Random Matrices & Beyond

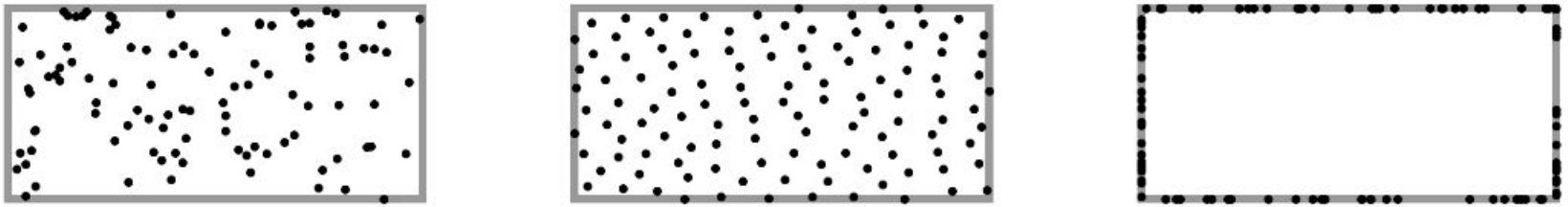
Big O Club @ Georgia Tech

Benjamin R. Bray

Basic Questions

- How can we efficiently generate random objects with a specific structure? e.g.
  - random points on a sphere, distributed uniformly
  - random positive-semidefinite matrix
  - random graphs
  - random differentiable function with bounded derivatives
- How does a deterministic algorithm behave when fed random inputs?
  - what if we generate a random matrix and run np.eig on it?
  - what if we generate a random graph and run Prim's algorithm to get a spanning tree?
- How can we **learn** a deterministic map capable of transforming uniformly random samples to a target distribution?

Three Types of Randomness



(a) Uniformly Random (b) Uniformly Spaced (c) Rare Events

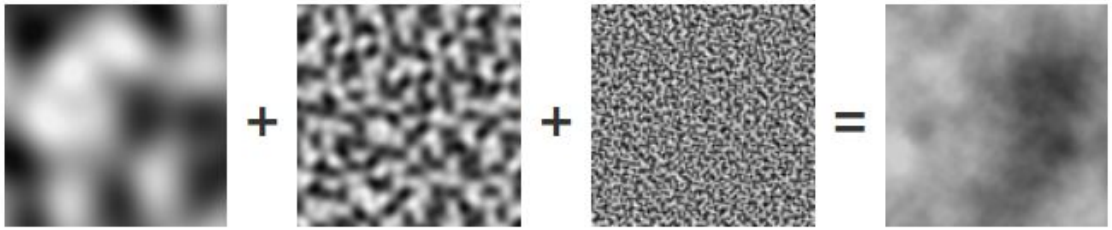
Fig. 1. Uniformly random samples naturally vary in density. The uniformly-spaced samples generated by the Poisson Disc algorithm better match our intuition, but are no longer independent. The last image is an informal depiction of "edge cases".

Why Random Matrices?

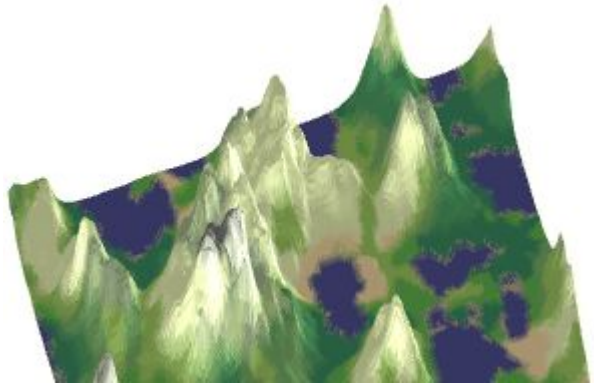
- Dimensionality reduction (Johnson-Lindenstrauss, random projections…)
- Testing e.g. numerical linear algebra implementations
  - LAPACK Benchmarks use a mix of random matrices + known ill-conditioned examples
  - Software fuzzing -- test large software against sampled range of valid inputs (corner cases!)
- Randomized / average-case analysis of algorithms
- Approximation algorithms for computationally hard problems
- Understanding deep learning
  - Rahimi, A., & Recht, B. (2009). Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In *Advances in neural information processing systems* (pp. 1313-1320).
  - Pennington, J., & Worah, P. (2017). Nonlinear random matrix theory for deep learning. In *Advances in Neural Information Processing Systems* (pp. 2634-2643).
  - Choromanski, K. M., Rowland, M., & Weller, A. (2017). The unreasonable effectiveness of structured random orthogonal embeddings. In *Advances in Neural Information Processing Systems* (pp. 218-227).

