A Faster Sampler for Discrete Determinantal Point Processes

Simon Barthelmé **Nicolas Tremblay** **Pierre-Olivier Amblard**
CNRS, Univ. Grenoble Alpes, Grenoble-INP, Gipsa-lab, Grenoble, France

Abstract

Discrete Determinantal Point Processes (DPPs) have a wide array of potential applications for subsampling datasets. They are however held back in some cases by the high cost of sampling. In the worst-case scenario, the sampling cost scales as $\mathcal{O}(n^3)$ where n is the number of elements of the ground set. A popular workaround to this prohibitive cost is to sample DPPs defined by low-rank kernels. In such cases, the cost of standard sampling algorithms scales as $\mathcal{O}(np^2 + nm^2)$ where m is the (average) number of samples of the DPP (usually $m \ll n$) and p the rank of the kernel used to define the DPP ($m \leq p \leq n$). The first term, $\mathcal{O}(np^2)$, comes from a SVD-like step. We focus here on the second term of this cost, $\mathcal{O}(nm^2)$, and show that it can be brought down to $\mathcal{O}(nm + m^3 \log m)$ without loss on the sampling's exactness. In practice, we observe very substantial speedups compared to the classical algorithm as soon as $n > 1,000$. The algorithm described here is a close variant of the standard algorithm for sampling continuous DPPs, and uses rejection sampling. In the specific case of projection DPPs, we also show that any additional sample can be drawn in time $\mathcal{O}(m^3 \log m)$. Finally, an interesting by-product of the analysis is that a realisation from a DPP is typically contained in a subset of size $\mathcal{O}(m \log m)$ formed using leverage score i.i.d. sampling.

Discrete Determinantal Point Processes have been advocated as a way of subsampling large datasets, because they produce samples that preserve some of the diversity of the original dataset (Kulesza et al., 2012). One impediment to their broad adoption in practice lies in their computational cost; aside from some special cases (*e.g.*, random spanning forests (Avena and Gaudillière, 2017)), exact sampling of a DPP with a large number of elements is rather expensive.

In this manuscript, we show that a simple modification of the standard algorithm yields a substantial improvement, without loss on the algorithm's exactness. The modification we suggest consists in using a form of rejection sampling. The idea is not very original, and indeed appears in works by Lavancier et al. (2015) for continuous DPPs, and more recently by Dereziński et al. (2019b) in the context of experimental design. What is striking is how effective this modification can be, when sampling discrete DPPs, especially given how easy it is to implement.

Here and throughout, let n designate the size of the ground set the DPP draws from, and m be the (average) size of the subsample produced by the DPP (usually $m \ll n$). In the worst-case, the cost of producing a sample may be as high as $\mathcal{O}(n^3)$, as it requires a full diagonalisation of the kernel (Hough et al., 2006). However, in the more realistic context of low-rank kernels and using standard exact sampling algorithms, this figure drops to $\mathcal{O}(np^2 + nm^2)$ where p is the rank of the kernel ($m \leq p \leq n$) (Gillenwater, 2014). We improve this to $\mathcal{O}(np^2 + nm + m^3 \log m)$. We readily see that, even though this is an improvement for sampling any DPP, the closer is p to m, the more substantial the improvement in practice, as np^2 stays the headline complexity. We identify three popular and general use-cases for which rejection sampling brings substantial speed-ups compared to the classical algorithm:

1. *(very significant speed-up: $p = m$ and orthogonalisation is already computed)* sample a DPP with kernel $\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{Q}^\top$, where $\boldsymbol{Q} \in \mathbb{R}^{n \times m}$ is orthonormal ($\boldsymbol{Q}^\top\boldsymbol{Q} = \boldsymbol{I}$) and given (for instance, the DPPs used by Launay et al. (2021)). As there is no orthogonalisation to compute, the total cost using rejection sampling is $\mathcal{O}(nm + m^3 \log m)$, substantially faster than the usual cost in $\mathcal{O}(nm^2)$.
2. *(significant speed-up: $p = m$ and orthogonalisation has yet to be computed)* in some cases, the orthonormal basis $\boldsymbol{Q}$ is not known from the start. A popular context is when one wishes to sample a fixed-size L-ensemble of size m with $\boldsymbol{L} = \boldsymbol{V}\boldsymbol{V}^\top$ with $\boldsymbol{V} \in \mathbb{R}^{n \times m}$ a matrix of features. In this case, one i/ first computes an orthonormal basis $\boldsymbol{Q}$ of the span of $\boldsymbol{V}$, before

Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS) 2023, Valencia, Spain. PMLR: Volume 206.

ii/ sampling a DPP with kernel $\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{Q}^\top$ (as in the previous case). Step i/ involves, e.g., a QR decomposition. Even though the cost of QR scales theoretically as $\mathcal{O}(nm^2)$, it is highly efficient (and parallelisable) in modern hardware such that the bottleneck in previous state-of-the-art is step ii/. Our improvement of step ii/ thus also has practical (possibly very large) speed-ups in this context. In addition, there are special cases of feature matrices $\boldsymbol{V}$ for which computing an orthogonal basis has cost less than $\mathcal{O}(nm^2)$; increasing further the potential benefits of our approach. This is the case, *e.g.*, for some classes of sparse $\boldsymbol{V}$ (Davis, 2008).

3. *(moderate speed-up: p equals a few times m)* Same context as 2/ but in the case where the feature matrix $\boldsymbol{V}$ is of size $n \times p$ with $p > m$. In this case, step i/ is to compute the SVD of $\boldsymbol{V}$. If p is too large, this will be the dominant step and our improvement over step ii/ will be negligible. However, in popular cases where p is only a few times m, the speed-up is appreciable (see Section 3 for details).

Moreover, in contexts where one needs several realisations of the same DPP, step i/ is computed once, and step ii/ as many times as the number of samples needed; such that our improvement over step ii/ becomes that much more useful.

Organisation of the paper. Section 1 briefly introduces the main objects and the state-of-the-art. Section 2 describes our algorithm and its runtime, and Section 3 presents empirical results. A corollary of our result states that DPPs are typically contained in a i.i.d. sample of size $\mathcal{O}(m \log m)$. Section 4 discusses this fact and offers concluding remarks.

1 Background and notation

1.1 Discrete DPPs

For more background on discrete DPPs, we refer readers to Kulesza et al. (2012) and Tremblay et al. (2023). Discrete DPPs are a specific instance of a *discrete point process*. We say $\mathcal{X}$ is a discrete point process on a ground set Ω, if it is a random subset of Ω. Without loss of generality, we let $\Omega = \{1, \dots, n\}$ so that $\mathcal{X}$ is a random subset of indices. Also, for two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ of same size, the notation $\boldsymbol{A} \preceq \boldsymbol{B}$ means that $\boldsymbol{B} - \boldsymbol{A}$ is positive semidefinite.

Definition 1.1 (DPP)**.** *$\mathcal{X}$ is a DPP on Ω with marginal kernel $\boldsymbol{K} \in \mathbb{R}^{n \times n}$ such that $\boldsymbol{0} \preceq \boldsymbol{K} \preceq \boldsymbol{I}$, noted $\mathcal{X} \sim DPP(\boldsymbol{K})$, if for all fixed subsets $S \subseteq \Omega$, we have*

$$p(S \subseteq \mathcal{X}) = \det \boldsymbol{K}_S \tag{1}$$

Here $\boldsymbol{K}_S$ is the principal submatrix of $\boldsymbol{K}$ with indices given by S. We shall use "Matlab" notation, where $\boldsymbol{K}_{A,B}$ denotes the submatrix with row indices A and column indices B, $\boldsymbol{K}_{A,:}$ means all columns and $\boldsymbol{K}_{:,B}$ all rows.

