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;
}

```

```
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
```