text{with} \quad \Xi_{-1} = [1] \quad \text{and} \quad \Xi_{N-1} = [\overline{\alpha}_{N-1}].$$

Hint: The effect of increasing the β parameter can be nicely visualized on this *Circular Ensemble*. Viewing β as the inverse temperature, the configuration of the eigenvalues crystallizes with β, see the figure below.

```
from dppy.beta_ensembles import CircularEnsemble

circular = CircularEnsemble(beta=2)  # beta must be >=0 integer, default beta=2

# See the cristallization of the configuration as beta increases
for b in [0, 1, 5, 10]:

    circular.beta = b
    circular.sample_banded_model(size_N=30)
    circular.plot()
```

(continues on next page)

(continued from previous page)

```
circular.beta = 2
circular.sample_banded_model(size_N=1000)
circular.hist()
```

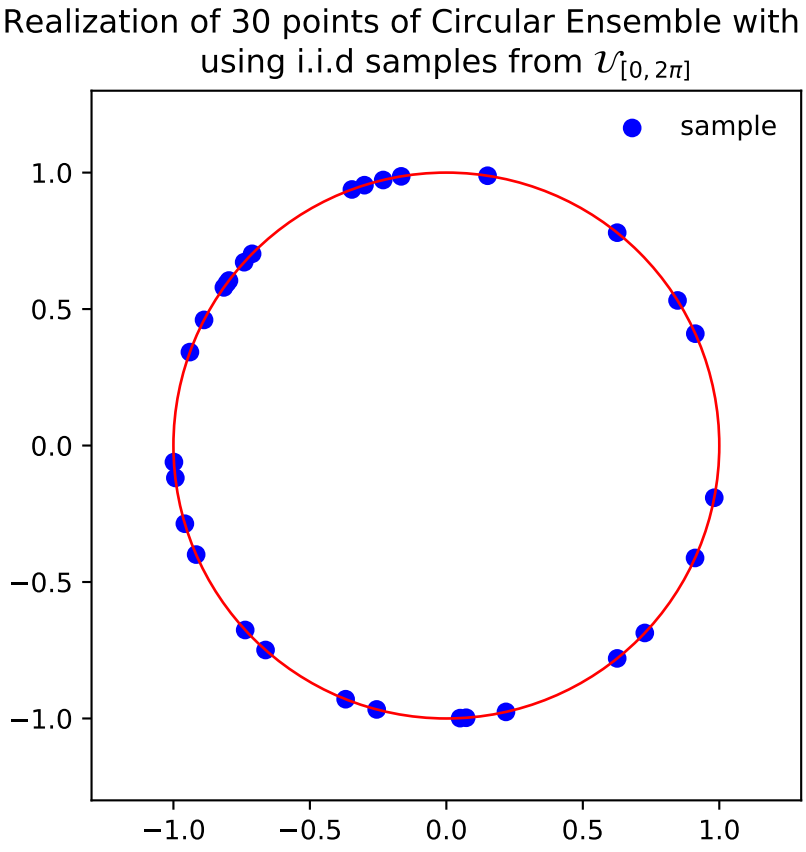


Fig. 3.17: Quindiagonal matrix model for the Circular ensemble

See also:

- [KN04] Theorem 1
- *Full matrix model* for Circular ensemble
- `CircularEnsemble` in API

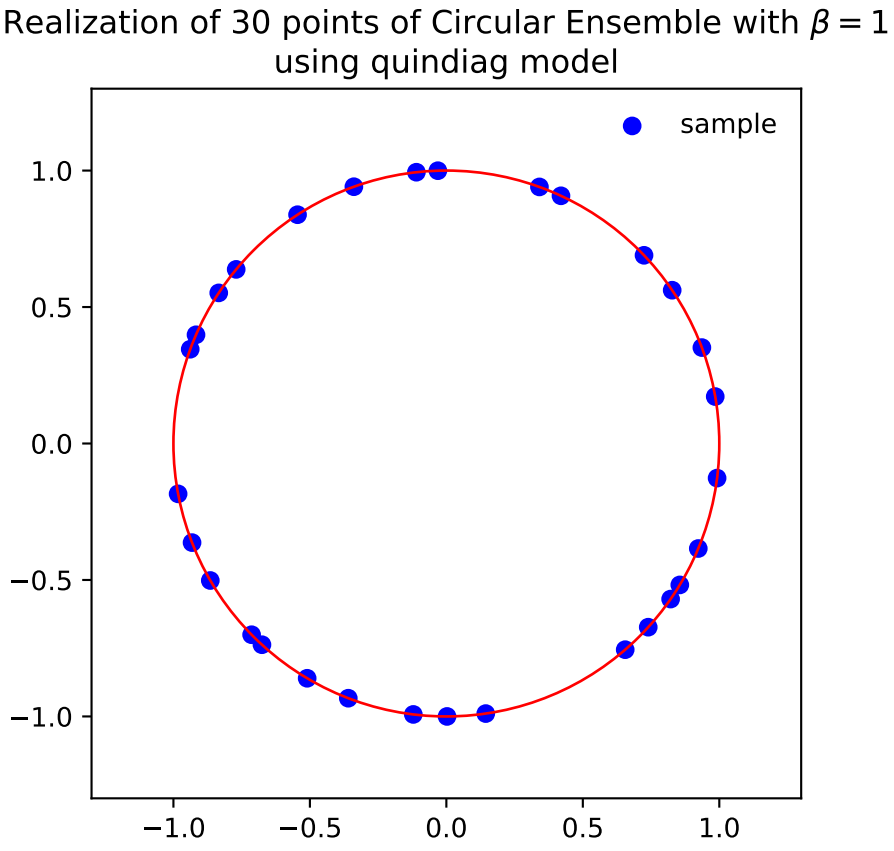


Fig. 3.18: Quindiagonal matrix model for the Circular ensemble

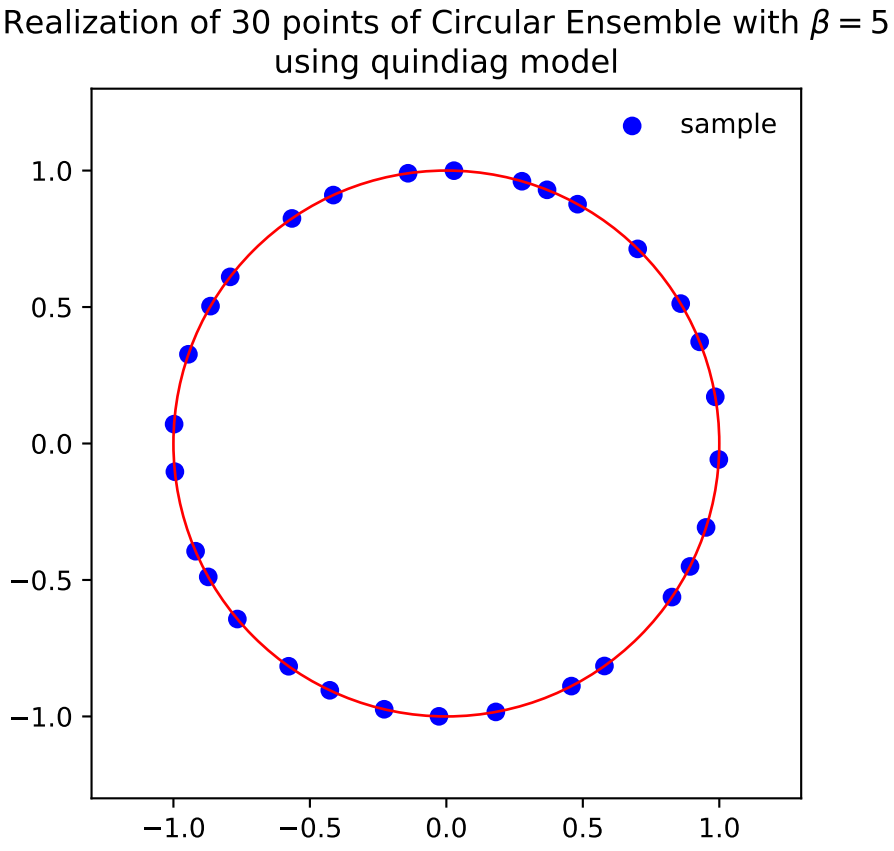


Fig. 3.19: Quindiagonal matrix model for the Circular ensemble

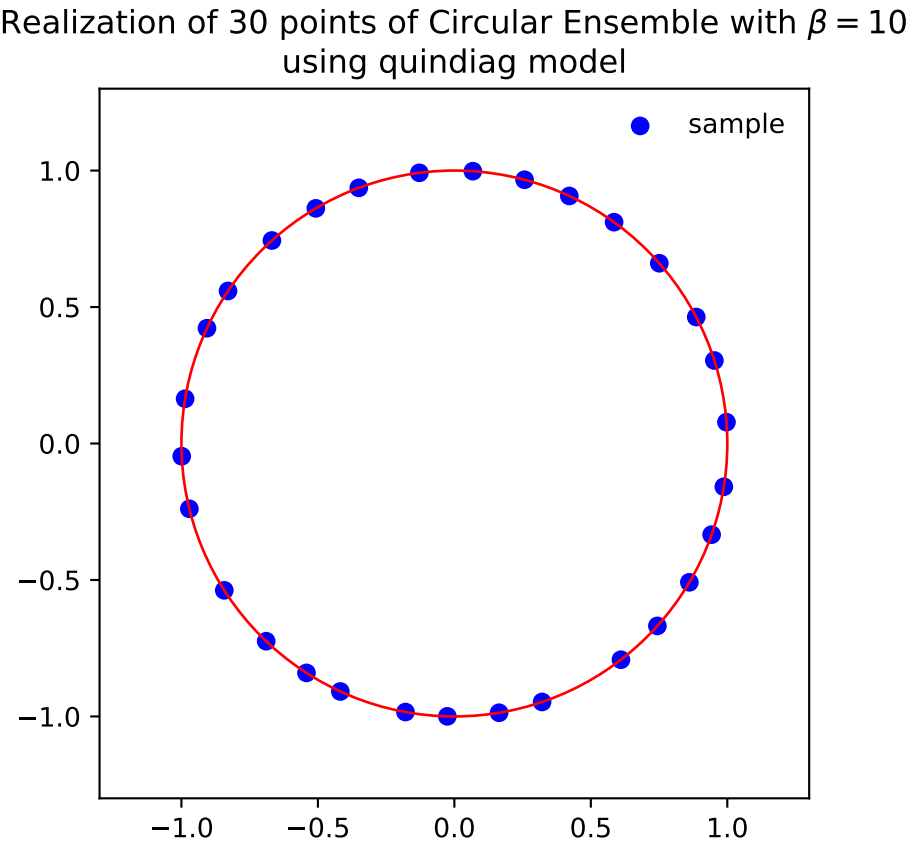


Fig. 3.20: Quindiagonal matrix model for the Circular ensemble

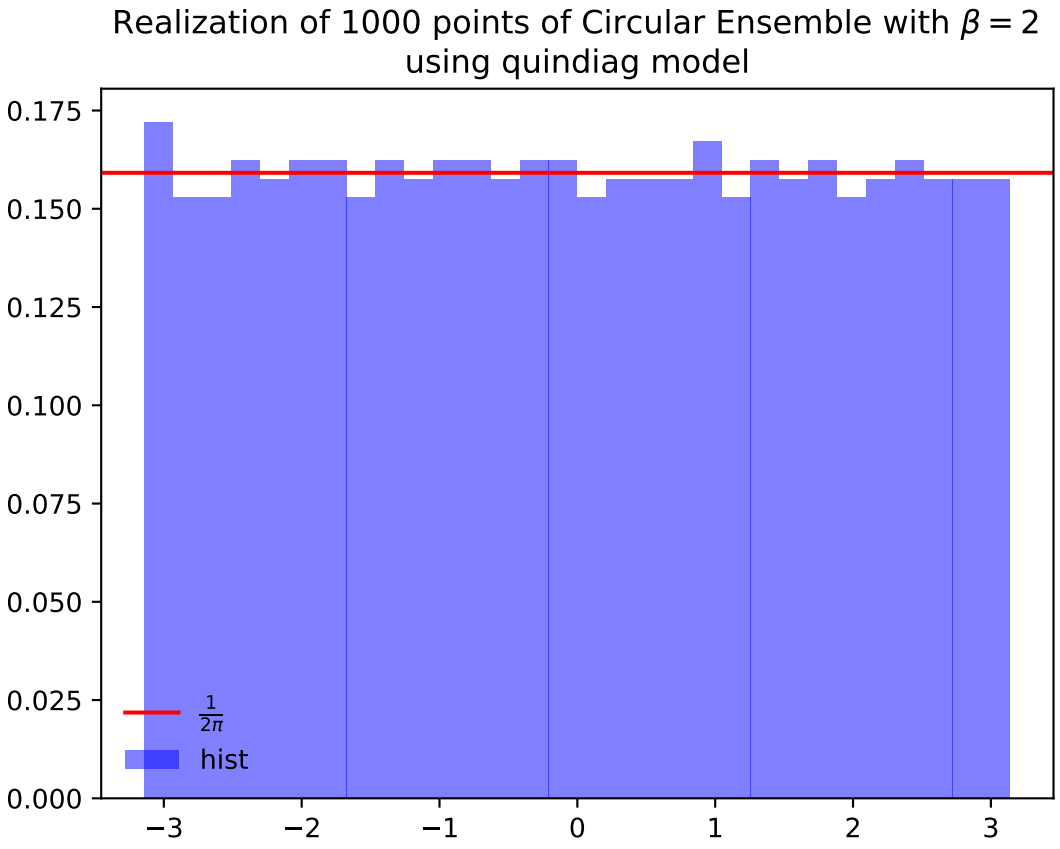


Fig. 3.21: Quindiagonal matrix model for the Circular ensemble

3.2.5 Multivariate Jacobi ensemble

Important: For the details please refer to:

