Problem 1

Lagrange's formula for the interpolating polynomial of degree N passing through $N+1$ points (x_n, y_n) for $n \in \mathbf{N} = \{0, 1, ..., N\}$ is

$$p_N(x) = \sum_{n \in \mathbf{N}} y_n L_n(x)$$

$$L_n(x) = \prod_{m \in \mathbf{N} \setminus n} \frac{x - x_m}{x_n - x_m}$$

Let $N = 2$ to find the quadratic interpolation. I take the derivative to find the formula for $p_2'(x)$. I focus on L_n' because, since $p_N(x)$ is just a linear combination of L_n terms, the same will go for their derivatives. I will use $\mathbf{M_n} = \mathbf{N} \setminus n$ to minimize clutter.

$$p_2'(x) = \sum_{n=0}^{2} y_n L_n'(x)$$

$$L_n'(x) = \frac{1}{(x_n - x_{m_1})(x_n - x_{m_2})} \quad \frac{d}{dx}\left[(x - x_{m_1})(x - x_{m_2})\right] \qquad m_1, m_2 \in \mathbf{M_n}$$

$$= \frac{2x - x_{m_1} - x_{m_2}}{(x_n - x_{m_1})(x_n - x_{m_2})}$$

Evaluate $L_n'(x)$ at $x^* = (x_{i-1} + x_i)/2, \quad i, i-1 \in \mathbf{N}$.

$$L_n'(x) = \frac{x_{i-1} + x_i - x_{m_1} - x_{m_2}}{(x_n - x_{m_1})(x_n - x_{m_2})}$$

There are three cases to consider:

$$\begin{array}{lll} (1) & n = i, & \mathbf{M_n} = \{i-1, j\}, \\ (2) & n = i-1, & \mathbf{M_n} = \{i, j\}, \\ (3) & n = j, & \mathbf{M_n} = \{i, i-1\} \end{array}$$

where $j \neq i, i-1, \ j \in \mathbf{N}$ represents the third given point.

Case 1: $(n = i)$

$$L_i'(x^*) = \frac{x_{i-1} + x_i - x_{i-1} - x_j}{(x_i - x_{i-1})(x_i - x_j)} = \frac{x_i - x_j}{(x_i - x_{i-1})(x_i - x_j)} = \frac{1}{x_i - x_{i-1}}$$

Case 2: $(n = i - 1)$

$$L'_{i-1}(x^*) = \frac{x_{i-1} + x_i - x_i - x_j}{(x_{i-1} - x_i)(x_{i-1} - x_j)} \;=\; \frac{x_i - x_j}{(x_{i-1} - x_i)(x_{i-1} - x_j)} \;=\; \frac{1}{x_{i-1} - x_i}$$

Case 3: $(n \neq i, i - 1)$

$$L'_j(x^*) = \frac{x_{i-1} + x_i - x_i - x_{i-1}}{(x_j - x_i)(x_j - x_{i-1})} \;=\; 0$$

10mm Substituting back in for $p'_2(x^*)$,

$$p'_2(x^*) = \sum_{n=0}^{2} y_n L'_n(x^*)$$

$$= y_i \left(\frac{1}{x_1 - x_{i-1}} \right) + y_{i-1} \left(\frac{1}{x_{i-1} - x_i} \right) + 0$$

$$= \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

$$\boxed{p'_2\left((x_{i-1} + x_i)/2\right) \;=\; f[x_{i-1}, x_i]}$$

Note: We know this holds for $i = 1, 2$ implicitly because the preceding statement holds for all values of i for which $i, i - 1 \in \mathbf{N}$. With $N = 2$ and $\mathbf{N} = \{0, ..., N\} = \{0, 1, 2\}$, this necessitates $i = 1, 2$.