Definition 1.2 (Projection DPP)**.** *A projection DPP is a DPP whose kernel is a projection matrix (ie. $\boldsymbol{K}^2 = \boldsymbol{K}$).*

If $\boldsymbol{K}$ is a projection matrix, it can be written as $\boldsymbol{Q}\boldsymbol{Q}^\top$ where $\boldsymbol{Q} \in \mathbb{R}^{n \times m}$ is an orthonormal basis for $\operatorname{span} \boldsymbol{K}$ ($\boldsymbol{Q}^\top \boldsymbol{Q} = \boldsymbol{I}$ and $\operatorname{span} \boldsymbol{Q} = \operatorname{span} \boldsymbol{K}$). Note that any orthonormal basis for $\boldsymbol{K}$ is enough, $\boldsymbol{Q}$ need not be a basis of eigenvectors.

Projection DPPs are important because of the following mixture decomposition, due to Hough et al. (2006).

Theorem 1.3 (mixture representation)**.** *Let $\mathcal{X} \sim DPP(\boldsymbol{K})$, and $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$ the eigendecomposition of $\boldsymbol{K}$, with $\boldsymbol{\Lambda}$ the diagonal matrix of eigenvalues $\{\lambda_j\}_{j=1,\dots,n}$ and $\boldsymbol{U} = (\boldsymbol{u}_1 | \boldsymbol{u}_2 | \dots | \boldsymbol{u}_n)$ the matrix of eigenvectors. Then the following process produces a sample from $\mathcal{X}$:*

1. *Sample a subset $\mathcal{Y}$ of eigenvectors by including each eigenvector $\boldsymbol{u}_j$ with probability λ_j*
2. *Form the projection kernel $\boldsymbol{K}_\mathcal{Y} = \boldsymbol{U}_{:,\mathcal{Y}}(\boldsymbol{U}_{:,\mathcal{Y}})^\top$*
3. *Sample $\mathcal{X} \sim DPP(\boldsymbol{K}_\mathcal{Y})$*

The cost of sampling a DPP when following this recipe equals the cost of computing the eigendecomposition of $\boldsymbol{K}$ ($\mathcal{O}(np^2)$ with p the rank of $\boldsymbol{K}$) followed by the cost of sampling a projection DPP (step 3). It is the latter step that we focus on here.

1.2 Fixed-size DPPs

The cardinal of a DPP is in general random. Such varying-sized samples are not practical in many applications, which led Kulesza and Taskar (2011) to define fixed-size DPPs[1]

Definition 1.4 (Fixed-size DPP)**.** *A fixed size DPP of size m is a DPP $\mathcal{X}$ conditioned on $|\mathcal{X}| = m$.*

To sample a fixed-size DPP with kernel $\boldsymbol{K}$, one follows the same recipe as in Theorem 1.3 except for the first step that is replaced by:

1. *Sample a subset $\mathcal{Y}$ of eigenvectors by including each eigenvector $\boldsymbol{u}_j$ with probability λ_j;* conditioned on $|\mathcal{Y}| = m$.

Performing such a conditioned sampling can be done by Algorithm 8 of Kulesza et al. (2012), which works by explicitly computing elementary polynomials. If n and/or m are too large, numerical instabilities usually arise with this method, and Barthelmé et al. (2019) propose a way to stabilize this conditioned sampling.

1.3 L-ensembles and fixed-size L-ensembles

L-ensembles are a subclass of DPPs popular in machine learning applications because of their intuitive definition:

[1] They are often called k-DPPs in the literature, but we prefer "fixed-size DPPs" in order not to overload the symbol k too much.

Definition 1.5 (L-ensemble)**.** *Let $\boldsymbol{L}$ be a positive semi-definite matrix. $\mathcal{X}$ is a L-ensemble on Ω if for all $X \subseteq \Omega$*

$$p(\mathcal{X} = X) = \frac{1}{\det(\boldsymbol{I} + \boldsymbol{L})} \det(\boldsymbol{L}_X) \tag{2}$$

L-ensembles are specified via their likelihood function, Eq. (2), which states that those subsets of Ω where the submatrix $\boldsymbol{L}_X$ is well-conditioned, are preferred. Intuitively, if $L_{i,j}$ represents a similarity between items i and j of Ω, then the L-ensemble favours subsets of Ω that hold dissimilar items.

As with DPPs, one defines fixed-size L-ensembles as:

Definition 1.6 (Fixed-size L-ensemble)**.** *A fixed size L-ensemble of size m is a L-ensemble $\mathcal{X}$ conditioned on $|\mathcal{X}| = m$.*

(Fixed-size) L-ensembles are (fixed-size) DPPs with kernel $\boldsymbol{K} = (\boldsymbol{I} + \boldsymbol{L})^{-1}\boldsymbol{L}$ (Kulesza et al., 2012; Tremblay et al., 2023), and the mixture representation thus applies.

To conclude this section, we have seen that all (fixed-size) L-ensembles and more generally all (fixed-size) DPPs have a mixture representation that divides the sampling algorithm in two steps: i/ a diagonalisation step that costs $\mathcal{O}(np^2)$, ii/ a step consisting of sampling a projection DPP. Step ii/ is known[2] to cost $\mathcal{O}(nm^2)$ in the literature. The purpose of this paper is to show that the cost of this second step can always (and easily) be reduced to $\mathcal{O}(nm + m^3 \log m)$.

1.4 State-of-the-art

Various directions have been explored when designing fast samplers for discrete DPPs. Some have focused on bypassing the eigendecomposition of $\boldsymbol{K}$ or $\boldsymbol{L}$ (Poulson, 2020; Launay et al., 2020; Dereziński et al., 2019a). If $\boldsymbol{K}$ is a sparse matrix, then the algorithms in Poulson (2020) can be quite advantageous compared to standard $\mathcal{O}(n^3)$ algorithms. These algorithms are difficult to adapt to L-ensembles in an efficient manner (for instance, they cannot take advantage of sparsity in $\boldsymbol{L}$). Random spanning forests (Avena and Gaudillière, 2017) are an example of a discrete DPP where a fast sampler (Wilson's algorithm, Wilson (1996)) is available, and sparsity in $\boldsymbol{L}$ seems to play a role. However, Wilson's algorithm does not extend readily to L-ensembles with arbitrary structures.

Another direction for generic DPP samplers is to give up on exactness. Approximate samplers are available, based on Markov Chain Monte Carlo methods. The most recent results in that direction are in Anari et al. (2022), where the authors show that given some preprocessing there are MCMC samplers that run in $\tilde{\mathcal{O}}(m^\omega)$, where the $\tilde{\mathcal{O}}$ is shorthand for $\mathcal{O}(.)$ "up to logarithmic factors", and ω is the exponent of matrix-multiplication time, which for practical values of m is effectively 3. The pre-processing consists essentially in estimating the inclusion probabilities, also known as the leverage scores, and its runtime is given by Anari et al. (2022) as $\tilde{\mathcal{O}}(nm^{\omega-1})$. Our results are essentially the same (preprocessing in $\mathcal{O}(nm^2)$, sampling in $\tilde{\mathcal{O}}(m^3)$), but our sampler is exact.

Also, two papers (Han et al., 2022a,b) extend the tree-based approach of Gillenwater et al. (2019) to obtain both approximate and exact samplers with complexity slightly larger than our proposal. They are also more complicated to implement. However, they can handle non-symmetric DPPs, which the algorithm given here cannot do.