a) the extensive documentation of *MultivariateJacobiOPE* below

b) the associated Jupyter notebook which showcases *MultivariateJacobiOPE*

c) our NeurIPS'19 paper [GBV19] *On two ways to use determinantal point processes for Monte Carlo integration*

d) our ICML'19 workshop paper

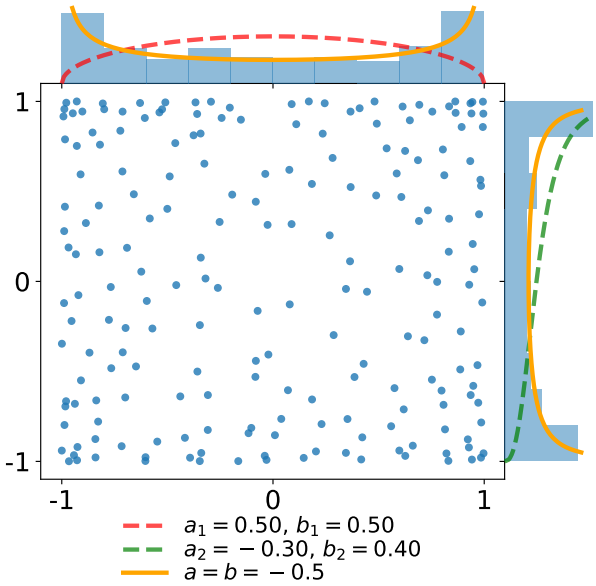
The figures below display a sample of a $d = 2$ dimensional Jacobi ensemble *MultivariateJacobiOPE* with $N = 200$ points. The red and green dashed curves correspond to the normalized base densities proportional to $(1-x)^{a_1}(1+x)^{b_1}$ and $(1-y)^{a_2}(1+y)^{b_2}$, respectively.

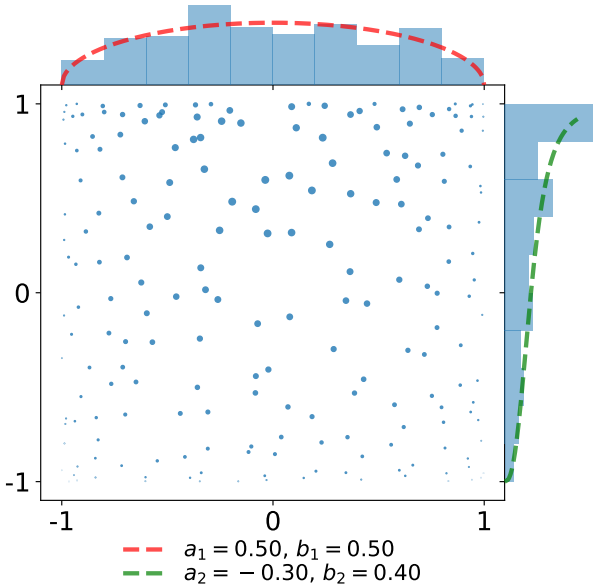
```
import numpy as np
import matplotlib.pyplot as plt
from dppy.multivariate_jacobi_ope import MultivariateJacobiOPE

# The .plot() method outputs smtg only in dimension d=1 or 2

# Number of points / dimension
N, d = 200, 2
# Jacobi parameters in [-0.5, 0.5]^{d x 2}
jac_params = np.array([[0.5, 0.5],
                       [-0.3, 0.4]])

dpp = MultivariateJacobiOPE(N, jac_params)

# Get an exact sample
sampl = dpp.sample()

# Display
# the cloud of points
# the base probability densities
# the marginal empirical histograms
dpp.plot(sample=sampl, weighted=False)
plt.tight_layout()

dpp.plot(sample=sampl, weighted='BH')
plt.tight_layout()

dpp.plot(sample=sampl, weighted='EZ')
plt.tight_layout()
```

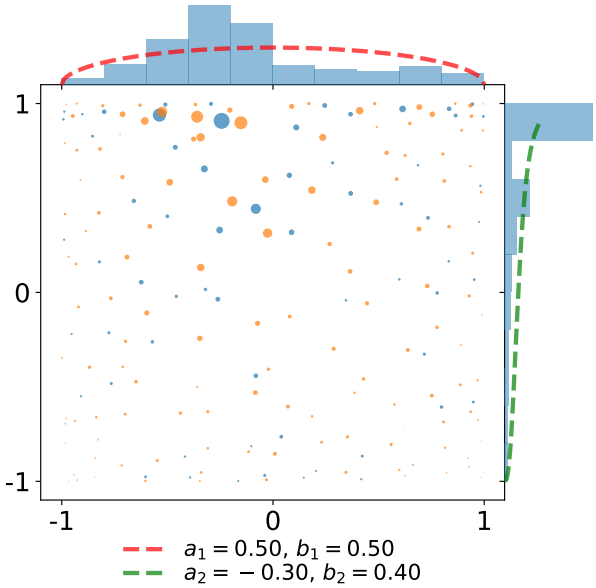
- In the first plot, observe that the empirical marginal density converges to the arcsine density $\frac{1}{\pi\sqrt{1-x^2}}$, displayed in orange.
- In the second plot, we take the same sample and attach a weight $\frac{1}{K(x,x)}$ to each of the points. This illustrates the choice of the weights defining the estimator of [BH16] as a proxy for the reference measure.

Implementation of the class *MultivariateJacobiOPE* used in [GBV19] for Monte Carlo with Determinantal Point Processes

It has 3 main methods:


1
0
-1
-1
0
1
$a_1 = 0.50, b_1 = 0.50$
$a_2 = -0.30, b_2 = 0.40$
$a = b = -0.5$


1
0
-1
-1
0
1
$a_1 = 0.50, b_1 = 0.50$
$a_2 = -0.30, b_2 = 0.40$



- *sample()* to generate samples
- *K()* to evaluate the corresponding projection kernel
- plot() to display 1D or 2D samples

class dppy.multivariate_jacobi_ope.**MultivariateJacobiOPE**(*N*, *jacobi_params*)

Bases: object

Multivariate Jacobi Orthogonal Polynomial Ensemble used in [GBV19] for Monte Carlo with Determinantal Point Processes

This corresponds to a continuous multivariate projection DPP with state space $[-1, 1]^d$ with respect to

- reference measure $\mu(dx) = w(x)dx$ (see also *eval_w()*), where

$$w(x) = \prod_{i=1}^{d} (1 - x_i)^{a_i} (1 + x_i)^{b_i}$$

- kernel K (see also *K()*)

$$K(x, y) = \sum_{\mathfrak{b}(k)=0}^{N-1} P_k(x) P_k(y) = \Phi(x)^\top \Phi(y)$$

where

- $k \in \mathbb{N}^d$ is a multi-index ordered according to the ordering $\mathfrak{b}$ (see *compute_ordering()*)
- $P_k(x) = \prod_{i=1}^{d} P_{k_i}^{(a_i,b_i)}(x_i)$ is the product of orthonormal Jacobi polynomials

$$\int_{-1}^{1} P_k^{(a_i,b_i)}(u) P_\ell^{(a_i,b_i)}(u)(1 - u)^{a_i}(1 + u)^{b_i} du = \delta_{k\ell}$$

so that (P_k) are orthonormal w.r.t $\mu(dx)$

- $\Phi(x) = \left(P_{\mathfrak{b}^{-1}(0)}(x), \dots, P_{\mathfrak{b}^{-1}(N-1)}(x)\right)^\top$

Parameters

- **N** (*int*) – Number of points $N \geq 1$
- **jacobi_params** (*array_like*) – Jacobi parameters $[(a_i, b_i)]_{i=1}^d$ The number of rows d prescribes the ambient dimension of the points i.e. $x_1, \dots, x_N \in [-1, 1]^d$. - when $d = 1$, $a_1, b_1 > -1$ - when $d \geq 2$, $|a_i|, |b_i| \leq \frac{1}{2}$

See also:

- *Multivariate Jacobi ensemble*
- when $d = 1$, the *univariate Jacobi ensemble* is sampled by computing the eigenvalues of a properly randomized *tridiagonal matrix* of [KN04]
- [BH16] initiated the use of the multivariate Jacobi ensemble for Monte Carlo integration. In particular, they proved CLT with variance decay of order $N^{-(1+1/d)}$ which is faster that the N^{-1} rate of vanilla Monte Carlo where the points are drawn i.i.d. from the base measure.

K(*X*, *Y=None*, *eval_pointwise=False*)

Evalute $(K(x, y))_{x \in X, y \in Y}$ if `eval_pointwise=False` or $(K(x, y))_{(x,y) \in (X,Y)}$ otherwise

$$K(x, y) = \sum_{\mathfrak{b}(k)=0}^{N-1} P_k(x) P_k(y) = \phi(x)^\top \phi(y)$$

where

- $k \in \mathbb{N}^d$ is a multi-index ordered according to the ordering $\mathfrak{b}$, *compute_ordering()*
- $P_k(x) = \prod_{i=1}^d P_{k_i}^{(a_i,b_i)}(x_i)$ is the product of orthonormal Jacobi polynomials

$$\int_{-1}^{1} P_k^{(a_i,b_i)}(u) P_\ell^{(a_i,b_i)}(u)(1-u)^{a_i}(1+u)^{b_i} du = \delta_{k\ell}$$

  so that (P_k) are orthonormal w.r.t $\mu(dx)$
- $\Phi(x) = \left(P_{\mathfrak{b}^{-1}(0)}(x), \dots, P_{\mathfrak{b}^{-1}(N-1)}(x)\right)$, see *eval_multiD_polynomials()*

  Parameters

  - **X** (*array_like*) – $M \times d$ array of M points $\in [-1, 1]^d$
  - **Y** (*array_like (default None)*) – $M' \times d$ array of M' points $\in [-1, 1]^d$
  - **eval_pointwise** (*bool (default False)*) – sets pointwise evaluation of the kernel, if `True`, X and Y must have the same shape, see Returns

  Returns

  If `eval_pointwise=False` (default), evaluate the kernel matrix

  $$(K(x, y))_{x \in X, y \in Y}$$

  If `eval_pointwise=True` kernel matrix Pointwise evaluation of K as depicted in the following pseudo code output

  - if `Y` is `None`
    - $(K(x, y))_{x \in X, y \in X}$ if `eval_pointwise=False`
    - $(K(x, x))_{x \in X}$ if `eval_pointwise=True`

- otherwise
  - $(K(x,y))_{x\in X, y\in Y}$ if `eval_pointwise=False`
  - $(K(x,y))_{(x,y)\in(X,Y)}$ if `eval_pointwise=True` (in this case X and Y should have the same shape)

Return type array_like

See also:

`eval_multiD_polynomials()`

`eval_multiD_polynomials`(*X*)

Evaluate

$$\mathbf{\Phi}(X) := \begin{pmatrix} \Phi(x_1)^\top \\ \vdots \\ \Phi(x_M)^\top \end{pmatrix}$$

where $\Phi(x) = \left(P_{\mathfrak{b}^{-1}(0)}(x), \dots, P_{\mathfrak{b}^{-1}(N-1)}(x)\right)^\top$ such that $K(x,y) = \Phi(x)^\top \Phi(y)$. Recall that $\mathfrak{b}$ denotes the ordering chosen to order multi-indices $k \in \mathbb{N}^d$.