Problem 2

The maximum interpolation error of a function $f(x)$ on interval S is

$$E^* = \max_{x \in S} |f(x) - g(x)|$$

where $g(x)$ is the interpolation function. Here, $f(x) = \cos x$ and $S = [0, \pi]$.

a. Given that $f(x)$ is twice-differentiable (true for cos$x$), the maximum error for piecewise-linear interpolation (adjacent table entries are interpolated by a straight line) is

$$\frac{h^2}{8} \max_{x \in S} |f''(x)|$$

where h is the x-length of the interval.[1] If we take our points to be evenly spaced and define each interval as between each pair of adjacent points, h is the spacing of the points.

If we want 6-decimal-place accuracy, we are looking for a maximum error less than 5e-7 (to make sure rounding agrees). We solve for h to find the necessary spacing to yield a maximum error under this value.

$$\text{5e-7} = \frac{h^2}{8} \max_{x \in [0,\pi]} |f''(x)|$$

$$h = \sqrt{\frac{\text{4e-6}}{\max\limits_{x \in [0,\pi]} |f''(x)|}}$$

Because cos$x$ is thrice-differentiable (in fact, $\cos x$ is infinitely differentiable), we can find the maximum of $f''(x) = -\cos x$ by finding the roots of $f'''(x)$ on the interval. $\frac{d}{dx}\big[-\cos x\big] = \sin x$, which has roots at $x = 0$ and $x = \pi$. We then evaluate $|f''(x)|$ at these values to find which is the maximum. It turns out that $|f''(0)| = |f''(\pi)| = 1$ is the maximum.

$$h = \sqrt{\frac{\text{4e-6}}{1}} = \text{2e-3}$$

(Because h represents point spacing, we know it must be the positive root.)

Thus, to guarantee 6-decimal-place accuracy, our lookup table must have points spaced (in x) at most 0.002 apart.

[1] Yano M., Penn, J. D., Konidaris G., & Patera, A.T.. Math, Numerics, & Programming (for Mechanical Engineers). August 2013. Massachusetts Institute of Technology: MIT OpenCouseWare, https://ocw.mit.edu/. License: Creative Commons BY-NC-SA.

b. We do much the same thing for piecewise-quadratic interpolation (a degree-2 polynomial is fit to every 3 consecutive points). As $f(x)$ is thrice-differentiable, the maximum interpolation error in the quadratic case is:[1]

$$E^* = \frac{h^3}{72\sqrt{3}} \max_{x\in[0,\pi]} |f'''(x)|$$

.

If we want the same accuracy, we again set $E^* =$ 5e-7 and solve for h.

$$\text{5e-7} = \frac{h^3}{72\sqrt{3}} \max_{x\in[0,\pi]} |f'''(x)|$$

$$h = \left(\frac{\text{3.6e-5}\sqrt{3}}{\max\limits_{x\in[0,\pi]} |f''(x)|} \right)^{1/3}$$

To find the maximum term, we find the roots of $f^{(4)}(x) = \cos x$, which exist at $x = \pi/2$ and $x = 3\pi/2$ on $[0, \pi]$. Evaluating $|f'''(x)| = |\sin x|$ at these points gives us the maximum value $|f'''(\pi/2)| = |f'''(3\pi/2)| = 1$.

$$h = \left(\frac{\text{3.6e-5}\sqrt{3}}{1} \right)^{1/3} \approx \text{3.9654e-2} \approx \text{4e-2}$$

Thus, using quadratic interpolation, we only need a spacing of approximately 4e-2, about $20\times$ coarser than for linear interpolation.

c. Assume we start entries with $x = 0$ and and place a point at $x = \pi$ (rounded to 6 decimal places) even if the interpolation doesn't land on it exactly.

- For linear interpolation, we use a spacing of 2e-3. Thus, there will be $(\pi - 0)/$2e-3 points in the table, which evaluates to approximately 1570.7. We round up to **1571 entries** to account for a point placed at π, which is slightly undershot by the 1570th entry.
- For quadratic interpolation, we use a spacing of approximately 4e-2. However, the *maximum* spacing we can use to guarantee 6 decimal places of accuracy is slightly below this. We will use the slight underestimated value 3.965e-2 to ensure proper accuracy. Thus, there will be $(\pi - 0)/$3.965e-2 points in the table, which evaluates to approximately 79.2! As before, we round up to **80 entries** to account for a point placed at π, which is slightly undershot by the 79th entry.

