1 Implementing Alternating Gradient Descent for Low Rank Matrix Completion in CUPY

1.1 Algorithm

We will use the algorithm produced by Jared Tanner and Ke Wei in their 2015 paper. Suppose we have a matrix $W \in \mathbb{R}^{m\times n}$ with a known max rank k but we only know a small portion of the indecies. If there are enough indecies (depending on the size, rank, and distribution of known indecies) then the following algorithm will recover the rest of them. Let $Z \in \mathbb{R}^{m\times n}$ be a matrix containing all known indecies of W with unkown indecies set to 0. We define $\Omega \in \{0,1\}^{m\times n}$ to be the boolean valued matrix that perfectly masks the known indecies of W, namely:

$$\Omega_{ij} = 1 \iff W_{ij} \text{ is known}$$

Then we use Ω to define the sparse projection operator

$$P_\Omega : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$$
$$P_\Omega(A)_{ij} := A_{ij} \cdot \Omega_{ij}$$

In particular we have that $P_\Omega(Z) = P_\Omega(W) = Z$. The number of known indecies is therefore given by $s = \sum_{ij} \Omega_{ij}$. We can represent arbitrary rank k matrices as a product XY with $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{k\times n}$ and our goal can be expressed as

$$W = XY \text{ given } \ P_\Omega(XY) = Z$$

We then relax the equality to an ℓ^2 norm to get a polynomial optimization problem:

$$\text{argmin}_{X,Y} ||P_\Omega(XY - Z)||_2^2 \quad X \in \mathbb{R}^{m\times k}, Y \in \mathbb{R}^{k\times n}$$

1.2 Power Factorization Algorithm (PF)

While this problem is very difficult to solve in X, Y simultaneously, it becomes a straightforward quadratic problem if either X or Y held fixed. Our strategy is then to alternate between fixing X and finding the best possible Y, then fixing Y and finding the best possible X. The process is described as:

Algorithm 1 Power Factorization

Input: $\Omega \in \{0,1\}^{m\times n}, P_\Omega(W) = Z, X_0 \in \mathbb{R}^{m\times r}, Y_0 \in \mathbb{R}^{r\times n}$

repeat

$$X_{i+1} \leftarrow \text{argmax}_X ||P_\Omega(XY_i - Z)||_2^2$$
$$Y_{i+1} \leftarrow \text{argmax}_Y ||P_\Omega(X_{i+1}Y - Z)||_2^2$$

until Termination condition is reached

There are a number of observations we can make in order to simplify the sub optimizations. First of all the optimization problems are indepedent for each row of X and each column of Y so we can simplifyto a large number of simpler subproblems. Let $\{x^i\}_{i=1}^m$ be the rows of X, and let $\{z^i\}_{i=1}^m$ be the rows of Z, we compute the optimal row data as:

$$x^i = z^i Y^t (Y_\Omega^i Y^t)^{-1} \text{ where } (Y_\Omega^i)_{rs} = \begin{cases} 0 & \text{if } \Omega_{is} = 0 \\ Y_{rs} & \text{otherwise} \end{cases}$$

The computation for the columns of Y are completely analogous. There are a number of issues that make this algorithm less desirable for large data. First of all we need to compute for each row (m times) the matrix Y_Ω^i, a k by k matrix inversion, and a $1-n, n-k, k-k$ matrix product. Second of all the inversion could cause arithmatic errors if the resultant matrix is close to singular. Finally we are making very poor use of the sparse structure. Note that computation of Y_Ω^i is equivalent to filtering out a sparse set of columns of Y.

1.3 Alternating Steepest Descent

The solution proposed by Tanner and Wei was to replace finding the best possible X or Y with the best possible step in the gradient direction. The proposed algorithm was:

Algorithm 2 Alternating Steepest Descent (ASD)