Randomness in Games & Art

- Procedural Noise
  - Terrain Generation (Red Blob Games, Minecraft)
  - Fog, fire, smoke, water, etc.
  - Procedural texture / material generation
  - L-systems for generating plants, trees, maps, etc.



```
elevation[y][x] =    1 * noise(1 * nx, 1 * ny);
                   + 0.5 * noise(2 * nx, 2 * ny);
                   + 0.25 * noise(4 * nx, 2 * ny);
```



Johnson-Lindenstrauss, Random Projections

A related lemma is the distributional JL lemma. This lemma states that for any $0 < \varepsilon, \delta < 1/2$ and positive integer d, there exists a distribution over $\mathbf{R}^{k \times d}$ from which the matrix A is drawn such that for $k = O(\varepsilon^{-2}\log(1/\delta))$ and for any unit-length vector $x \in \mathbf{R}^d$, the claim below holds.[3]

$$P(|\|Ax\|_2^2 - 1| > \varepsilon) < \delta$$

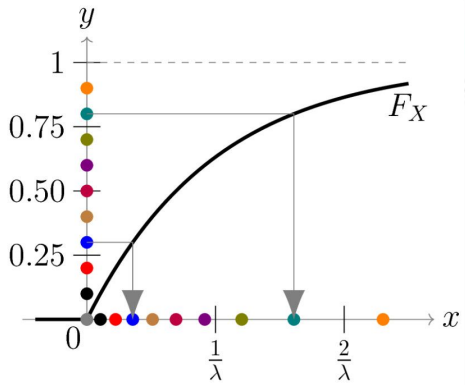
One can obtain the JL lemma from the distributional version by setting $x = (u - v)/\|u - v\|_2$ and $\delta < 1/n^2$ for some pair u,v both in X. Then the JL lemma follows by a union bound over all such pairs.

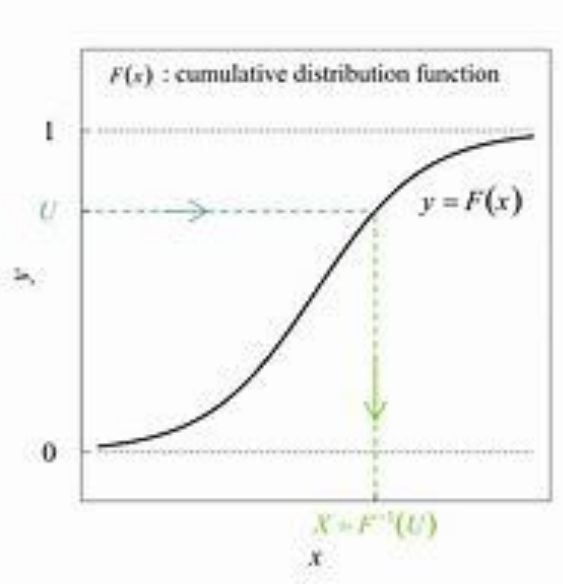
Sampling Fundamentals

- Assume we have access to uniform random samples **X~U[0,1]**
  - Assume infinite precision for our analysis.
  - In practice, pseudo-random samples with a close-to-uniform distribution are good enough.
- How do we use **U[0,1]** to generate samples from other distributions?
  - Gaussian, Poisson, Exponential, etc.
  - Discrete distributions?
- Easy Transforms: Scale / Translate / Rotate the input space
  - Arbitrary uniform distribution: **U[a,b] = a + (b-a) U[0,1]** for any a < b

Fundamentals: Inverse-Transform Sampling

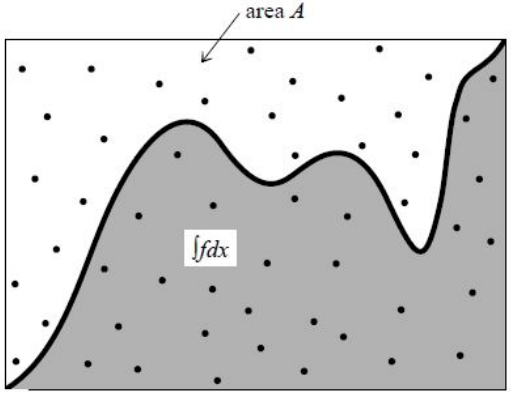
- **Algorithm:** Sampling from a continuous distribution with cdf **F**
  - Generate a uniform random sample **u** ← **U[0,1]**
  - Find X such that F(X) = u (compute the inverse cdf)
- Pro: Relatively simple, efficient if inverse is known analytically.
- Con: Need to compute inverse.
- Con: Higher dimensions?

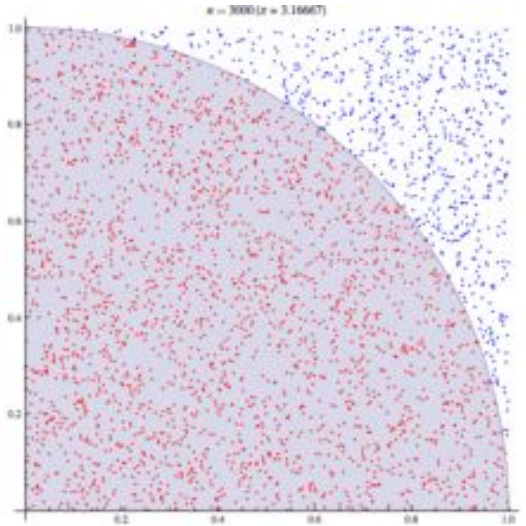




Fundamentals: Rejection Sampling

- **Basic Idea:** Suppose we want random points from a region S in the plane.
  - Generate points from the “bounding box” containing S, and throw out any that aren’t in S!
- **Refinement:** Sample from density f(x) via a proposal distribution g(x)
  - g(x) should “envelope” f(x), that is, graph of f(x) lies under g(x)
  - sample **Y ~ g(y)** and **U ~ U[0,1]**
  - if U < f(Y) / g(Y), accept Y as a sample of f
  - otherwise, reject Y and re-sample
- More Sampling:
  - Importance Sampling
  - MCMC