This is done by evaluating each of the three-term recurrence relations satisfied by each univariate orthogonal Jacobi polynomial, using the dedicated see also SciPy `scipy.special.eval_jacobi()` satistified by the respective univariate Jacobi polynomials $P_{k_i}^{(a_i,b_i)}(x_i)$. Then we use the slicing feature of the Python language to compute $\Phi(x) = \left(P_k(x) = \prod_{i=1}^d P_{k_i}^{(a_i,b_i)}(x_i)\right)_{k=\mathfrak{b}^{-1}(0),\dots,\mathfrak{b}^{-1}(N-1)}^\top$

Parameters **`X`** (*`array_like`*) – $M \times d$ array of M points $\in [-1,1]^d$

Returns $\mathbf{\Phi}(X)$ - $M \times N$ array

Return type array_like

See also:

- evaluation of the kernel *`K()`*

`eval_w`(*X*)

Evaluate $w(x) = \prod_{i=1}^d (1-x_i)^{a_i}(1+x_i)^{b_i}$ which corresponds to the density of the base measure $\mu(dx) = w(x)dx$

Parameters **`X`** (*`array_like`*) – $M \times d$ array of M points $\in [-1,1]^d$

Returns $w(x) = \prod_{i=1}^d (1-x_i)^{a_i}(1+x_i)^{b_i}$

Return type array_like

`sample`(*nb_trials_max=10000*, *random_state=None*, *tridiag_1D=True*)

Use the chain rule [HKPVirag06] (Algorithm 18) to sample $(x_1, \dots, x_N)$ with density

$$\begin{aligned}
&\frac{1}{N!}\left(K(x_n,x_p)\right)_{n,p=1}^N \prod_{n=1}^N w(x_n) \\
&= \frac{1}{N} K(x_1,x_1) w(x_1) \prod_{n=2}^N \frac{K(x_n,x_n) - K(x_n,x_{1:n-1})\left[(K(x_k,x_l))_{k,l=1}^{n-1}\right]^{-1} K(x_{1:n-1},x_n)}{N-(n-1)} w(x_n) \\
&= \frac{\|\Phi(x)\|^2}{N} \omega(x_1)dx_1 \prod_{n=2}^N \frac{\mathrm{distance}^2(\Phi(x_n), \mathrm{span}\{\Phi(x_p)\}_{p=1}^{n-1})}{N-(n-1)} \omega(x_n)dx_n
\end{aligned}$$

The order in which the points were sampled can be forgotten to obtain a valid sample of the corresponding DPP

- $x_1 \sim \frac{1}{N} K(x,x)w(x)$ using `sample_chain_rule_proposal()`
- $x_n|Y = \{x_1, \dots, x_{n-1}\}$, is sampled using rejection sampling with proposal density $\frac{1}{N} K(x,x)w(x)$ and rejection bound frac{N}{N-(n-1)}

$$\frac{1}{N-(n-1)}[K(x,x) - K(x,Y)K_Y^{-1}K(Y,x)]w(x) \leq \frac{N}{N-(n-1)}\frac{1}{N}K(x,x)w(x)$$

Note: Using the gram structure $K(x,y) = \Phi(x)^\top \Phi(y)$ the numerator of the successive conditionals reads

$$\begin{aligned} K(x,x) - K(x,Y)K(Y,Y)^{-1}K(Y,x) &= \text{distance}^2(\Phi(x_n), \text{span}\{\Phi(x_p)\}_{p=1}^{n-1}) \\ &= \left\| (I - \Pi_{\text{span}\{\Phi(x_p)\}_{p=1}^{n-1}} \phi(x) \right\|^2 \end{aligned}$$

which can be computed simply in a vectorized way. The overall procedure is akin to a sequential Gram-Schmidt orthogonalization of $\Phi(x_1), \dots, \Phi(x_N)$.

See also:

- *Projection DPPs: the chain rule*
- `sample_chain_rule_proposal()`

sample_chain_rule_proposal(*nb_trials_max=10000*, *random_state=None*)

Use a rejection sampling mechanism to sample

$$\frac{1}{N}K(x,x)w(x)dx = \frac{1}{N}\sum_{\mathfrak{b}(k)=0}^{N-1} \left(\frac{P_k(x)}{\|P_k\|}\right)^2 w(x)$$

with proposal distribution

$$w_{eq}(x)dx = \prod_{i=1}^{d} \frac{1}{\pi\sqrt{1-(x_i)^2}} dx_i$$

Since the target density is a mixture, we can sample from it by

1. Select a multi-index k uniformly at random in $\left\{\mathfrak{b}^{-1}(0), \dots, \mathfrak{b}^{-1}(N-1)\right\}$
2. Sample from $\left(\frac{P_k(x)}{\|P_k\|}\right)^2 w(x)dx$ with proposal $w_{eq}(x)dx$.

   The acceptance ratio writes

   $$\frac{\left(\frac{P_k(x)}{\|P_k\|}\right)^2 w(x)}{w_{eq}(x)} = \prod_{i=1}^{d} \pi \left(\frac{P_{k_i}^{(a_i,b_i)}(x)}{\left\|P_{k_i}^{(a_i,b_i)}\right\|}\right)^2 (1-x_i)^{a_i+\frac{1}{2}}(1+x_i)^{b_i+\frac{1}{2}} \leq C_k$$

   which can be bounded using the result of [Gau09] on Jacobi polynomials.

   Note: Each of the rejection constant C_k is computed at initialization of the `MultivariateJacobiOPE` object using `compute_rejection_bounds()`

Returns A sample $x \in [-1,1]^d$ with probability distribution $\frac{1}{N}K(x,x)w(x)$

Return type array_like

See also:

- *compute_rejection_bounds()*
- *sample()*

dppy.multivariate_jacobi_ope.**compute_degrees_1D_polynomials**(*max_degrees*)

deg[i, j] = i if i <= max_degrees[j] else 0

dppy.multivariate_jacobi_ope.**compute_norms_1D_polynomials**(*jacobi_params*, *deg_max*)

Compute the square norms $\|P_k^{(a_i,b_i)}\|^2$ of each (univariate) orthogoanl Jacobi polynomial for $k = 0$ to `deg_max` and $a_i, b_i =$ `jacobi_params[i, :]` Recall that the Jacobi polynomials $\left(P_k^{(a_i,b_i)}\right)$ are orthogonal w.r.t. $(1-u)^{a_i}(1+u)^{b_i}du$.

$$\begin{aligned}\|P_k^{(a_i,b_i)}\|^2 &= \int_{-1}^{1}\left(P_k^{(a_i,b_i)}(u)\right)^2(1-u)^{a_i}(1+u)^{b_i}du \\ &= \frac{2^{a_i+b_i+1}}{2k+a_i+b_i+1}\frac{\Gamma(k+a_i+1)\Gamma(k+b_i+1)}{\Gamma(k+a_i+b_i+1)n!}\end{aligned}$$

Parameters

- **jacobi_params** (*array_like*) – Jacobi parameters $[(a_i,b_i)]_{i=1}^d \in [-\frac{1}{2},\frac{1}{2}]^{d\times 2}$ The number of rows d prescribes the ambient dimension of the points i.e. $x_1,\dots,x_N \in [-1,1]^d$
- **deg_max** (*int*) – Maximal degree of 1D Jacobi polynomials

Returns Array of size `deg_max + 1` $\times d$ with entry k, i given by $\|P_k^{(a_i,b_i)}\|^2$

Return type array_like

See also:

- Wikipedia Jacobi polynomials
- *compute_ordering()*

dppy.multivariate_jacobi_ope.**compute_ordering**(*N*, *d*)

Compute the ordering of the multi-indices $\in \mathbb{N}^d$ defining the order between the multivariate monomials as described in Section 2.1.3 of [BH16].

Parameters

- **N** (*int*) – Number of polynomials (P_k) considered to build the kernel *K()* (number of points of the corresponding *MultivariateJacobiOPE*)
- **d** (*int*) – Size of the multi-indices $k \in \mathbb{N}^d$ characterizing the _degree_ of P_k (ambient dimension of the points x_{1}, dots, x_{N} in [-1, 1]^d)

Returns Array of size $N \times d$ containing the first N multi-indices $\in \mathbb{N}^d$ in the order prescribed by the ordering $\mathfrak{b}$ [BH16] Section 2.1.3

Return type array_like

For instance, for $N = 12, d = 2$

```
[(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2), (0, 3),␣
↪(1, 3), (2, 3)]
```

See also:

- [BH16] Section 2.1.3

dppy.multivariate_jacobi_ope.**compute_rejection_bounds**(*jacobi_params*, *ordering*, *log_scale=True*)

Compute the rejection constants for the acceptance/rejection mechanism used in `sample_chain_rule_proposal()` to sample

$$\frac{1}{N}K(x,x)w(x)dx = \frac{1}{N}\sum_{\mathfrak{b}(k)=0}^{N-1}\left(\frac{P_k(x)}{\|P_k\|}\right)^2 w(x)$$

with proposal distribution

$$w_{eq}(x)dx = \prod_{i=1}^{d}\frac{1}{\pi\sqrt{1-(x_i)^2}}dx_i$$

To get a sample:

1. Draw a multi-index k uniformly at random in $\left\{\mathfrak{b}^{-1}(0),\dots,\mathfrak{b}^{-1}(N-1)\right\}$
2. Sample from $P_k(x)^2 w(x)dx$ with proposal $w_{eq}(x)dx$.

The acceptance ratio writes

$$\frac{\left(\frac{P_k(x)}{\|P_k\|}\right)^2 w(x)}{w_{eq}(x)} = \prod_{i=1}^{d}\pi\left(\frac{P_{k_i}^{(a_i,b_i)}(x)}{\left\|P_{k_i}^{(a_i,b_i)}\right\|}\right)^2 (1-x_i)^{a_i+\frac{1}{2}}(1+x_i)^{b_i+\frac{1}{2}} \leq C_k$$

- For $k_i > 0$ we use a result on Jacobi polynomials given by, e.g., [Gau09], for $|a|,|b| \leq \frac{1}{2}$

$$\pi(1-u)^{a+\frac{1}{2}}(1+u)^{b+\frac{1}{2}}\left(\frac{P_n^{(a,b)}(u)}{\left\|P_n^{(a,b)}\right\|}\right)^2$$
$$\leq \frac{2}{n!(n+(a+b+1)/2)2^{\max(a,b)}}\frac{\Gamma(n+a+b+1)\Gamma(n+\max(a,b)+1)}{\Gamma(n+\min(a,b)+1)}$$

- For $k_i = 0$, we use less involved properties of the Jacobi polynomials:
  - $P_0^{(a,b)} = 1$
  - $\|P_0^{(a,b)}\|^2 = 2^{a+b+1}\,\mathrm{B}(a+1,b+1)$
  - $m = \frac{b-a}{a+b+1}$ is the mode of $(1-u)^{a+\frac{1}{2}}(1+u)^{b+\frac{1}{2}}$ (valid since $a+\frac{1}{2}, b+\frac{1}{2} > 0$)

  So that,

$$\pi(1-u)^{a+\frac{1}{2}}(1+u)^{b+\frac{1}{2}}\left(\frac{P_0^{(a,b)}(u)}{\|P_0^{(a,b)}\|}\right)^2 = \frac{\pi(1-u)^{a+\frac{1}{2}}(1+u)^{b+\frac{1}{2}}}{\|P_0^{(a,b)}\|^2}$$
$$\leq \frac{\pi(1-m)^{a+\frac{1}{2}}(1+m)^{b+\frac{1}{2}}}{2^{a+b+1}\,\mathrm{B}(a+1,b+1)}$$

Parameters

- **jacobi_params** (*array_like*) – Jacobi parameters $[(a_i,b_i)]_{i=1}^d \in [-\frac{1}{2},\frac{1}{2}]^{d\times 2}$.