Finally, the use of rejection sampling is not new in the context of DPP sampling, since algorithms for sampling continuous DPPs use this strategy out of necessity (Lavancier et al., 2015). More recently, Dereziński et al. (2019b) describe a similar algorithm to ours, in the context of volume sampling for experimental design. Our contribution compared to Dereziński et al. (2019b) is to i/ lay out a more refined analysis: better bound on the total number of proposals in Theorem 2.1, novel investigation in the theoretical implications of this theorem in Section 4, ii/ keep an eye on practical implementations: see Sections 2.3 and 3. What we would like to stress is that rejection sampling leads to an algorithm that is much faster in practice, but no more complicated to implement, than the traditional discrete sampler.

2 Sampling via accept-reject

In this section the goal is to formulate and analyse an algorithm for sampling a projection DPP $\mathcal{X} \sim DPP(\boldsymbol{Q}\boldsymbol{Q}^\top)$ with $\boldsymbol{Q} \in \mathbb{R}^{n\times m}$, verifying $\boldsymbol{Q}^\top\boldsymbol{Q} = \boldsymbol{I}$. The first exact such algorithm was described by Hough et al. (2006), and adapted in Kulesza et al. (2012) to the discrete case. The first efficient version appeared in Gillenwater (2014). It is effectively a variant of the Gram-Schmidt algorithm.

For completeness we give an easy derivation of this classical algorithm in the next section (Section 2.1), and the notation will serve to describe our own variant, in Section 2.2.

2.1 State-of-the-art algorithm

A projection DPP has size m almost surely (Kulesza et al., 2012). We shall sample the m elements successively. Let $\vec{\mathcal{X}} = (x_1, \ldots, x_m)$ be an ordered version of $\mathcal{X}$; we can go from $\vec{\mathcal{X}}$ to $\mathcal{X}$ by forgetting the order and from $\mathcal{X}$ to $\vec{\mathcal{X}}$ by ordering randomly. The latter can be achieved by picking a first item uniformly from $\mathcal{X}$, then a second, then a third etc. The sampling algorithm proceeds via the following

[2] This is an average (resp. deterministic) cost for DPPs (resp. fixed-size DPPs) for which m refers to the average (resp. desired) size of the sample.

decomposition:

$$p(\vec{\mathcal{X}}) = p(x_1)p(x_2|x_1)p(x_3|x_1,x_2)\dots p(x_m|x_1\dots x_{m-1}) \tag{3}$$

The algorithm samples x_1 first, then x_2 given x_1 has been selected, etc. The law of x_1 is the law of the first element of $\vec{\mathcal{X}}$, a randomly ordered version of $\mathcal{X}$. That is equivalent to x_1 being sampled uniformly at random from $\mathcal{X}$, and so:

$$p(x_1 = i) = \frac{1}{m}p(i \in \mathcal{X}) = \frac{K_{i,i}}{m}$$

We can similarly obtain the law of x_2 given x_1, as two elements drawn randomly (without replacement) from $\mathcal{X}$:

$$\begin{aligned} p(x_2 = i|x_1 = j) &= \frac{p(x_1 = j, x_2 = i)}{p(x_1 = j)} \\ &= \frac{p(i \in \mathcal{X}, j \in \mathcal{X})}{m(m-1)} \times \frac{m}{p(j \in \mathcal{X})} \\ &= \frac{1}{(m-1)}\frac{\det \boldsymbol{K}_{\{i,j\}}}{K_{j,j}} \end{aligned}$$

The formula for determinants of block matrices yields:

$$\frac{\det \boldsymbol{K}_{\{i,j\}}}{K_{j,j}} = K_{i,i} - \frac{K_{i,j}^2}{K_{j,j}}$$

and so:

$$p(x_2 = i|x_1 = j) = \frac{1}{(m-1)}\left(K_{i,i} - \frac{K_{i,j}^2}{K_{j,j}}\right)$$

For the general term in the chain rule decomposition (Eq. (3)), the same reasoning applies. We obtain:

$$\begin{aligned} &p\left(x_t = i|\vec{\mathcal{X}}_{1:(t-1)} = X_t\right) \\ &\qquad = \frac{1}{(m-t)}\left(K_{i,i} - \boldsymbol{K}_{i,X_t}(\boldsymbol{K}_{X_t})^{-1}\boldsymbol{K}_{X_t,i}\right) \end{aligned} \tag{4}$$

Eq. (4) is enough to give us a sampling algorithm, since at each step we have an explicit (discrete) probability distribution to sample from. However, implementing Eq. (4) naïvely, we would be computing a matrix inverse at each step, which would turn out to be quite expensive for large m. To get an efficient algorithm a bit more work is needed.

Let us reexpress Eq. (4) in terms of $\boldsymbol{Q}$. Recalling $\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{Q}^\top$, we obtain:

$$\begin{aligned} &p\left(x_t = i|\vec{\mathcal{X}}_{1:(t-1)} = X_t\right) \\ &\qquad = \frac{1}{(m-t)}\left(K_{ii} - \boldsymbol{Q}_{i,:}\boldsymbol{M}_t(\boldsymbol{Q}_{i,:})^\top\right) \end{aligned} \tag{5}$$

where $\boldsymbol{M}_t = (\boldsymbol{Q}_{X_t,:})^\top\left(\boldsymbol{Q}_{X_t,:}(\boldsymbol{Q}_{X_t,:})^\top\right)^{-1}\boldsymbol{Q}_{X_t,:}$ is a projection matrix ($\boldsymbol{M}_t^2 = \boldsymbol{M}_t$) of size m and rank $|X_t| = t-1$, and so can be rewritten

$$\boldsymbol{M}_t = \sum_{i=1}^{t-1}\boldsymbol{s}_i\boldsymbol{s}_i^\top$$

where $\boldsymbol{s}_1 \dots \boldsymbol{s}_t$ form an orthonormal basis for $\operatorname{span}\boldsymbol{M}_t = \operatorname{span}\boldsymbol{Q}_{X_t,:}^\top$, the linear subspace spanned by the rows of $\boldsymbol{Q}$ selected so far. A first source of computational savings comes from realising that $\boldsymbol{M}_t$ can be computed iteratively via the Gram-Schmidt process. Notice that $\boldsymbol{M}_t = \boldsymbol{M}_{t-1} + \boldsymbol{s}_{t-1}\boldsymbol{s}_{t-1}^\top$, and $\boldsymbol{M}_{t-1}$ spans $\operatorname{span}\boldsymbol{Q}_{X_{t-1},:}^\top$. We obtain $\boldsymbol{s}_{t-1}$ via Gram-Schmidt: first we compute the residual

$$\boldsymbol{z}_{t-1} = (\boldsymbol{I} - \boldsymbol{M}_{t-1})\boldsymbol{Q}_{x_{t-1},:}^\top$$

and then we normalise:

$$\boldsymbol{s}_{t-1} = \frac{\boldsymbol{z}_{t-1}}{\|\boldsymbol{z}_{t-1}\|}$$

At each step t this costs $\mathcal{O}(m(t-1))$ operations, and we will need to do this $m-1$ times at a total cost of $\mathcal{O}(m^3)$.

