Problem 7.1 We are tasked with finding all 3 solutions for the function $f(x) = x^3 - 5x^2 - 12x + 19 = 0$ via newtons method, for this problem I figured the best way forward was to use a modified and redacted version of my C code with improvements made thanks to suggestions by Ed Bueler.

```c
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include <stdbool.h>

//------------------------------INTERFACE-----------------------------------
#define MIN -10.0
#define MAX 10.0
#define STEP 0.05
#define EPSILON 1e-6
#define STORAGE 100

double f(double x)
{
    return pow(x, 3) - 5 * pow(x, 2) - 12 * x + 19;
}

//--------------------------------------------------------------------------

// Derivative using central finite differences
double deriv(double x, double h)
{
    return (f(x+h) - f(x-h)) / (2.0*h);
}

bool signCheck(double prev, double curr)
{
    // Check if the sign has changed, considering EPSILON for near-zero values
    if ((prev > EPSILON && curr < -EPSILON) || (prev < -EPSILON && curr > EPSILON))
     {
        return true;
    }
    return false;
}

// Newton's Method for finding zeros
double duke_newton(double x0, double h, double tolerance, int max_iterations)
{
    double x = x0;
    double x_new;

    for (int i = 0; i < max_iterations; i++) {
        x_new = x - f(x) / deriv(x, h);

        if (fabs(x_new - x) < tolerance) {
            return x_new;
        }
        x = x_new;
    }
    printf("Did not converge within %d iterations for given range.\n",
    max_iterations);
    return x_new;
}

```

```c
int main()
{
    double h = EPSILON; // For notational convenience
    double x;
    int zeroCount = 0;
    int i = 0;
    double zeros[STORAGE];
    zeros[i] = MIN; // Start with MIN to capture potential zero at the boundary
    ++i;

    // Find zeros by checking where f(x) changes sign
    double prev_val = f(MIN);
    for (x = MIN + STEP; x <= MAX; x += STEP) {
        double curr_val = f(x);
        if (signCheck(prev_val, curr_val)) {
            zeros[i] = x - STEP; // Adjust back to where the sign change was
    detected
            i++;
            zeroCount++;
        }
        prev_val = curr_val;
    }
    zeros[i] = MAX; // End with MAX to capture potential zero at the boundary

    // Refine the zero estimates using Newton's method
    for(i = 1; i < zeroCount + 1; i++) { // Start from 1 to skip the MIN boundary
    check
        double guess = zeros[i];
        double out = duke_newton(guess, h, EPSILON, 6);
        printf("Zero at x: %.5f, f(x): %.5f\n", out, f(out));
    }

    printf("Number of zeros found: %d\n", zeroCount);
    return 0;
}
```

Our output ends up being:

```
Zero at x: -2.56524, f(x): -0.00000
Zero at x: 1.15555, f(x): 0.00000
Zero at x: 6.40970, f(x): -0.00000
Number of zeros found: 3
```

Note that the reason f(x) values end up as positive or negative zero, is because these are approximations and are not actually true zero, rather they are values which are sufficiently close to zero and are simply cut off within the first 5 digits. Source code is available in https://github.com/Funkematics/Optimizations/tree/m as "NewtProb.c", you may compile by running "make".

Problem 7.10 For this problem we we are tasked with solving the following system of 2 variable equations using Newtons method and we are told there are 4 solutions.

$$f_1(x_1, x_2) = x_1^2 + x_2^2 - 1 = 0$$
$$f_2(x_1, x_2) = 5x_1^2 - x_2 - 2 = 0$$

The following is a modified version of our C code, though you should note that I opted to compute the Jacobian manually, and this is because it is faster to compute manually in this case than it is to write the code to do it numerically for this problem. We evoke Cramers rule here to solve the system given by our Jacobian.

```c
#include <stdio.h>
#include <math.h>

//-------------------------INTERFACE-------------------------
#define TOLERANCE 1e-6
#define MAX_ITER 100

// Function to calculate f1 and f2
void f(double x1, double x2, double *f1, double *f2)
{
    *f1 = pow(x1, 2) + pow(x2, 2) - 1;
    *f2 = 5* pow(x1, 2) - x2 - 2;
}
//Consolidated
//-----------------------------------------------------------

// Function to calculate Jacobian
void jacobian(double x1, double x2, double J[2][2])
{
    J[0][0] = 2*x1;
    J[0][1] = 2*x2;
    J[1][0] = 10*x1;
    J[1][1] = -1;
}

// Function to solve linear equations using Cramers rule(not the seinfeld one)
void CramerCramer(double A[2][2], double b[2], double dx[2])
{
    double det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
    if (fabs(det) > TOLERANCE) {
        dx[0] = (b[0]*A[1][1] - b[1]*A[0][1]) / det;
        dx[1] = (A[0][0]*b[1] - A[1][0]*b[0]) / det;
    } else {
        printf("Jacobian singular, cannot proceed.\n");
        dx[0] = dx[1] = 0;
    }
}

// Newton's method
void duke_newton2(double *x1, double *x2, double x1_init, double x2_init)
{
    double x[2] = {x1_init, x2_init};
    double f1, f2, J[2][2], dx[2];
    int iter = 0;

    do {
        f(x[0], x[1], &f1, &f2);
        jacobian(x[0], x[1], J);

        double F[2] = {-f1, -f2};
        CramerCramer(J, F, dx);

        x[0] += dx[0];
```

```
        x[1] += dx[1];

        iter++;
        if (iter > MAX_ITER) {
            printf("Max iterations reached.\n");
            break;
        }
    } while (fabs(f1) > TOLERANCE || fabs(f2) > TOLERANCE);

    *x1 = x[0];
    *x2 = x[1];
}

int main()
{
    double x1, x2;

    // Initial guesses, I cheated and used a graph to pick out places closish to
    the intersections(but not too close)
    double initial_guesses[][2] = {
        {0.9, 0.5},
        {-0.9, 0.5},
        {-0.5, -1.5},
        {0.5, -1.5}
    };

    for (int i = 0; i < 4; i++) {
        duke_newton2(&x1, &x2, initial_guesses[i][0], initial_guesses[i][1]);
        printf("Solution %d: x1 = %f, x2 = %f\n", i+1, x1, x2);
    }

    return 0;
}
```

