MA 580 Assignment 3

Kyle Hansen

17 October 2025

AI Use Statement: No AI used for this assignment

Question 1 Rank 1 matrices

1(a) If $v \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$, then the product $A = uv^T$ can be expressed visually as:

$$A = uv^T = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix} \begin{pmatrix} v_1 v_2 \cdots v_n \end{pmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & \\ u_2v_1 & u_2v_2 & & \\ \vdots & & \ddots & \\ & & & u_mv_n \end{bmatrix} \tag{1}$$

clearly each column is the vector u, scaled by constant $v_1, v_2, \ldots, v_n$ and each row is a similarly scaled vector v^T. Since all rows and all columns are linarly dependent on each other (there are no two linearly independent rows or columns), the matrix has rank 1.

The column and row vectors of any matrix rank 1 $A \in \mathbb{R}^{m \times n}$ must be scalar multiples of each other (linear dependence, by the definiiton of rank 1), so it may be written, using appropriate definitions of u_i and v_i, as

$$A = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & \\ u_2v_1 & u_2v_2 & & \\ \vdots & & \ddots & \\ & & & u_mv_n \end{bmatrix}. \tag{2}$$

This defines such a matrix where all rows and all columns are scalar mulitples of each other, and can be alternatively be rewritten $A = uv^T$ where $v \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$ and both vectors are nonzero. Thus if the matrix A has rank 1, it can be written as $A = uv^T$. From Equation 1, the reverse is also true– if a matrix $A \in \mathbb{R}^{m \times n}$ is defined as $A = uv^T$ for any nonzero $v \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$, then A has rank 1.

Then the original problem statement has been proven: a matrix $A \in \mathbb{R}^{m \times n}$ is rank 1 if and only if $A = uv^T$ for nonzero $v \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$.

1(b) From the previous part, a rank 1 matrix A can be written using nonzero $v \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$.

$\hat{U}$ and $\hat{V}$ are defined using u and v from the previous part, where $\hat{U}$ is the vector u, normalized and repeated m times, and $\hat{V}$ is the vector v, normalized and repeated n:

$$\begin{aligned} \hat{U} &= \begin{bmatrix} u/\|u\| & u/\|u\| & \cdots & u/\|u\| \end{bmatrix}^{m \times m} \\ \hat{V} &= \begin{bmatrix} v/\|v\| & v/\|v\| & \cdots & v/\|v\| \end{bmatrix}^{n \times n} \end{aligned} \tag{3}$$

This satisfies the dimension of $\hat{U}$ and $\hat{V}$, as well as the condition that they have unitary columns.

Since a rank 1 matrix only has one non-zero singular value σ, the singular value matrix $\hat{\Sigma}$ is composed of all zero elements, except $\Sigma_{1,1} = \sigma$, where in this case

$$\sigma = \|u\| \cdot \|v\| . \tag{4}$$

Alternatively, all three matrices can be further reduced, defining the 1×1 matrix $\hat{\Sigma} = \sigma$, and $\hat{U}$ and $\hat{V}$ can be reduced to $m \times 1$ and $n \times 1$ matrices, respectively. Then this condensed SVD is:

$$A = \frac{u}{\|u\|} \|u\| \|v\| \frac{v}{\|v\|}. \tag{5}$$

1(c) In this matrix, $u = \begin{bmatrix} 3 & 4 \end{bmatrix}^T$ and $v = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^T$. Then, since $\|u\|_2 = 5$ and $\|v\|_2 = \sqrt{14}$, the condensed SVD (using $\hat{U} \in \mathbb{R}^{m \times 1}$ and $\hat{V} \in \mathbb{R}^{n \times 1}$ as before) is:

$$A = (5\sqrt{14}) \begin{bmatrix} \frac{3}{5} \\ \frac{4}{5} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{14}} & \frac{2}{\sqrt{14}} & \frac{3}{\sqrt{14}} \end{bmatrix} \tag{6}$$

and $\|A\|_2 = 5\sqrt{14}$.

