1. Algorithm implementation.

a) Research and implement the Secant root-finding method.

First, I created a function that calculates the value of a given equation based on the coefficients, the value of x and the degree of the polynomial.

```
double calculatePolynomial(double coefficients[], int degree, double x) {
    double result = 0.0;

    for (int i = degree; i >= 0; i--) {
        result += coefficients[i] * pow(x, i);
    }

    return result;
}
```

Next, based on the given formula, I created a function to directly calculate the Secant root-finding method.

The input parameters of the function are:
- Coefficients
- Degree of the polynomial
- Initial values of x0 and x1
-Tolerance with which accuracy the function should be calculated
-Maximum number of iterations

If the function computes an example with the given precision, it returns that computed number. Otherwise, it will return NAN. This allows us to process this value further in the code. For example, to throw a message that the calculation failed.

```
double secantMethod(double coefficients[], int degree, double x0, double x1, double tolerance, int maxIterations) {
    double x2, f0, f1;

    for (int i = 0; i < maxIterations; i++) {
        f0 = calculatePolynomial(coefficients, degree, x: x0);
        f1 = calculatePolynomial(coefficients, degree, x: x1);

        x2 = x1 - (f1 * ((x1 - x0) / (f1 - f0)));

        if (fabs(x2 - x1) < tolerance) {
            printf("Root found!\n");
            return x2;
        }

        x0 = x1;
        x1 = x2;
    }

    printf("Secant method did not converge within the specified iterations.\n");
    return NAN;
}
```

b) Research and implement the Newton-Raphson root-finding method.

Now I need to create a function that calculates the value of the derived polynomial equation.

```
double calculateDerivedPolynomial(double coefficients[], int degree, double x) {
    double result = 0.0;

    for (int i = degree - 1; i >= 0; i--) {
        result += coefficients[i + 1] * (i + 1) * pow(x, i);
    }

    return result;
}
```

Using the same procedure as in the previous example, I will create a function for the Newton-Raphson root-finding method.

```
double newtonRaphsonMethod(double coefficients[], int degree, double x, double tolerance, int maxIterations) {
    double x1, fx, fpx;

    for (int i = 0; i < maxIterations; i++) {
        fx = calculatePolynomial(coefficients, degree, x);
        fpx = calculateDerivedPolynomial(coefficients, degree, x);

        if (fabs(fpx) < tolerance) {
            printf("Derivative is close to zero. Newton-Raphson method cannot continue.\n");
            return NAN;
        }

        x1 = x - (fx / fpx);

        if (fabs(x1 - x) < tolerance) {
            printf("Root found at x = %lf\n", x1);
            return x1;
        }

        x = x1;
    }

    printf("Newton-Raphson method did not converge within the specified iterations.\n");
    return NAN;
}
```

c) Write a program that presents the end-user with a command-line menu, repeatedly asks the user to execute the above functions, and displays their output accordingly, or else to quit. Proper validation of user input is required.

Functions for creating command-line menu.

```
void printMenu() {
    printf("1. Secant root-finding method\n");
    printf("2. Newton-Raphson root-finding method\n");
    printf("3. Quit\n");
    printf("-----------------------------------------------\n");
    printf("Enter a choice (1, 2, or 3): ");
}
```

Now we know that we will need 2 int values from the user - the menu option and the degree of the polynomial. But both numbers must be in some range only. Therefore, we will create this function:

(This function checks if the given number is int, if it is in the given range and if there are no extra characters after the number. If everything is OK, it returns the number. Otherwise, it writes that the input is not valid and requires further input from the user)

```
int validateAndReturnInt(int minRange, int maxRange) {
    int returnValue;
    char extraChar;

    while (1) {
        if (scanf("%d%c", &returnValue, &extraChar) != 2 || extraChar != '\n' || returnValue < minRange || returnValue > maxRange) {
            printf("Invalid input. Try again: ");
            while (getchar() != '\n'); // Clear input buffer
            continue;
        }

        break;
    }

    return returnValue;
}
```