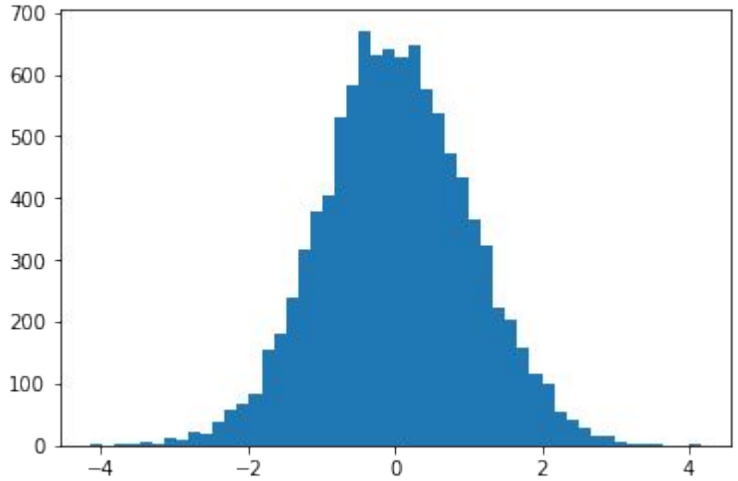
Fundamentals: Box-Muller Transform

- **Algorithm:** Box-Muller for Sampling from Unit Gaussian
  - Sample X,Y ~ Unif[0,1]
  - The following are guaranteed to be independent samples with Normal(0,1) distribution

$$Z_0 = R\cos(\Theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$
$$Z_1 = R\sin(\Theta) = \sqrt{-2\ln U_1}\sin(2\pi U_2).$$

- Non-unit Gaussian?
  - Translate and scale!

```
# Box-Muller
def normal_sample(num, a):
    num //= 2;
    first = np.sqrt(-2*np.log(np_rnd.random(num)));
    u2 = 2*np.pi*np_rnd.random(num);
    return np.concatenate([first*np.cos(u2), first*np.sin(u2)]);

plt.hist(normal_sample(10000, 1), bins=50);
```



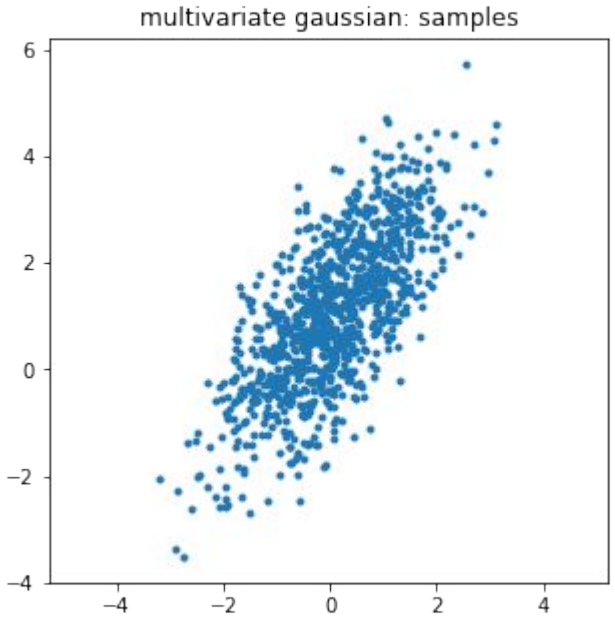
Fundamentals: Multivariate Normal

- **Def:** Random vector X is *normally distributed* if every linear combination of its components ($Y = a_1 X_1 + \cdots + a_k X_k$) is normally distributed.

To sample from a multivariate Gaussian $\mathcal{N}(\mu, \Sigma)$ for $\Sigma \in \mathbb{R}^{d \times d}$,

1. Compute the Cholesky decomposition $\Sigma = AA^T$ for lower-triangular $A \in \mathbb{R}^{d \times d}$
2. Sample d independent standard normals $Z_1, \ldots, Z_d \overset{iid}{\sim} \mathcal{N}(0, 1)$
3. Compute $X_k = \mu + AZ_k$. The vector $(X_1, \ldots, X_d)$ will be a sample from $\mathcal{N}(\mu, \Sigma)$.