The solutions we get as our output ends up being

```
Solution 1: x1 = 0.732260, x2 = 0.681025
Solution 2: x1 = -0.732260, x2 = 0.681025
Solution 3: x1 = -0.473070, x2 = -0.881025
Solution 4: x1 = -0.473070, x2 = -0.881025
```

The code can be found in https://github.com/Funkematics/Optimizations/tree/main/Code/DoubNewt and can be compiled with a simple "make" command.

Problem 2.4 Corollary 6.7: If x is a feasible solution to the primal, y is a feasible solution to the dual, and $c^T x = b^T y$, then x and y are optimal for their respective problems.

Proof: Lets consider a Linear Program in standard form and its dual

$$\begin{aligned} \text{minimize} \quad & Z = c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

and

$$\begin{aligned} \text{maximize} \quad & W = b^T y \\ \text{subject to} \quad & A^T y \geq c \\ & y \geq 0 \end{aligned}$$

Let x be feasible for the primal problem and y be feasible for the dual problem given the constraints. By definition for any feasible x and y, the duality gap is $x^T s = c^T x - b^T y$, optimality ensures that $x^T s = 0$. Weak duality also states that $b^T y \leq c^T x$ which holds because $c^T x = b^T y$.

Now lets assume by contradiction that x in our primal is not optimal, Then there exists an $\hat{x}$ such that $c^T \hat{x} < c^T x$, since our $\hat{x}$ is feasible then by weak duality $b^T y \leq c^T \hat{x}$ which implies that $c^T x = b^T y < c^T \hat{x}$. This contradicts $b^T \hat{y} > b^T y$ leading to $b^T \hat{y} > b^T y = c^T x$, this contradicts weak duality $b^T \hat{y} \leq c^T x$. Therefore y must be optimal. Thus both x and y must be optimal for their respective problems. □

Problem 2.11 I need to come back to this

Problem 11 For this problem we run the kleeminty.m program in octave and get our maximum value in under 10 seconds using an n of 13 in my case. There is an exponential growth in the number of iterations needed as n increases so even though we are well below 10 seconds, when we use an $n = 14$ we are slightly above 10 seconds. The output of the last iteration as well as runtime are below:

```
z = -1.2207e+09
iters = 8192
PASS
Elapsed time is 5.2905 seconds.
```

The number of variables in standard form for n is $2n + 1$, in our case this is 27 variables in standard form for our Klee-Minty cube with $2n + 1$ constraints or 27 constraints. The runtime roughly doubles for each increment of n making it computationally infeasible to use simplex method in basic form.

Problem 12 We are tasked with writing psuedo-code that implement a phase-1 strategy that generates an initial BFS. We similarly use GNU Octave.

```
function [x, isFeasible] = phaseone(A, b)
    % A: m x n matrix full of constraints
    % b: m x 1 vector, the right side of our equations
    % x: n x 1, the starting point for our real problem
    % Ideally this should tell us if we can even solve the problem

    % phase-1 problem setup
    m = size(A, 1);   % Num of constraints
    n = size(A, 2);   % Num of variables

    I = eye(m);  % Identity matrix

    % Mix our original A with the identity
    A_prime = [A, I];
```

```
15 15
16 16      % new extended objective function
17 17      c_prime = [zeros(1, n), ones(1, m)];
18 18
19 19      % Next step, solve phaseone linear problem
20 20      % Suppose there exists a simplex solver such that
21 21      [solution, success] = simplex_solver(A_prime, b, c_prime);
22 22
23 23      % Analysis
24 24      if success
25 25          artificial_vars = solution(n + 1:end);  % Get artificial variables
26 26          original_vars = solution(1:n);          % Our original variables
27 27
28 28          % Check feasibility
29 29          if sum(artificial_vars) == 0
30 30              x = original_vars;  % Case where its feasible
31 31              isFeasible = true;
32 32          else
33 33              x = [];             % Case where its not feasible
34 34              isFeasible = false;
35 35          end
36 36      else
37 37          % This is in case of failure.
38 38          x = [];
39 39          isFeasible = false;
40 40      end
41 41  end
```

We start by adding artificial variables to our constraints A, each with a coefficient of 1. We then should use a simplex solver to minimize the sum of artificial variables. We know that we've found a solution if the sum of all artificial variables is zero, if any artificial variables are non-zero then the problem itself is infeasible or the program has failed.