We will also have many more inputs from the user in double format. We need to validate these inputs as well. However, for some inputs there is no restriction on what range they must be in, so we will create these 2 functions:

```c
double validateAndReturnDouble() {
    double returnValue;
    char extraChar;

    while (1) {
        if (scanf("%lf%c", &returnValue, &extraChar) != 2 || !isspace( c: extraChar)) {
            printf("Invalid input. Try again: ");
            while (getchar() != '\n'); // Clear input buffer
            continue;
        }

        break;
    }

    return returnValue;
}
double validateAndReturnDoubleInRange(double minRange, double maxRange) {
    double returnValue;
    char extraChar;

    while (1) {
        if (scanf("%lf%c", &returnValue, &extraChar) != 2 || !isspace( c: extraChar) || returnValue < minRange || returnValue > maxRange) {
            printf("Invalid input. Try again: ");
            while (getchar() != '\n'); // Clear input buffer
            continue;
        }

        break;
    }

    return returnValue;
}
```

Now all that is left is to select which method to use for the calculation based on the option selected by the user, retrieve the inputs from the user and execute the method.
This could already be done in main based on switch/case or if/else. However, since reading user input is very similar in both cases I created another function. This function is no longer necessary, but it significantly reduces the length of the code and in my opinion improves readability.

```c
void rootFindingMethod(int method) {
    int degree,maxIterations;
    double x0, x1, tolerance,root;

    printf("Enter max iterations (1-%d): ",MAX_ITERATIONS);
    maxIterations = validateAndReturnInt( minRange: 1,  maxRange: MAX_ITERATIONS);


    printf("Enter tolerance(0-1): ");
    tolerance = validateAndReturnDoubleInRange( minRange: 0,  maxRange: 1);

    printf("Enter the degree of the polynomial (1-%d): ", MAX_DEGREE);
    degree = validateAndReturnInt( minRange: 1,  maxRange: MAX_DEGREE);

    double coefficients[degree + 1];
    printf("Enter the coefficients of the polynomial (from highest degree to constant term):\n");

    for (int i = degree; i >= 0; i--) {
        printf("x%d: ", i);
        coefficients[i] = validateAndReturnDouble();
    }
```

```
    printf("Enter x0: ");
    x0 = validateAndReturnDouble();

    if (method == 1) {
        printf("Enter x1: ");
        x1 = validateAndReturnDouble();

        root = secantMethod(coefficients, degree, x0, x1, tolerance, maxIterations);
    }

    if (method == 2) {
        root = newtonRaphsonMethod(coefficients, degree, x: x0, tolerance, maxIterations);
    }

    if (!isnan(root)) {
        printf("Approximate root: %lf\n", root);
    }

    printf("Returning back to menu.\n\n");
}
```

Finally, the main() function:

```
int main() {
    int option;

    while (1) {
        printMenu();

        option = validateAndReturnInt( minRange: 1,  maxRange: 3);

        switch (option) {
            case 1:
                rootFindingMethod( method: option);
                break;

            case 2:
                rootFindingMethod( method: option);
                break;

            case 3:
                printf("Exiting program");
                return 0;

            default:
                printf("Invalid option. Try again.\n");
        }
    }
}
```

2) A Generic Set library.

a) Implement a simplified version of GenSet_t that is instantiated with either elements of integer type or of fixed-sized 64-character strings. Both options need to be provided. Test all functionality.

In this case, I first created a DataType enum and a GenSet_t, which is the generic set itself that will hold the elements.

The GenSet_t stores information about the maximum capacity (this is immutable in my case, information about the current size and the Data Type. This will make it easier to create functions in the future.
Last, it stores a double pointer to the data itself. In the case of this task, we could also use an ordinary pointer. However, since we will be extending the set to hold any data type, it is preferable to use a double pointer directly.

By using a double pointer void** data, it allows for the creation of a dynamic array of pointers to void. This offers the flexibility to store and manipulate a collection of elements with varying data types.

```
typedef enum {
    DATATYPE_INT,
    DATATYPE_STRING,
} DataType;

typedef struct {
    int capacity;
    int size;
    void** data;
    DataType type;
} GenericSet_t;
```

initSet() - Creates a new set for a specific element type and allocates associated memory resources.