Next, we can show that the probability distribution we sample from at step t can be easily obtained from the one we had at step $t-1$. It is more convenient to write this using unnormalised versions of the densities. Let $\pi^{(1)}(i) = mp(x_1 = i) = K_{i,i}$. Next, we define:

$$\begin{aligned} \pi^{(2)}(i) &= (m-1)p(x_2 = i|x_1 = j) \\ &= K_{i,i} - \frac{K_{i,j}^2}{K_{i,i}} = \pi^{(1)}(i) - \frac{K_{i,j}^2}{K_{i,i}} \end{aligned}$$

Note that we have suppressed the dependency on the past in the notation $\pi^{(2)}(i)$: it is to be understood as the (unnormalised) density we draw from at the second step of the algorithm. In the general case, we define:

$$\pi^{(t)}(i) = (m-t+1)\ p\left(x_t = i|\vec{\mathcal{X}}_{1:(t-1)} = X_t\right) \tag{6}$$

Injecting Eq. (4) and Eq. (5), we find

$$\begin{aligned} \pi^{(t)}(i) &= \pi^{(1)}(i) - \boldsymbol{Q}_{i,:}\boldsymbol{M}_t(\boldsymbol{Q}_{i,:})^\top \\ &= \pi^{(1)}(i) - \boldsymbol{Q}_{i,:}(\boldsymbol{M}_{t-1} + \boldsymbol{s}_{t-1}\boldsymbol{s}_{t-1}^\top)(\boldsymbol{Q}_{i,:})^\top \\ &= \pi^{(t-1)}(i) - (\boldsymbol{Q}_{i,:}\boldsymbol{s}_{t-1})^2 \end{aligned} \tag{7}$$

All we need to do at each step of the algorithm is to

1. pick an item i according to $\pi^{(t)}$
2. perform a step of the Gram-Schmidt algorithm to update $\boldsymbol{M}_t$ based on the new vector $\boldsymbol{Q}_{i,:}$
3. Update the probability distribution to $\pi^{(t+1)}$ according to Eq. (7)

Sampling from a discrete distribution (step 1 above) can be done at cost $\mathcal{O}(n)$, and is needed m times, for a total cost of $\mathcal{O}(nm)$. We have already established that the cost of the Gram-Schmidt algorithm is $\mathcal{O}(m^3)$. It is the update to the probability distribution that is the most costly, with each step costing $\mathcal{O}(nm)$ (n dot products in $\mathbb{R}^m$) for a total of $\mathcal{O}(nm^2)$. Since $n > m$ the cost of the algorithm scales as $\mathcal{O}(nm^2)$. We show pseudo-code for this standard algorithm as Alg. 1.

In the next section, we move on to the core of our contribution, showing that this $\mathcal{O}(nm^2)$ cost can be reduced to $\mathcal{O}(nm + m^3\log m)$ via rejection sampling.

Algorithm 1: Sampling from a projection DPP $\mathcal{X} \sim DPP(\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{Q}^\top)$, standard algorithm

Initialise $\pi(x) \leftarrow \sum_{j=1}^m Q_{x,j}^2$, $\mathcal{X} = \emptyset$,
Gram-Schmidt basis $\boldsymbol{S} = []$;
foreach $t \in 1 \dots m$ **do**
  Sample x from $\frac{\pi(x)}{m-(t-1)}$, add to $\mathcal{X}$;
  Compute residual $\boldsymbol{z}_t = (\boldsymbol{I} - \boldsymbol{S}\boldsymbol{S}^\top)(\boldsymbol{Q}_{x,:})^\top$;
  Add column $\boldsymbol{s}_t = \frac{\boldsymbol{z}_t}{\|\boldsymbol{z}_t\|}$ to $\boldsymbol{S}$;
  Compute $\boldsymbol{v} = \boldsymbol{Q}\boldsymbol{s}_t \in \mathbb{R}^n$;
  Update probabilities $\pi(x) \leftarrow \pi(x) - (v_x)^2$;
end

2.2 Using rejection sampling

As discussed above, the most expensive part of alg. 1, lies in updating the probability distribution to sample from (last step of the *for* loop). It turns out that the accept-reject method lets us bypass this step.

To briefly recall the rejection sampling idea, suppose we have an unnormalised density $\pi(x)$ (the target) we wish to draw from, and a proposal $q(x)$, also unnormalised, but that we know how to draw from, and is not too far from π. Further, $q(x)$ has support at least as wide as $\pi(x)$, and upper bounds it ($\pi(x) \leq q(x)$ over the support). We may then draw x from $q(x)$ and accept the sample with probability $\frac{\pi(x)}{q(x)}$. The accepted samples then have density $\pi(x)$. If $q(x)$ is a good bound for $\pi(x)$, then the rejection sampler will be quite efficient. In the limit where $\pi = q$, the acceptance probability goes to 1. If on the other hand $q(x)$ is quite loose, the acceptance probability may be bad.

In the DPP sampling algorithm, we need to sample from $\pi^{(1)}$, then $\pi^{(2)}$, etc. up to $\pi^{(m)}$. Our proposal is to compute $\pi^{(1)}$ exactly for all n entries, then use $\pi^{(1)}$ as proposal distribution for the rest of the sequence.

Recall that $\pi^{(t)}$ is the unnormalised density defined in Eq. (6). From the recursion in Eq. (7), we have that

$$\pi^{(t)}(x) \geq \pi^{(t+1)}(x)$$

for all x and $1 \leq t \leq m-1$. This is true in particular for $\pi^{(1)}$, which can therefore be used as a proposal distribution for all subsequent $\pi^{(t)}$. We can directly compute the probability of accepting a proposed sample. At step t, we sample x from the normalised density $\frac{1}{m}\pi^{(1)}(x)$ and accept it with probability $\frac{\pi^{(t)}(x)}{\pi^{(1)}(x)}$. The acceptance probability equals:

$$\rho_t = \sum_{i=1}^n \frac{\pi^{(t)}(i)}{\pi^{(1)}(i)} \frac{\pi^{(1)}(i)}{m} = \frac{1}{m}\sum_{i=1}^n \pi^{(t)}(i) = \frac{(m-t+1)}{m} \tag{8}$$

This probability decreases at each step of the algorithm, but at the final step it is still positive and equals $\frac{1}{m}$.

Let us outline the proposed algorithm. First, one computes every entry of $\pi^{(1)}$. For this we use the following formula

$$\pi^{(1)}(i) = K_{i,i} = \sum_{i=1}^n Q_{ij}^2 \tag{9}$$

The computation is equivalent to computing the norm of each row of $\boldsymbol{Q}$, at cost $\mathcal{O}(nm)$. Because we need to sample from $\pi^{(1)}$ repeatedly, it pays to use Walker's alias method (Walker, 1977) (see also Chapter III.4 of Devroye (1986)). Given a preprocessing cost of $\mathcal{O}(n)$, the alias method gives us all subsequent samples at cost $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$.

At the first step we sample our first item from $\pi^{(1)}$. At step 2, and all subsequent steps, we use rejection sampling, which involves computing the ratio

$$\frac{\pi^{(t)}(x)}{\pi^{(1)}(x)} = 1 - \frac{1}{\pi^{(1)}(x)}\sum_{j=1}^{t-1}(\boldsymbol{Q}_{x,:}\boldsymbol{s}_j)^2 \tag{10}$$

by Eq. (7). Computing the acceptance ratio has cost $\mathcal{O}(m(t-1))$ at step t, the cost of $t-1$ dot products in $\mathbb{R}^m$. We do this repeatedly until a proposal is accepted, at which point we need to perform a Gram-Schmidt step to update $\boldsymbol{M}_t$ to $\boldsymbol{M}_{t+1}$. We then move on to the next iteration, or stop if $t = m$.

We summarise the whole process as Alg. 2. To recapitulate the different computational costs:

- Preprocessing cost: computing $\pi^{(1)}$ for all entries comes at cost $\mathcal{O}(nm)$ and setting up Walker's alias method at cost $\mathcal{O}(n)$
- The Gram-Schmidt process (computing $\boldsymbol{z}_t$ then $\boldsymbol{s}_t$) costs $\mathcal{O}(tm)$ at step t. Summing this figure for $t = 1$ to m gives a cost of $\mathcal{O}(m^3)$
- We now need to compute the average cost of the *while* loop. At step t, the rejection sampler has probability ρ_t of succeeding, given by Eq. (8). The number of proposals R_t that are required until acceptance is thus a random variable that follows a geometric distribution with success probability ρ_t. One thus has:

$$\mathrm{E}(R_t) = \frac{1}{\rho_t} = \frac{m}{m-t+1}. \tag{11}$$

  Since computing the acceptance ratio costs $\mathcal{O}(mt)$ for each trial, the expected cost of the *while* loop at step t scales as $\mathcal{O}\left(\frac{m^2 t}{m-t+1}\right)$. Summing this figure over t:

$$\sum_{t=1}^m \frac{m^2 t}{m-t+1} \leq m^3 \sum_{t=1}^m \frac{1}{m-t+1}$$

  yields a total expected cost scaling as [3] $\mathcal{O}(m^3 \log m)$.