  The number of rows d prescribes the ambient dimension of the points i.e. $x_1,\dots,x_N \in [-1,1]^d$

- **ordering** (*array_like*) – Ordering of the multi-indices $\in \mathbb{N}^d$ defining the order between the multivariate monomials (see also *compute_ordering()*)
  - the number of rows corresponds to the number N of monomials considered.
  - the number of columns $= d$
- **log_scale** (*bool*) – If True, the rejection bound is computed using the logarithmic versions `betaln`, `gammaln` of `beta` and `gamma` functions to avoid overflows

Returns The rejection bounds C_k for $k = \mathfrak{b}^{-1}(0), \dots, \mathfrak{b}^{-1}(N-1)$

Return type array_like

See also:

- [Gau09] for the domination when $k_i > 0$
- `compute_poly1D_norms()`

3.2.6 API

Implementation of the meta-class *BetaEnsemble* see $\beta-$*Ensembles* with children:

- *HermiteEnsemble*
- *LaguerreEnsemble*
- *JacobiEnsemble*
- *CircularEnsemble*
- *GinibreEnsemble*

Such objects have 4 main methods:

- `sample_full_model()`
- `sample_banded_model()`
- `plot()` to display a scatter plot of the last sample and eventually the limiting distribution (after normalization)
- `hist()` to display a histogram of the last sample and eventually the limiting distribution (after normalization)

class dppy.beta_ensembles.**BetaEnsemble**(*beta=2*)

Bases: `object`

β-Ensemble object parametrized by

Parameters **beta** (int, float, default 2) – $\beta >= 0$ inverse temperature parameter.

The default `beta`$= 2$ corresponds to the DPP case, see *Orthogonal Polynomial Ensembles*

See also:

- β-Ensembles *definition*

flush_samples()

Empty the `list_of_samples` attribute.

abstract hist()

Display histogram of the last realization of the underlying β-Ensemble. For some β-Ensembles, a normalization argument is available to display the limiting (or equilibrium) distribution and scale the points accordingly.

abstract normalize_points()

Normalize points ormalization argument is available to display the limiting (or equilibrium) distribution and scale the points accordingly.

abstract plot()
Display last realization of the underlying β-Ensemble. For some β-Ensembles, a normalization argument is available to display the limiting (or equilibrium) distribution and scale the points accordingly.

abstract sample_banded_model()
Sample from underlying β-Ensemble using the corresponding banded matrix model. Arguments are the associated reference measure's parameters, or the matrix dimensions used in *sample_full_model()*

abstract sample_full_model()
Sample from underlying β-Ensemble using the corresponding full matrix model. Arguments are the associated matrix dimensions

class dppy.beta_ensembles.**CircularEnsemble**(*beta=2*)
Bases: *dppy.beta_ensembles.BetaEnsemble*

Circular Ensemble object

See also:

- *Full matrix model* associated to the Circular ensemble
- *Quindiagonal matrix model* associated to the Circular ensemble

flush_samples()
Empty the `list_of_samples` attribute.

hist(*normalization=True*)
Display the histogram of the angles $\theta_1, \dots, \theta_N$ associated to the last realization $\left\{e^{i\theta_1}, \dots, e^{i\theta_N}\right\}$ object.

Parameters **normalization** (bool, default `True`) – When `True`, the limiting distribution of the angles, i.e., the uniform distribution in $[0, 2\pi]$ is displayed

See also:

- *sample_full_model()*, *sample_banded_model()*
- *plot()*
- *Full matrix model* associated to the Circular ensemble
- *Quindiagonal matrix model* associated to the Circular ensemble

normalize_points(*points*)
No need to renormalize the points

plot(*normalization=True*)
Display the last realization of the *CircularEnsemble* object.

Parameters **normalization** (bool, default `True`) – When `True`, the unit circle is displayed

See also:

- *sample_full_model()*, *sample_banded_model()*
- *hist()*
- *Full matrix model* associated to the Circular ensemble
- *Quindiagonal matrix model* associated to the Circular ensemble

sample_banded_model(*size_N=10*, *random_state=None*)
Sample from *Quindiagonal matrix model* associated to the Circular Ensemble. Available for `beta` $\in \mathbb{N}^*$, and the degenerate case `beta` $= 0$ corresponding to i.i.d. uniform points on the unit circle

Parameters **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized

---

Note: To compare *sample_banded_model()* with *sample_full_model()* simply use the `size_N` parameter.

See also:

- *Quindiagonal matrix model* associated to the Circular ensemble
- *sample_full_model()*

sample_full_model(*size_N=10*, *haar_mode='Hermite'*, *random_state=None*)

Sample from *tridiagonal matrix model* associated to the Circular ensemble. Only available for `beta` $\in \{1, 2, 4\}$ and the degenerate case `beta` $= 0$ corresponding to i.i.d. uniform points on the unit circle

Parameters

- **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized
- **haar_mode** (*str, default 'hermite'*) – Sample Haar measure i.e. uniformly on the orthogonal/unitary/symplectic group using: - 'QR', - 'Hermite'

See also:

- *Full matrix model* associated to the Circular ensemble
- *sample_banded_model()*

class dppy.beta_ensembles.**GinibreEnsemble**(*beta=2*)

Bases: *dppy.beta_ensembles.BetaEnsemble*

Ginibre Ensemble object

See also:

- *Full matrix model* associated to the Ginibre ensemble

flush_samples()

Empty the `list_of_samples` attribute.

hist(*normalization=True*)

Display the histogram of the radius of the points the last realization of the *GinibreEnsemble* object

Parameters **normalization** (bool, default `True`) – When `True`, the points are first normalized so as to concentrate in the unit disk (see *normalize_points()*) and the limiting density $2r1_{[0,1]}(r)$ of the radii is displayed

See also:

- *normalize_points()*
- *sample_full_model()*
- *Full matrix model* associated to the Ginibre ensemble ensemble

normalize_points(*points*)

Normalize points to concentrate in the unit disk.

$$x \mapsto \frac{x}{\sqrt{N}}$$

See also:

- *plot()*
- *hist()*

plot(*normalization=True*)
Display the last realization of the *GinibreEnsemble* object

Parameters **normalization** (bool, default `True`) – When `True`, the points are first normalized so as to concentrate in the unit disk (see *normalize_points()*) and the unit circle is displayed

See also:

- *normalize_points()*
- *sample_full_model()*
- *Full matrix model* associated to the Ginibre ensemble ensemble

sample_banded_model(**args*, ***kwargs*)
No banded model is known for Ginibre, use *sample_full_model()*

sample_full_model(*size_N=10*, *random_state=None*)
Sample from *full matrix model* associated to the Ginibre ensemble. Only available for `beta` $= 2$

Parameters **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized

See also:

- *Full matrix model* associated to the Ginibre ensemble

class dppy.beta_ensembles.**HermiteEnsemble**(*beta=2*)
Bases: *dppy.beta_ensembles.BetaEnsemble*

Hermite Ensemble object

See also:

- *Full matrix model* associated to the Hermite ensemble
- *Tridiagonal matrix model* associated to the Hermite ensemble

flush_samples()
Empty the `list_of_samples` attribute.

hist(*normalization=True*)
Display the histogram of the last realization of the *HermiteEnsemble* object.

Parameters **normalization** (bool, default `True`) – When `True`, the points are first normalized (see *normalize_points()*) so that they concentrate as

- If `beta` $= 0$, the $\mathcal{N}(0, 2)$ reference measure associated to full *full matrix model*
- If `beta` > 0, the limiting distribution, i.e., the semi-circle distribution

in both cases, the corresponding p.d.f. is displayed

See also:

- *sample_full_model()*, *sample_banded_model()*
- *normalize_points()*
- *plot()*
- *Full matrix model* associated to the Hermite ensemble
- *Tridiagonal matrix model* associated to the Hermite ensemble

normalize_points (*points*)

Normalize points obtained after sampling to fit the limiting distribution, i.e., the semi-circle

$$f(x) = \frac{1}{2\pi}\sqrt{4-x^2}$$

Parameters **points** (*array_like*) – A sample from Hermite ensemble, accessible through the `list_of_samples` attribute

- If sampled using *sample_banded_model()* with reference measure $\mathcal{N}(\mu, \sigma^2)$
  1. Normalize the points to fit the p.d.f. of $\mathcal{N}(0, 2)$ reference measure of the *full matrix model*

$$x \mapsto \sqrt{2}\frac{x-\mu}{\sigma}$$

  2. If `beta` > 0, normalize the points to fit the semi-circle distribution

$$x \mapsto \frac{x}{\beta N}$$

  Otherwise if `beta` $= 0$ do nothing more
- If sampled using *sample_full_model()*, apply 2. above

Note: This method is called in *plot()* and *hist()* when `normalization=True`

---

plot (*normalization=True*)

Display the last realization of the *HermiteEnsemble* object

Parameters **normalization** (bool, default `True`) – When `True`, the points are first normalized (see *normalize_points()*) so that they concentrate as

- If `beta` $= 0$, the $\mathcal{N}(0, 2)$ reference measure associated to full *full matrix model*
- If `beta` > 0, the limiting distribution, i.e., the semi-circle distribution

in both cases, the corresponding p.d.f. is displayed

See also:

- *sample_full_model()*, *sample_banded_model()*
- *normalize_points()*
- *hist()*

- *Full matrix model* associated to the Hermite ensemble
- *Tridiagonal matrix model* associated to the Hermite ensemble

sample_banded_model(*loc=0.0*, *scale=1.4142135623730951*, *size_N=10*, *random_state=None*)

Sample from *tridiagonal matrix model* associated to the Hermite Ensemble. Available for `beta` > 0 and the degenerate case `beta` $= 0$ corresponding to i.i.d. points from the Gaussian $\mathcal{N}(\mu, \sigma^2)$ reference measure

Parameters

- **loc** (float, default 0) – Mean μ of the Gaussian $\mathcal{N}(\mu, \sigma^2)$
- **scale** (float, default $\sqrt{2}$) – Standard deviation σ of the Gaussian $\mathcal{N}(\mu, \sigma^2)$
- **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized

---

Note: The reference measure associated with the *full matrix model* is $\mathcal{N}(0, 2)$. For this reason, in the `sampling_params` attribute, the default values are set to `loc`$= 0$ and `scale`$= \sqrt{2}$.

To compare *sample_banded_model()* with *sample_full_model()* simply use the `size_N` parameter.

See also:

- *Tridiagonal matrix model* associated to the Hermite ensemble
- [DE02] II-C
- *sample_full_model()*

sample_full_model(*size_N=10*, *random_state=None*)

Sample from *full matrix model* associated to the Hermite ensemble. Only available for `beta` $\in \{1, 2, 4\}$ and the degenerate case `beta` $= 0$ corresponding to i.i.d. points from the Gaussian $\mathcal{N}(\mu, \sigma^2)$ reference measure

Parameters **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized

---

Note: The reference measure associated with the *full matrix model* is $\mathcal{N}(0, 2)$. For this reason, in the `sampling_params` attribute, the values of the parameters are set to `loc`$= 0$ and `scale`$= \sqrt{2}$.

To compare *sample_banded_model()* with *sample_full_model()* simply use the `size_N` parameter.

See also:

- *Full matrix model* associated to the Hermite ensemble
- *sample_banded_model()*

class dppy.beta_ensembles.**JacobiEnsemble**(*beta=2*)

Bases: *dppy.beta_ensembles.BetaEnsemble*

Jacobi Ensemble object

See also:

- *Full matrix model* associated to the Jacobi ensemble
- *Tridiagonal matrix model* associated to the Jacobi ensemble

flush_samples()
Empty the `list_of_samples` attribute.

hist(*normalization=True*)
Display the histogram of the last realization of the *JacobiEnsemble* object.

