CPSC 302 - Assignment 4

Tristan Rice, q7w9a, 25886145

1. Data Fitting

1.a

The equation $u(t)$ isn't linear, thus you can't find a perfect match using linear least squares.

1.b

We're trying to solve for $v(t) = x_1 + x_2 t$.

Normal equations:

$$X^T X b = X^T y$$

$$X = \begin{bmatrix} 1 & 0.0 \\ 1 & 1.0 \\ 1 & 2.0 \end{bmatrix}$$

$$y = \begin{bmatrix} 0.1 \\ 0.9 \\ 2 \end{bmatrix}$$

Solving the normal equations gives us

$$b = \begin{bmatrix} 0.05 \\ 0.95 \end{bmatrix}$$

$$v(t) = 0.05 + 0.95t$$

$$u(t) = e^{v(t)} = e^{0.05+0.95t}$$
$$u(t) = e^{v(t)} = e^{0.05} e^{0.95t}$$
$$u(t) = 1.0512711 e^{0.95t}$$

2. Classical Gram-Schmidt vs. Modified Gram-Schmidt

2.a

$$A = \begin{bmatrix} 1 & 1 & 1 \\ \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{bmatrix}$$

Since we have a 4x3 matrix, we can decompose it into a 4x3 matrix and a 3x3 matrix.

$$A = \begin{bmatrix} \frac{1}{\sqrt{1+\epsilon^2}} & 0 & 1 \\ \frac{\epsilon}{\sqrt{1+\epsilon^2}} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \epsilon \end{bmatrix} \begin{bmatrix} \sqrt{1+\epsilon^2} & \epsilon & \epsilon \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

TODO: finish up

2.b

TODO

2.c

TODO

3. Comparison of Algorithms

3.a

$$A^T A x = A^T b$$

$$A = \begin{bmatrix} 1 & 1 & 1 \\ \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{bmatrix}$$

$$b = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^T b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$A^T A = \begin{bmatrix} 1+\epsilon^2 & 1 & 1 \\ 1 & 1+\epsilon^2 & 1 \\ 1 & 1 & 1+\epsilon^2 \end{bmatrix}$$

For ease of writing, $a = x_1, b = x_2, c = x_3$:

$$(1+\epsilon^2)a + b + c = 1$$
$$a + (1+\epsilon^2)b + c = 1$$
$$a + b + (1+\epsilon^2)c = 1$$
$$a = 1 - b - (1+\epsilon^2)c$$

$$(1+\epsilon^2)(1 - b - (1+\epsilon^2)c) + b + c = 1$$
$$1 - b - (1+\epsilon^2)c + (1+\epsilon^2)b + c = 1$$
$$-b - (1+\epsilon^2)c + (1+\epsilon^2)b + c = 0$$
$$\epsilon^2 b + \epsilon^2 c = 0$$
$$b = -c$$
$$(1+\epsilon^2)(1 + c - (1-\epsilon^2)c) = 1$$
$$1 + c - (1-\epsilon^2)c = \frac{1}{1+\epsilon^2}$$
$$c\epsilon^2 = \frac{1}{1+\epsilon^2} - 1$$
$$c = \frac{1}{\epsilon^2 + \epsilon^4} - \frac{1}{\epsilon^2}$$
$$a = 1 + c - (1-\epsilon^2)c$$

$$a = 1 + \epsilon^2 c$$
$$a = \frac{1}{1+\epsilon^2}$$

Thus,

$$x_1 = \frac{1}{1+\epsilon^2}$$
$$x_2 = -\frac{1}{\epsilon^2+\epsilon^4} + \frac{1}{\epsilon^2}$$
$$x_3 = \frac{1}{\epsilon^2+\epsilon^4} - \frac{1}{\epsilon^2}$$

TODO: check this

3.b

4. Regularization

4.a

$$A = U\Sigma V^T$$
$$A^TA + \gamma I = (U\Sigma V^T)^T U\Sigma V^T + \gamma I$$
$$A^TA + \gamma I = (U\Sigma V^T)^T U\Sigma V^T + \gamma I$$

Since U is an orthoganal matrix, $U^TU = UU^T = I$.

$$A^TA + \gamma I = V\Sigma^T\Sigma V^T + \gamma I$$

Since V is likewise an orthoganal matrix, we can represent $\gamma I = V\gamma V^T$.

$$A^TA + \gamma I = V\Sigma^T\Sigma V^T + V\gamma V^T$$
$$A^TA + \gamma I = V(\Sigma^T\Sigma + \gamma I)V^T$$