-Data is allocated based on capacity and data type.

```
GenericSet_t initSet(int capacity, DataType type) {
    GenericSet_t set;
    set.capacity = capacity;
    set.size = 0;

    if (type == DATATYPE_INT) {
        set.data = malloc( size: capacity * sizeof(int*));
    } else if (type == DATATYPE_STRING) {
        set.data = malloc( size: capacity * sizeof(char[64]));
    }

    set.type = type;
    return set;
}
```

deinitSet() - Destroys an existing set and relinquishes associated memory resources.

```
void deInit(GenericSet_t* set) {
    free(set->data);
    set->data = NULL;
    set->capacity = 0;
    set->size = 0;
    set->type = 0;
}
```

addToSet() - Adds a non-duplicate element to the set.

-Here it is useful to create a helper function contains(), which finds out if the element is in our set. Using dereference, its values are compared with the element in the list. In the case of a string, we use the strcmp() function from the string.h library.

```
bool contains(GenericSet_t set, void* element) {
    for (int i = 0; i < set.size; i++) {
        switch (set.type) {
            case DATATYPE_INT:
                if (*((int*)set.data[i]) == *((int*)element)) {
                    return true;  // Element found
                }
                break;
            case DATATYPE_STRING:
                if (strcmp((char*)set.data[i], (char*)element) == 0) {
                    return true;  // Element found
                }
                break;
        }
    }
    return false;  // Element not found
}
```

-Here is the addToSet() function itself. If the capacity is exceeded, or the item is already in the set, it is not inserted. Using dereference, the value is added to the set again. We can also see that we need to allocate memory for the element before inserting it into the set. This is needed because we are using a double pointer to the data.

```
void addToSet(GenericSet_t* set, void* element) {
    if (set->size >= set->capacity) {
        printf("Set is full, cannot add more elements\n");
        return;  // Set is full, do nothing
    }

    if (contains(*set, element)) {
        return;  // Element already exists, do nothing
    }
    switch (set->type) {
        case DATATYPE_INT:
            set->data[set->size] = malloc( size: sizeof(int));
            *((int*)set->data[set->size]) = *((int*)element);
            break;
        case DATATYPE_STRING:
            set->data[set->size] = malloc( size: strlen( s: (char*)element) + 1);
            strcpy((char*)set->data[set->size], (char*)element);
            break;
    }
    set->size++;
}
```

displaySet() - Outputs all elements of a set in no particular order to the standard output.

```
void displaySet(GenericSet_t set) {
    for (int i = 0; i < set.size; i++) {
        switch (set.type) {
            case DATATYPE_INT:
                printf("%d ", *((int*)set.data[i]));
                break;
            case DATATYPE_STRING:
                printf("%s ", (char*)set.data[i]);
                break;
        }
    }
    printf("\n");
}
```

unionSet() - Returns a newly created set, the union of two input sets.

-Here, just create a new set (with the size of the sum of the sizes of both sets) and insert elements from both sets into it. We can do this because the addToSet() function already ignores duplicate elements. Of course, the operation is only performed if the sets have the same data type. Otherwise, an empty set with a data type equal to the first set is returned.

```
GenericSet_t unionSet(GenericSet_t set1, GenericSet_t set2) {
    GenericSet_t result;
    if (set1.type != set2.type) {
        printf("Error: Sets have different data types\n");
        result = initSet( capacity: 0, set1.type);  // Return an empty set of the same type as set1
        return result;
    }

    result = initSet( capacity: set1.size + set2.size, set1.type);

    for (int i = 0; i < set1.size; i++) {
        addToSet( set: &result,  element: set1.data[i]);
    }
    for (int i = 0; i < set2.size; i++) {
        addToSet( set: &result,  element: set2.data[i]);
    }

    return result;
}
```

intersectSet() - Returns a newly created set, the intersection of two input sets.

-Here the same principle as in the previous case is used. But now only the elements that are also contained in the second set are inserted.

```
GenericSet_t intersectSet(GenericSet_t set1, GenericSet_t set2) {
    GenericSet_t result;
    if (set1.type != set2.type) {
        printf("Error: Sets have different data types\n");
        result = initSet( capacity: 0, set1.type);  // Return an empty set of the same type as set1
        return result;
    }

    result = initSet( capacity: set1.size, set1.type);

    for (int i = 0; i < set1.size; i++) {
        if (contains( set: set2,  element: set1.data[i])) {
            addToSet( set: &result,  element: set1.data[i]);
        }
    }

    return result;
}
```

diffSet() - Returns a newly created set, the difference between two input sets.