Input: $\Omega \in \{0,1\}^{m \times n}, P_\Omega(W) = Z, X_0 \in \mathbb{R}^{m \times r}, Y_0 \in \mathbb{R}^{r \times n}$

repeat

$$\nabla f_{Y_i}(X_i) \leftarrow P_\Omega(X_i Y_i - Z) Y_i^T, \qquad t_{x_i} \leftarrow \frac{||\nabla f_{Y_i}(X_i)||_2^2}{||P_\Omega(\nabla f_{Y_i}(X_i) Y_i)||_2^2}$$

$$X_{i+1} \leftarrow X_i - t_{x_i} \nabla f_{Y_i}(X_i)$$

$$\nabla f_{X_{i+1}}(Y_i) \leftarrow X_{i+1}^T P_\Omega(X_{i+1} Y_i - Z), \qquad t_{y_i} \leftarrow \frac{||\nabla f_{X_{i+1}}(Y_i)||_2^2}{||P_\Omega(X_{i+1} \nabla f_{X_{i+1}}(Y_i))||_2^2}$$

$$Y_{i+1} \leftarrow Y_i - t_{y_i} \nabla f_{X_{i+1}}(Y_i)$$

until Termination condition is reached

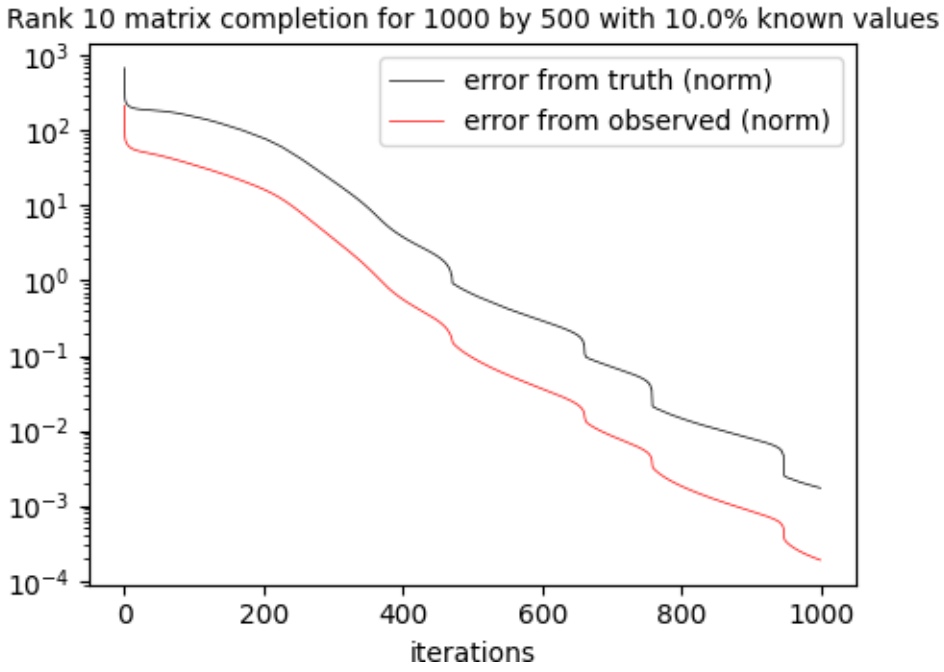
This algorithm makes excellent use of the sparse structure. When computing terms like $P_\Omega(AB)$ where $A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}$. Instead of allocating $m \times n \times 4$ bytes of memory (assuming float32 calculations) to compute the matrix product and then throwing out most of the coefficients we can instead only compute the s coefficients we end up using. While you might hope that having a sparsity of 1% means that the matrix multiplication is also 100 times faster than the naive approach, in practise we lose a lot of performance from not working on contiguous blocks of memory.