Question 2 SVD

Using some definitions

$$\begin{aligned} A &= U\Sigma V^T \\ A^T &= V\Sigma U^T \\ A^\dagger &= V\Sigma^\dagger U^T \end{aligned} \tag{7}$$

And letting the product $\Sigma^\dagger\Sigma = \tilde{I}$ be defined by:

$$\Sigma^\dagger\Sigma = \begin{bmatrix} \hat{\Sigma}^{-1} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{\Sigma} & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} I^{r\times r} & 0 \\ 0 & 0 \end{bmatrix} = \tilde{I} \tag{8}$$

where $\tilde{I}$ is the $m \times m$ matrix where the first r diagonal entries are ones, and the rest are zeros– the result of this operation is retaining only the first r rows when multipling on the right, and only the first r columns when multiplying on the right. Since both Σ and $\Sigma^\dagger$ contain only r nonzero rows and columns, $\tilde{I}\Sigma = \Sigma$, and $\tilde{I}\Sigma^\dagger = \Sigma^\dagger$

Then the proof can be shown with some algebraic manipulation:

$$\begin{aligned} A^TA &= V\Sigma\cancelto{I}{U^TU}\Sigma V^T \\ (A^TA) &= V\Sigma\Sigma V^T \\ (A^TA)^{-1} &= V\Sigma^\dagger\Sigma^\dagger V^T \\ (A^TA)^{-1}A^T &= V\Sigma^\dagger\Sigma^\dagger\cancelto{I}{V^TV}\Sigma U^T \\ (A^TA)^{-1}A^T &= V\Sigma^\dagger\cancelto{\tilde{I}}{\Sigma^\dagger\Sigma}U^T \\ (A^TA)^{-1}A^T &= V\Sigma^\dagger U^T \\ (A^TA)^{-1}A^T &= A^\dagger. \end{aligned} \tag{9}$$

This is equivalent to directly solving the normal equations for overdetermined least squares:

$$\begin{aligned} Ax &= b \\ A^TAx &= A^Tb \\ x &= (A^TA)^{-1}A^Tb \\ x &= A^\dagger b. \end{aligned} \tag{10}$$

2(a) Similar to before,

Table 1: Results from Least-Squares Polynomial Fits

n	ε_{rel}(Direct)	ε_{rel}(QR)	$\kappa_2(A^TA)$
1	0.9926	0.9926	1.000E+00
2	0.6996	0.6996	1.567E+01
3	0.2963	0.2963	3.891E+02
4	0.14397	0.14397	1.245E+04
5	0.12144	0.12144	4.126E+05
6	0.12051	0.12051	1.725E+07
7	0.082461	0.082461	6.264E+08
8	0.071419	0.071419	1.646E+10
9	0.070484	0.070484	7.477E+11
10	0.063618	0.063618	2.268E+13
11	0.06326	0.06326	6.717E+14
12	0.063257	0.063257	4.893E+16
13	0.064037	0.052675	3.882E+17
14	0.065135	0.012167	1.496E+18

$$\begin{aligned} AA^T &= U\Sigma\cancel{V^TV}\Sigma U^T \\ (AA^T)^{-1} &= U\Sigma^\dagger\Sigma^\dagger U^T \\ A^T(AA^T)^{-1} &= V\Sigma\cancel{U^TU}\Sigma^\dagger\Sigma^\dagger U^T \\ A^T(AA^T)^{-1} &= V\cancel{\Sigma\Sigma^{-1}}\Sigma^{-1}U^T \\ A^T(AA^T)^{-1} &= V\Sigma^\dagger U^T \\ A^T(AA^T)^{-1} &= A^\dagger \end{aligned} \tag{11}$$