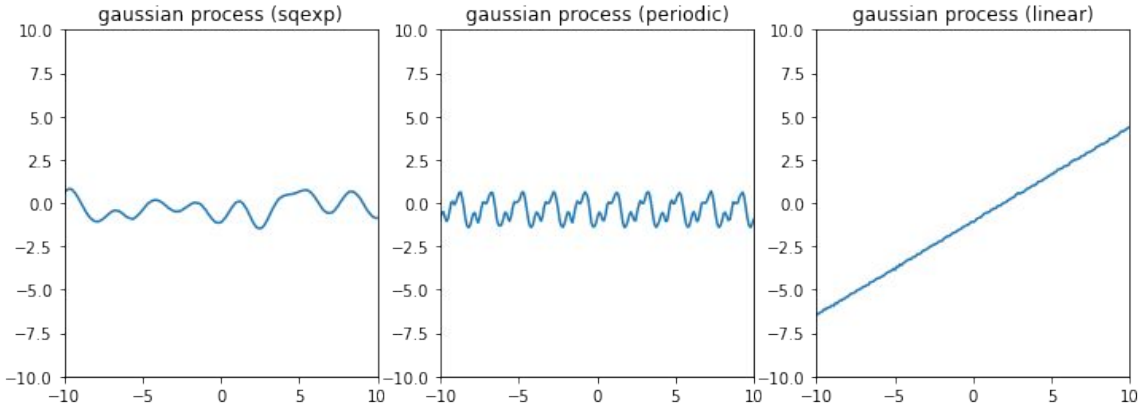
```python
def gaussian_sample(mean_vec, cov_mat, num_samples=1):
    d = len(mean_vec);
    assert(cov_mat.shape == (d,d));

    # Step 1:  compute cholesky factor
    A = np.linalg.cholesky(cov_mat);
    # Step 2:  sample d independent normals
    z = np.random.randn(d, num_samples);
    # Step 3:  skew independent samples by A to correlate them
    return A @ z + mean_vec[:,np.newaxis];
```



Fundamentals: Gaussian Process

A Gaussian process is a stochastic process $(X_t)_{t\in I}$ such that any finite subcollection of the X_t forms a Gaussian vector. In particular, $(X_t)_{t\in I} \sim \mathcal{GP}(\mu, k)$ is a Gaussian process with **mean function** $\mu : I \to \mathbb{R}$ and **covariance function** $k : I \times I \to \mathbb{R}$ if for any finite index set $\{t_1, t_2, \ldots, t_n\} \subset I$,

$$\begin{bmatrix} X_{t_1} \\ \vdots \\ X_{t_n} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(t_1) \\ \vdots \\ \mu(t_n) \end{bmatrix}, \begin{bmatrix} k(t_1,t_1) & \cdots & k(t_1,t_m) \\ \vdots & \ddots & \vdots \\ k(t_m,t_1) & \cdots & k(t_m,t_m) \end{bmatrix} \right)$$




```
def gp_sample(x, mean_func, cov_func, **kwargs):
    mean_vec = mean_func(x);
    cov_mat  = cov_func(x, **kwargs);
    return gaussian_sample(mean_vec, cov_mat);
```

Random Matrices

(in not nearly enough detail)

Matrix Ensembles

A **matrix ensemble** is a probability distribution over a family of matrices.

- **Gaussian Orthogonal Ensemble (GOE):** $\mathbb{R}^{n\times n}$ with real Gaussian entries
- **Gaussian Unitary Ensemble (GUE):** $\mathbb{C}^{n\times n}$ with complex Gaussian entries
- **Gaussian Symplectic Ensemble:** Gaussian quaternion entries

> *Note: GOE and GUE are invariant under orthogonal and unitary transformations, respectively. They **do not** necessarily produce random orthogonal and unitary matrices)*