1.4 Implementing ASD in python on CPU

```
import scipy as sp
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

# maximum number of iterations
MAX_ITERATIONS = 1000
# number of rows
M = 1000
# number of columns
N = 500
# max rank of matrix, must be less than or equal to n and m
R = 10

# density
D = 0.1

# produce a random matrix with the specified rows, columns, and max rank
def createRandomMaxRank(rows: int, columns: int, rank: int) -> np.matrix:
    return np.matrix(np.random.rand(rows, rank)) * np.matrix(np.random.rand(rank, columns))

# produce a random sparse mask with specified dimensions and fill percentage
def createRandomSparseMask(rows: int, columns: int, density: float ) ->sp.sparse:
    return sp.sparse.random(rows, columns, density = density,
                            random_state=None, data_rvs= (lambda l: np.array([1]*l)))

# generate a random matrix with specified max rank
# true rank of the matrix may end up being lower but it is unlikely
# however distribution of singular values will probably not be very uniform
```

```
truth = createRandomMaxRank(M, N, R)

# generate a random sparse mask to act as a random sample of our low rank matrix
mask = createRandomSparseMask(M, N, D)

# perform a Hadamard (componentwise) multiplication to get a sparse sampling of our low rank matrix
observed = mask.multiply(truth)

# X: reduced rank left hand operand M*R matrix from previous step or random guess
# Y: reduced rank right hand operand R*N matrix from previous step or random guess
# mask: sparse {0,1} valued matrix aligned with sparse target
# Z: M*N sparse matrix with values aligned with mask.
def ASD_iteration(X: np.matrix, Y: np.matrix, mask: sp.sparse, Z_0: sp.sparse):
    # gradients of steppest ascent for objective function with that argument held fixed
    grad_Y_f = (mask.multiply(X@Y) - Z_0)@Y.T
    # computed step sizes
    t_x = np.linalg.norm(grad_Y_f) ** 2 / sp.sparse.linalg.norm(mask.multiply(grad_Y_f @ Y)) ** 2
    X_1  = X - t_x * grad_Y_f
    grad_X_f = X_1.T @ (mask.multiply(X_1 @ Y) - Z_0)
    t_y = np.linalg.norm(grad_X_f) ** 2 / sp.sparse.linalg.norm(mask.multiply(X_1 @ grad_X_f)) ** 2
    Y_1 = Y - t_y * grad_X_f
    return X_1, Y_1

# create a random guess for X_0 and Y_0 as our starting point
# in practise these should be scaled to be comparable to the data
X_0 = np.random.rand(M, R);
Y_0 = np.random.rand(R, N);

errors = np.zeros(MAX_ITERATIONS)
errors2 = np.zeros(MAX_ITERATIONS)
startTime = datetime.now()
for x in range(MAX_ITERATIONS):
    errors[x] = np.linalg.norm((X_0 @ Y_0) - truth)
    errors2[x] = sp.sparse.linalg.norm(mask.multiply((X_0 @ Y_0) - truth))
    X_0, Y_0 = ASD_iteration(X_0, Y_0, mask, observed)
plt.plot(errors, label="error from truth (norm)", color= (0,0,0), linewidth=0.5,)
plt.plot(errors2, label ="error from observed (norm)", color= (1,0,0), linewidth=0.5)
plt.yscale("log")
plt.xlabel("iterations", fontsize='8')
plt.legend(fontsize='10')
plt.title("Rank " + str(R) +" matrix completion for " + str(M) + " by "
          + str(N) + " with " + str(D*100) + "% known values", fontsize='10')
print("completed in: ")
print(datetime.now() - startTime)
plt.show()
```

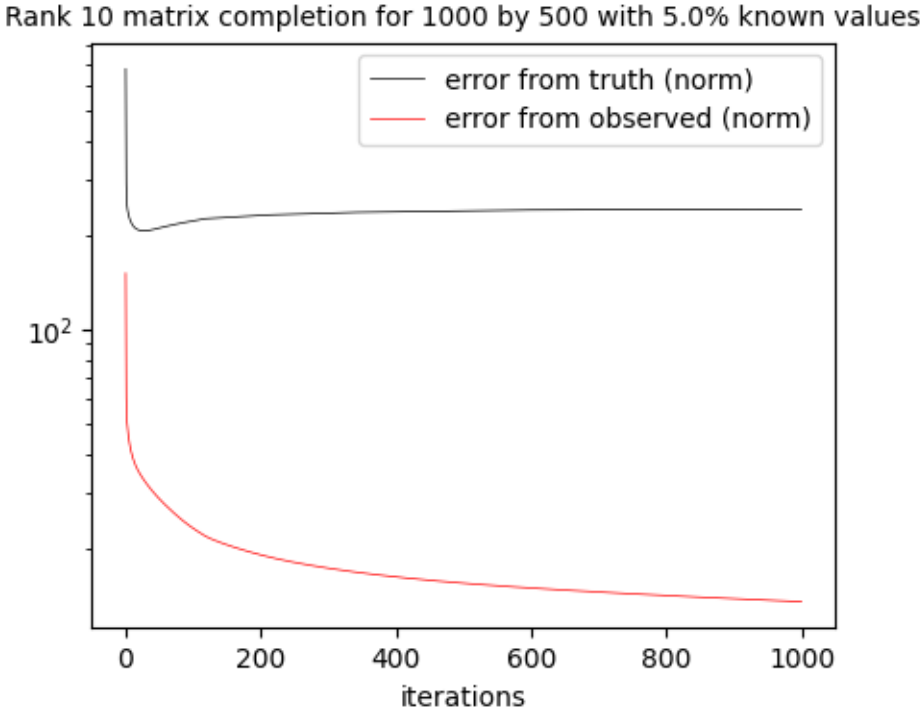
Python is excellent for rapid prototyping but trying to do any native arithmatic is prohibitively slow. To that end we use the numpy and scipy packages to execute matrix algebra routines using precompiled c code. The example code above took 1 minute to run on a 4670k CPU and produced the following output:



