
HPC for numerical methods and data analysis

Fall Semester 2024

Prof. Laura Grigori

Assistant: Mariana Martínez Aguilar

Session 10 – November 21, 2023

Deterministic rank revealing factorizations for low rank approximation

Exercise 0 (Optional): Numerical stability of randomized Nystrom

Just as in last week's exercise, consider the MNIST data set and the matrix A as defined in the project or in last week's exercise sheet (with a fixed value of c). **You'll have to be careful with this data set here and in your project because you have to make sure the data set is normalized, i.e. all the entries are between 0 and 1.** Take a relatively small sample of the training set. In this section you can use either last week's code or your code from your project. Consider two different types of sketching matrices, Ω_1, Ω_2 . For different sketching dimensions of both types of matrices, compute the relative error of the low rank approximation in terms of the nuclear norm. Provide a graph that compares these errors. Comment on your findings.

Exercise 1: Tournament pivoting

The truncated SVD provides the best low rank approximation in terms of the Frobenius and L2 norms. The QR decomposition with column pivoting relies on a permutation matrix Π_c and computes the decomposition $A\Pi_c = QR$, where $A \in \mathbb{R}^{m \times m}$, $Q \in \mathbb{R}^{m \times n}$, $R \in \mathbb{R}^{n \times n}$. There are many ways of building such permutational matrix. The strong rank revealing QR, (Strong RRQR) is designed to more effectively identify the numerical rank of a matrix and ensure a robust separation between the significant and insignificant columns in the matrix. For $A \in \mathbb{R}^{m \times n}$ the QR decomposition with column pivoting $A\Pi_c = QR$ can be written as

$$A\Pi_c = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \quad (1)$$

where $Q_1 \in \mathbb{R}^{m \times k}$, $Q_2 \in \mathbb{R}^{m \times (n-k)}$, $R_{11} \in \mathbb{R}^{k \times k}$, $R_{12} \in \mathbb{R}^{k \times (n-k)}$, $R_{22} \in \mathbb{R}^{(n-k) \times (n-k)}$. Let the singular values of A be ordered as

$$\sigma_{\max}(A) = \sigma_1(A) \geq \dots \geq \sigma_{\min}(A) = \sigma_n(A).$$

The factorization in (??) is **strong rank revealing** if for $1 \leq i \leq k$ and $1 \leq j \leq n - k$ we have

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}$$

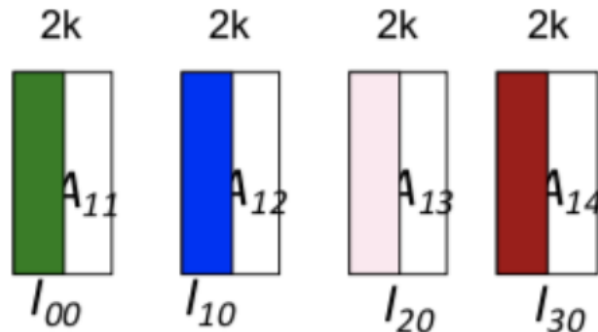
$$\frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \gamma_1(n, k)$$

$$\|R_{11}^{-1}R_{12}\|_{\max} \leq \gamma_2(n, k),$$

where γ_1, γ_2 are low degree polynomials in n and k .

Recall tournament pivoting for deterministic column selection. (Refer to the image below).

- Consider a matrix A partitioned into 4 column blocks. Each processor has one of these blocks.
- Implement strong RRQR (there is a MATLAB implementation, you can use that as a template for building your own Python implementation).
- For each block of columns, select k columns using strong RRQR, save their indices are given in I_{i0} . Be careful with these indices, you might have to deal with "global" and "local" indices.
- In each processor output these indices $I_{00}, I_{10}, I_{20}, I_{30}$.
- Test your method with three different matrices:
 - $A = H_n D H_n^T$, where H_n is the normalized Hadamard matrix of dimension n , D is a diagonal matrix of your choice. Pick n to be "small".
 - Load the normalized MNIST data set and build A as in the project (or last week's exercises). Select a few columns and rows.
 - Build A from the MNIST data set with 2^{11} data points.
- Comment your results with the different matrices. Do you notice a problem when you take more rows and columns using the MNIST data set?
- Using $I_{00}, I_{10}, I_{20}, I_{30}$ build a low rank approximation of A . Check the L2 norm of the error with respect to the error of the truncated SVD.
- Check if the singular values of these selected columns approximate well the singular values of A .
- Check if the diagonal elements of R_{11} approximate well the singular values of A .