Tallying everything we obtain the following theorem.

[3] $\sum_{t=1}^m \frac{1}{m-t+1} = \sum_{t=1}^m \frac{1}{t}$ scales as $\mathcal{O}(\log m)$: see, *e.g.*, Chapter 6 of Abramowitz and Stegun (1964)

Algorithm 2: Sampling from a projection DPP $\mathcal{X} \sim DPP(\boldsymbol{K} = \boldsymbol{Q}\boldsymbol{Q}^\top)$ with rejection sampling

Initialise $\pi^{(1)}(x) \leftarrow \sum_{j=1}^m Q_{x,j}^2$, $\mathcal{X} \leftarrow \emptyset$, Gram-Schmidt basis $\boldsymbol{S} \leftarrow []$;
Initialise alias table for Walker's alias algorithm to sample from $\pi^{(1)}$.
foreach $t \in 1 \dots m$ **do**
  accept $\leftarrow$ false ;
  while *not accept* **do**
    Draw x from $\pi^{(1)}$ using the alias method;
    Compute acceptance ratio $r = 1 - \frac{1}{\pi^{(1)}(x)} \sum_{j=1}^{t-1} (\boldsymbol{Q}_{x,:}\boldsymbol{s}_t)^2$;
    if *rand()* $< r$ **then**
      accept $\leftarrow$ true
    end
  end
  Add x to $\mathcal{X}$;
  Compute residual $\boldsymbol{z}_t = (\boldsymbol{I} - \boldsymbol{S}\boldsymbol{S}^\top)(\boldsymbol{Q}_{x,:})^\top$;
  Add column $\boldsymbol{s}_t = \frac{\boldsymbol{z}_t}{\|\boldsymbol{z}_t\|}$ to $\boldsymbol{S}$;
end

Theorem 2.1. *Alg. 2 samples a projection DPP, with an expected runtime scaling as $\mathcal{O}(nm + m^3 \log m)$. Also, any additional sample from the same DPP can be obtained in an extra $\mathcal{O}(m^3 \log m)$ expected runtime.*
Moreover, these expected runtimes are representative. Indeed, $\forall \delta \in \left(0, \frac{1}{2}\right)$, the total number of proposals $R = \sum_{t=1}^m R_t$ satisfies, with probability greater than $1 - \delta$:

$$R \leq 2m \log m + 3m \log \frac{1}{\delta}$$

Proof. The fact that Alg. 2 samples a projection DPP in $\mathcal{O}(nm + m^3 \log m)$ expected runtime is proven above the Theorem's statement. The fact that any additional sample from the same DPP only costs an extra $\mathcal{O}(m^3 \log m)$ expected runtime comes from the observation that all initialisation steps (the computation of $\pi^{(1)}$ and the setting-up cost of Walkers' algorithm) have already been computed for the first sample. To obtain any extra sample, one only needs to run the *for* loop once more, costing $\mathcal{O}(m^3 \log m)$.
The final statement relates to concentration properties of R. For $m = 1$, it is trivial (log refers to the natural logarithm in the result) as $R = 1$ with probability 1 in this case. We now show the result for $m \geq 2$. At step t of Alg. 2 the number of proposals R_t is a random geometric variable with parameter $(m-t+1)/m$. Note that all the R_t's are independent. We study here the behavior of $R = \sum_{t=1}^m R_t$, the total number of rejection sampling steps in the whole course of Alg. 2. The following one-tailed upper bound is drawn from Thm 2.3 in Janson (2018):

$$\forall \lambda \geq 1, \quad p\left(R \geq \lambda E[R]\right) \leq \frac{1}{\lambda} \left(1 - \frac{1}{m}\right)^{(\lambda - 1 - \log \lambda)\mathrm{E}[R]}$$

Let $\delta \in \left(0, \frac{1}{2}\right)$. We look for λ large enough such that $p\left(R \geq \lambda E[R]\right) \leq \delta$, that is:

$$-\log \lambda + (\lambda - 1 - \log \lambda) E[R] \log\left(1 - \frac{1}{m}\right) \leq \log \delta$$

As $\lambda \geq 1$, $-\log \lambda \leq 0$. It thus suffices to seek λ verifying:

$$(\lambda - 1 - \log \lambda) E[R] \log\left(1 - \frac{1}{m}\right) \leq \log \delta \qquad (12)$$

Note that $E[R] \log\left(1 - \frac{1}{m}\right)$ is negative so we seek a lower bound of $\lambda - 1 - \log \lambda$. One has[4]:

$$\forall \lambda \geq 1, \qquad \frac{1}{3}\lambda - \log \frac{3}{2} \leq \lambda - 1 - \log \lambda$$

such that Eq. (12) is verified provided that:

$$\lambda \geq 3 \left(\log \frac{3}{2} - \frac{\log \delta^{-1}}{\mathrm{E}[R] \log\left(1 - \frac{1}{m}\right)}\right).$$

Stated differently, setting λ to this lower bound implies $p(R \leq \lambda \mathrm{E}[R]) \geq 1 - \delta$. All is left to show is that $\forall m \geq 2$:

$$3 \left(\log \frac{3}{2} - \frac{\log \delta^{-1}}{\mathrm{E}[R] \log\left(1 - \frac{1}{m}\right)}\right) \mathrm{E}[R] \leq 2m \log m + 3m \log \delta^{-1}. \qquad (13)$$

For this, we use two upper bounds. The first one is

$$\forall m > 1, \qquad -\frac{1}{\log\left(1 - \frac{1}{m}\right)} \leq m - 1/2.$$

The second one is the following bound on $\mathrm{E}[R]$. As $\mathrm{E}[R_t] = m/(m - t + 1)$, one has (see, *e.g.*, Chapter 6 of Abramowitz and Stegun (1964)):

$$\frac{\mathrm{E}[R]}{m} = \sum_{t=1}^m \frac{1}{m - t + 1} = \sum_{t=1}^m \frac{1}{t} = \gamma + \psi(m) + \frac{1}{m}$$

where $\gamma \approx 0.577$ is Euler's constant and $\psi(m)$ the digamma function. Now, a known bound on $\psi(m)$ is $\psi(m) \leq \log m - \frac{1}{2m}$, which gives $\mathrm{E}[R] \leq m(\log m + \gamma) + \frac{1}{2}$. Applying these two upper bounds yields Eq. (13). □

2.3 Some refinements

Alg. 2 works well enough as is but there are some refinements that can further reduce the computational cost.

[4] In fact, the function $\lambda - 1 - \log \lambda$ is convex for all $\lambda \geq 1$ and thus lower-bounded by all its tangents. The one we use is the tangent in $3/2$. Other choices lead to other constants in the result.

2.3.1 Preprocessing for general DPPs

In some applications we require several samples from the same DPP, and algorithms have been described that trade higher set-up cost for a lower cost per sample (see, *e.g.*, Gillenwater et al. (2019)). If the target DPP is a projection DPP, then setting up Alg. 2 for repeated sampling could not be easier, as stated in Thm 2.1: computing $\pi^{(1)}$ and setting up the alias table is part of the preprocessing, so the first sample from the DPP costs $\mathcal{O}(nm + m^3 \log m)$ but after that the cost is just $\mathcal{O}(m^3 \log m)$ per sample.