Note that the average size of indecies in truth was 2.5 so if we want to ensure that at most 0.1% of predicted values had an error greater than 0.01 the total error should be smaller than

$$\sqrt{n \cdot m \cdot \text{error\_threshold}^2 \cdot \text{error\_frequency}}$$

which is 0.2236 in this example. As can be seen it takes almost 800 iterations to reach this point. Since the code uses a different random matrix as the target each time there is some variance in the convergence. As can be seen from the graph the error decays exponentially in the good case. If there is an insufficient amount of information provided (i.e the sparsity is too low) then the output looks more like this:



In practise we do not have the true values to compare against but we can simply look at the error curve to determine whether we are in a convergent regime or not. As for our initial guess, a cursory analysis suggests that random matrices work fine as long as our guess and the truth has a similar singular value profile.

1.5 Running Algebra on the GPU with CUPY

I chose CUPY for a GPU implementation because it was designed to be most compatible with the numpy and scipy libraries. Due to installation issues I ended up using NVIDIA GPU Computing Toolkit v11.0 and Visual studio 2017 for the c++ compiler. The GPU used was a gtx 1070 with 8GB of ram, 1920 CUDA cores and compute capability 6.1 with CUDA support 8+. The code provided should be compatible with any NVIDIA GPU from that generation onward using toolkit 12.x and visual studio 2022.

```
import cupy as cp
from cupyx.scipy.sparse import linalg
from cupyx.scipy.sparse import random as srandom
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

# maximum number of iterations
MAX_ITERATIONS = 1000
# number of rows
M = 1000
# number of columns
N = 500
# max rank of matrix, must be less than or equal to n and m
R = 10

# density
D = 0.1

# produce a random matrix with the specified rows, columns, and max rank
def createRandomMaxRank(rows: int, columns: int, rank: int) -> cp.ndarray:
    return cp.matmul(cp.array(cp.random.rand(rows, rank)),cp.array(cp.random.rand(rank, columns)))

# produce a random sparse mask with specified dimensions and fill percentage
def createRandomSparseMask(rows: int, columns: int, density: float ) ->cp.sparse:
    return srandom(rows, columns, density = density,
                   random_state=None, data_rvs= (lambda l: cp.array([1]*l)))

# generate a random matrix with specified max rank
# true rank of the matrix may end up being lower but it is unlikely
# however distribution of singular values will probably not be very uniform
truth = createRandomMaxRank(M, N, R)

# generate a random sparse mask to act as a random sample of our low rank matrix
mask = createRandomSparseMask(M, N, D)

print(cp.shape(truth))
print(cp.shape(mask))

# perform a Hadamard (componentwise) multiplication to get a sparse sampling of our low rank matrix
observed = mask.multiply(truth)


# X: reduced rank left hand operand N*R matrix from previous step or random guess
# Y: reduced rank right hand operand R*M matrix from previous step or random guess
```

```
# mask: sparse {0,1} valued matrix aligned with sparse target
# Z: N*M sparse matrix with values aligned with mask.
def ASD_iteration(X: cp.ndarray, Y: cp.ndarray, mask: cp.sparse, Z_0: cp.sparse):
    # gradients of steppest ascent for objective function with that argument held fixed
    grad_Y_f = (mask.multiply(X @ Y) - Z_0) @ (Y.T)
    # computed step sizes
    t_x = cp.linalg.norm(grad_Y_f) ** 2 / linalg.norm(mask.multiply(grad_Y_f @ Y)) ** 2
    X_1  = X - t_x * grad_Y_f
    grad_X_f = X_1.T @ (mask.multiply(X_1 @ Y) - Z_0)
    t_y = cp.linalg.norm(grad_X_f) ** 2 / linalg.norm(mask.multiply(X_1 @ grad_X_f)) ** 2
    Y_1 = Y - t_y * grad_X_f
    return X_1, Y_1

# create a random guess for X_0 and Y_0 as our starting point
# in practise these should be scaled to be comparable to the data
X_0 = cp.random.rand(M, R);
Y_0 = cp.random.rand(R, N);

errors = np.zeros(MAX_ITERATIONS)
errors2 = np.zeros(MAX_ITERATIONS)
#benchmarking
startTime = datetime.now()
mempool = cp.get_default_memory_pool()
pinned_mempool = cp.get_default_pinned_memory_pool()
for x in range(MAX_ITERATIONS):
    errors[x] = cp.linalg.norm((cp.matmul(X_0, Y_0)) - truth)
    errors2[x] = linalg.norm(mask.multiply((cp.matmul(X_0, Y_0)) - truth))
    X_0, Y_0 = ASD_iteration(X_0, Y_0, mask, observed)
plt.plot(errors, label="error from truth (norm)", color= (0,0,0), linewidth=0.5,)
plt.plot(errors2, label ="error from observed (norm)", color= (1,0,0), linewidth=0.5)
plt.yscale("log")
plt.xlabel("iterations", fontsize='8')
plt.legend(fontsize='10')
plt.title("Rank " + str(R) +" matrix completion for " + str(M) + " by "
          + str(N) + " with " + str(D*100) + "% known values", fontsize='10')
print("completed in: ")
print(datetime.now() - startTime)
plt.show()
```