Since this is a $m \times n$ matrix of rank n,

$$\kappa_2 = \frac{\sigma_1}{\sigma_n}$$

Thus, the condition number of A is just the largest singular value of A divided by the smallest singular value.

$$\kappa_2^2(A) = (\frac{\sigma_1}{\sigma_n})^2$$
$$\kappa_2(A^TA + \gamma I) = \frac{\sigma_1^2 + \gamma}{\sigma_n^2 + \gamma}$$

Thus,

$$\kappa_2^2(A) \geq \kappa_2(A^TA + \gamma I)$$
$$\frac{\sigma_1^2}{\sigma_n^2} \geq \frac{\sigma_1^2 + \gamma}{\sigma_n^2 + \gamma}$$

Thus $+\gamma$ term makes the condition number closer to 1, and thus, we see that

$$\kappa_2^2(A) \geq \kappa_2(A^TA + \gamma I)$$

4.b

Normal equations

$$A^T Ax = A^T b$$

becomes

$$\min_x ||b - Ax||_2$$

We want to convert $(A^T A + \gamma I)x_\gamma = A^T b$ into something similar.

$$(A^T A + \gamma I)x_\gamma = A^T b$$
$$(A + \gamma (A^T)^{-1})x_\gamma = b$$

Now we have something in the form $Cx = d$ where our new $C = A + \gamma (A^T)^{-1}$.

Thus,

$$\min_{x_\gamma} ||b - (A + \gamma (A^T)^{-1})x_\gamma||_2$$

4.c

We can use the normal equations and the regularized normal equations to show that $||x_\gamma||_2 \leq ||x||_2$.

$$A^T Ax = A^T b$$
$$(A^T A + \gamma I)x_\gamma = A^T b$$
$$(A^T A + \gamma I)x_\gamma = A^T Ax$$
$$(I + \gamma (A^T A)^{-1})x_\gamma = x$$

We know that $A^T A$ is positive definite, so it's inverse must also be positive definite. Adding another positive definite matrix (the identity matrix in this case) results in it being a positive definite matrix.

Thus, $||I + \gamma (A^T A)^{-1}||_2 \geq 1$ and $I + \gamma (A^T A)^{-1}$ is positive definite.

Since positive definite is an extension of a scalar to matrixes, we know that since the coefficient A is positive definite that given $Ab = c, b \leq c$. Thus, $x_\gamma \leq x$.

4.d

Running the regularized least squares it for the specified values gives us:

```
j = Inf:
  - gamma = 0
    <missing>
j = 0:
  - gamma = 1
  - err = 5.0286
j = 3:
  - gamma = 0.001
  - err = 5.025
j = 6:
  - gamma = 1e-06
  - err = 5.025
j = 12:
  - gamma = 1e-12
  - err = 5.025
TSVD:
  - err = 5.025
```

We can't directly compute $\gamma = 0$ using Matlab since A is very close to singular. However, some small non-zero value of γ for regularization gives decent results. If it's too large it increases the error.

Using TSVD gives a very similar result to using regularization. Both TSVD and regularization seem to give decent results for close to singular matrices that you wouldn't be otherwise able to solve.

5. Compressing Image Information

5.a

TSVD Mandrill

TSVD Durer

5.b

Mandrill requires $480 * 500 = 240000$ locations. The compressed version requires $r * (480 + 500 + 1)$ storage locations.

Durer requires $648 * 509 = 329832$ storage locations. The compressed version requires $r * (648 + 509 + 1)$ storage locations.