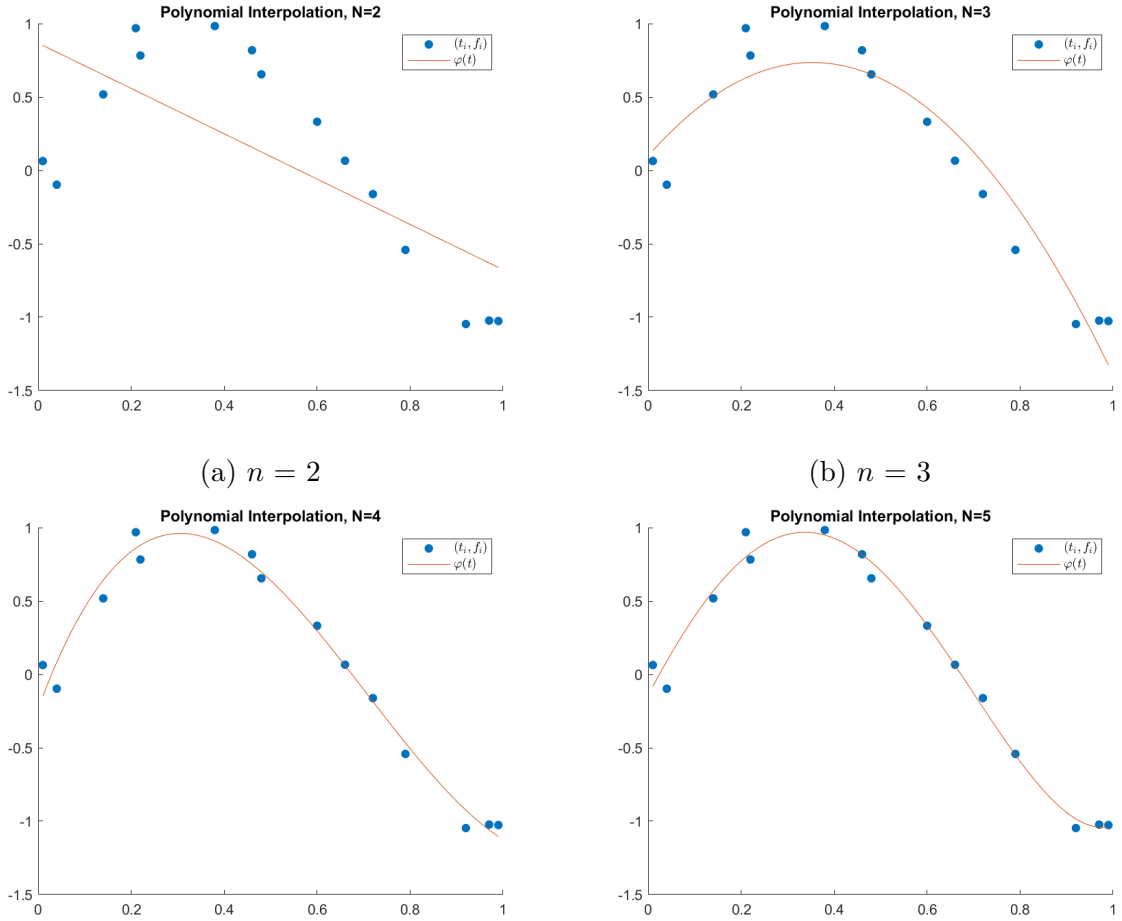
2(b) The third part can be shown similarly to the previous two:

$$\begin{aligned} A^\dagger A &= V\Sigma^\dagger\cancel{U^TU}\Sigma V^T \\ A^\dagger A &= V\cancel{\Sigma^\dagger\Sigma}V^T \\ A^\dagger A &= VV^T \\ A^\dagger A &= I \\ A^\dagger &= A^{-1}. \end{aligned} \tag{12}$$

Question 3 Least Squares using QR Factorization

My MATLAB code (and a back-substitution algorithm from K. Ming Leung, 2003) is listed in Appendix A. The relative errors from 14 solutions are provided in Table 1.