Parameters **normalization** (bool, default `True`) – When `True`

- If `beta` $= 0$, display the p.d.f. of the $\mathrm{Beta}(a, b)$
- If `beta` > 0, display the limiting distribution, i.e., the Wachter distribution

See also:

- *sample_full_model()*, *sample_banded_model()*
- *normalize_points()*
- *plot()*
- *Full matrix model* associated to the Jacobi ensemble
- *Tridiagonal matrix model* associated to the Jacobi ensemble

normalize_points(*points*)
No need to renormalize the points

plot(*normalization=True*)
Display the last realization of the *JacobiEnsemble* object

Parameters **normalization** (bool, default `True`) – When `True`

- If `beta` $= 0$, display the p.d.f. of the $\mathrm{Beta}(a, b)$
- If `beta` > 0, display the limiting distribution, i.e., the Wachter distribution

See also:

- *sample_full_model()*, *sample_banded_model()*
- *hist()*
- *Full matrix model* associated to the Jacobi ensemble
- *Tridiagonal matrix model* associated to the Jacobi ensemble

sample_banded_model(*a=1.0*, *b=2.0*, *size_N=10*, *size_M1=None*, *size_M2=None*, *random_state=None*)
Sample from *tridiagonal matrix model* associated to the Jacobi ensemble. Available for `beta` > 0 and the degenerate case `beta` $= 0$ corresponding to i.i.d. points from the $\mathrm{Beta}(a, b)$ reference measure

Parameters

- **shape** (float, default 1) – Shape parameter k of $\Gamma(k, \theta)$ reference measure
- **scale** (float, default 2.0) – Scale parameter θ of $\Gamma(k, \theta)$ reference measure
- **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized. Equivalent to the first dimension N of the matrices used in the *full matrix model*.
- **size_M1** (*int*) – Equivalent to the second dimension M_1 of the first matrix used in the *full matrix model*.

- **size_M2** (*int*) – Equivalent to the second dimension M_2 of the second matrix used in the *full matrix model*.

Note: The reference measure associated with the *full matrix model* is :

$$\text{Beta}\left(\frac{\beta}{2}(M_1 - N + 1), \frac{\beta}{2}(M_2 - N + 1)\right)$$

For this reason, in the `sampling_params` attribute, the values of the parameters are set to `a`$= \frac{\beta}{2}(M_1 - N + 1)$ and `b`$= \frac{\beta}{2}(M_2 - N + 1)$.

To compare *`sample_banded_model()`* with *`sample_full_model()`* simply use the `size_N`, `size_M2` and `size_M2` parameters.

- If `size_M1` and `size_M2` are not provided:

  In the `sampling_params` attribute, `size_M1,2` are set to `size_M1`$= \frac{2a}{\beta} + N - 1$ and `size_M2`$= \frac{2b}{\beta} + N - 1$, to give an idea of the corresponding second dimensions $M_{1,2}$.

- If `size_M1` and `size_M2` are provided:

  In the `sampling_params` attribute, `a` and `b` are set to: `a`$= \frac{\beta}{2}(M_1 - N + 1)$ and `b`$= \frac{\beta}{2}(M_2 - N + 1)$.

See also:

- *Tridiagonal matrix model* associated to the Jacobi ensemble
- [KN04] Theorem 2
- *`sample_full_model()`*

sample_full_model(*size_N=100*, *size_M1=150*, *size_M2=200*, *random_state=None*)

Sample from *full matrix model* associated to the Jacobi ensemble. Only available for `beta` $\in \{1, 2, 4\}$ and the degenerate case `beta` $= 0$ corresponding to i.i.d. points from the $\text{Beta}(a, b)$ reference measure

Parameters

- **size_N** (int, default 100) – Number N of points, i.e., size of the matrix to be diagonalized. First dimension of the matrix used to form the covariance matrix to be diagonalized, see *full matrix model*.
- **size_M1** (int, default 150) – Second dimension M_1 of the first matrix used to form the matrix to be diagonalized, see *full matrix model*.
- **size_M2** (int, default 200) – Second dimension M_2 of the second matrix used to form the matrix to be diagonalized, see *full matrix model*.

Note: The reference measure associated with the *full matrix model* is

$$\text{Beta}\left(\frac{\beta}{2}(M_1 - N + 1), \frac{\beta}{2}(M_2 - N + 1)\right)$$

For this reason, in the `sampling_params` attribute, the values of the parameters are set to `a`$= \frac{\beta}{2}(M_1 - N + 1)$ and `b`$= \frac{\beta}{2}(M_2 - N + 1)$.

To compare *sample_banded_model()* with *sample_full_model()* simply use the `size_N`, `size_M2` and `size_M2` parameters.

See also:

- *Full matrix model* associated to the Jacobi ensemble
- *sample_banded_model()*

class dppy.beta_ensembles.**LaguerreEnsemble**(*beta=2*)

Bases: *dppy.beta_ensembles.BetaEnsemble*

Laguerre Ensemble object

See also:

- *Full matrix model* associated to the Laguerre ensemble
- *Tridiagonal matrix model* associated to the Laguerre ensemble

flush_samples()

Empty the `list_of_samples` attribute.

hist(*normalization=True*)

Display the histogram of the last realization of the *LaguerreEnsemble* object.

Parameters **normalization** (bool, default `True`) – When `True`, the points are first normalized (see *normalize_points()*) so that they concentrate as

- If `beta` $= 0$, the $\Gamma(k, 2)$ reference measure associated to full *full matrix model*
- If `beta` > 0, the limiting distribution, i.e., the Marcenko-Pastur distribution

in both cases the corresponding p.d.f. is displayed

See also:

- *sample_full_model()*, *sample_banded_model()*
- *normalize_points()*
- *plot()*
- *Full matrix model* associated to the Laguerre ensemble
- *Tridiagonal matrix model* associated to the Laguerre ensemble

normalize_points(*points*)

Normalize points obtained after sampling to fit the limiting distribution, i.e., the Marcenko-Pastur distribution

$$\frac{1}{2\pi}\frac{\sqrt{(\lambda_+ - x)(x - \lambda_-)}}{cx}1_{[\lambda_-,\lambda_+]}dx$$

where $c = \frac{M}{N}$ and $\lambda_\pm = (1 \pm \sqrt{c})^2$

Parameters **points** (*array_like*) – A sample from Laguerre ensemble, accessible through the `list_of_samples` attribute

- If sampled using *sample_banded_model()* with reference measure $\Gamma(k, \theta)$

$$x \mapsto \frac{2x}{\theta} \quad \text{and} \quad x \mapsto \frac{x}{\beta M}$$

- If sampled using *sample_full_model()*

$$x \mapsto \frac{x}{\beta M}$$

Note: This method is called in *plot()* and *hist()* when `normalization=True`.

plot (*normalization=True*)

Display the last realization of the *LaguerreEnsemble* object

Parameters **normalization** (bool, default `True`) – When `True`, the points are first normalized (see *normalize_points()*) so that they concentrate as

- If `beta` $= 0$, the $\Gamma(k, 2)$ reference measure associated to full *full matrix model*
- If `beta` > 0, the limiting distribution, i.e., the Marcenko-Pastur distribution

in both cases the corresponding p.d.f. is displayed

See also:

- *sample_full_model()*, *sample_banded_model()*
- *normalize_points()*
- *hist()*
- *Full matrix model* associated to the Laguerre ensemble
- *Tridiagonal matrix model* associated to the Laguerre ensemble

sample_banded_model (*shape=1.0*, *scale=2.0*, *size_N=10*, *size_M=None*, *random_state=None*)

Sample from *tridiagonal matrix model* associated to the Laguerre ensemble. Available for `beta` > 0 and the degenerate case `beta` $= 0$ corresponding to i.i.d. points from the $\Gamma(k, \theta)$ reference measure

Parameters

- **shape** (float, default 1) – Shape parameter k of $\Gamma(k, \theta)$ reference measure
- **scale** (float, default 2.0) – Scale parameter θ of $\Gamma(k, \theta)$ reference measure
- **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized. Equivalent to the first dimension N of the matrix used to form the covariance matrix in the *full matrix model*.
- **size_M** (*int, default None*) – Equivalent to the second dimension M of the matrix used to form the covariance matrix in the *full matrix model*.

- If `size_M` is not provided:

  In the `sampling_params`, `size_M` is set to `size_M`$= \frac{2k}{\beta} + N - 1$, to give an idea of the corresponding second dimension M.

- If `size_M` is provided:

  In the `sampling_params`, `shape` and `scale` are set to: `shape`$= \frac{1}{2}\beta(M - N + 1)$ and `scale`$= 2$

Note: The reference measure associated with the *full matrix model* is $\Gamma\left(\frac{\beta}{2}(M-N+1), 2\right)$. This explains the role of the `size_M` parameter.

To compare `sample_banded_model()` with `sample_full_model()` simply use the `size_N` and `size_M` parameters

See also:

- *Tridiagonal matrix model* associated to the Laguerre ensemble
- [DE02] III-B
- `sample_full_model()`

sample_full_model (*size_N=10*, *size_M=100*, *random_state=None*)

Sample from *full matrix model* associated to the Laguerre ensemble. Only available for `beta` $\in \{1, 2, 4\}$ and the degenerate case `beta` $= 0$ corresponding to i.i.d. points from the $\Gamma(k, \theta)$ reference measure

Parameters

- **size_N** (int, default 10) – Number N of points, i.e., size of the matrix to be diagonalized. First dimension of the matrix used to form the covariance matrix to be diagonalized, see *full matrix model*.
- **size_M** (int, default 100) – Second dimension M of the matrix used to form the covariance matrix to be diagonalized, see *full matrix model*.

Note: The reference measure associated with the *full matrix model* is $\Gamma\left(\frac{\beta}{2}(M-N+1), 2\right)$. For this reason, in the `sampling_params`, the values of the parameters are set to `shape`$= \frac{\beta}{2}(M-N+1)$ and `scale`$= 2$.

To compare `sample_banded_model()` with `sample_full_model()` simply use the `size_N` and `size_M` parameters.

See also:

- *Full matrix model* associated to the Laguerre ensemble
- `sample_banded_model()`

3.3 Exotic DPPs

3.3.1 Uniform Spanning Trees

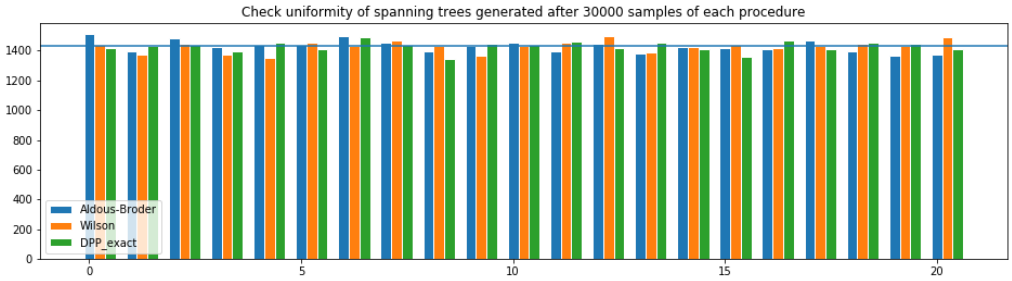
The Uniform measure on Spanning Trees (UST) of a directed connected graph corresponds to a projection DPP with kernel the transfer current matrix of the graph. The later is actually the orthogonal projection matrix onto the row span of the vertex-edge incidence matrix. In fact, one can discard any row of the vertex-edge incidence matrix - note A the resulting matrix - to compute $\mathbf{K} = A^\top [AA^\top]^{-1} A$.