It seems to be more effective for Mandrill. If we look at how much data is lost for a given r value we see that for mandrill it's 26.1% ($64 * (480 + 500 + 1)/(480 * 500) = 0.2616$) of the original storage size. For durer it's 22.5% ($64 * (648 + 509 + 1)/(648 * 509) = 0.2247$) of the original size. This is due to the fact that mandrill is lower rank to start with so it doesn't have to lose as much data to get to rank 64. Mandrill appears better at a given r value since it's closer to the original image and it has fewer small details.

```
b = [1; 0; 0; 0];

for epsilon = [2*sqrt(eps) sqrt(eps) 2*eps, eps]
  disp("Epsilon = " + eps + ":")
  A = genA(epsilon);
  xreal = [
    1/(1+epsilon^2);
    -1/(epsilon^2+epsilon^4) + 1/(epsilon^2);
    1/(epsilon^2+epsilon^4) - 1/(epsilon^2)
  ];

  x = solveNormal(A, b);
  print_error("Normal equations", x, xreal, A, b)

  x = solveClassicalGS(A, b);
  print_error("ClassicalGS", x, xreal, A, b)

  x = solveModifiedGS(A, b);
  print_error("ModifiedGS", x, xreal, A, b)

  x = solveHouseholder(A, b);
  print_error("Householder", x, xreal, A, b)

  x = solveGivens(A, b);
  print_error("Givens", x, xreal, A, b)

  x = solveQR(A, b);
  print_error("QR", x, xreal, A, b)

  x = solveBackslash(A, b);
  print_error("Backslash", x, xreal, A, b)

  x = solveSVD(A, b);
  print_error("SVD", x, xreal, A, b)

  x = solveTSVD(A, b);
  print_error("TSVD", x, xreal, A, b)
end

function print_error(name, x, xreal, A, b)
  err = norm(x - xreal, 2);
  res = norm(b - A * x, 2);
  disp("  - " + name + ": err = " + err + ", residual = " + res);
end

% genA generates the matrix A with the given value of epsilon.
function A = genA(epsilon)
  A = [1 1 1; epsilon 0 0; 0 epsilon 0; 0 0 epsilon];
end

% solveNormal solves least squares via the normal equations.
function x = solveNormal(A, b)
  % Form B = A^TA and y = A^Tb
  B = A' * A;
  y = A' * b;
  % solve the system Bx = y
  x = B\y;
end

% solveClassicalGS solves least squares via classical Gram-Schmidt
% orthogonalization.
function x = solveClassicalGS(A, b)
  % compute size of matrix
  [rows, cols] = size(A);
  % initialize Q, R
  Q = zeros(rows, cols);
  R = zeros(cols, cols);
  % iterate though each column
  for j=1:cols
    % initialize Q column j from A
    Q(:, j) = A(:, j);
    for i=1:(j-1)
      % compute the dot product between A j and Q i
```

```
      R(i, j) = A(:, j)' * Q(:, i);
      Q(:, j) = Q(:, j) - R(i, j) * Q(:, i);
    end
    % compute R jj value
    R(j, j) = norm(Q(:, j));
    % normalize column j
    Q(:, j) = Q(:, j)./R(j, j);
  end
  % use the QR decomposition to solve
  x = R\(Q' * b);
end

% solveModifiedGS solves least squares via modified Gram-Schmidt
% orthogonalization.
function x = solveModifiedGS(A, b)
  % compute size of matrix
  [rows, cols] = size(A);
  % initialize Q, R
  Q = zeros(rows, cols);
  R = zeros(cols, cols);
  % iterate though each column
  for j=1:cols
    % initialize Q column j from A
    Q(:, j) = A(:, j);
    for i=1:(j-1)
      % compute the dot product between Q j and Q i
      R(i, j) = Q(:, j)' * Q(:, i);
      Q(:, j) = Q(:, j) - R(i, j) * Q(:, i);
    end
    % compute R jj value
    R(j, j) = norm(Q(:, j));
    % normalize column j
    Q(:, j) = Q(:, j)./R(j, j);
  end
  % use the QR decomposition to solve
  x = R\(Q' * b);
end

% solveHouseholder solves least squares via Householder Transformations QR.
% This is based off of the sample code in the textbook.
function x = solveHouseholder(A, b)
  [m,n] = size(A);
  p = zeros(1,n);
  for k = 1:n
    % define u of length = m-k+1
    z = A(k:m, k);
    e1 = [1; zeros(m-k,1)];
    u = z+sign(z(1)) * norm(z) * e1;
    u = u/norm(u);
    % update nonzero part of A by I-2uu^T
    A(k:m,k:n) = A(k:m,k:n)-2 * u*(u'*A(k:m,k:n));
    % store u
    p(k) = u(1);
    A(k+1:m, k) = u(2:m-k+1);
  end
  y = b(:);
  % transform b
  for k=1:n
    u = [p(k);
    A(k+1:m,k)];
    y(k:m) = y(k:m) - 2 * u * (u' * y(k:m));
  end
  % form upper triangular R and solve
  R = triu(A(1:n,:));
  x = R \ y(1:n);