For increasing values of n, the matrix A^TA becomes increasingly ill-conditioned, which can be seen when comparing results of the direct solve to QR– the relative error at $n = 14$ is 0.065 for the direct solve, but only 0.012 for the QR solve. After about $n = 10$, the condition number becomes so large that increasing the size of the matrix returns no additional precision for the direct solve, since at this point, noise from roundoff error is enough to greately perturb the solution. MATLAB's backslash operator likely has reduced the rank of the matrix for these solutions, which is why they change little with different values of n. The difference between the QR solution, which makes no approximation, and the backslash solution, which accounts for this instability, can be seen in Figure 1.



(a) $n = 2$ (b) $n = 3$

(c) $n = 4$ (d) $n = 5$

Figure 1: Data and Least-Squares interpolation using QR method

3(a) For this example, using $n = 4$ is a good choice for interpolating the data, since it retains enough information about the trend of the data without being overfit, though different sources of data may justify different degrees of interpolation (for example, if the

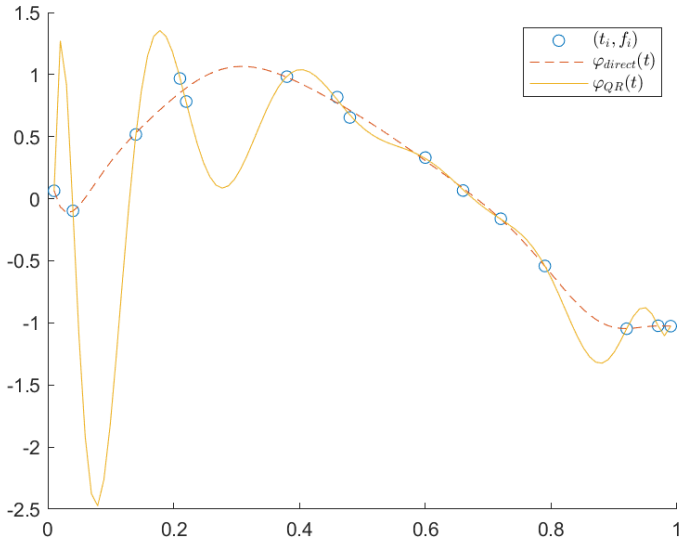


Figure 2: Comparison of direct and QR solve

data were taken from an experiment that models a physical phenomenon that is expected to have t^2 behavior, there is no physical justification for any more precision than that in the least squares fit).

Question 4 Cholesky Heat Equation

The provided code uses matlab backslash to solve the system $Gu^{n+1} = u^n$, specifically using the code snippet:

Direct solve (runs in 95.21 seconds):

```
   .
   .
   .
U = zeros((nx-1)^2,nt+1);
U(:,1) = U0(:);
I = speye((nx-1)^2);

fprintf('time integration in progress ...\n');
tic;
G = I + dt * A;
for n = 1 : nt
   % show progress
   if mod(ti(n), 1) == 0
      fprintf('t = %4.2f\n', ti(n));
   end
   % euler step
```

```
   U(:,n+1) = G \ U(:,n);
end
compute_time = toc;
fprintf('time integration complete in %g seconds\n', compute_time);
  .
  .
  .
```

The computational cost of solving the system can be reduced by computing the Cholesky factor L of the SPD matrix G, $G = L^T L$. The speed can be further improved by permuting G before computing its Cholesky factor, in order to produce a sparse Cholesky factor, which can be solved in fewer flops. In the provided MATLAB code, this is implemented as follows:

Cholesky (runs in 25.26 seconds):

```
  .
  .
  .
U = zeros((nx-1)^2,nt+1);
U(:,1) = U0(:);
I = speye((nx-1)^2);

fprintf('time integration in progress ...\n');
tic;

%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%
G = I + dt * A;
L = chol(G);
%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%

for n = 1 : nt
  .
  .
    Z = L' \ U(:, n);
    U(:, n+1) = L \ Z;
  .
  .
```