See also:

- *UST*
- Wilson algorithm [PW98]

- Aldous-Broder [Ald90]
- [Lyo02]

Important: DPPy uses the `networkx` library to handle DPPs related to trees and graphs, but `networkx` is not installed by default when installing DPPy. Please refer to the installation instructions on GitHub for more details on how to install the necessary dependencies.




```
from networkx import Graph
from dppy.exotic_dpps import UST


# Build graph
g = Graph()
edges = [(0, 2), (0, 3), (1, 2), (1, 4), (2, 3), (2, 4), (3, 4)]
g.add_edges_from(edges)

# Initialize UST object
ust = UST(g)
```

```
# Display underlying kernel i.e. transfer current matrix
ust.plot_graph()
```

```
# Display underlying kernel i.e. transfer current matrix
ust.plot_kernel()
```

```
# Display some samples
for md in ('Wilson', 'Aldous-Broder', 'GS'):
    ust.sample(md)
    ust.plot()
```

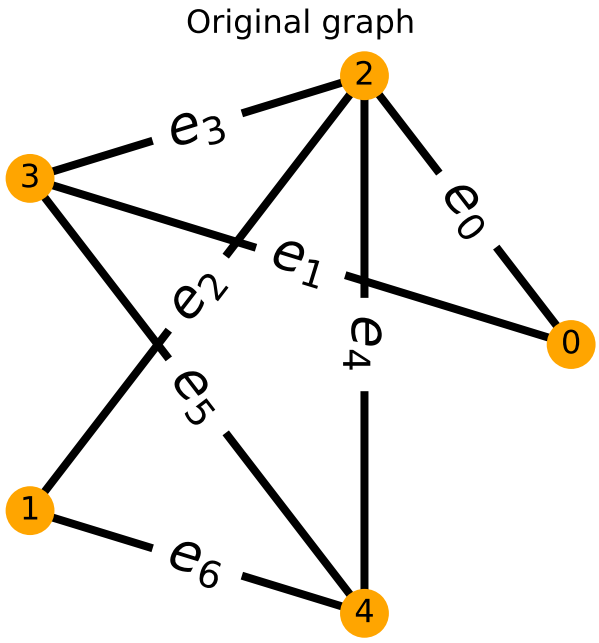
3.3.2 Stationary 1-dependent process

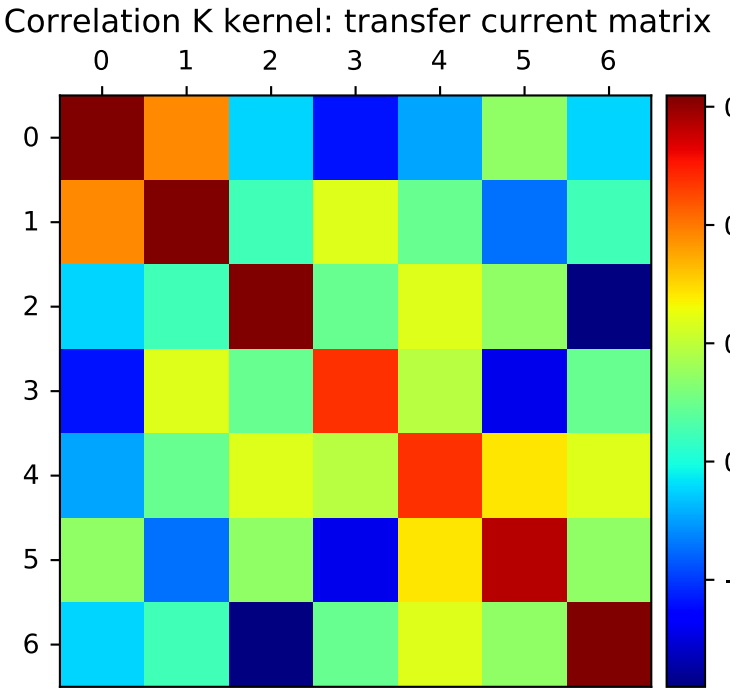
A point process $\mathcal{X}$ on $\mathbb{Z}$ (resp. $\mathbb{N}$) is called 1-dependent if for any $A, B \subset \mathbb{Z}$ (resp. $\mathbb{N}$), such as the distance between A and B is greater than 1,

$$\mathbb{P}(A \cup B \subset \mathcal{X}) = \mathbb{P}(A \subset \mathcal{X})\mathbb{P}(B \subset \mathcal{X}).$$

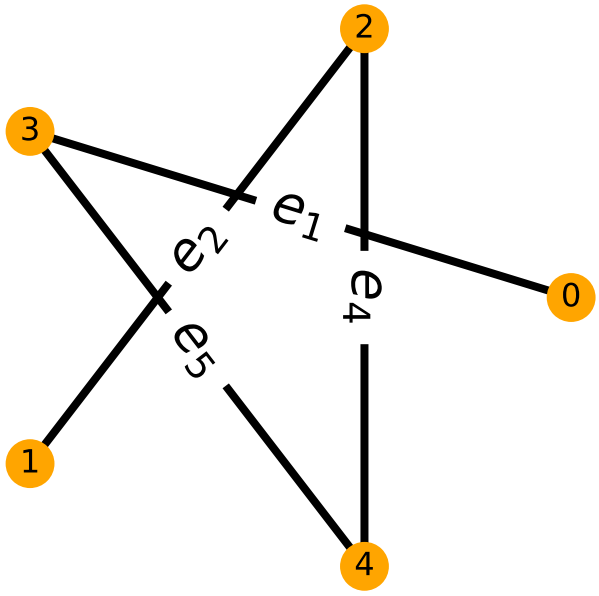
If $\mathcal{X}$ is stationary and 1-dependent then $\mathcal{X}$ forms a DPP.

The following 3 examples are stationary and 1-dependent process.

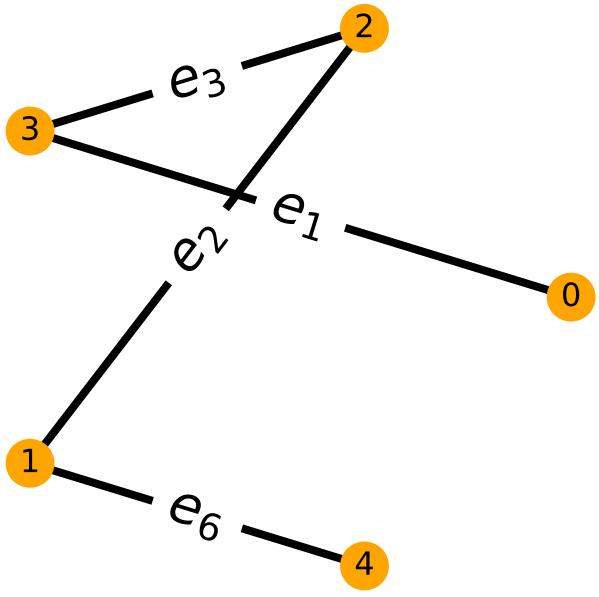

Original graph
2
3
0
1
4
e_3
e_0
e_1
e_2
e_4
e_5
e_6

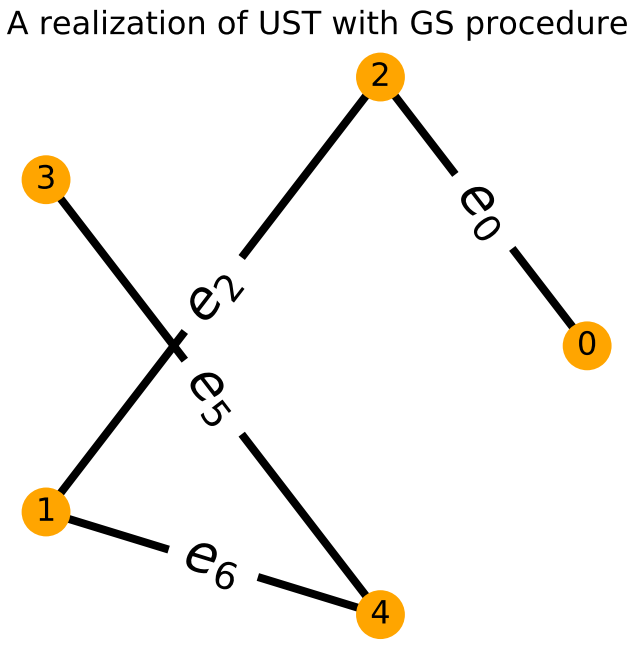

Correlation K kernel: transfer current matrix
0
1
2
3
4
5
6

A realization of UST with Wilson procedure



A realization of UST with Aldous-Broder procedu





Carries process

The sequence of carries appearing when computing the cumulative sum (in base b) of a sequence of i.i.d. digits forms a DPP on $\mathbb{N}$ with non symmetric kernel.

```
from dppy.exotic_dpps import CarriesProcess


base = 10  # base
cp = CarriesProcess(base)

size = 100
cp.sample(size)

cp.plot(vs_bernoullis=True)
```

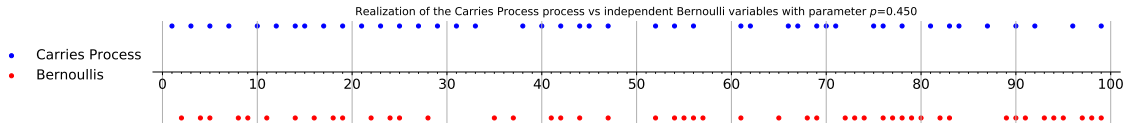


Fig. 3.22: Carries process

See also:

- *CarriesProcess*
- [BDF10]

Descent process

The descent process obtained from a uniformly chosen permutation of $\{1, 2, \dots, n\}$ forms a DPP on $\{1, 2, \dots, n-1\}$ with non symmetric kernel. It can be seen as the limit of the carries process as the base goes to infinity.

```
from dppy.exotic_dpps import DescentProcess


dp = DescentProcess()

size = 100
dp.sample(size)

dp.plot(vs_bernoullis=True)
```

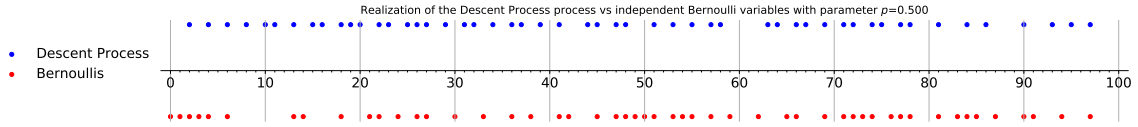


Fig. 3.23: Descent process

See also:

- *DescentProcess*
- [BDF10]

Limiting Descent process for virtual permutations

For non uniform permutations the descent process is not necessarily determinantal but in the particular case of virtual permutations with law stable under conjugation of the symmetric group the limiting descent process is a mixture of determinantal point processes.

```
from dppy.exotic_dpps import VirtualDescentProcess


vdp = VirtualDescentProcess(x_0=0.5)

size = 100
vdp.sample(size)

vdp.plot(vs_bernoullis=True)
```

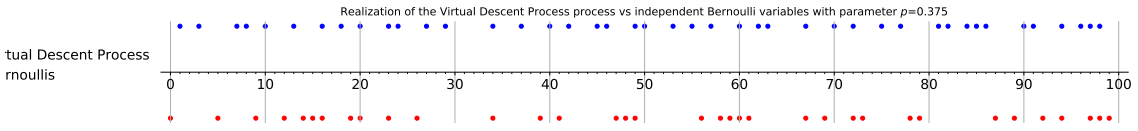


Fig. 3.24: Virtual descent process

See also:

- *VirtualDescentProcess*
- [Kam18]

3.3.3 Poissonized Plancherel measure

The poissonized Plancherel measure is a measure on partitions $\lambda = (\lambda_1 \geq \lambda_2 \geq \cdots \geq 0) \in \mathbb{N}^{\mathbb{N}^*}$. Samples from this measure can be obtained in the following way

- Sample $N \sim \mathcal{P}(\theta)$
- Sample a uniform permutation $\sigma \in \mathfrak{S}_N$
- Compute the sorting tableau P associated to the RSK (Robinson-Schensted-Knuth correspondence) applied to σ
- Consider only the shape λ of P.