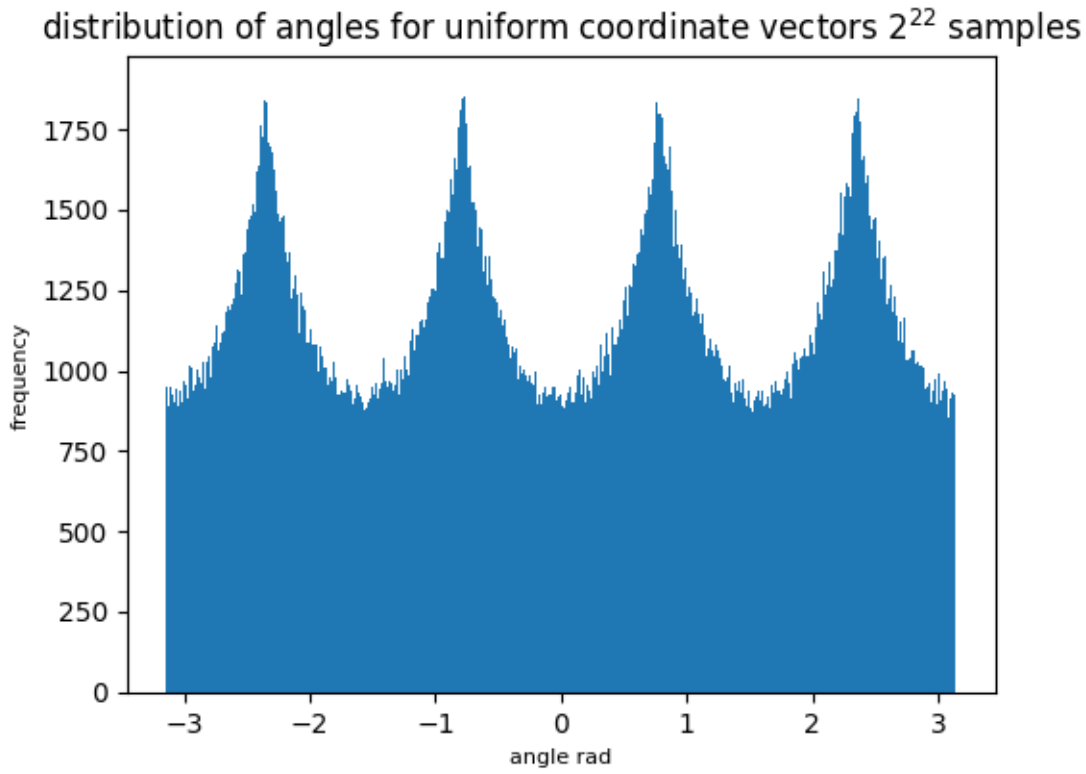
This code runs slower overall for small matrices because each matrix algebra operation needs to be compiled into a GPU kernel.