If the target DPP is not a projection DPP, then one has to use the mixture representation (Thm 1.3): draw a random set of eigenvectors $\mathcal{Y}$ and run Alg. 2 with $\boldsymbol{Q} = \boldsymbol{U}_{\mathcal{Y},:}$. Since the kernel changes every time, so does $\pi^{(1)}$, and it cannot be computed as part of pre-processing. However, the kernels encountered in practice tend to have rapidly decreasing eigenvalues, so that the variance in $\mathcal{Y}$ is quite small and the DPP is close to a projection DPP. Without getting into too much detail, it is possible to pre-compute the partial sum

$$\widehat{\pi^{(1)}}(i) = \sum_{j \in \hat{\mathcal{Y}}} U_{i,j}^2$$

for some highly likely subset of $\mathcal{Y}$, denoted here $\hat{\mathcal{Y}}$. $\pi^{(1)}$ for the actual sampled $\mathcal{Y}$ can be obtained efficiently by removing the extra entries and adding the missing ones. The alias table can be computed from scratch. This type of preprocessing brings down the cost to $\mathcal{O}(ns + m^3 \log m)$ per sample, where $m = \mathrm{E}(|\mathcal{Y}|)$ and s is the expected size of the symmetric difference between $\hat{\mathcal{Y}}$ and $\mathcal{Y}$.

2.3.2 Caching computations and updating the proposal distribution

Clearly, Alg. 2 has some wasted computation, since all the computations done when a proposal x is rejected are performed again should x come up a second time. When n is small, or when $\pi^{(1)}$ has low entropy, this may indeed happen several times. Caching is one way of reducing the amount of redundant computations that are performed. Going back to the recursive formula for $\pi^{(t)}$ (Eq. (7)), we see that it is cheaper to compute $\pi^{(t)}$ from $\pi^{(t-1)}$ than it is to compute it from scratch. A reasonable caching strategy is then to keep track for every point x of the last density evaluation performed for that point. If x comes up again, the evaluation of the acceptance ratio is simplified.

Another natural idea is to update the proposal distribution over the course of the algorithm (instead of sticking with $\pi^{(1)}$ throughout). The most basic version is that any x that has already been selected has an acceptance probability of 0, so we may as well not suggest them. Another is that points similar to a selected point are quite unlikely to come up further down, and so it may be worth computing $\pi^{(t)}$ for these neighbours to tighten the bound. Finally, we may combine this idea with the caching idea, which provides a better bound for every point that has ever been suggested. Unfortunately this runs against the difficulty of updating the alias table in Walker's algorithm, which one would have to compute from scratch at every update (at cost $\mathcal{O}(n)$). A better way would be to use a binary tree representation (Devroye, 1986), which can be updated at cost $\mathcal{O}(\log n)$ and provides samples also at cost $\mathcal{O}(\log n)$. The implementation complexity increases a lot however, and we have not pursued this further. As we shall see below, Alg. 2 is quite fast in practice, and implementation time may be better invested in feature computation and orthogonalisation.

3 Empirical results

We compare the Accept/Reject algorithm (Alg. 2) to its classical counterpart (Alg. 1) for different values of n and m. Both algorithms are implemented in the Julia language and are publicly available[5]. For each value of n, we sample a random projection matrix of size $n \times m$ (via QR decomposition of a matrix with Gaussian entries). We then pre-compute the leverage scores, and run each algorithm 100 times. Fig. 1 a) and b) show the measured median runtimes. All tests are run on a 2017 Linux laptop with i7-8550U Intel CPU and 8 Go of RAM.

For very small values of n, the classical algorithm is faster, which can be explained by the efficiency of BLAS calls. At each step the whole conditional distribution is computed, the main cost being a matrix multiplication (i.e., a BLAS call), which benefits from efficient multithreaded code. However, the different asymptotic scalings ($\mathcal{O}(nm^2)$ vs. $\mathcal{O}(nm)$) soon makes the classical algorithm uncompetitive. The cross-over point in our simulations is at around $n = 1,000$, and by $n = 10,000$ the difference is stark. This illustrates the "very significant speed-up" scenario described in the introduction: $p = m$ and orthogonalisation is already computed. We now move on to illustrate a less favorable case related to the "moderate speed-up" scenario of the introduction: p equals a few times m.

In many cases, sampling a projection DPP is only one of the steps in a process that involves feature computation and orthogonalisation. For instance, in Tremblay et al. (2019), a DPP based on the Gaussian kernel is used to produce a subset of the data suitable for running clustering algorithms. Starting from n points, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ in $\mathbb{R}^d$ they use a DPP with L-ensemble given by $L_{ij} = \gamma\kappa(\boldsymbol{x}, \boldsymbol{y}) = \gamma \exp\left(-\frac{1}{\sigma^2} \|\boldsymbol{x} - \boldsymbol{y}\|^2\right)$, where γ is a tuning parameter that determines the expected size. Producing a sample from this

[5] we've added a folder in our DPP.jl repository containing the code necessary to reproduce the figures, available here: `https://github.com/dahtah/DPP.jl/tree/main/misc/sampling_paper`. In addition, the A/R sampler is available as part of the DPP.jl software package, `https://github.com/dahtah/DPP.jl/`.

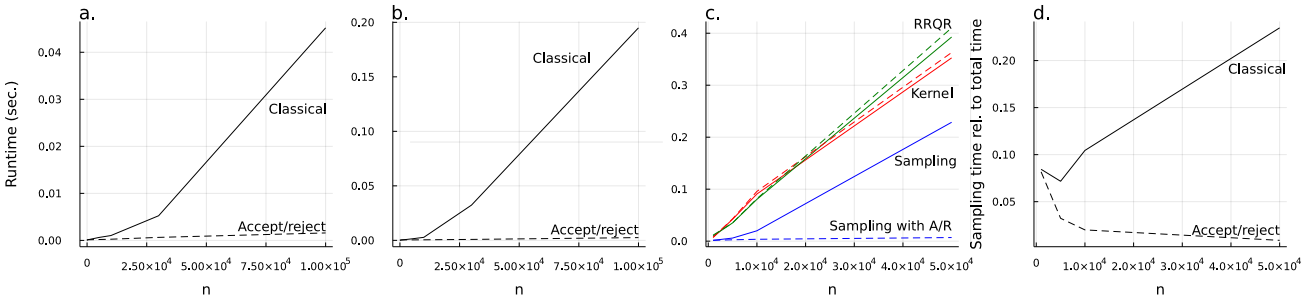

Figure 1: *Left panels*: median time needed to sample a projection DPP, using the standard approach (Alg. 1) vs. A/R (Alg. 2). For two values of m: a) $m = 30$, b) $m = 60$. Note that here the time taken to compute an orthogonal basis is not taken into account (see text). *Right panels*: simulation of a full workload that includes feature generation and orthogonalisation (see text). c) time taken by each step in the computation as a function of n. Solid lines are runtimes when sampling using the classical method, dashed, when using A/R. Note that the first two steps (labeled "kernel" and "RRQR", in red and green respectively) are identical. A/R sampling becomes beneficial at around $n = 1,000$ and is orders of magnitude faster at $n = 100,000$. d) Sampling time as a function of total computation time.

exact DPP requires the eigendecomposition of $\boldsymbol{L}$ which is impractical; however, $\boldsymbol{L}$ is numerically low-rank for relevant values of σ and this can be exploited. In Tremblay et al. (2019) a low-rank approximation of $\boldsymbol{L}$ is used, based on random Fourier features (Rahimi and Recht, 2007) followed by a SVD, bringing down the total cost to $\mathcal{O}(nm^2)$. We now sketch (without any formal justification) another procedure which gives comparable results at lower cost.