Finally, the point process formed by $\{\lambda_i - i + \frac{1}{2}\}_{i \geq 1}$ is a DPP on $\mathbb{Z} + \frac{1}{2}$.

```
from dppy.exotic_dpps import PoissonizedPlancherel


theta = 500  # Poisson parameter
pp = PoissonizedPlancherel(theta=theta)
pp.sample()
pp.plot_diagram(normalization=True)
```

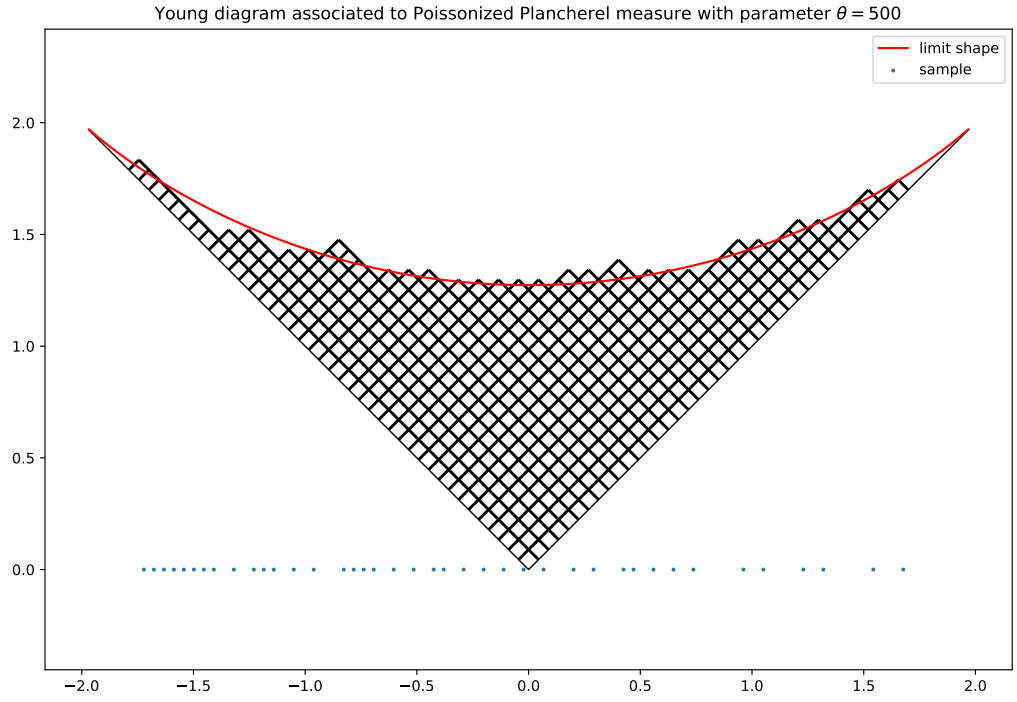


Fig. 3.25: Poissonized Plancherel measure

See also:

- *PoissonizedPlancherel*
- [Bor09] Section 6

3.3.4 API

Implementation of exotic DPP objects:

- Uniform spanning trees *UST*
- Descent procresses `Descent`:
  - *CarriesProcess*
  - *DescentProcess*
  - *VirtualDescentProcess*
- *PoissonizedPlancherel* measure

class dppy.exotic_dpps.**CarriesProcess**(*base=10*)
Bases: dppy.exotic_dpps.Descent

DPP on $\{1, \dots, N-1\}$ (with a non symmetric kernel) derived from the cumulative sum of N i.i.d. digits in $\{0, \dots, b-1\}$.

Parameters **base** (*int, default 10*) – Base/radix

See also:

- [BDF10]
- *Carries process*

flush_samples()
Empty the `list_of_samples` attribute.

plot(*vs_bernoullis=True*, *random_state=None*)
Display the last realization of the process. If `vs_bernoullis=True` compare it to a sequence of i.i.d. Bernoullis with parameter `_bernoulli_param`

See also:

- *sample()*

sample(*size=100*, *random_state=None*)
Compute the cumulative sum (in base b) of a sequence of i.i.d. digits and record the position of carries.

Parameters **size** (*int*) – size of the sequence of i.i.d. digits in $\{0, \dots, b-1\}$

class dppy.exotic_dpps.**DescentProcess**
Bases: dppy.exotic_dpps.Descent

DPP on $\{1, \dots, N-1\}$ associated to the descent process on the symmetric group $\mathfrak{S}_N$.

See also:

- [BDF10]
- *Descent process*

flush_samples()
Empty the `list_of_samples` attribute.

plot(*vs_bernoullis=True*, *random_state=None*)
Display the last realization of the process. If `vs_bernoullis=True` compare it to a sequence of i.i.d. Bernoullis with parameter `_bernoulli_param`

See also:

- *sample()*

sample (*size=100*, *random_state=None*)
Draw a permutation $\sigma \in \mathfrak{S}_N$ uniformly at random and record the descents i.e. $\{i\ ;\ \sigma_i > \sigma_{i+1}\}$.

Parameters **size** (*int*) – size of the permutation i.e. degree N of $\mathfrak{S}_N$.

class dppy.exotic_dpps.**PoissonizedPlancherel** (*theta=10*)
Bases: object

DPP on partitions associated to the Poissonized Plancherel measure

Parameters **theta** (*int, default 10*) – Poisson parameter i.e. expected length of permutation

See also:

- [Bor09] Section 6
- *Poissonized Plancherel measure*

plot (*title=''*)
Display the process on the real line

Parameters **title** (*string*) – Plot title

See also:

- *sample()*

plot_diagram (*normalization=False*)
Display the Young diagram (russian convention), the associated sample and potentially rescale the two to visualize the limit-shape theorem [Ker96]. The sample corresponds to the projection onto the real line of the descending surface edges.

Parameters **normalization** (*bool, default False*) – If normalization=True, the Young diagram and the corresponding sample are scaled by a factor $\sqrt{\theta}$ and the limiting

See also:

- *sample()*
- *plot()*
- [Ker96]

sample (*random_state=None*)
Sample from the Poissonized Plancherel measure.

Parameters **random_state** (*None, np.random, int, np.random.RandomState*) –

class dppy.exotic_dpps.**UST** (*graph*)
Bases: object

DPP on edges of a connected graph G with correlation kernel the projection kernel onto the span of the rows of the incidence matrix Inc of G.

This DPP corresponds to the uniform measure on spanning trees (UST) of G.

Parameters **graph** (*networkx graph*) – Connected undirected graph

See also:

- *Uniform Spanning Trees*
- *Definition of DPP*

compute_kernel()
Compute the orthogonal projection kernel $\mathbf{K} = \text{Inc}^{+}\text{Inc}$ i.e. onto the span of the rows of the vertex-edge incidence matrix Inc of size $|V| \times |E|$.

In fact, for a connected graph, Inc has rank $|V| - 1$ and any row can be discarded to get an basis of row space. If we note A the amputated version of Inc, then $\text{Inc}^{+} = A^{\top}[AA^{\top}]^{-1}$.

In practice, we orthogonalize the rows of A to get the eigenvectors U of $\mathbf{K} = UU^{\top}$.

See also:

- *plot_kernel()*

compute_kernel_eig_vecs()
See explaination in *compute_kernel*

flush_samples()
Empty the `list_of_samples` attribute.

plot(*title=''*)
Display the last realization (spanning tree) of the corresponding *UST* object.

Parameters **title** (*string*) – Plot title

See also:

- *sample()*

plot_graph(*title=''*)
Display the original graph defining the *UST* object

Parameters **title** (*string*) – Plot title

See also:

- *compute_kernel*

plot_kernel(*title=''*)
Display a heatmap of the underlying orthogonal projection kernel $\mathbf{K}$ associated to the DPP underlying the *UST* object

Parameters **title** (*string*) – Plot title

See also:

- *compute_kernel*

sample(*mode='Wilson'*, *root=None*, *random_state=None*)
Sample a spanning of the underlying graph uniformly at random. It generates a networkx graph object.

Parameters

- **mode** (string, default `'Wilson'`) – Markov-chain-based samplers:
  - `'Wilson'`, `'Aldous-Broder'`

  Chain-rule-based samplers:
  - `'GS'`, `'GS_bis'`, `'KuTa12'` from eigenvectors
  - `'Schur'`, `'Chol'`, from $\mathbf{K}$ correlation kernel

- **root** (*int*) – Starting node of the random walk when using Markov-chain-based samplers
- **random_state** (*None, np.random, int, np.random.RandomState*) –

See also:

- Wilson [PW98]
- Aldous-Broder [Ald90]
- `sample()`

class dppy.exotic_dpps.**VirtualDescentProcess**(*x_0=0.5*)

Bases: dppy.exotic_dpps.Descent

This is a DPP on $\{1, \dots, N-1\}$ with a non symmetric kernel appearing in (or as a limit of) the descent process on the symmetric group $\mathfrak{S}_N$.

See also:

- [Kam18]
- *Limiting Descent process for virtual permutations*
- *DescentProcess*

flush_samples()

Empty the `list_of_samples` attribute.

plot(*vs_bernoullis=True*, *random_state=None*)

Display the last realization of the process. If `vs_bernoullis=True` compare it to a sequence of i.i.d. Bernoullis with parameter `_bernoulli_param`

See also:

- *sample()*

sample(*size=100*, *random_state=None*)

Draw a permutation uniformly at random and record the descents i.e. indices where $\sigma(i+1) < \sigma(i)$ and something else...

Parameters **size** (*int*) – size of the permutation i.e. degree N of $\mathfrak{S}_N$.

See also:

- [Kam18], Sec ??

Todo: ask @kammmoun to complete the docsting and Section in see also

3.4 Bibliography

BIBLIOGRAPHY

[AKFT13] Raja Hafiz Affandi, Alex Kulesza, Emily B Fox, and Ben Taskar. Nyström Approximation for Large-Scale Determinantal Processes. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 31, 85–98. 2013. URL: http://proceedings.mlr.press/v31/affandi13a.

[AM15] Ahmed El Alaoui and Michael W. Mahoney. Fast randomized kernel ridge regression with statistical guarantees. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, 775–783. Montreal, Canada, December 2015.

[Ald90] David J Aldous. The Random Walk Construction of Uniform Spanning Trees and Uniform Labelled Trees. *SIAM Journal on Discrete Mathematics*, 3(4):450–465, nov 1990. URL: http://epubs.siam.org/doi/10.1137/0403039, doi:10.1137/0403039.

[AGR16] Nima Anari, Shayan Oveis Gharan, and Alireza Rezaei. Monte Carlo Markov Chain Algorithms for Sampling Strongly Rayleigh Distributions and Determinantal Point Processes. In *Conference on Learning Theory (COLT)*, 103–115. New York, USA, 2016. PMLR. URL: http://proceedings.mlr.press/v49/anari16, arXiv:1602.05242.

[AGaudilliere13] Luca Avena and Alexandre Gaudillière. On some random forests with determinantal roots. *e-prints*, 2013. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.740.6173\protect\T1\textbraceleft\T1\textbackslash{}&\protect\T1\textbracerightrep=rep1\protect\T1\textbraceleft\T1\textbackslash{}&\protect\T1\textbracerighttype=pdf.

[BT05] Adrian Baddeley and Rolf Turner. spatstat : An R Package for Analyzing Spatial Point Patterns. *Journal of Statistical Software*, 12(6):1–42, jan 2005. URL: http://www.jstatsoft.org/v12/i06/, doi:10.18637/jss.v012.i06.

[BH16] Rémi Bardenet and Adrien Hardy. Monte Carlo with Determinantal Point Processes. *ArXiv e-prints*, 2016. URL: http://arxiv.org/abs/1605.00361, arXiv:1605.00361.

[Bor09] Alexei Borodin. Determinantal point processes. *ArXiv e-prints*, 2009. URL: http://arxiv.org/abs/0911.1153, arXiv:0911.1153.

[BDF10] Alexei Borodin, Persi Diaconis, and Jason Fulman. On adding a list of numbers (and other one-dependent determinantal processes). *Bulletin of the American Mathematical Society*, 47(4):639–670, 2010. URL: http://www.ams.org/journals/bull/2010-47-04/S0273-0979-2010-01306-9/S0273-0979-2010-01306-9.pdf, arXiv:0904.3740.

[BRW19] David Burt, Carl Edward Rasmussen, and Mark Van Der Wilk. Rates of Convergence for Sparse Variational Gaussian Process Regression. In *International Conference on Machine Learning (ICML)*, 862–871. may 2019. URL: http://proceedings.mlr.press/v97/burt19a.html, arXiv:1903.03571.

[CDerezinskiV20] Daniele Calandriello, Michal Dereziński, and Michal Valko. Sampling from a k-DPP without looking at all items. In *Advances in Neural Information Processing Systems*. 2020.

[CLV17] Daniele Calandriello, Alessandro Lazaric, and Michal Valko. Distributed adaptive sampling for kernel matrix approximation. In *Artificial Intelligence and Statistics*, 1421–1429. 2017.

[DVJ03] Daryl J. Daley and David Vere-Jones. *An Introduction to the Theory of Point Processes. Volume I: Elementary Theory and Methods*. Probability and its Applications. Springer-Verlag New York, New York, USA, 2 edition, 2003. ISBN 0-387-95541-0. URL: http://link.springer.com/10.1007/b97277, doi:10.1007/b97277.