Problem 3

Newton's method finds the roots of a function by iterating $x_{n+1} = x_n - f(x_n)/f'(x_n)$ starting with an initial estimate of the root x_0. Newton's method only works on equations that we can state as $f(x) = 0$. We can find solutions to $x = \tan x$ by defining a function $f(x) = \tan x - x$ and finding its roots.

I first graphed $f(x)$, as I know $\tan x$ is a fairly ill-behaved function; it is periodic and approaches $\pm\infty$ asymptotically at the bounds of each period. Figure 1 shows $f(x)$ graphed using the online Desmos Graphing Calculator[2] zoomed around the roots of interest.

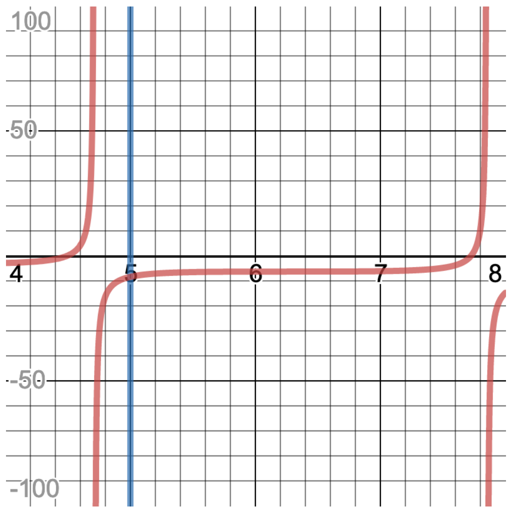

Figure 1 Graph of $f(x) = \tan x - x$ (red), zoomed around the roots closest to $x = 5$ (highlighted in blue). Note that the x and y axes use very different scales.

Via rough visual estimate, the roots of interest appear to occur around $x = 4.5$ and $x = 7.75$, with an asymptote between them at approximately $x = 5$. To account for this, my program runs Newton's method from a multitude of starting points. Since the period of $\tan x$ is π, I sample starting points on the interval $5 \pm \pi$ with a spacing of 0.1. This results in 63 runs. The window is overly generous, but it ensures that the program samples starting points on either side of the roots, and the code is simple enough that it doesn't add a significant delay.

From each starting point, the program runs for 20 iterations, or until consecutive steps converge to estimates within 5e-7 of each other (to ensure 6-decimal-place accuracy). If the estimate was over π away from 5, it implied that the root estimate was either a) converging on a different root than the two closest to 5, or b) Newton's Method diverged. Either way, these solutions were not the roots asked for, and were thrown out. The runs that converged within π of 5 did so onto two distinct points near these estimates that I take to be my solutions (rounded to 6 decimal places): $\mathbf{x} = \mathbf{4.493409}$ and $\mathbf{x} = \mathbf{7.725252}$.

[2]Desmos Graphing Calculator. 2020. *Desmos Graphing Calculator.* [online] Available at: https://www.desmos.com/calculator [Accessed 5 November 2020].

Problem 4

Lagrange's formula for the interpolating polynomial of degree N passing through $N+1$ points (x_n, y_n) for $n \in \mathbf{N} = \{0, 1, ..., N\}$ is

$$p_N(x) = \sum_{n \in \mathbf{N}} y_n L_n(x)$$

$$L_n(x) = \prod_{m \in \mathbf{N} \setminus n} \frac{x - x_m}{x_n - x_m}$$

With $x_0 = 58,\ x_1 = 108,\ x_2 = 149.5,\ x_3 = 227,\ x_4 = 778$ (and numbers presented to 6 significant digits for conciseness):

