Problem 1:

In this problem, I had to show that the nth Fibonacci number can be obtained by taking the nth derivative of the exponential generating function and evaluating it at t = 0. The exponential generating function is a power series where the coefficient of each term is the Fibonacci number divided by the factorial of the index. Taking the nth derivative eliminates all terms except the one involving t^n, and evaluating at t = 0 returns F_n. This was demonstrated in MATLAB by comparing the Fibonacci numbers obtained from the exponential generating function definition to the actual values; based on the MATLAB code and its solution, they match exactly. This showed how the derivative property allows us to recover each Fibonacci number directly from the structure of the generating function.

Problem 2:

This problem asked to prove that the second derivative of the exponential generating function equals the first derivative plus the function itself. In MATLAB, I built the function numerically using the exponential generating function definition with the first 16 Fibonacci numbers. Then I used finite differences to estimate the first and second derivatives of the function at various values of t. After this, I compared the left-hand side to the right-hand side and found that the difference was minimal. This confirmed that the identity holds. This proved that the exponential generating function satisfies a second-order differential equation, which reflects the recursive nature of the Fibonacci sequence.

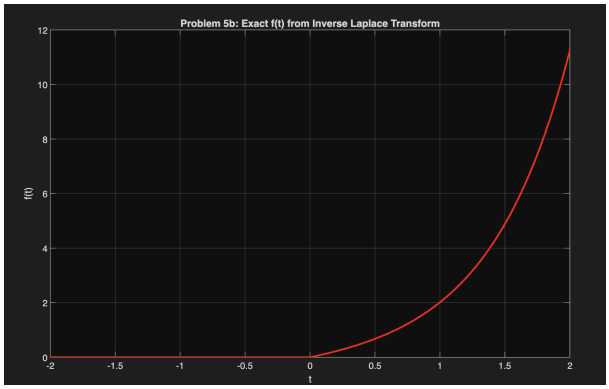
Problem 3:

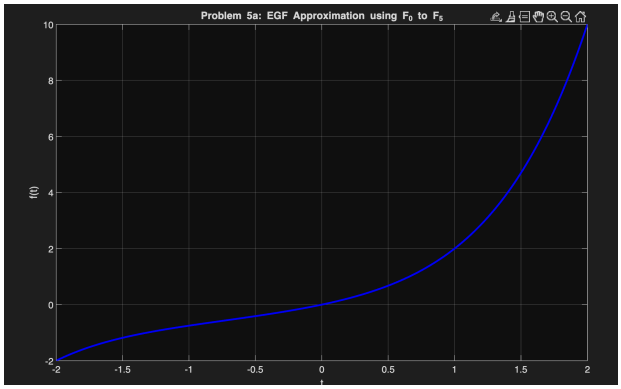
In this problem, I applied the Laplace transform to the differential equation from problem 2. I used the Laplace transform method and plugged in the initial conditions to solve the algebraic equation. After doing this, I found that F(s) = 1 divided by (s squared minus s minus 1). Therefore, this represents the Laplace domain of the exponential generating function.

Problem 4:

In this problem, we had to find the poles and zeros of F(s). Since the numerator of F(s) is 1, there are no finite zeros. I used the MATLAB roots function to assist in solving the denominator equation. This equation is s squared minus s minus 1. The two roots are real and distinct, and correspond to the golden ratio and its conjugate, being 1.618 and -0.618. The poles we found are important because they determine the exponential terms that will appear in the time domain expression of the function, which is used in the next problem.

Problem 5:





In part 5a, I used the exponential generating function to approximate f(t) to make the series only include the first six Fibonacci numbers. I implemented this as a polynomial in MATLAB and plotted it over the range -2 to 2. The result gave an approximation near t = 0 but diverged away from the exact behavior as t increased over time. In part 5b, I used the inverse Laplace expression for f(t). This is a combination of exponentials involving the golden ratio and its conjugate. Furthermore, I included a unit step function to ensure the function is zero for t less than 0. Plotting this version proved it to be accurate over the full interval. Comparing both plots confirmed that the approximation is only truly valid near zero. The inverse Laplace result captures the entire true behavior of f(t).

Problem 6:

This problem asked us to derive a closed-form formula for the Fibonacci number F_n using the expression for f(t) and the derivative identity. Since f(t) is a combination of exponentials, taking the nth derivative of each exponential gives a factor of the base raised to the nth power. Evaluating this result a t = 0 yields the Fibonacci number, which equals the golden ratio raised to the nth power minus its conjugate raised to the nth power, which is all divided by the square root of 5. This was implemented in my MATLAB code and tested for n = 12, for which the test gave a true result.

Problem 7:

In this problem, I reused the closed-form formula from the previous problem to compute the 19th Fibonacci number. I substituted n = 19 into the expression and used MATLAB to compute the result. The result that MATLAB returned was 4181, which is the correct value of the 19th Fibonacci number. This therefore confirms that the closed-form expression remains accurate and stable for larger values of n.