[DFL13] Laurent Decreusefond, Ian Flint, and Kah Choon Low. Perfect Simulation of Determinantal Point Processes. *ArXiv e-prints*, 2013. URL: http://arxiv.org/abs/1311.1027, arXiv:1311.1027.

[DWH18] Michal Derezinski, Manfred K Warmuth, and Daniel J Hsu. Leveraged volume sampling for linear regression. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2505–2514. Curran Associates, Inc., 2018. URL: http://papers.nips.cc/paper/7517-leveraged-volume-sampling-for-linear-regression.pdf.

[Derezinski19] Michał Dereziński. Fast determinantal point processes via distortion-free intermediate sampling. In Alina Beygelzimer and Daniel Hsu, editors, *Proceedings of the Thirty-Second Conference on Learning Theory*, volume 99 of Proceedings of Machine Learning Research, 1029–1049. Phoenix, USA, 25–28 Jun 2019. PMLR. URL: http://proceedings.mlr.press/v99/derezinski19a.html.

[DerezinskiCV19] Michal Dereziński, Daniele Calandriello, and Michal Valko. Exact sampling of determinantal point processes with sublinear time preprocessing. In *Advances in Neural Information Processing Systems*. 2019.

[DE15] Alexander Dubbs and Alan Edelman. Infinite Random Matrix Theory, Tridiagonal Bordered Toeplitz Matrices, and the Moment Problem. *Linear Algebra and its Applications*, 467:188–201, 2015. arXiv:1502.04931, doi:10.1016/j.laa.2014.11.006.

[DE02] Ioana Dumitriu and Alan Edelman. Matrix Models for Beta Ensembles. *Journal of Mathematical Physics*, 43(11):5830–5847, 2002. URL: https://sites.math.washington.edu/\protect\T1\textbraceleft~\protect\T1\textbracerightdumitriu/JMathPhys\protect\T1\textbraceleft\T1\textbackslash{}_\protect\T1\textbraceright43\protect\T1\textbraceleft\T1\textbackslash{}_\protect\T1\textbraceright5830.pdf, arXiv:0206043, doi:10.1063/1.1507823.

[DB18] Christophe Dupuy and Francis Bach. Learning Determinantal Point Processes in Sublinear Time. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 84, 244–257. Lanzarote, Spain, 2018. PMLR. URL: http://proceedings.mlr.press/v84/dupuy18a, arXiv:1610.05925.

[GBDK19] Mike Gartrell, Victor-Emmanuel Brunel, Elvis Dohmatob, and Syrine Krichene. Learning Nonsymmetric Determinantal Point Processes. *ArXiv e-prints*, may 2019. URL: http://arxiv.org/abs/1905.12962, arXiv:1905.12962.

[GPK16] Mike Gartrell, Ulrich Paquet, and Noam Koenigstein. Low-Rank Factorization of Determinantal Point Processes for Recommendation. In *AAAI Conference on Artificial Intelligence*, 1912–1918. 2016. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/download/14657/14354, arXiv:1602.05436.

[GBV17] Guillaume Gautier, Rémi Bardenet, and Michal Valko. Zonotope hit-and-run for efficient sampling from projection DPPs. *International Conference on Machine Learning (ICML)*, pages 1223–1232, may 2017. URL: http://proceedings.mlr.press/v70/gautier17a, arXiv:1705.10498.

[GBV19] Guillaume Gautier, Rémi Bardenet, and Michal Valko. On two ways to use determinantal point processes for Monte Carlo integration. In *Neural Information Processing Systems (NeurIPS)*. 2019. URL:.

[GPBV19] Guillaume Gautier, Guillermo Polito, Rémi Bardenet, and Michal Valko. DPPy: DPP Sampling with Python. *Journal of Machine Learning Research - Machine Learning Open Source Software (JMLR-MLOSS), in press*, 2019.

[Gau09] Walter Gautschi. How sharp is Bernstein's Inequality for Jacobi polynomials? *Electronic Transactions on Numerical Analysis*, 36:1–8, 2009. URL: http://emis.ams.org/journals/ETNA/vol.36.2009-2010/pp1-8.dir/pp1-8.pdf.

[Gil14] Jennifer Gillenwater. *Approximate inference for determinantal point processes*. PhD thesis, University of Pennsylvania, 2014. URL: https://repository.upenn.edu/edissertations/1285.

[HKPVirag06] J. Ben Hough, Manjunath Krishnapur, Yuval Peres, and Bálint Virág. Determinantal Processes and Independence. In *Probability Surveys*, volume 3, 206–229. The Institute of Mathematical Statistics and the Bernoulli Society, 2006. URL: http://arxiv.org/abs/math/0503110, arXiv:0503110, doi:10.1214/154957806000000078.

[Joh06] Kurt Johansson. Random matrices and determinantal processes. *Les Houches Summer School Proceedings*, 83(C):1–56, 2006. arXiv:0510038, doi:10.1016/S0924-8099(06)80038-7.

[Kam18] Mohamed Slim Kammoun. Monotonous subsequences and the descent process of invariant random permutations. *Electronic Journal of Probability*, 2018. URL: https://projecteuclid.org/euclid.ejp/1543287754, arXiv:1805.05253, doi:10.1214/18-EJP244.

[KDK16] Tarun Kathuria, Amit Deshpande, and Pushmeet Kohli. Batched Gaussian Process Bandit Optimization via Determinantal Point Processes. In *Neural Information Processing Systems (NIPS)*, 4206–4214. 2016. URL: http://papers.nips.cc/paper/6452-batched-gaussian-process-bandit-optimization-via-determinantal-point-processes, arXiv:1611.04088.

[Ker96] Sergei Kerov. A Differential Model Of Growth Of Young Diagrams. *Proceedings of St.Petersburg Mathematical Society*, 1996. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.7744.

[KN04] Rowan Killip and Irina Nenciu. Matrix models for circular ensembles. *International Mathematics Research Notices*, 2004(50):2665, 2004. URL: https://academic.oup.com/imrn/article-lookup/doi/10.1155/S1073792804141597, arXiv:0410034, doi:10.1155/S1073792804141597.

[KT12] Alex Kulesza and Ben Taskar. Determinantal Point Processes for Machine Learning. *Foundations and Trends in Machine Learning*, 5(2-3):123–286, 2012. URL: http://arxiv.org/abs/1207.6083, arXiv:1207.6083, doi:10.1561/2200000044.

[Konig04] Wolfgang König. Orthogonal polynomial ensembles in probability theory. *Probab. Surveys*, 2:385–447, 2004. URL: http://arxiv.org/abs/math/0403090, arXiv:0403090, doi:10.1214/154957805100000177.

[LGD18] Claire Launay, Bruno Galerne, and Agnès Desolneux. Exact Sampling of Determinantal Point Processes without Eigendecomposition. *ArXiv e-prints*, feb 2018. URL: http://arxiv.org/abs/1802.08429, arXiv:1802.08429.

[LMollerR12] Frédéric Lavancier, Jesper Møller, and Ege Rubak. Determinantal point process models and statistical inference : Extended version. *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, 77(4):853–877, may 2012. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/rssb.12096, arXiv:1205.4818, doi:10.1111/rssb.12096.

[LJS16a] Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Efficient Sampling for k-Determinantal Point Processes. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 1328–1337. Cadiz, Spain, 2016. URL: http://proceedings.mlr.press/v51/li16f, arXiv:1509.01618.

[LJS16b] Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Fast DPP Sampling for Nyström with Application to Kernel Methods. In *International Conference on Machine Learning (ICML)*, 2061–2070. New York, USA, 2016. URL: http://proceedings.mlr.press/v48/lih16, arXiv:1603.06052.

[LJS16c] Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Fast Mixing Markov Chains for Strongly Rayleigh Measures, DPPs, and Constrained Sampling. In *Neural Information Processing Systems (NIPS)*, 4188–4196. Barcelona, Spain, 2016. URL: https://papers.nips.cc/paper/6182-fast-mixing-markov-chains-for-strongly-rayleigh-measures-dpps-and-constrained-sampling, arXiv:1608.01008.

[LJS16d] Chengtao Li, Stefanie Jegelka, and Suvrit Sra. Fast Sampling for Strongly Rayleigh Measures with Application to Determinantal Point Processes. *ArXiv e-prints*, 2016. URL: http://arxiv.org/abs/1607.03559, arXiv:1607.03559.

[Lyo02] Russell Lyons. Determinantal probability measures. *Publications mathématiques de l'IHÉS*, 98(1):167–212, apr 2002. URL: http://link.springer.com/10.1007/s10240-003-0016-0, arXiv:0204325, doi:10.1007/s10240-003-0016-0.

[Mac75] Odile Macchi. The coincidence approach to stochastic point processes. *Advances in Applied Probability*, 7(01):83–122, 1975. URL: https://www.cambridge.org/core/product/identifier/S0001867800040313/type/journal\protect\T1\textbraceleft\T1\textbackslash{}_\protect\T1\textbracerightarticle, doi:10.2307/1425855.

[MCA19] Adrien Mazoyer, Jean-François Coeurjolly, and Pierre-Olivier Amblard. Projections of determinantal point processes. *ArXiv e-prints*, 2019. URL: https://arxiv.org/pdf/1901.02099.pdf, arXiv:1901.02099v3.

[Mez06] Francesco Mezzadri. How to generate random matrices from the classical compact groups. *Notices of the American Mathematical Society*, 54:592–604, sep 2006. URL: http://arxiv.org/abs/math-ph/0609050, arXiv:0609050.

[MollerW04] Jesper. Møller and Rasmus Plenge. Waagepetersen. *Statistical inference and simulation for spatial point processes*. Volume 23. Chapman & Hall/CRC, 2004. ISBN 1584882654. URL: https://www.crcpress.com/Statistical-Inference-and-Simulation-for-Spatial-Point-Processes/Moller-Waagepetersen/p/book/9781584882657, doi:10.1201/9780203496930.

[PB11] Raj K. Pathria and Paul D. Beale. *Statistical Mechanics*. Academic Press, 2011. ISBN 0123821894. URL: http://linkinghub.elsevier.com/retrieve/pii/B9780123821881000207, doi:10.1016/B978-0-12-382188-1.00020-7.

[Pou19] Jack Poulson. High-performance sampling of generic Determinantal Point Processes. *ArXiv e-prints*, apr 2019. URL: http://arxiv.org/abs/1905.00165, arXiv:1905.00165.

[PW98] James Gary Propp and David Bruce Wilson. How to Get a Perfectly Random Sample from a Generic Markov Chain and Generate a Random Spanning Tree of a Directed Graph. *Journal of Algorithms*, 27(2):170–217, may 1998. URL: https://www.sciencedirect.com/science/article/pii/S0196677497909172, doi:10.1006/JAGM.1997.0917.

[RW06] Carl Edward. Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. MIT Press, 2006. ISBN 026218253X. URL: http://www.gaussianprocess.org/gpml/.

[RCCR18] Alessandro Rudi, Daniele Calandriello, Luigi Carratino, and Lorenzo Rosasco. On fast leverage score sampling and optimal learning. In *Advances in Neural Information Processing Systems 31*, pages 5672–5682. 2018.

[Sos00] Alexander Soshnikov. Determinantal random point fields. *Russian Mathematical Surveys*, 55(5):923–975, feb 2000. URL: http://dx.doi.org/10.1070/RM2000v055n05ABEH000321, arXiv:0002099, doi:10.1070/RM2000v055n05ABEH000321.

[TAB17] Nicolas Tremblay, Pierre-Olivier Amblard, and Simon Barthelme. Graph sampling with determinantal processes. In *European Signal Processing Conference (EUSIPCO)*, 1674–1678. IEEE, aug 2017. URL: http://ieeexplore.ieee.org/document/8081494/, arXiv:1703.01594, doi:10.23919/EUSIPCO.2017.8081494.

[TBA18] Nicolas Tremblay, Simon Barthelme, and Pierre-Olivier Amblard. Optimized Algorithms to Sample Determinantal Point Processes. *ArXiv e-prints*, feb 2018. URL: http://arxiv.org/abs/1802.08471, arXiv:1802.08471.

[Wig67] Eugene P. Wigner. Random Matrices in Physics. *SIAM Review*, 9(1):1–23, 1967. doi:10.1137/1009001.

PYTHON MODULE INDEX

d

INDEX

U

V