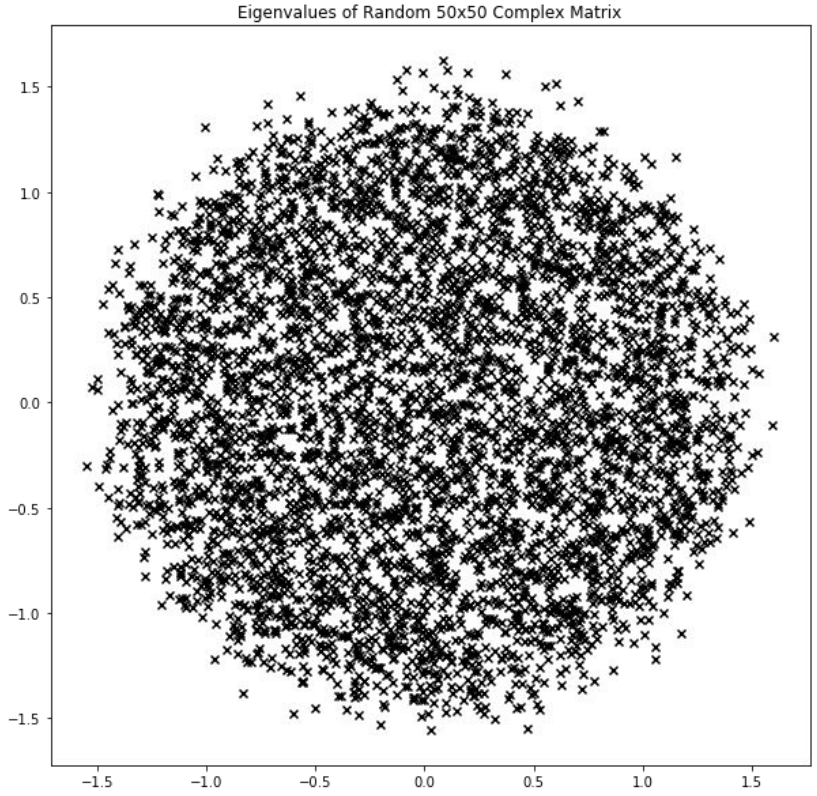
Matrix Ensembles: Circular Laws

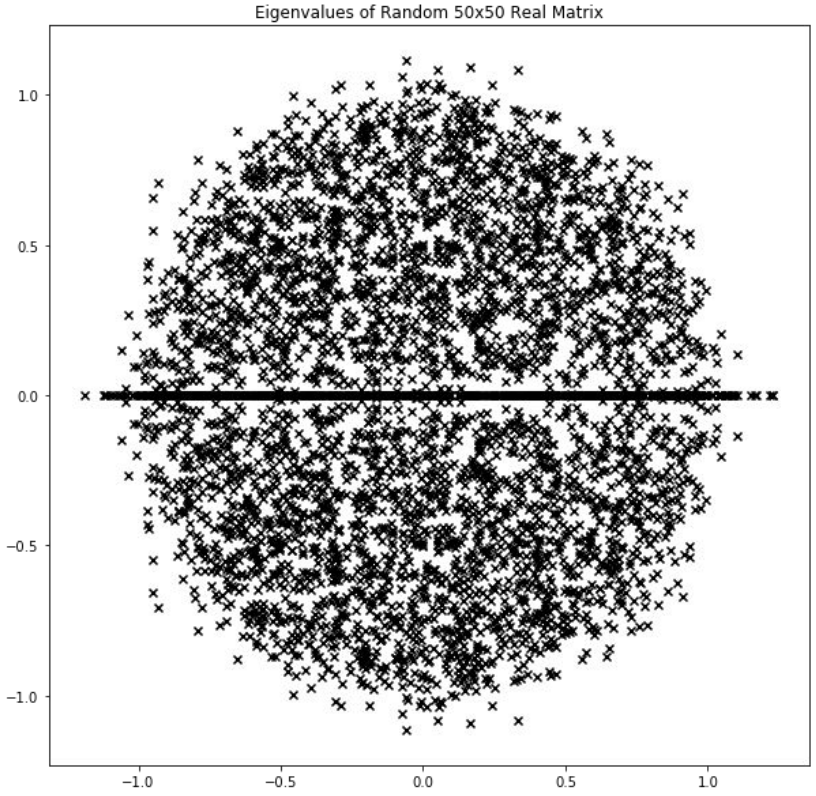
How are the eigenvalues of a random matrix distributed?

> **Thm:** (Tao 2007) Let $M_n \in \mathbb{C}^{n \times n}$ have complex entries drawn iid from a distribution with mean 0 and variance 1. Then as $n \to \infty$, the distribution of the eigenvalues of $\frac{1}{\sqrt{n}} M$ tends to the uniform distribution on the unit disk.

- Relatively easy to show for GOE, but not proven in general until recently.
- Like a **“Central Limit Theorem”** for random matrices.