end

% solveGivens solves least squares via Givens Rotation QR factorization.
function x = solveGivens(A, b)
  [rows, cols] = size(A);
  Q = eye(rows);
  % eliminate column by column
  for i=1:cols
```

```
    % then eliminate row by row
    for j=(i+1):rows
      % compute s and c values so we can make the rotation matrix
      alpha = sqrt(A(i,i)^2 + A(j, i));
      s = A(j,i)/alpha;
      c = A(i,i)/alpha;
      % create the rotation matrix
      G = eye(rows);
      G(i, i) = c;
      G(j, i) = -s;
      G(i, j) = s;
      G(j, j) = c;
      % rotate A and Q with the rotation matrix
      A = G * A;
      Q = G * Q;
    end
  end
  % transpose Q to get the actual Q matrix
  Q = Q';
  R = A;
  x = R\(Q' * b);
end

% solveQR solves least squares by using Matlab's qr function.
function x = solveQR(A, b)
  % compute the QR factorization
  [Q, R] = qr(A);
  % compute c = Q^Tb
  c = Q' * b;
  % solve the system Rx = c
  x = R\c;
end

% solveBackslash solves least squares by using Matlab's backslash operator.
function x = solveBackslash(A, b)
  % solve the system Ax = b
  x = A\b;
end

% solveSVD solves least squares via SVD.
function x = solveSVD(A, b)
  % compute SVD of A
  [U, S, V] = svd(A);
  % compute z = U^Tb
  z = U' * b;
  % compute pseudo inverse of S and apply it to z
  [rows, cols] = size(S);
  y = z(1:cols);
  diagS = diag(S);
  % divide y by non-zero elements of S
  y(diagS~=0) = y(diagS~=0)./diagS(diagS~=0);
  % Compute x = Vy
  x = V * y;
end

% solveTSVD solves least squares via TSVD.
function x = solveTSVD(A, b)
  % compute SVD of A
  [U, S, V] = svd(A);
  % truncate small values of S
  S(S<10^-6) = 0;
  % compute z = U^Tb
  z = U' * b;
  % compute pseudo inverse of S and apply it to z
  [rows, cols] = size(S);
  y = z(1:cols);
  diagS = diag(S);
  % divide y by non-zero elements of S
  y(diagS~=0) = y(diagS~=0)./diagS(diagS~=0);
  % Compute x = Vy
  x = V * y;
end
```

```
A = [1 0 1 2; 2 3 5 10; 5 3 -2 6; 3 5 4 12; -1 6 3 8];
b = [4; -2; 5;-2; 1];

% problems asks for a bunch of different regularization values. inf is
% equivalent to no regularization since 10^-inf = 0.
for j = [inf 0 3 6 12]
  % gamma is the amount of regularization to use while computing least squares
  gamma = 10^-j;
  % compute x_gamma using the regularized least squares normal equations
  [rows, cols] = size(A);
  x = (A'*A + gamma * eye(cols))\(A' * b);
  % compute the residual
  r = b - A*x;
  % compute the norm of the residual
  err = norm(r, 2);
  % display the results
  disp("j = " + j + ":")
  disp("  - gamma = " + gamma)
  disp("  - err = " + err)
end


% compute TSVD for this problem
% first compute the SVD
[U, S, V] = svd(A);
% truncate lower singular values
S(S < 10^-10) = 0;
% compute A from the truncated SVD
Atrunc = U * S * V';
% solve the least squares using the TSVD formed A matrix
x = (Atrunc' * Atrunc)\(Atrunc' * b);
% compute the residual
r = b - A*x;
% compute the norm of the residual
err = norm(r, 2);

disp("TSVD:")
disp("  - err = " + err)
```

```
% generate plots for mandrill
load mandrill
plot_all_tsvd("mandrill", X)

% generate plots for durer
load durer
plot_all_tsvd("durer", X)

% plot_all_tsvd generates the rank 2-64 plots and saves them to a file.
function plot_all_tsvd(name, X)
  f = figure()
  for i = 1:6
    plot_tsvd(X, 2^i, i)
  end
  saveas(f, "q5-" + name + ".png")
end

% plot_tsvd takes the TSVD of the matrix X resulting in a SVD of rank r and
% displays it in a 3x2 grid in the specified slot.
function plot_tsvd(X, r, slot)
  % Compute the SVD of the matrix X.
  [U,S,V] = svd(X);

  % Figure out how big the decomposition is.
  [rows, cols] = size(S);
  % Truncate the singular values so it has rank R
  S((r+1):rows, :) = 0;

  % Compute the compressed version from the TSVD.
  compressed_X = U * S * V';

  % Check that the truncation is correct.
  assert(rank(S) == r)
  assert(rank(compressed_X) == r)

  % Display the TSVD of X.
  colormap(gray);
  subplot(3, 2, slot);
  image(compressed_X);
  title("TSVD rank = " + r);
end
```