A Python implementation for strong RRQR is found below (adapted after the MATLAB implementation found in <https://www.mathworks.com/matlabcentral/fileexchange/69139-strong-rank-revealing-q>

```
# Implementation of the strong rank revealing QR with fixed rank 'k'

import numpy as np
from math import sqrt
from scipy.linalg import solve_triangular
import pdb
from numpy.linalg import norm
from scipy.linalg import qr
import pdb

def givens(a, b):
    """
    Submatrix for a Givens rotation
    """
    if b == 0:
        c = 1;
        s = 0;
    elif abs(b) >= abs(a):
        cotan = a/b;
        s = 1/sqrt(1 + cotan**2)
        c = s*cotan
    else:
        tang = b/a;
        c = 1/sqrt(1 + tang**2)
        s = c*tang
    G = np.array([[c, s], [-s, c]])
    return G

def sRRQR_rank(A, f, k):
    """
    Strong Rank Revealing QR with fixed rank 'k'. The name of the variables
    are the same as in the paper. (Combination of algorithm 4 and 5).
    A is R11, B is R12, C is R22
    gamma is the norm of R22's columns
    omega is the reciprocal of R11^-1's row norm

    Reference:
    Gu, Ming, and Stanley C. Eisenstat. "Efficient algorithms for
    computing a strong rank-revealing QR factorization." SIAM Journal
    on Scientific Computing 17.4 (1996): 848-869.
    Paper found in: https://math.berkeley.edu/~mgu/MA273/Strong-RRQR.pdf
    Translated from matlab implementation
    https://www.mathworks.com/matlabcentral/fileexchange/69139-strong-rank-revealing-qr-decomposition
    """
    # Check constant bound f
    if f < 1:
        print("Parameter f is less than 1. Automatically set f = 2")
        f = 2

    # Dimension of the given matrix
    m, n = A.shape

    # Modify rank k if necessary
    k = min(k, m, n)
```

```

# Pivoting QR first (generally most time-consuming step)
Q, R, p = qr(A, pivoting = True, mode='economic')
# Check special case: rank equals the number of columns.
if k == n:
    # No need for SRRQR.
    return Q, R, p

# Make diagonals of R positive
ss = np.sign(np.diag(R))
ss = np.diag(ss)
R = ss@R
Q = Q@ss

# Initialization of  $A^{-1}B$  (A refers to R11, B refers to R12)
AB = np.linalg.solve(R[:k, :k], R[:k, k:])

# Initialization of gamma, i.e., norm of C's columns (C refers to R22)
gamma = np.zeros(n - k)
if k != R.shape[0]:
    gamma = np.linalg.norm(R[k:, k:], axis=0)

# Initialization of omega, i.e., reciprocal of  $\text{inv}(A)$ 's row norm
tmp = np.linalg.solve(R[:k, :k], np.eye(k, k))
omega = np.reciprocal(np.sqrt(np.sum(tmp**2, axis=1)))
omega = np.reshape(omega, (len(omega), 1))
gamma = np.reshape(gamma, (len(gamma), 1))

# KEY STEP in Strong RRQR:
# "while" loop for interchanging the columns from the first k columns and
# the remaining (n-k) columns.
Rm, _ = R.shape
while True:
    # Identify interchanging columns
    tmp = (1 / omega * np.transpose(gamma))**2 + AB**2
    i, j = np.unravel_index(np.argmax(tmp > f**2), tmp.shape)

    # If no entry of tmp greater than f*f, no interchange needed and the
    # present Q, R, p is a strong RRQR.
    if tmp[i, j] <= f**2:
        #print("NO ENTRY OF TMP GREATER")
        break

    # Interchange the i-th and (k+j)-th column of the target matrix A and
    # update QR decomposition (Q, R, p), AB, gamma, and omega.
    # First step: interchanging the k+1 and k+j-th columns
    if j > 0:
        AB[:, [0, j]] = AB[:, [j, 0]]
        gamma[[0, j]] = gamma[[j, 0]]
        R[:, [k, k+j]] = R[:, [k+j, k]]
        p[[k, k+j]] = p[[k+j, k]]

    # Second step: interchanging the i and k-1-th columns
    if i < k:
        #print("USING SECOND STEP")
        ind = list(range(i+1, k)) + [i]
        p[i:k] = p[ind]
        R[:, i:k] = R[:, ind]
        omega[i:k] = omega[ind]