Matrix Ensembles: Circular Laws

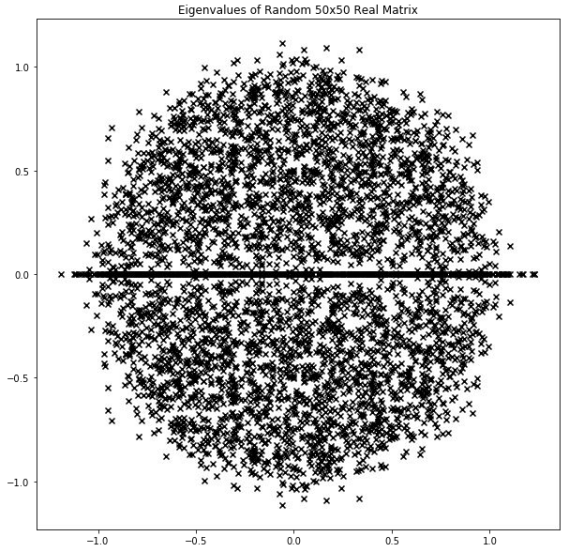




Matrix Ensembles: Circular Laws

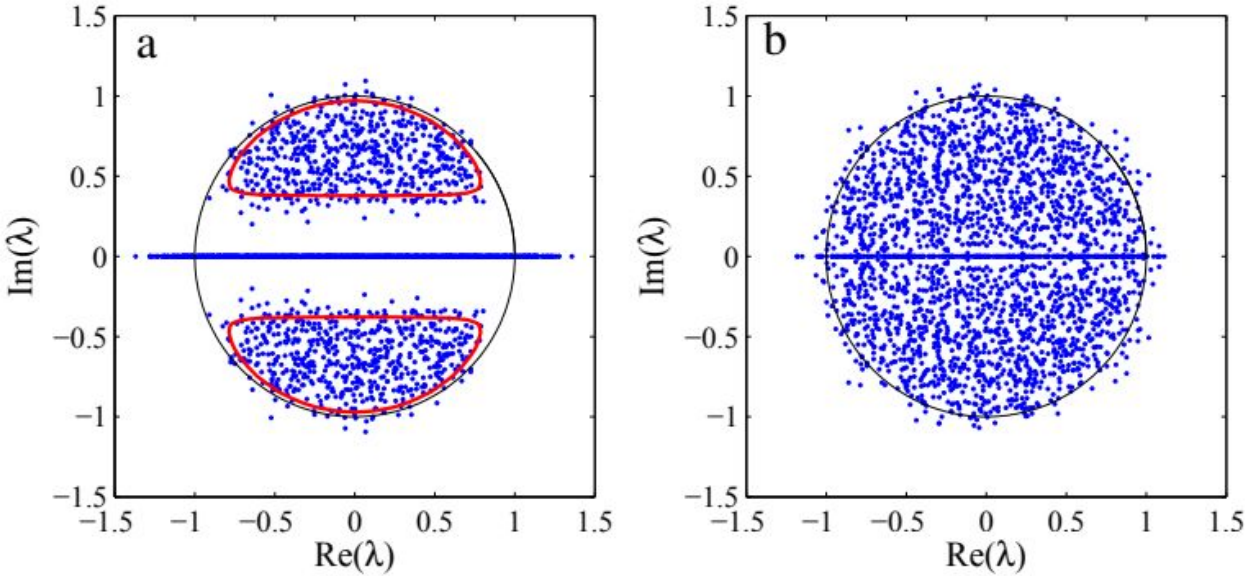
The circular law is verified empirically in Figure 4 for the Gaussian ensembles. For the real case on the right, there is an unexpected concentration of eigenvalues exactly on the real line, suggesting that the eigenvalue distribution is *not absolutely continuous* with respect to Lebesgue measure. The joint eigenvalue density for the GOE was worked out explicitly by (Edelman 1997) and integrated, leading to the following computation:

Theorem 2. *(Edelman 1997) Let $M \in \mathbb{R}^{n \times n}$ have independent standard normal entries. The probability that M has k eigenvalues has the form $r + s\sqrt{2}$ for some rational $r, s \in \mathbb{Q}$. In particular, the probability that a random matrix has all real eigenvalues is $1/2^{n(n-1)/4}$.*



More Hard Questions

- What's the distribution of eigenvalues of a random matrix, conditioned on having k real eigenvalues? **(del Molino et al. 2015)**
  - Answer: *It's complicated!!*



More Hard Questions

- What is the characteristic polynomial of a random unitary matrix?
  - see e.g. [Bourgade et al. 2008]
- How do we generalize the normal distribution to manifolds?
  - [Rains 1997] “Combinatorial Properties of Brownian Motion on the Compact Classical Groups”
- If we solve Ax=b for random A, what is the distribution of floating-point error?
  - [Hennig 2015] “Probabilistic Interpretation of Linear Solvers”
- How to sample a random graph where each node has a specified degree?
  - [Zhao 2014] “A Linear Time Algorithm for Sampling Graphs with Given Degrees”

Random Structured Matrices

How can we generate random matrices with a specific structure?

- Random diagonal matrix?
- Random upper-triangular matrix?
- Random symmetric / Hermitian matrix?
- Random orthogonal / unitary matrix?

What exactly does it mean for a matrix to be random?

Random Structured Matrices

Given a matrix with random entries `A = np.random.randn(n,n)`, we might try:

- Random diagonal matrix `diag(randn(n))`
- Random symmetric matrix `A + A.T`
- Random lower-triangular matrix `L = np.chol(A)`
- Random upper-triangular matrix `_,R = np.linalg.qr(A)`
- Random orthogonal matrix `Q,_ = np.linalg.qr(A)`

What is the resulting distribution? Is it sufficiently *"uniform"*? Can we do better?