-Same procedure. All elements from the first set that are not included in the second set are added to the new set.

```
GenericSet_t diffSet(GenericSet_t set1, GenericSet_t set2) {
    GenericSet_t result;
    if (set1.type != set2.type) {
        printf("Error: Sets have different data types\n");
        result = initSet( capacity: 0, set1.type);  // Return an empty set of the same type as set1
        return result;
    }

    result = initSet( capacity: set1.size, set1.type);

    for (int i = 0; i < set1.size; i++) {
        if (!contains( set: set2,  element: set1.data[i])) {
            addToSet( set: &result,  element: set1.data[i]);
        }
    }

    return result;
}
```

countSet() - Returns the element count, or cardinality, of an input set.

- Because we keep track of how many elements we have in the set. This function is very simple:

```
int countSet(GenericSet_t set) {
    return set.size;
}
```

isSubsetSet() - Returns whether the first set is a subset of the second.

-True is returned if the sets have the same data type, set1 is not larger than set2 and all elements of set1 are contained in set2

```
bool isSubsetSet(GenericSet_t set1, GenericSet_t set2) {
    if (set1.type != set2.type) {
        // Error: Sets have different data types
        return false;
    }

    if (set1.size > set2.size) {
        // set1 cannot be a subset if it has more elements than set2
        return false;
    }

    for (int i = 0; i < set1.size; i++) {
        if (!contains( set: set2,  element: set1.data[i])) {
            return false;  // Element from set1 is not found in set2
        }
    }

    return true;  // All elements of set1 are found in set2
}
```

isEmptySet() - Returns whether a set is empty.

-If the size is 0 returns true, otherwise false

```
bool isEmptySet(GenericSet_t set) {
    return set.size == 0;
}
```

b) Extend the implementation in 2(a) so that there are no limitations on the supported element types this time. Additionally, support for the following additional operation is required:

The first thing to do is to extend enum.

```
typedef enum {
    DATATYPE_CHAR,
    DATATYPE_INT,
    DATATYPE_FLOAT,
    DATATYPE_DOUBLE,
    DATATYPE_STRING,
    DATATYPE_CUSTOM
} DataType;
```

initSet()

Next, we modify the initSet() function. Now we no longer choose from two data types, but it can be any. Including custom data types.
Therefore, we only allocate pointers to void (regardless of the specific size of the data the pointer points to).

```
GenericSet_t initSet(int capacity, DataType type) {
    GenericSet_t set;
    set.capacity = capacity;
    set.size = 0;
    set.data = malloc( size: capacity * sizeof(void*));
    set.type = type;
    return set;
}
```

deinitSet()
-This function remains as in a)

addToSet()
Again, we will create the contains() function.
Now we're going to use pointers to functions. Since a set can now contain any data type, including custom, we can no longer choose the behavior of the code based on switch/case. Since the custom data types can be any structure, the only possible solution is to use the function pointer to insert another function into the contains() function header to ensure correct element comparing.

```
bool contains(GenericSet_t set, void* element, bool (*compareFunc)(void*, void*)) {
    for (int i = 0; i < set.size; i++) {
        if (compareFunc(set.data[i], element)) {
            return true;  // Element found
        }
    }
    return false;  // Element not found
}
```

The comparison functions inserted into the contains() function header can look like this (within my file I created these functions for the data types: string, char, int, double, float):

```
bool compareChar(void* item1, void* item2) {
    char* char1 = (char*)item1;
    char* char2 = (char*)item2;
    return (*char1 == *char2);
}
```

If we call a function that calls the contains() function we have to insert the comparison function into the header again. The addToSet() function itself looks like this:

```
void addToSet(GenericSet_t* set, void* element, bool (*compareFunc)(void*, void*)) {
    if (set->size >= set->capacity) {
        printf("Set is full, cannot add more elements\n");
        return;  // Set is full, do nothing
    }

    if (contains(*set, element, compareFunc)) {
        return;  // Element already exists, do nothing
    }

    // Add the new element to the set
    set->data[set->size] = element;
    set->size++;
}
```

displaySet() - Outputs all elements of a set in no particular order to the standard output.