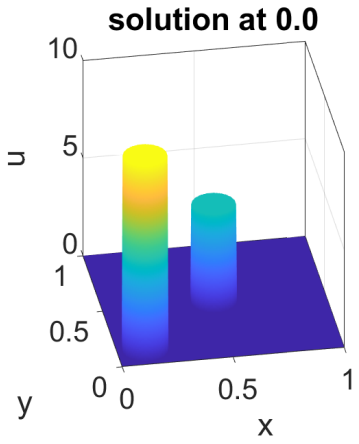
Cholesky and symamd (runs in 4.65 seconds):

```
  .
  .
  .
U = zeros((nx-1)^2,nt+1);
U(:,1) = U0(:);
I = speye((nx-1)^2);
```

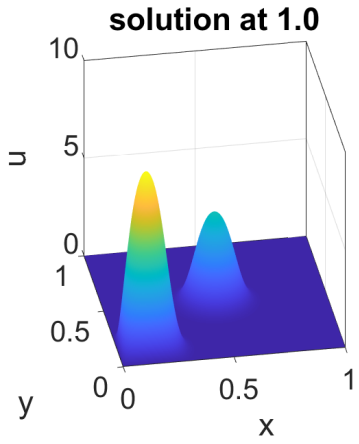
```
fprintf('time integration in progress ...\n');
tic;

%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%
G = I + dt * A;
p = symamd(G);
L = chol(G(p, p));
%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%

for n = 1 : nt
  .
  .
    Z = L' \ U(p, n);
    U(p, n+1) = L \ Z;
  .
  .
```

The results of the solve (using Cholesky & symamd) are shown in Figure 3. In this example, it is clearly most appropriate to optimize the linear system before attempting to solve it, by pre-factoring a sparse Cholesky factor, but in some cases this may not be beneficial– the time to factorize the sparse factor using symamd was 2.63449e-01 seconds, about twice the time of solving for a single time step without Cholesky.

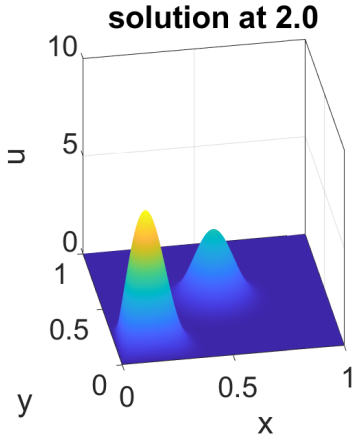
In this case, with 500 time steps, the extra computational cost of pre-factorizing is worth the time saved in solving, but would not be appropriate for a system with only a few time steps, where regular Cholesky or direct solve may be more appropriate.



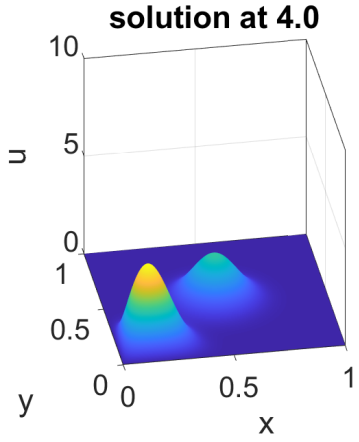
(a)



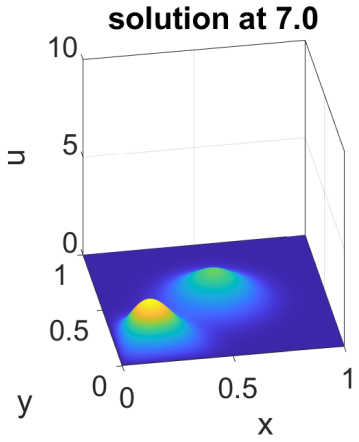
(b)



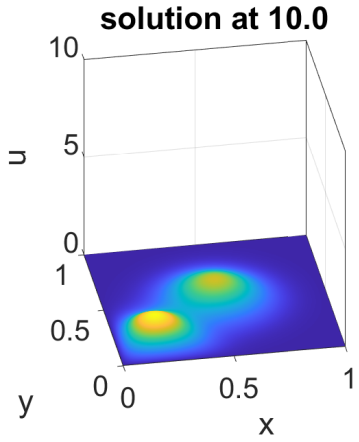
(c)