$$L_0(x) = \frac{(x-108)(x-149.5)(x-227)(x-778)}{(58-108)(58-149.5)(58-227)(58-778)} = \frac{(x-108)(x-149.5)(x-227)(x-778)}{5.56686 \times 10^8}$$

$$L_1(x) = \frac{(x-58)(x-149.5)(x-227)(x-778)}{(108-58)(108-149.5)(108-227)(108-778)} = \frac{(x-58)(x-149.5)(x-227)(x-778)}{-1.65440 \times 10^8}$$

$$L_2(x) = \frac{(x-58)(x-108)(x-227)(x-778)}{(149.5-58)(149.5-108)(149.5-227)(149.5-778)} = \frac{(x-58)(x-108)(x-227)(x-778)}{1.84959 \times 10^8}$$

$$L_3(x) = \frac{(x-58)(x-108)(x-149.5)(x-778)}{(227-58)(227-108)(227-149.5)(227-778)} = \frac{(x-58)(x-108)(x-149.5)(x-778)}{-8.58790 \times 10^8}$$

$$L_4(x) = \frac{(x-58)(x-108)(x-149.5)(x-227)}{(778-58)(778-108)(778-149.5)(778-227)} = \frac{(x-58)(x-108)(x-149.5)(x-227)}{1.67057 \times 10^{11}}$$

Therefore, with $y0 = 88,\ y_1 = 224.7\ y_2 = 365.3\ y_3 = 687,\ y_4 = 4331.5$, the Lagrange polynomial $P_4(x)$ is:

$$\begin{aligned}
P_4(x) = &\ 88 \times \frac{(x-108)(x-149.5)(x-227)(x-778)}{5.56686 \times 10^8} \\
-&\ 224.7 \times \frac{(x-58)(x-149.5)(x-227)(x-778)}{1.65440 \times 10^8} \\
+&\ 365.3 \times \frac{(x-58)(x-108)(x-227)(x-778)}{1.84959 \times 10^8} \\
-&\ 687 \times \frac{(x-58)(x-108)(x-149.5)(x-778)}{8.58790 \times 10^8} \\
+&\ 4331.5 \times \frac{(x-58)(x-108)(x-149.5)(x-227)}{1.67057 \times 10^{11}}
\end{aligned}$$

Problem 5

Given the derivative $f'(x)$ and the initial condition $f(0)$, we can integrate $f'(x)$ to uniquely define $f(x)$.

$$\begin{aligned}
f'(x) &= e^x - 4 \\
f(0) &= 2
\end{aligned}$$

$$\begin{aligned}
F(x) &= \int (e^x - 4)\ dx\ , \quad \text{the family of functions with derivative } f'(x). \\
&= e^x - 4x + c
\end{aligned}$$

$$\begin{aligned}
f(0) &= e^0 - 4(0) + c \\
(2) &= 1 + c \\
c &= 1
\end{aligned}$$

$$f(x) = e^x - 4x + 1$$

We have one point, $f(0) = 2$, by definition. This is positive, so to find a zero crossing we will look for a value of x such that $f(x) < 0$.

Let's take the next most easily evaluated point, at $x = 1$.

$$\begin{aligned} f(1) &= e^1 - 4(1) + 1 \\ &= e - 4 + 1 \\ &\approx -0.281718 \end{aligned}$$

Luckily, we find $f(1)$ is negative, so given that $f(x)$ is continuous, there are an odd number of zero-crossings, and therefore at least one root, between $x = 0$ and $x = 1$.