- Here we use the same procedure as before:

```
void displaySet(GenericSet_t set, void (*displayFunc)(void*)) {
    for (int i = 0; i < set.size; i++) {
        if (displayFunc != NULL) {
            displayFunc(set.data[i]);
        }
    }
    printf("\n");
}

void displayChar(void* data) {
    char value = *((char*)data);
    printf("%c ", value);
}
```

unionSet() - Returns a newly created set, the union of two input sets.
intersectSet() - Returns a newly created set, the intersection of two input sets.
diffSet() - Returns a newly created set, the difference between two input sets.
countSet() - Returns the element count, or cardinality, of an input set.
isSubsetSet() - Returns whether the first set is a subset of the second.
isEmptySet() - Returns whether a set is empty.

All these features remain unchanged from the previous section.
The only difference is that functions that use contains() or addToSet() must have pointers to the necessary functions in their header.

export() - Exports the Generic Set elements to a text file, with each line displaying a textual representation of each element in no partic- ular order.

- Here you also need to use pointers to the function:

```
void export(GenericSet_t set, const char* filename, void (*exportFunc)(void*, FILE*)) {
    FILE* file = fopen(filename, mode: "w");  // Open the file in write mode
    if (file == NULL) {
        printf("Error opening file\n");
        return;
    }

    for (int i = 0; i < set.size; i++) {
        exportFunc(set.data[i], file);
    }

    fclose(file);
}

void exportChar(void* element, FILE* file) {
    char* ch = (char*)element;
    fprintf(file, "%c\n", *ch);
}
```

c) Compile the full implementation as a shared library and provide a proper Abstract Data Type interface. The test application should link with the shared library. In case of difficulties with task 2(b), you can compile the library from the simpler version developed in 2(a), with a maximum of 10 marks.

We split our file into a header file and a source file. Then we use CMakeList to create a shared library. We'll link another file I've named "GenSetInterface.c" to the shared library as follows:

```
add_library(GenSet SHARED GenSetLibrary/GenSet.c)

add_executable(testAPP GenSetInterface.c)
target_link_libraries(testAPP GenSet)
```