(d)



(e)



(f)

Figure 3: Solutions of Heat Equation using sparse Cholesky

Question A Least-Squares MATLAB Code

```
T = [0.0100 0.0400 0.1400 0.2100 0.2200 0.3800 0.4600 0.4800 0.6000 0.6600 0.7200 0.7900
F = [0.0639 -0.0973 0.5178 0.9686 0.7827 0.9832 0.8183 0.6544 0.3314 0.0661 -0.1612 -0.5

N = 1:14;
data = zeros(14, 4); % Ouput table for report
data(:, 1) = N';

for n = N
    [x_direct, c] = direct_solve(T, F, n)
    x_qr     = qr_solve(T, F, n)

    soln_direct = polyval(x_direct, T)
    soln_qr     = polyval(x_qr, T)

    err_direct  = norm(soln_direct - F, 2)/norm(F, 2)
    err_qr      = norm(soln_qr     - F, 2)/norm(F, 2)



    data(n, 2:4) = [err_direct err_qr c]

end

N = 2:5;
plt_t = linspace(min(T), max(T), 100);

%% plots
for n=N
    x_qr     = qr_solve(T, F, n)
    phi_t = polyval(x_qr, plt_t);

    figure()
    hold on
    scatter(T, F, "filled")
    plot(plt_t, phi_t)
    hold off
    legend("$(t_i, f_i)$", "$\varphi(t)$", interpreter="latex")
    title(sprintf("Polynomial Interpolation, N=%i", n))
    saveas(gcf, "fig_"+n+".png")
end

%%
x_qr     = qr_solve(T, F, 14);
```

```
x_direct = direct_solve(T, F, 14);
phi_qr = polyval(x_qr, plt_t);
phi_direct = polyval(x_direct, plt_t);

figure()
hold on
scatter(T, F)
plot(plt_t, phi_direct, LineStyle="--")
plot(plt_t, phi_qr)
saveas(gcf, "n14.png")
legend("$(t_i, f_i)$", "$\varphi_{direct}(t)$","$\varphi_{QR}(t)$", interpreter="latex")


%% print data to latex
output_string = "";
for i = 1:14
    for j = 1:3
        output_string = output_string +"&"+ data(i, j) ;
    end
    output_string = output_string +"&"+ sprintf("%.3E", data(i, 4)) ;
    output_string = output_string + "\\"+ newline;
end
output_string

%% functions
function [x, c] = direct_solve(t, f, n)
    m = length(t);
    t = reshape(t, [m 1]);
    f = reshape(f, [m 1]);
    A = ones(m, n);

    for i = 1:n-1
        A(:, n-i) = t.^(i);
    end
    x = (A' * A)\(A' * f);

    c = cond((A' * A), 2);
end

function x = qr_solve(t, f, n)
    m = length(t);
    t = reshape(t, [m 1]);
    f = reshape(f, [m 1]);
    A = ones(m, n);
    for i = 1:n-1
```

```
89          A(:, n-i) = t.^(i);
90      end
91      [Q,R] = qr(A, 0);
92      z = Q' * f;
93      x = UTriSol(R, z, length(z)); % from K. Ming Leung, 01/26/03
94  end
```

Included is the function `backSubstitution` (implemented as `UTriSol` in my code), from cse.engineering.nyu.edu/ mleung/:

```
1  function x=backSubstitution(U,b,n)
2  % Solving an upper triangular system by back-substitution
3  % Input matrix U is an n by n upper triangular matrix
4  % Input vector b is n by 1
5  % Input scalar n specifies the dimensions of the arrays
6  % Output vector x is the solution to the linear system
7  % U x = b
8  % K. Ming Leung, 01/26/03
9
10 x=zeros(n,1);
11 for j=n:-1:1
12     if (U(j,j)==0) error('Matrix is singular!'); end;
13     x(j)=b(j)/U(j,j);
14     b(1:j-1)=b(1:j-1)-U(1:j-1,j)*x(j);
15 end
```