We then use the bisection method to estimate a root of $f(x)$ starting at the interval [0,1]. Thus, at each step, we 1) find the midpoint x_n of our interval (a, b), 2) evaluate $f(x_n)$, and 3) redefine our interval such that x_n is one endpoint, and the other endpoint has the opposite sign. We stop iterating when $\Delta_n = |f(x_n) - f(x_{n-1})| < \delta$. I will use $\delta =$ 1e-3. A trace of the states of the bisection algorithm are tabulated below.

n	(a,b)	x_n	$f(x_n)$	Δ_n
0	(0,1)	0.5	+0.648721	—
1	(0.5,1)	0.75	+0.117000	0.5317
2	(0.75,1)	0.875	-0.101124	0.2181
3	(0.75,0.875)	0.8125	+0.003535	0.1047
4	(0.8125,0.875)	0.84375	-0.049930	0.0535
5	(0.8125,0.84375)	0.828125	-0.023477	0.0265
6	(0.8125,0.828125)	0.8203125	-0.010041	0.0134
7	(0.8125,0.8203125)	0.81640625	-0.00327	0.0068
8	(0.8125,0.81640625)	0.814453125	+0.000128	0.0034
9	(0.814453125,0.81640625)	0.8154296875	-0.001572	0.0017
10	(0.814453125,0.8154296875)	0.81494140625	-0.000722	0.0008

Thus, there is a root of $f(x)$ at approximately $x = 0.81494140625$.

Appendix: Problem 3 Jupyter Notebook

[1]:
```
#Imports
import numpy as np
```

Function definitions

[2]:
```
def f(x):
    '''
    Rewrite tan(x) = x as f(x) = 0 = tan(x) - x to find solutions with Newton's␣
 ↪Method.
    Zeros of f(x) are solutions to tan(x) = x.
    '''

    return np.tan(x) - x

def dfdx(x):
    '''
    First derivative of f(x) w.r.t. x.
    f'(x) = sec^2(x) - 1
    '''
    sec = 1/np.cos(x)
    return np.power(sec,2) - 1

def NM_step(x0,fx,df):
    '''
    Given an x0 value, calculate the next step using Newton's method.
    '''
    x1 = x0 - fx(x0)/df(x0)
    return x1

def run_NM(x0,fx,df,n=20,tolerance=5e-7):
    '''
    Run Newton's method to find the roots of fx, with derivative dx,
    using initial estimate x0, for n iterations or until consecutive
    steps converge to within tolerance.
    '''
    xlast = x0
    for i in range(n):
        # Run one step on Newton's Method
        x = NM_step(x0=xlast,fx=fx,df=df)
        # Check for convergence, otherwise keep iterating
        if (np.abs(x-xlast)<tolerance):
            return x
        # Update xlast
        xlast = x
    return x
```

"main()" program

[3]:
```
################################################################################
# Initialize parameters and initial conditions
################################################################################

# Define window of interest (length of window will actually be 2*WINDOW)
WINDOW = np.pi

# Define a tolerance within which we consider two root estimates to be the same
TOLERANCE = 5e-7 # inspired by problem 2

# If NM hasn't converged in MAX_ITERS iterations, consider it divergent
MAX_ITERS = 20

# Initialize root estimate array
root_estimates = np.array([])

# Array of first estimates to initialize Newton's Method
x0 = np.arange(5-WINDOW,5+WINDOW,0.1)


################################################################################
# Run Newton's Method
################################################################################
# Iterate through initial estimates
for x in x0:
    x_est = run_NM(x0=x,fx=f,df=dfdx,n=MAX_ITERS,tolerance=TOLERANCE)
    # If the estimate is ever over WINDOW of 5, we can assume that NM is either␣
 ↪converging
    # on the incorrect root or diverging, and abandon the run
    if np.abs(x_est-5) < WINDOW:
        # Check if this root is already recorded
        if root_estimates.size == 0: # If we have no estimates yet, append this␣
 ↪root
            root_estimates = np.append(root_estimates,x_est)
        # Otherwise, if it's more than TOLERANCE different than recorded roots,␣
 ↪append this root
        elif np.all(np.abs(root_estimates-x_est)>TOLERANCE):
            root_estimates = np.append(root_estimates,x_est)

# Print roots (rounded to 6 decimal places for presentation)
print(np.round(root_estimates,6))
```

```
[4.493409 7.725252]
```