1.6 Generating Large Random Low Rank Matrices

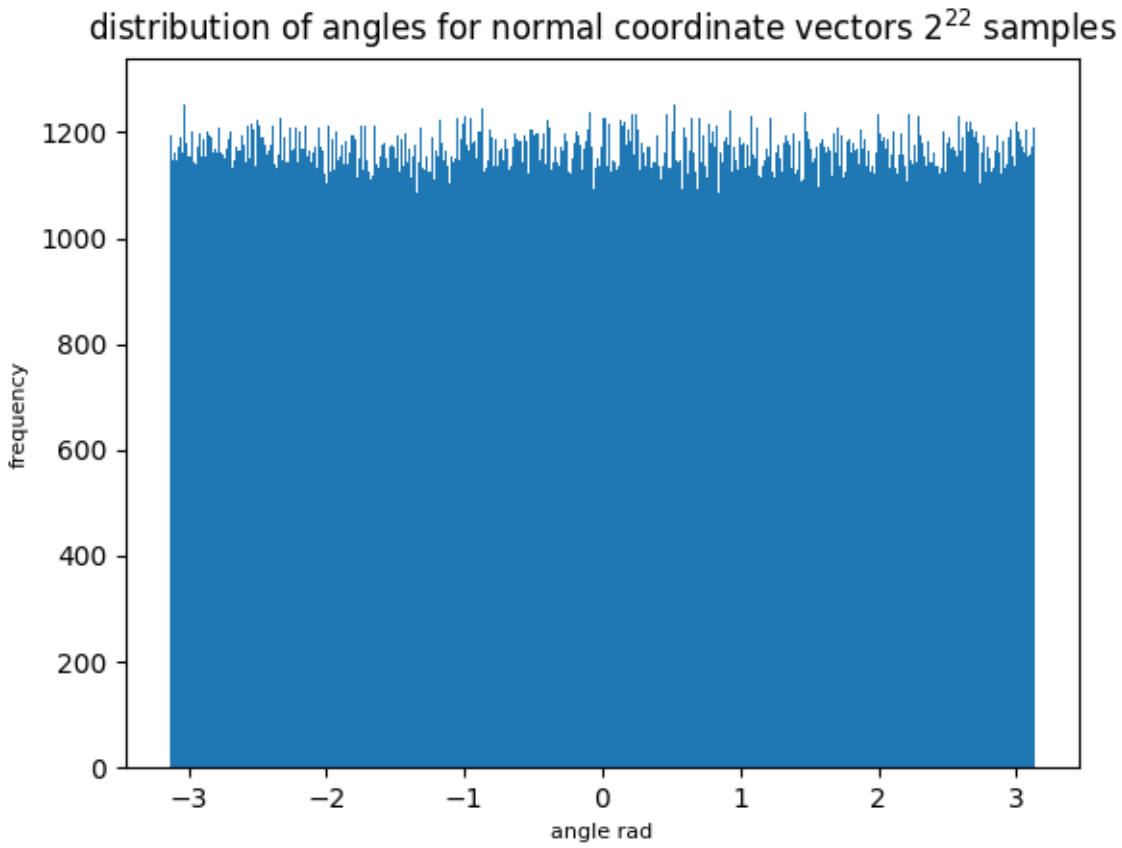
In order to test our algorithm a good starting point is randomly generated matrices. However we must be careful, we need to make sure that the matrices we generate are a fair representation of the parameter space. We also want to make sure that the random matrices we use as our starting seeds are also uniformly distributed in our parameter space. In our case there are two important considerations: do the random matrices sweep out completely random subspaces and what does their distribution of singular values look like. While subspaces are difficult to vizualize in high dimensions it is sufficient to look at what happens in two dimensions. If we create a random vector in $\mathbb{R}^2$ we would hope that it is equally likely to point in any possible direction. Rather than worry about pullback measures and other advanced mathematical concepts we will just run a very large simulation and look at the result:

```
import numpy as np
import matplotlib.pyplot as plt
SIZE = 2**22
x = (np.random.uniform(low= -0.5, high=0.5, size=SIZE))
```

```
y = (np.random.uniform(low= -0.5, high=0.5,size=SIZE))
angles = np.arctan2(y,x);

plt.hist(angles, 360*10)
plt.title("distribution of angles for normal coordinate vectors $2^{22}$ samples")
plt.xlabel('angle rad', fontsize='8')
plt.ylabel('frequency', fontsize='8')
plt.show()
```