```

```

AB[i:k, :] = AB[ind, :]
# Givens rotation for the triangulation of R(1:k, 1:k)
for ii in range(i, k-1):
    G = np.array([R[ii, ii] / np.sqrt(R[ii, ii]**2 + R[ii+1, ii]**2),
                  R[ii+1, ii] / np.sqrt(R[ii, ii]**2 + R[ii+1, ii]**2)],
                  [-R[ii+1, ii] / np.sqrt(R[ii, ii]**2 + R[ii+1, ii]**2),
                  R[ii, ii] / np.sqrt(R[ii, ii]**2 + R[ii+1, ii]**2)])
    if G[0, :] @ np.array([R[ii, ii], R[ii+1, ii]]) < 0:
        G = -G # guarantee R(ii,ii) > 0
    R[ii:ii+2, :] = G @ R[ii:ii+2, :]
    Q[:, ii:ii+2] = Q[:, ii:ii+2] @ G.T
if R[k-1, k-1] < 0:
    R[k-1, :] = -R[k-1, :]
    Q[:, k-1] = -Q[:, k-1]

# Third step: zeroing out the below-diagonal of k-th columns
if k < Rm-1:
    #print("USING THIRD STEP")
    for ii in range(k+1, Rm):
        G = np.array([R[k-1, k-1] / np.sqrt(R[k-1, k-1]**2 + R[ii, k-1]**2),
                      R[ii, k-1] / np.sqrt(R[k-1, k-1]**2 + R[ii, k-1]**2)],
                      [-R[ii, k-1] / np.sqrt(R[k-1, k-1]**2 + R[ii, k-1]**2),
                      R[k-1, k-1] / np.sqrt(R[k-1, k-1]**2 + R[ii, k-1]**2)])
        if G[0, :] @ np.array([R[k, k], R[ii, k]]) < 0:
            G = -G # guarantee R(k,k) > 0
        R[[k, ii], :] = G @ R[[k, ii], :]
        Q[:, [k, ii]] = Q[:, [k, ii]] @ G.T

# Fourth step: interchanging the k and k+1-th columns
#print("USING FOURTH STEP")
p[[k-1, k]] = p[[k, k-1]]
ga = R[k-1, k-1]
mu = R[k-1, k] / ga
if k-1 < Rm:
    nu = R[k, k] / ga
else:
    nu = 0
rho = np.sqrt(mu**2 + nu**2)
ga_bar = ga * rho
b1 = R[:k-1, k-1]
b2 = R[:k-1, k]
c1T = R[k, k+1:]
if k+1 > Rm:
    c2T = np.zeros(len(c1T))
else:
    c2T = R[k, k+1:]
c1T_bar = (mu * c1T + nu * c2T) / rho
c2T_bar = (nu * c1T - mu * c2T) / rho
c1T_bar = np.reshape(c1T_bar, (1, -1))
c2T_bar = np.reshape(c2T_bar, (1, -1))

# Modify R
R[:k-1, k-1] = b2
R[:k-1, k] = b1
R[k-1, k-1] = ga_bar
R[k-1, k] = ga * mu / rho
R[k, k] = ga * nu / rho
R[k-1, k+1:] = c1T_bar
R[k, k+1:] = c2T_bar

```

```

# Update AB
u = np.linalg.solve(R[:k-1, :k-1], b1)
u = np.reshape(u, (-1, 1))
u1 = AB[:k-1, 0]
u1 = np.reshape(u1, (-1, 1))
AB[:k-1, 0] = ((nu * nu * u - mu * u1) / rho**2)[: , 0]
AB[k-1, 0] = mu / rho**2
AB[k-1, 1:] = (c1T_bar / ga_bar)[0, :]
AB[:k-1, 1:] = AB[:k-1, 1:] + (nu * u*c2T_bar - u1*c1T_bar) / ga_bar

# Update gamma
gamma[0] = ga * nu / rho
gamma[1:, 0] = np.sqrt( np.power(gamma[1:], 2)[: , 0] + np.power(np.transpose(c2T_bar), 2)[: , 0])

# Update omega
u_bar = u1 + mu * u
omega[k-1] = ga_bar
omega[:k-1] = np.reciprocal(np.sqrt(omega[:k-1]**(-2) + u_bar**2 / ga_bar**2 - u**2 / ga**2))

# Eliminate new R(k+1, k) by orthogonal transformation
Gk = np.array([[mu/rho, nu/rho], [nu/rho, -mu/rho]])
if k < Rm:
    Q[:, [k-1, k]] = Q[:, [k-1, k]] @ Gk.T

# Only return the truncated version of the strong RRQR decomposition
Q = Q[:, :k]
R = R[:k, :]

return Q, R, p