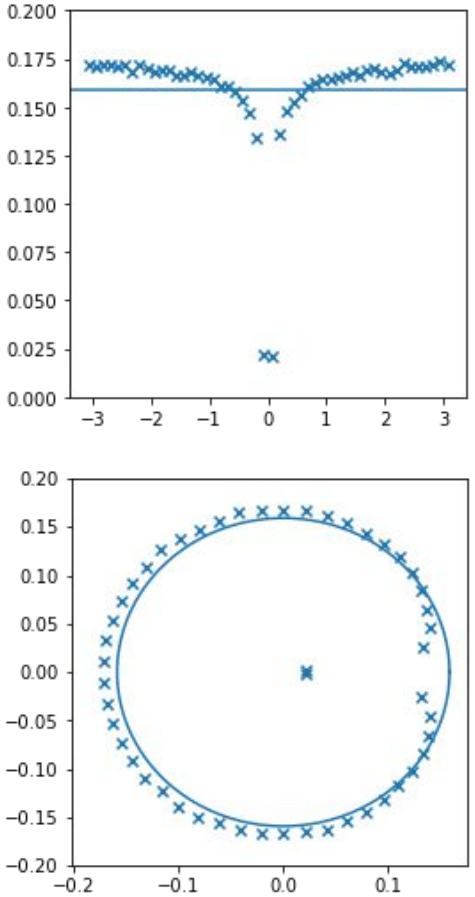
Q: How do deterministic algorithms behave when fed random inputs?

Random Unitary Matrices

A matrix $U \in \mathbb{C}^{n \times n}$ is **unitary** if it has orthonormal columns $UU^* = I$.

- Eigenvalues $\lambda = e^{i\theta}$ have unit modulus and $|\det U| = 1$
- In the real case, can *roughly* be thought of as rotations/reflections

```
# Mezzadri 2006
def standard_qr(n):
    # sample from GUE
    Z = randn(n,n) + 1j*randn(n,n) / np.sqrt(2);
    # standard qr factorization
    Q,R = np.linalg.qr(Z);
    return Q;
```



Why isn't the eigenvalue phase distribution uniform?

Random Unitary Matrices

The QR decomposition is not unique. Let $Z \in \mathrm{GL}(n, \mathbb{C})$ and suppose $Z = QR$, where Q is unitary and R upper-triangular. Then for any $\Lambda = \mathrm{diag}(e^{i\theta_1}, \ldots, e^{i\theta_n})$,

$$Z = QR = (Q\Lambda^{-1})(\Lambda R)$$

is a different valid QR factorization of Z. Therefore QR factorization is a **multi-valued** map

$$QR : GL(n, \mathbb{C}) \to \mathrm{Unitary}(n) \times \mathrm{InvUpperTri}(n)$$

Different factorization algorithms choose different principal values.

Either 1) Design an algorithm to pick principal values intelligently
Or 2) Find a way to correct the output of existing algorithms.

Corrected QR

Lemma 2. *Let $Z \in \mathbb{C}^{n\times n}$ have two valid QR decompositions $Z = Q_1R_1 = Q_2R_2$. Then, there is $\Lambda \in \Lambda_n(\mathbb{C})$ such that $Q_2 = Q_2\Lambda^{-1}$ and $R_2 = \Lambda R_1$.*

Consider the quotient group $\Gamma_n(\mathbb{C}) = T_n(\mathbb{C})/\Lambda_n(\mathbb{C})$. Our corrected algorithm will be a one-to-one map

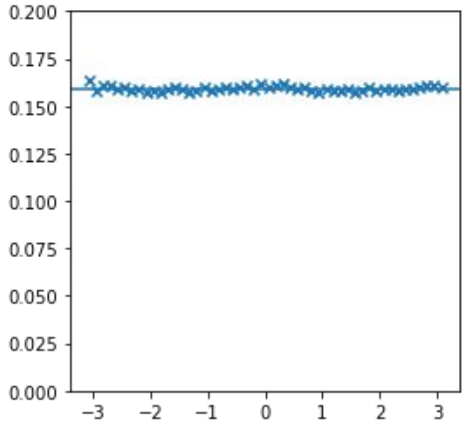
$$\widehat{\mathrm{QR}} : \mathrm{GL}_n(\mathbb{C}) \to \mathrm{U}_n(\mathbb{C}) \times \Gamma_n(C)$$

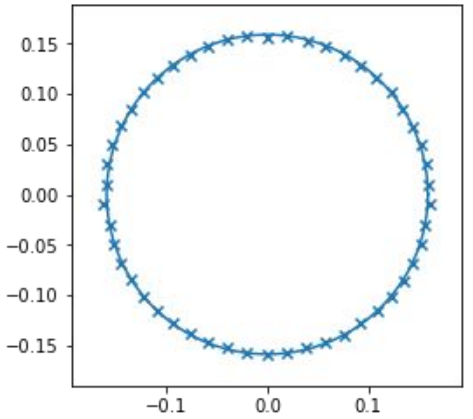
The upper-rectangular matrix returned by the corrected algorithm will be a representative γ of $\Gamma_n(\mathbb{C})$. If we choose the map $\widehat{\mathrm{QR}}$ to be unitarily invariant with respect to γ, that is,

$$Z \mapsto (Q, \gamma) \implies UZ \mapsto (UQ, \gamma) \text{ for all } Uin\mathrm{U}_n(\mathbb{C})$$

then by Lemma 1, the algorithm $\widehat{\mathrm{QR}}$ with input from a Gaussian Unitary Ensemble will be distributed according to the Haar measure on $U_n(\mathbb{C})$. It can be shown that the

```
def haar_measure(n):
    # sample from Ginibre ensemble
    Z = randn(n,n) + 1j*randn(n,n) / np.sqrt(2);
    # standard qr factorization
    Q,R = np.linalg.qr(Z);
    # correction
    d = np.diag(R);
    PH = np.diag(d) / np.absolute(d);
    return Q @ PH;
```