Gaussian kernel matrices have rapidly decaying spectra (see, *e.g.*, Wathen and Zhu (2015)), which implies in particular that DPPs sampled from a Gaussian L-ensemble are well approximated by projection DPPs with kernels $\boldsymbol{P}_m = \boldsymbol{U}_m\boldsymbol{U}_m^\top$ where $\boldsymbol{P}_m$ projects onto the dominant eigenspace of $\boldsymbol{L}$ of order m. Thus, all we need is a good basis for the dominant eigenspace. Methods from randomised linear algebra offer good practical tools ("range finders") to obtain a basis for such a space (Martinsson and Tropp, 2020). For these simulations, we used the following approximation:

1. Select (and compute) $5m$ columns uniformly from $\boldsymbol{L}$. Call this matrix $\boldsymbol{A}$. We call this the "kernel step".
2. Use Rank-Revealing QR (RRQR, Chan (1987)) and random projections, as implemented in the Julia package LowRankApprox.jl, to produce $\boldsymbol{Q}$, an orthonormal matrix of size $n \times m$ that approximates the image of $\boldsymbol{A}$. We call this the "RRQR step".
3. Sample a DPP with projection kernel $\boldsymbol{Q}\boldsymbol{Q}^\top$ using either the classical or the A/R algorithm.

We set $m = 100$ and time each step. This results in a total runtime of around 1.2 sec. at $n = 10^5$ with the A/R sampler, which challenges the notion that DPPs are very slow to sample from. With this procedure, the time spent sampling the actual DPP goes up to 20% of total time for the classical algorithm at $n = 10^5$, but using the A/R sampler sampling time becomes negligible. See Fig. 1 c) and d) to see how these times vary with n. This indicates that for some computations the implementation effort may be better allocated to speeding up the linear algebra and feature computation part rather than the sampling part. In this particular instance, step (1) at least could be sped up by exploiting parallelism, or the GPU, which we did not attempt.

4 Discussion and perspectives

On top of the improvement on the sampling time of DPPs, our results imply the following intriguing by-product. Let $\mathcal{X}$ be a projection DPP of size m. A set of $\mathcal{O}(m \log m)$ points sampled i.i.d. from the inclusion probability distribution $p(i \in \mathcal{X}) = \pi^{(1)}(i)/m$ (also known as *leverage scores*) contains with high probability a realisation from the DPP. The consequences of this fact are worth discussing. First, let us put the result a bit more formally.

Definition 4.1. *Let $\mathcal{X}$ be a DPP on Ω and $\mathcal{Y} \subseteq \Omega$. We call $\Phi(\mathcal{Y})$ a* thinning algorithm *if it returns a subset of $\mathcal{Y}$. Moreover, we say $\Phi(\mathcal{Y})$ is* successful *when it returns a realisation from $\mathcal{X}$.*

Corollary 4.2. *Let $\mathcal{X}$ be a projection DPP of size m, and $\mathcal{Y}$ be a set of i.i.d. points sampled with replacement with probability proportional to the leverage scores: $p(i \in \mathcal{X}) = \pi^{(1)}(i)/m$. Let $\delta \in (0, 1/2)$. A simple modification of Alg. 2 gives a thinning algorithm Φ that verifies: $\Phi(\mathcal{Y})$ is successful with probability greater than $1 - \delta$ provided that $|\mathcal{Y}| \geq 2m \log m + 3m \log\left(\frac{1}{\delta}\right)$.*

Proof. Let $\mathcal{Y}$ be drawn i.i.d. with replacement from the leverage score distribution $\pi^{(1)}$. Φ is the following simple modification of Alg. 2. Instead of drawing a new proposal x using the alias method at the beginning of the *while* loop as in Alg. 2, draw uniformly and without replacement from $\mathcal{Y}$. If Φ finishes before emptying $\mathcal{Y}$, then it is successful. If $\mathcal{Y}$ is empty and Φ is not terminated, then it fails. The probability that Φ succeeds is thus equal to the probability that

$|\mathcal{Y}|$ is larger than the number of proposals R of Alg. (2). By Thm 2.1, setting $|\mathcal{Y}| = 2m\log m + 3m\log\left(\frac{1}{\delta}\right)$ yields $p(|\mathcal{Y}| \geq R) \leq 1-\delta$ and ends the proof. □

Note that this is a substantial improvement over the work of Dereziński et al. (2019a), which gives this result only for $|\mathcal{Y}| \geq \mathcal{O}(m^2)$ i.i.d. points.

A natural question is to ask if this result is optimal: can we find a thinning algorithm that succeeds with high probability for even smaller i.i.d sets? The answer is no in general:

Proposition 4.3. *Corollary 4.2 is optimal in the following sense. Let $\mathcal{X}$ and $\mathcal{Y}$ be as previously. There does not exist a generic thinning algorithm able to succeed with fixed non-null probability if $|\mathcal{Y}| = o(m\log m)$.*

Proof. We show the proposition by exhibiting a type of DPP for which there does not exist a thinnning algorithm that succeeds with a non-null probability if $|\mathcal{Y}|$ is asymptotically smaller than $m\log m$.

It is well-known in the folklore that a form of *stratified sampling* is a special case of projection DPPs. In stratified sampling, we partition the ground set Ω into m classes, and sample an item uniformly from each segment of the partition. To simplify the argument, assume Ω can be cut into m subsets of equal size, and define vector $\boldsymbol{e}_j$ as the (normalised) indicator of segment j, i.e. $\boldsymbol{e}_j(i) = \sqrt{\frac{n}{m}}$ if item i is in segment j and 0 otherwise. Let $\boldsymbol{E} = [\boldsymbol{e}_1 \dots \boldsymbol{e}_m]$. Then it is easy to show that stratified sampling is equivalent to a DPP $\mathcal{X}$ with marginal kernel $\boldsymbol{K} = \boldsymbol{E}\boldsymbol{E}^\top$. Since $\boldsymbol{E}^\top\boldsymbol{E} = \boldsymbol{I}$, the DPP in question is a projection DPP.

Because of the nature of stratified sampling, we know that $\mathcal{X}$ contains a point from each one of the segments, and that $p(i \in \mathcal{X}) = \frac{m}{n}$ for all i. Now in order for any thinning algorithm to produce a stratified sample, the i.i.d. sample $\mathcal{Y}$ needs to contain at least one point from each segment. Let $l \overset{\text{def}}{=} |\mathcal{Y}|$: how large does l need to be so that $\mathcal{Y}$ contains at least one point from each segment with probability at least α ($\alpha > 0$ fixed)? This is an instance of the coupon collector's problem. Assume that at each time t we add a ball to one of m urns with equal probability, and call T the smallest t such all urns have at least one ball. We show that for any $l(m) = o(m\log m)$, $\lim_{m\to\infty} p(T < l(m)) = 0$.