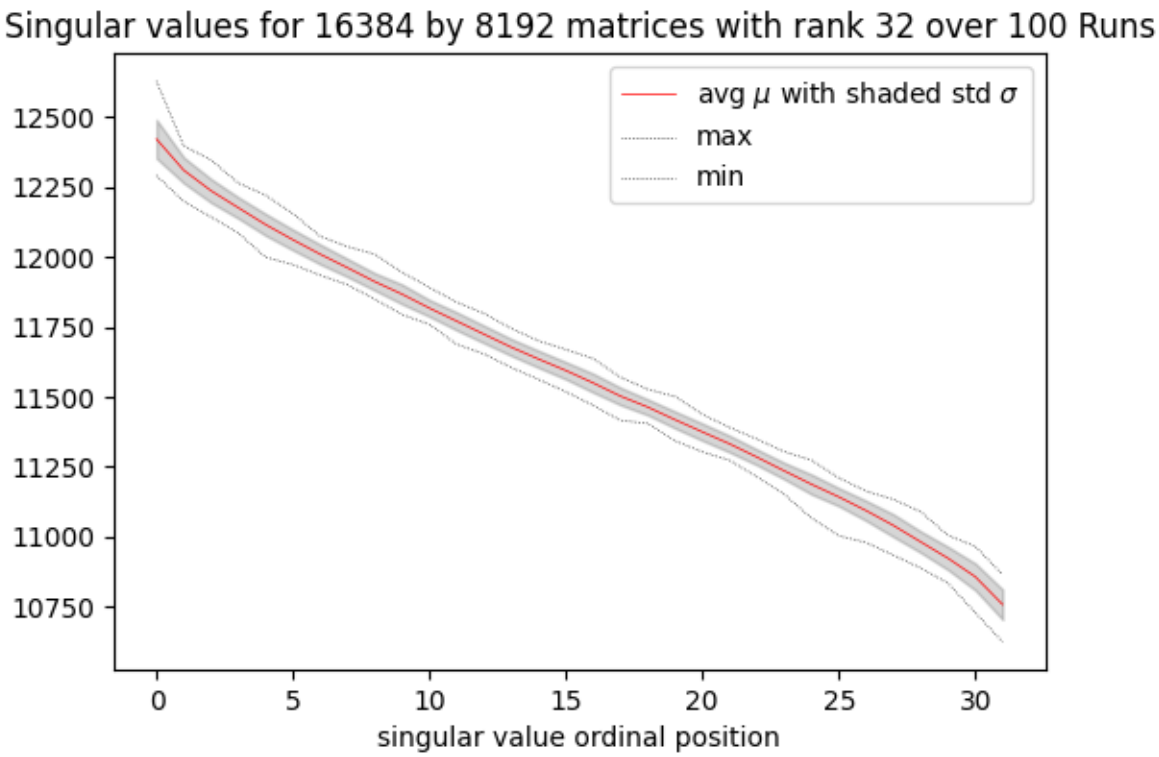
It looks like using a uniform distribution biases our subspaces towards the intercardinal directions. Luckily using a basic normal distribution gets rid of most of the bias.