```

Implementation of deterministic column selection: tournament pivoting

```

import numpy as np
from numpy.linalg import norm
import matplotlib.pyplot as plt
from copy import deepcopy
from scipy.linalg import hadamard
from math import ceil, log, exp
import pandas as pd

plt.ion()

# Functions to build the matrix A (see last week's exercises)
def readData(filename, size = 784, save = False):
    '''
    Read MNIST sparse data from filename
    and transforms this into a dense
    matrix, each line representing an entry
    of the database (i.e. a "flattened" image)
    '''
    dataR = pd.read_csv(filename, sep=',', header = None)
    n = len(dataR)
    data = np.zeros((n, size))
    labels = np.zeros((n, 1))

```

```

# Format accordingly
for i in range(n):
    l = dataR.iloc[i, 0]
    labels[i] = int(l[0]) # We know that the first digit is the label
    l = l[2:]
    indices_values = [tuple(map(float, pair.split(':')) for pair in l.split())]
    # Separate indices and values
    indices, values = zip(*indices_values)
    indices = [int(i) for i in indices]
    # Fill in the values at the specified indices
    data[i, indices] = values
if save:
    data.tofile('./denseData.csv', sep = ',', format='%10.f')
    labels.tofile('./labels.csv', sep = ',', format='%10.f')
return data, labels

def buildA_sequential(data, c = 1e3, save = False):
    '''
    Function to build A out of a data base
    using the RBF exp( -||x_i - x_j||/c)
    Notice that we only need to fill in the
    upper triangle part of A since it's symmetric
    and its diagonal elements are all 1.
    '''
    n = data.shape[0]
    A = np.zeros((n, n))
    for j in range(n):
        for i in range(j):
            A[i, j] = exp( -norm( data[i, :] - data[j, :])*2/c)
    A = A + np.transpose(A)
    np.fill_diagonal(A, 1.0)
    if save:
        A.tofile('./A.csv', sep=',', format='%10.f')
    return A

filename = "mnist_780"
data, labels = readData(filename)
A = buildA_sequential(data)
A = A[0:16, 0:32]

Q, R, P = sRRQR_rank(A, 1.05, 5)

```

The tournament pivoting for deterministic column selection can be found below:

```

# Implementation of deterministic column selection: tournament pivoting

from mpi4py import MPI
import numpy as np
from numpy.linalg import norm, svd
import matplotlib.pyplot as plt
from copy import deepcopy
from scipy.linalg import hadamard
from math import sqrt, exp, ceil, log
import pandas as pd

```

```

# Custom library
from sRRQR import sRRQR.rank

plt.ion()

# Functions to build the matrix A (see last week's exercises)
def readData(filename, size = 784, save = False):
    """
    Read MNIST sparse data from filename
    and transforms this into a dense
    matrix, each line representing an entry
    of the database (i.e. a "flattened" image)
    """
    dataR = pd.read_csv(filename, sep=',', header = None)
    n = len(dataR)
    data = np.zeros((n, size))
    labels = np.zeros((n, 1))
    # Format accordingly
    for i in range(n):
        l = dataR.iloc[i, 0]
        labels[i] = int(l[0]) # We know that the first digit is the label
        l = l[2:]
        indices_values = [tuple(map(float, pair.split(':')) for pair in l.split())]
        # Separate indices and values
        indices, values = zip(*indices_values)
        indices = [int(i) for i in indices]
        # Fill in the values at the specified indices
        data[i, indices] = values
    if save:
        data.tofile('./denseData.csv', sep = ',', format='%10.0f')
        labels.tofile('./labels.csv', sep = ',', format='%10.0f')
    return data, labels

def buildA_sequential(data, c = 1e3, save = False):
    """
    Function to build A out of a data base
    using the RBF  $\exp(-||x_i - x_j||/c)$ 
    Notice that we only need to fill in the
    upper triangle part of A since it's symmetric
    and its diagonal elements are all 1.
    """
    n = data.shape[0]
    A = np.zeros((n, n))
    for j in range(n):
        for i in range(j):
            A[i, j] = exp(-norm(data[i, :] - data[j, :])*2/c)
    A = A + np.transpose(A)
    np.fill_diagonal(A, 1.0)
    if save:
        A.tofile('./A.csv', sep=',', format='%10.0f')
    return A

def matTranspose(A, n):
    """
    Since we want to distribute A's columns
    we need to manipulate our data into the correct
    order before sending it.
    """
    arrs = np.split(A, n, axis = 1)