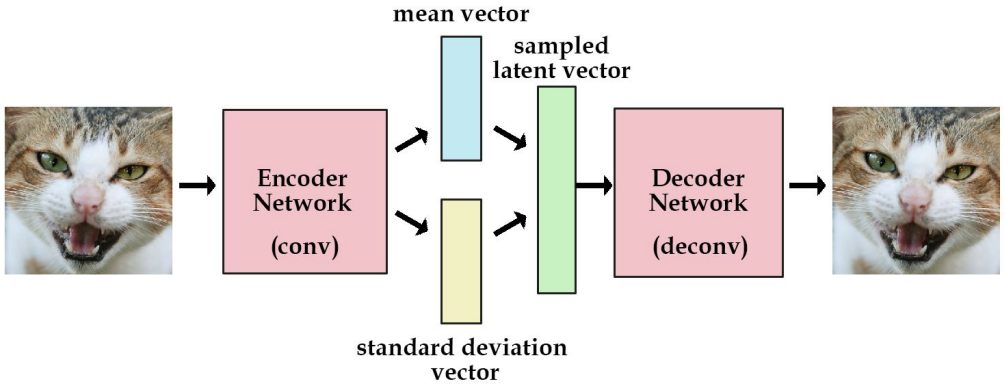




Machine Learning

Generative Modeling

- Modern machine learning is all about **learning** deterministic functions capable of mapping **U[0,1]^d** or **N[0,1]^d** to arbitrary output distributions
  - Variational Auto-Encoders
  - Generative Adversarial Networks
- More buzzwords:
  - reparameterization trick
  - deep convolutional networks
  - amortization gap



http://kvfrans.com/variational-autoencoders-explained/

Log-Sum-Exp Trick (see Tim Viera's blog)

```
def exp_normalize(x):
    b = x.max()
    y = np.exp(x - b)
    return y / y.sum()

>>> exp_normalize(x)
array([0., 0., 1.])
```

Supposed you'd like to evaluate a probability distribution $\boldsymbol{\pi}$ parametrized by a vector $\boldsymbol{x} \in \mathbb{R}^n$ as follows:

$$\pi_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

The exp-normalize trick leverages the following identity to avoid numerical overflow. For any $b \in \mathbb{R}$,

$$\pi_i = \frac{\exp(x_i - b)\exp(b)}{\sum_{j=1}^{n} \exp(x_j - b)\exp(b)} = \frac{\exp(x_i - b)}{\sum_{j=1}^{n} \exp(x_j - b)}$$

In other words, the $\boldsymbol{\pi}$ is shift-invariant. A reasonable choice is $b = \max_{i=1}^{n} x_i$.

Gumbel-Max Trick (see Tim Viera's Blog)

The Gumbel-max trick:

$$y = \underset{i \in \{1,\cdots,K\}}{\operatorname{argmax}} \; x_i + z_i$$

where $z_1 \cdots z_K$ are i.i.d. $\textbf{Gumbel}(0, 1)$ random variates. It turns out that y is distributed according to π. (See the short derivations in this blog post.)

Implementing the Gumbel-max trick is remarkable easy:

```
def gumbel_max_sample(x):
    z = gumbel(loc=0, scale=1, size=x.shape)
    return (x + z).argmax(axis=1)
```

If you don't have access to a Gumbel random variate generator, you can use $-\log(-\log(\text{Uniform}(0, 1))$

Gumbel-Max Trick

The **Gumbel-Max trick** is a way to sample from log-parameterized ($\alpha_k = \log \pi_k$) discrete distributions.

$$z_1, \ldots, z_n \overset{iid}{\sim} \text{Gumbel}(0, 1)$$
$$y = \arg \max_{j=1,\ldots,n} \{\alpha_j + z_j\} \sim \text{Discrete}(\pi_1, \ldots, \pi_n)$$

Comparison to inverse sampling:

- Gumbel-max requires N uniform samples to generate **one** discrete sample, compared to one uniform sample for the inverse CDF method.
- If we are sampling K values, using uniform variables to generate Gumbel samples, we require $2K$ calls to log, whereas the inverse method requires K calls to exp. Both log and exp are expensive.
- Gumbel-max can be used to sample in a streaming fashion. Notice that the *probabilities are unnormalized* and the terms in the argmax don't depend on each other. If we partially evaluate the arg max without seeing all indices, our y will be a discrete sample from the indices seen so far! See this blog post by Tim Viera for an application to reservoir sampling.

```
def discrete_sample_gumbelmax(log_probs):
    # 1. sample gumbel variables
    z = gumbel_sample_invcdf(log_probs.shape[0]);
    # 2. arg max
    return np.argmax(log_probs + z);
```

Concrete Distribution

https://casmls.github.io/general/2017/02/01/GumbelSoftmax.html

Thank You!