To do this, we need an upper bound for $p(T < l(m))$. One could work with results from Witt (2014) for instance. However, we prefer an elegant line of proof inspired by a contribution of a stackexchange user called "cardinal"[6]. T can be viewed as a sum of m geometric variables: $T = \sum_{i=1}^m T_i$, where T_i is the time at which i urns have at least one ball. All the T_i's are independent geometric random variables with success probability $p_i = 1 - \frac{i-1}{m}$. Indeed, the same representation is obtained by considering Alg. 2 in the special case of stratified sampling (each iteration fills one urn). Now, Markov's inequality gives:

$$\begin{aligned}\forall s > 0, \qquad p(T < l) &= p(\exp(-sT) > \exp(-sl)) \\ &\leq \exp(sl)\mathrm{E}\left(\exp(-sT)\right)\end{aligned}$$

Since T is a sum of independent geometric variables, $\mathrm{E}(\exp(-sT))$ is easy to compute[7]:

$$\mathrm{E}(\exp(-sT)) = \prod_{i=1}^m \frac{i}{m(e^s-1)+i}$$

Picking $s = \frac{1}{m}$, we obtain:

$$p(T < l) \leq \exp^{\frac{l}{m}} \prod_{i=1}^m \frac{i}{m(e^{1/m}-1)+i}$$

Since $e^{\frac{1}{m}} \geq 1 + \frac{1}{m}$, we upper bound the right-hand side to:

$$p(T < l) \leq \exp^{\frac{l}{m}} \prod_{i=1}^m \frac{i}{1+i} = \frac{\exp(\frac{l}{m})}{m+1}$$

Thus, any choice of sample size $l(m)$ such that $\frac{\exp(\frac{l(m)}{m})}{m+1}$ goes to 0 in the limit is asymptotically too small (the probability of success goes to 0). Noting that $\frac{\exp(\frac{l}{m})}{m+1} = o(1)$ is equivalent to $l(m) = o(m\log m)$ yields the claim. □

This transition occuring at $\mathcal{O}(m\log m)$ calls for discussion, and paves the way to future interesting lines of research. First of all, Corollary 4.2 shows, from an original angle, that the repulsiveness of DPPs is weak. Indeed, other repulsive processes such as hard-core processes cannot verify such property in all generality. For instance, in the high density limit of a hard-sphere model, the probability that the position of m non-overlapping spheres can be found within a set of only $\mathcal{O}(m\log m)$ iid points drawn uniformly, tends to 0. In addition, these results ask the following question: in what cases should one pay the extra cost of sampling m elements from a DPP, rather than simply sampling $\mathcal{O}(m\log m)$ elements i.i.d. from the leverage score distribution? Of course, when the objective is to sample a diverse set, such as in search engines, it is always worthwhile to sample the DPP. However, in the case of integration (Bardenet and Hardy, 2020; Coeurjolly et al., 2021); or in the case of coresets (Tremblay et al., 2019), the answer is not so clear and requires further investigation.

Acknowledgements

We thank the five anonymous reviewers for their helpful comments that led to an improved version of this manuscript. This work was partially supported by the ANR project GRANOLA (ANR-21-CE48-0009), as well as the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) and MIAI@Grenoble Alpes (ANR-19-P3IA-0003).

[6] see `https://stats.stackexchange.com/q/7774`

[7] Using $E\left(\exp^{-sT}\right) = \prod_{j=1}^m E\left(\exp^{-sT_j}\right) = \prod_{j=1}^m \frac{p_j \exp^{-s}}{1-(1-p_j)\exp^{-s}}$ and changing variable $i \leftarrow m-j+1$

References

Abramowitz, M. and Stegun, I. A. (1964). *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, volume 55. US Government printing office.

Anari, N., Liu, Y. P., and Vuong, T.-D. (2022). Optimal sublinear sampling of spanning trees and determinantal point processes via average-case entropic independence. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 123–134.

Avena, L. and Gaudillière, A. (2017). Two Applications of Random Spanning Forests. *Journal of Theoretical Probability*.

Bardenet, R. and Hardy, A. (2020). Monte Carlo with determinantal point processes. *The Annals of Applied Probability*, 30(1):368 – 417.

Barthelmé, S., Amblard, P.-O., and Tremblay, N. (2019). Asymptotic equivalence of fixed-size and varying-size determinantal point processes. *Bernoulli*, 25(4B):3555–3589.

Chan, T. F. (1987). Rank revealing qr factorizations. *Linear algebra and its applications*, 88:67–82.

Coeurjolly, J.-F., Mazoyer, A., and Amblard, P.-O. (2021). Monte carlo integration of non-differentiable functions on $[0,1]\iota$, $iota = 1, \ldots, d$, using a single determinantal point pattern defined on $[0,1]^d$. *Electronic Journal of Statistics*, 15(2):6228 – 6280.

Davis, T. A. (2008). Algorithm 8xx: Suitesparseqr, a multifrontal multithreaded sparse qr factorization package. *ACM Trans. Math. Software*.

Dereziński, M., Calandriello, D., and Valko, M. (2019a). Exact sampling of determinantal point processes with sublinear time preprocessing. *Advances in neural information processing systems*, 32.

Dereziński, M., Clarkson, K. L., Mahoney, M. W., and Warmuth, M. K. (2019b). Minimax experimental design: Bridging the gap between statistical and worst-case approaches to least squares regression. In *Conference on Learning Theory*, pages 1050–1069. PMLR.

Devroye, L. (1986). *Non-Uniform Random Variate Generation*. Springer-Verlag.

Gillenwater, J. (2014). *Approximate inference for determinantal point processes*. PhD thesis, University of Pennsylvania.

Gillenwater, J., Kulesza, A., Mariet, Z., and Vassilvtiskii, S. (2019). A tree-based method for fast repeated sampling of determinantal point processes. In *International Conference on Machine Learning*, pages 2260–2268. PMLR.

Han, I., Gartrell, M., Dohmatob, E., and Karbasi, A. (2022a). Scalable mcmc sampling for nonsymmetric determinantal point processes. In *International Conference on Machine Learning*, pages 8213–8229.

Han, I., Gartrell, M., Gillenwater, J., Dohmatob, E., and Karbasi, A. (2022b). Scalable sampling for nonsymmetric determinantal point processes. In *International Conference on Learning Representations*.

Hough, J. B., Krishnapur, M., Peres, Y., and Virág, B. (2006). Determinantal Processes and Independence. *Probability Surveys*, 3:206–229.

Janson, S. (2018). Tail bounds for sums of geometric and exponential variables. *Statistics & Probability Letters*, 135:1–6.

Kulesza, A. and Taskar, B. (2011). k-dpps: fixed-size determinantal point processes. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 1193–1200.

Kulesza, A., Taskar, B., et al. (2012). Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning*, 5(2–3):123–286.

Launay, C., Desolneux, A., and Galerne, B. (2021). Determinantal point processes for image processing. *SIAM Journal on Imaging Sciences*, 14(1):304–348.

Launay, C., Galerne, B., and Desolneux, A. (2020). Exact sampling of determinantal point processes without eigendecomposition. *Journal of Applied Probability*, 57(4):1198–1221.

Lavancier, F., Møller, J., and Rubak, E. (2015). Determinantal point process models and statistical inference. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 77(4):853–877.

Martinsson, P.-G. and Tropp, J. A. (2020). Randomized numerical linear algebra: Foundations and algorithms. *Acta Numerica*, 29:403–572.

Poulson, J. (2020). High-performance sampling of generic determinantal point processes. *Philosophical Transactions of the Royal Society A*, 378(2166):20190059.

Rahimi, A. and Recht, B. (2007). Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems*, volume 20.

Tremblay, N., Barthelmé, S., Usevich, K., and Amblard, P.-O. (2023). Extended L-ensembles: A new representation for determinantal point processes. *The Annals of Applied Probability*, 33(1):613 – 640.

Tremblay, N., Barthelmé, S., and Amblard, P.-O. (2019). Determinantal Point Processes for Coresets. *Journal of Machine Learning Research*, 20(168):1–70.

Walker, A. J. (1977). An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256.

Wathen, A. J. and Zhu, S. (2015). On spectral distribution of kernel matrices related to radial basis functions. *Numerical Algorithms*, 70(4):709–726.

Wilson, D. B. (1996). Generating random spanning trees more quickly than the cover time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 296–303.

Witt, C. (2014). Fitness levels with tail bounds for the analysis of randomized search heuristics. *Information Processing Letters*, 114(1-2):38–41.