```



```

    raveled = [np.ravel(arr) for arr in arrs]
    A.transpose = np.concatenate(raveled)
    return A.transpose

# Initialize MPI (world)

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

A = None
m = None
n = None
local_sizeCols = None
A_T = None
f = 1.05
k = 4

if rank == 0:
    # Read the files and build the matrix A here (see last week's exercises)
    filename = "mnist.780"
    data, labels = readData(filename)
    A = buildA_sequential(data)
    #A = A[0:16, 0:16]
    # D = np.diag([1, 0.9, 0.8, 0.7, 0.65, 0.4, 0.1, 0.001])
    # H = hadamard(8)
    # H = 1/norm(H[:, 0])*H
    # A = H@D@np.transpose(H)
    # A = np.arange(0, 108)
    # A = np.reshape(A, (9, 12)) + 0.0
    m = A.shape[0]
    n = A.shape[1]
    print("m: ", m, "n: ", n)
    local_sizeCols = n//size # number of columns
    A_T = matTranspose(A, n)

# Step one: partition A into 4 column blocks
local_sizeCols = comm.bcast(local_sizeCols, root=0)
m = comm.bcast(m, root=0)
A_local = np.empty((local_sizeCols, m))
comm.Scatter(A_T, A_local, root = 0)
# Step 0
# From each column block Ali, i = 1, . . . , 4, k columns are selected by using
# strong RRQR, and their indices are given in Ii0.
Q, R, P = sRRQR_rank(np.transpose(A_local), f, k)
P = P[0:k]
Pall = deepcopy(P) + rank*local_sizeCols
#Start iterating
for i in range(1, ceil(log(size)) + 1):
    #print("i: ", i)
    if rank%(2**i) == 0 and rank + 2**(i-1)<size:
        j = rank + 2**(i-1)
        Phere = np.copy(P)
        Psave = deepcopy(Pall)
        Ahere = np.copy(A_local)
        Ahere = Ahere[Phere, :]
        # We receive the first k indices from Pj from processor j
        #print("Receiving and factorizing at: ", rank)
        comm.Recv(P, source = j, tag = 77)

```

```

        comm.Recv(A_local, source = j, tag = 88)
        comm.Recv(Pall, source = j, tag = 99)
        A_local = A_local[P, :]
        A_local = np.concatenate((A_here, A_local))
        Pall = np.concatenate((P_save, Pall))
        # strong RRQR
        Q, R, P = sRRQR.rank(np.transpose(A_local), f, k)
        P = P[0:k]
        Pall = Pall[P]
    elif rank%(2**i) == 2**(i-1):
        # Send the local P
        #print("Sending from: ", rank)
        comm.Send(P, dest = rank - 2**(i-1), tag = 77)
        comm.Send(A_local, dest = rank - 2**(i-1), tag = 88)
        comm.Send(Pall, dest = rank - 2**(i-1), tag = 99)

if rank == 0:
    # Print the selected columns I02
    # Make the low rank approximation
    Pall = Pall.flatten()
    print("Selected columns: ", Pall)
    Atilde = A[:, Pall]
    U, Sigma, V = svd(Atilde)
    U = U[:, 0:k]
    Sigma = Sigma[0:k]
    V = V[0:k, 0:k]
    appsRRQR = U@np.diag(Sigma)@V
    # Singular values of full A
    Uf, Sigmaf, Vf = svd(A)
    Uf = Uf[:, 0:k]
    Sigmaf = Sigma[0:k]
    Vf = Vf[0:k, 0:k]
    appfull = Uf@np.diag(Sigmaf)@Vf
    # Compute the theoretical bound
    gamma = sqrt( 1 + f**2*k*(n-k) )
    print("L2 error with truncated SVD: ",
          norm( appsRRQR - appfull ) )
    print(r'$\gamma$ (n, k) = $')
    print(gamma)
    print(r'$\sigma_i(A)/\sigma_i(R_{\{11\}}) = $')
    print(np.divide(Sigma, Sigmaf))

```