Appendix:

```
close all;

clear;

clc;

% PROBLEM 1

n = 30; % Change this to any n

% Approximate EGF: f(t) = sum F_k / k! * t^k

% Since f^(n)(0) = F_n, we simulate it directly

% We'll hardcode Fibonacci values up to n

F = zeros(1, n+1);

F(2) = 1;

for i = 3:n+1

   F(i) = F(i-1) + F(i-2);

end

% f(t) = sum_{k=0}^{n} (F_k / k!) * t^k

% The coefficient of t^n is F_n / n!

% So: F_n = coeff * n!

coeff = F(n+1) / factorial(n);

Fn = coeff * factorial(n);

fprintf('F_%d = %d\n', n, Fn);

% PROBLEM 2

% Fibonacci numbers F_0 to F_15

F = zeros(1, 16);

F(2) = 1;

for i = 3:16

   F(i) = F(i-1) + F(i-2);

end

% Time point to test (can use any small value near 0)
```

```
t = 0.1;
% Compute f(t)
f = 0;
for k = 0:15
   f = f + (F(k+1) / factorial(k)) * t^k;
end
% Numerical derivatives using finite difference
h = 1e-5;
% f'(t)
f_plus = 0;
f_minus = 0;
for k = 0:15
   f_plus = f_plus + (F(k+1) / factorial(k)) * (t + h)^k;
   f_minus = f_minus + (F(k+1) / factorial(k)) * (t - h)^k;
end
df = (f_plus - f_minus) / (2*h);
% f''(t)
f_plus2 = 0;
f_minus2 = 0;
for k = 0:15
   f_plus2 = f_plus2 + (F(k+1) / factorial(k)) * (t + h)^k;
   f_minus2 = f_minus2 + (F(k+1) / factorial(k)) * (t - h)^k;
end
d2f = (f_plus2 - 2*f + f_minus2) / (h^2);
% Check identity
check = abs(d2f - (df + f)) < 1e-6;
fprintf("f''(t) ≈ f'(t) + f(t) at t = %.4f: %s\n", t, string(check));
% PROBLEM 3
```

```
% Define s range (for possible later evaluation or plotting)
s = linspace(-5, 5, 1000);  % Just a dummy range if needed
% Manually compute F(s) from:
% (s^2 - s - 1) * F(s) = 1  -->  F(s) = 1 / (s^2 - s - 1)
% Example: Evaluate F(s) at a few points for verification
s_vals = [2, 3, 4];
for i = 1:length(s_vals)
   s_i = s_vals(i);
   Fs = 1 / (s_i^2 - s_i - 1);
   fprintf('F(%d) = %.5f\n', s_i, Fs);
end
% PROBLEM 4
% Define numerator and denominator
num = [1];           % Numerator: constant → no finite zeros
den = [1 -1 -1];     % Denominator: s^2 - s - 1
% Compute zeros and poles
zeros_F = roots(num);   % Zeros of F(s)
poles_F = roots(den);   % Poles of F(s)
% Display results
fprintf('Zeros of F(s):\n');
if isempty(zeros_F)
   fprintf('  None (numerator is constant)\n');
else
   disp(zeros_F);
end
fprintf('Poles of F(s):\n');
disp(poles_F);
% PROBLEM 5a
```

```
% Approximate f(t) using EGF with F_0 to F_5

% Define Fibonacci numbers F_0 to F_5

F = [0 1 1 2 3 5];

% Time range

t = linspace(-2, 2, 400);

f_approx = zeros(size(t));

% Compute EGF: f(t) = sum_{k=0}^5 (F_k / k!) * t^k

for k = 0:5

   f_approx = f_approx + (F(k+1) / factorial(k)) * t.^k;

end

% Plot the result

figure;

plot(t, f_approx, 'b', 'LineWidth', 2);

xlabel('t');

ylabel('f(t)');

title('Problem 5a: EGF Approximation using F_0 to F_5');

grid on;

% PROBLEM 5b

% Plot exact f(t) from inverse Laplace

phi1 = (1 + sqrt(5)) / 2;

phi2 = (1 - sqrt(5)) / 2;

% Unit step: u(t) = 1 when t >= 0

u = double(t >= 0);

% Compute f(t)

f_exact = (exp(phi1 * t) - exp(phi2 * t)) / sqrt(5) .* u;

% Plot exact f(t)

figure;

plot(t, f_exact, 'r', 'LineWidth', 2);
```

```
xlabel('t');

ylabel('f(t)');

title('Problem 5b: Exact f(t) from Inverse Laplace Transform');

grid on;

% PROBLEM 6

n = 12;  % or any n ≥ 0

phi1 = (1 + sqrt(5)) / 2;

phi2 = (1 - sqrt(5)) / 2;

F_n = (phi1^n - phi2^n) / sqrt(5);

fprintf('F_%d = %.0f\n', n, F_n);

% PROBLEM 7

n = 19;

phi1 = (1 + sqrt(5)) / 2;

phi2 = (1 - sqrt(5)) / 2;

F_19 = (phi1^n - phi2^n) / sqrt(5);

fprintf('F_%d = %.0f\n', n, F_19);
```