In the file "GenSetInterface.c" we will test the whole functionality. For the demonstration I created my own data type "Movies", the necessary functions and tested all the functions from the library:

```
typedef struct {
    char name[100];
    double rating;
} Movies;
```

```
void exportMovie(void* element, FILE* file) {
    Movies* movie = (Movies*)element;
    fprintf(file, "Name: %s\nRating: %lf\n\n", movie->name, movie->rating);
}

// Custom display function for the Movies struct
void displayMovies(void* item) {
    Movies* movie = (Movies*) item;
    printf("Movie: %s, Rating: %.1f\n", movie->name, movie->rating);
}

bool compareMovies(void* item1, void* item2) {
    Movies* movie1 = (Movies*)item1;
    Movies* movie2 = (Movies*)item2;
    return (movie1->rating == movie2->rating && strcmp(movie1->name, movie2->name) == 0);
}
```

```c
void testMovieSet() {
    // Create sets
    GenericSet_t set1 = initSet( capacity: 5,  type: DATATYPE_CUSTOM);
    GenericSet_t set2 = initSet( capacity: 5,  type: DATATYPE_CUSTOM);
    GenericSet_t set3 = initSet( capacity: 5,  type: DATATYPE_CUSTOM);
    GenericSet_t set4 = initSet( capacity: 5,  type: DATATYPE_CUSTOM);

    // Add elements to set1
    Movies movie1 = { .name: "Movie 1",  .rating: 7.5};
    addToSet( set: &set1,  element: &movie1,  compareFunc: compareMovies);
    Movies movie2 = { .name: "Movie 2",  .rating: 8.2};
    addToSet( set: &set1,  element: &movie2,  compareFunc: compareMovies);
    Movies movie3 = { .name: "Movie 3",  .rating: 6.9};
    addToSet( set: &set1,  element: &movie3,  compareFunc: compareMovies);

    // Call the `export()` function with the relative file path
    export( set: set1,  filename: "../TextFiles/movies.txt",  exportFunc: exportMovie);

    // Add elements to set2
    Movies movie4 = { .name: "Movie 2",  .rating: 8.2};
    addToSet( set: &set2,  element: &movie4,  compareFunc: compareMovies);
    Movies movie5 = { .name: "Movie 3",  .rating: 6.9};
    addToSet( set: &set2,  element: &movie5,  compareFunc: compareMovies);
    Movies movie6 = { .name: "Movie 4",  .rating: 7.8};
    addToSet( set: &set2,  element: &movie6,  compareFunc: compareMovies);

    // Add elements to set3
    Movies movie7 = { .name: "Movie 1",  .rating: 7.5};
    addToSet( set: &set3,  element: &movie7,  compareFunc: compareMovies);
    Movies movie8 = { .name: "Movie 2",  .rating: 8.2};
    addToSet( set: &set3,  element: &movie8,  compareFunc: compareMovies);
// Display sets
printf("Set 1:\n");
displaySet( set: set1,  displayFunc: displayMovies);
printf("Set 2:\n");
displaySet( set: set2,  displayFunc: displayMovies);
printf("Set 3:\n");
displaySet( set: set3,  displayFunc: displayMovies);
printf("Set 4:\n");
displaySet( set: set4,  displayFunc: displayMovies);

// Check if set3 is a subset of set1
bool isSubset = isSubsetSet( set1: set3,  set2: set1,  compareFunc: compareMovies);
if (isSubset) {
    printf("Set 3 is a subset of Set 1\n");
} else {
    printf("Set 3 is not a subset of Set 1\n");
}

// Check if set4 is empty
bool empty = isEmptySet( set: set4);
if (empty) {
    printf("Set 4 is empty\n");
} else {
    printf("Set 4 is not empty\n");
}

// Perform union of set1 and set2
GenericSet_t unionResult = unionSet(set1, set2,  compareFunc: compareMovies);
printf("Union of Set 1 and Set 2:\n");
displaySet( set: unionResult,  displayFunc: displayMovies);

    // Perform intersection of set1 and set2
    GenericSet_t intersectResult = intersectSet(set1, set2,  compareFunc: compareMovies);
    printf("Intersection of Set 1 and Set 2:\n");
    displaySet( set: intersectResult,  displayFunc: displayMovies);

    // Perform difference of set1 and set2
    GenericSet_t difference = diffSet(set1, set2,  compareFunc: compareMovies);
    printf("Result of Set 1 - Set 2:\n");
    displaySet( set: difference,  displayFunc: displayMovies);

    // Count elements in set1
    int count = countSet( set: set1);
    printf("Number of elements in Set 1: %d\n", count);

    // Deallocate memory
    deInit( set: &set1);
    deInit( set: &set2);
    deInit( set: &set3);
    deInit( set: &set4);
    deInit( set: &unionResult);
    deInit( set: &intersectResult);
    deInit( set: &difference);
}
```

The test function is called in main()

```
int main() {
    testMovieSet();
    return 0;
}
```

```
Set 1:
Movie: Movie 1, Rating: 7.5
Movie: Movie 2, Rating: 8.2
Movie: Movie 3, Rating: 6.9

Set 2:
Movie: Movie 2, Rating: 8.2
Movie: Movie 3, Rating: 6.9
Movie: Movie 4, Rating: 7.8

Set 3:
Movie: Movie 1, Rating: 7.5
Movie: Movie 2, Rating: 8.2

Set 4:

Set 3 is a subset of Set 1
Set 4 is empty
Union of Set 1 and Set 2:
Movie: Movie 1, Rating: 7.5
Movie: Movie 2, Rating: 8.2
Movie: Movie 3, Rating: 6.9
Movie: Movie 4, Rating: 7.8

Intersection of Set 1 and Set 2:
Movie: Movie 2, Rating: 8.2
Movie: Movie 3, Rating: 6.9

Result of Set 1 - Set 2:
Movie: Movie 1, Rating: 7.5

Number of elements in Set 1: 3
```

So everything is running properly. Of course, there are additional test functions for other data types in the source code, so if you need them you can find them there.

GitLab link:

https://gitlab.com/school9862942/cps1011assignment