The second consideration is the resultant distribution of singular values. While we expect the final outcome of our algorithm to have a rapidly decaying set of singular values, using that as a starting point would not be wise. Let us go ahead and visualize the distribution of singular values:

```
import cupy as cp
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable

# number of rows
M = 2**15
# number of columns
N = 2**15
# max rank of matrix, must be less than or equal to n and m
R = 2**5

def computeSingularValuesLR(L: cp.ndarray, R: cp.ndarray) -> cp.ndarray:
    (U, lt) = cp.linalg.qr(L)
    (V, ut) = cp.linalg.qr(R.T)
    return cp.linalg.svd(lt@ut.T, full_matrices=False,compute_uv=False)

def adjustSingularValuesLR(L: cp.ndarray, R: cp.ndarray, func: Callable, args)
        -> (cp.ndarray, cp.ndarray, cp.ndarray):
    # compute SVD of LR pair the fast way
    (U, lt) = cp.linalg.qr(L)
    (V, ut) = cp.linalg.qr(R.T)
    (u,s,v) = cp.linalg.svd(lt@ut.T, full_matrices=True,compute_uv=True)
```

```
    # adjust the singular values according to the provided function
    s = func(s, args)
    # return a new LR pair using the same orthogonal spaces but adjusted singular values
    # note that this operation has low numerical precision
    return ((U @ u @ cp.diag(cp.sqrt(s))),(cp.diag(cp.sqrt(s)) @ v @ V.T), s)

def computeLRMatrixCoincidence(L: cp.ndarray, R: cp.ndarray, L2: cp.ndarray, R2: cp.ndarray):
    left,right,s  = adjustSingularValuesLR(L, R, lambda x, y: x, None)
    left2,right2,s2  = adjustSingularValuesLR(L2, R2, lambda x, y: x, None)
    norm = (cp.prod(s)**0.5 * cp.prod(s2)**0.5)
    return (cp.linalg.det(left.T @ left2) / norm, cp.linalg.det(right @ right2.T) / norm)

def testSingularValueDistribution(MAX_ITERATIONS: int):
    # test breakdown of singular values
    svd_data = cp.zeros((MAX_ITERATIONS, R), dtype=cp.float32)
    for i in range(MAX_ITERATIONS):
        left = (cp.random.normal(size = (M, R), dtype=cp.float32))
        right = (cp.random.normal(size = (R, N), dtype=cp.float32))
        svd_data[i, :] = computeSingularValuesLR(left, right)
    # compute statistics for each singular value
    svd_data = svd_data.get()

    mu = np.mean(svd_data, axis=0)
    sigma = np.std(svd_data, axis=0)
    max = np.max(svd_data, axis = 0)
    min = np.min(svd_data, axis = 0)

    plt.plot(mu, color= (1,0,0), linewidth=0.5, label="avg $\mu$ with shaded std $\sigma$")
    plt.fill_between(range(mu.size), mu + sigma, mu - sigma, color = (0.2, 0.2, 0.2, 0.2))
    plt.plot(max, ':',label ="max", color=(0.1, 0.1, 0.1), linewidth=0.5)
    plt.plot(min, ':',label ="min", color=(0.1, 0.1, 0.1), linewidth=0.5)
    plt.xlabel("singular value ordinal position", fontsize = '10')
    plt.legend(fontsize = '10')
    plt.title("Singular values for " + str(M) + " by " + str(N) + " matrices with rank " + str(R) +
        " over " + str(MAX_ITERATIONS) + " Runs", fontsize='12')
    plt.show()

def testRandomMatrixCoincidence(MAX_ITERATIONS: int):
    coincidence_data_L = np.zeros(MAX_ITERATIONS, dtype=cp.float64)
    coincidence_data_R = np.zeros(MAX_ITERATIONS, dtype=cp.float64)
    for i in range(MAX_ITERATIONS):
        # express one random matrix in LR orthogonal form, do not change the singular values
        left = (cp.random.normal(size = (M, R), dtype=cp.float64))
        right = (cp.random.normal(size = (R, N), dtype=cp.float64))

        coincidence_data_L[i], coincidence_data_R[i] = computeLRMatrixCoincidence(
                cp.random.normal(size = (M, R), dtype=cp.float64),
                     cp.random.normal(size = (R, N), dtype=cp.float64),
                     cp.random.normal(size = (M, R), dtype=cp.float64),
                     cp.random.normal(size = (R, N), dtype=cp.float64))
    print(coincidence_data_L)
    print(coincidence_data_R)
```



We see that despite the indecies of our matrix being independent and normally distributed the singular values are actually very consistent. This is to be expected, since each random vector we add to the matrix is of roughly similar size, what we end up measuring is how close each new vector is to being orthogonal to the rest. The larger the rank of the matrix (relative to its size), the harder it is for later vectors to be orthogonal. In order to compute the singular values for our LR decomposition we took a QR transform which essentially performs the Gram-Schmidt procedure and spits out the result and the lower triangular matrix that generated it. If we do that to both L and R (applying QR to R^T because we really want an RQ transformation) we can then apply the singular value decomposition to the product of the upper and lower triangular matrices (an $r \times r$ matrix). This process however is not very numerically stable and best not performed as part of an itterative algorithm with float32 precision. For preprocessing or post processing though it is more than adequate.

We can think of L as represented by a k dimensional subspace of an m dimensional space and the singular values as weights attached to orthogonal directions. If L_0 is our guess and L_1 is the best possible value, then the efficiency of the algorithm will depend on how well the two already align. In fact with both matrices in orthogonal form (coming from SVD) we can estimate the alignment as the determinant of $L_1 L_0^t$ normalized by the product of square roots of singular values. This quantity for two random matrices becomes very very small and should only be computed with double precision. The code is provided in computeLRMatrixCoincidence but isn't actively used.