1 Problem 1

1.1 Overview of the Problem and Scenario

The main issue is to develop a C programme that can find out if a certain `n x n` symmetric matrix is definite. Using the signs of the determinants of every leading primary minors, the programme must determine whether the matrix is positive definite, negative definite, positive semidefinite, or negative semidefinite. Handling matrices up to `3 x 3` in real time is necessary for this, along with duties like matrix input, symmetry checking, calculating determinants, and determining definiteness. For the programme to be successfully implemented, each of these tasks is a subproblem.

1.2 Subproblem 1: Input and Creation of a Symmetric Matrix

The purpose of this subproblem is to randomly assign memory to a `n x n` symmetric matrix and fill it with values between 1 and 3, where `n` is user-supplied. A user is prompted to input the matrix size as the first step in the process. C uses dynamic allocation of matrices of suitable size using double pointers once the size is known. Following this, the user inputs values into each cell of the matrix. The user inputs each unique element just once and it is mirrored across the diagonal, simplifying the procedure, but only provided the matrix stays symmetric, which requires care to guarantee.

Approach. After verifying that the matrix size is within the permitted range of `3x3`, the programme should ask the user to enter the size of the matrix and subsequently each element of the matrix. The symmetry is maintained by automatically mirroring the lower triangle components and just entering the upper triangle elements (including the diagonal)—no need to input the bottom triangle elements. To do this, the matrix structure must be properly formed, which requires meticulous management of memory allocation and initialization.

```c
double **matrix = (double **)malloc(n * sizeof(double *));
for (int i = 0; i < n; i++) {
    matrix[i] = (double *)malloc(n * sizeof(double));
    for (int j = 0; j < n; j++) {
        printf("Enter element [%d][%d]: ", i + 1, j + 1);
        scanf("%lf", &matrix[i][j]);
    }
}
```

1.3 Subproblem 2: Check Matrix Symmetry

Checking whether the given matrix is symmetric is the objective here. For every `i` and `j`, element `(i, j)` should equal to element `(j, i)`, indicating that the matrix is symmetric if it is equal to its transpose. This is verified by iteratively comparing each element below the diagonal with its equivalent transposed element in a nested loop, as items above the diagonal are reflections of these. The function returns false (or 0 in C) if an element does not match its counterpart, which indicates that the matrix is not symmetric.

Approach. Make use of a looping function that compares each member below the diagonal of the matrix with its symmetrical counterpart above the diagonal. The matrix is considered non-symmetric if any of its elements do not match, and the user should be warned or further definiteness tests should be stopped by the programme.

```c
int isSymmetric(double **matrix, int n) {
    // Iterate over rows of the matrix
    for (int i = 0; i < n; i++) {
        // Only check elements below the diagonal (j < i)
        for (int j = 0; j < i; j++) {
            // Check if the element at (i, j) is not equal to the element at (j, i)
            if (matrix[i][j] != matrix[j][i]) {
                // If any element is not equal to its transpose counterpart, return 0 (not
                    symmetric)
                return 0;
            }
        }
    }
    // If all checked elements are equal to their transposed counterparts, return 1 (symmetric)
    return 1;
}
```

1.4 Subproblem 3: Check Matrix Definiteness

Finding out whether the matrix is positive definite, negative definite, positive semidefinite, or negative semidefinite is the goal of this subproblem. This is established by taking into consideration the signs of the determinants of all leading main minors, which are submatrices obtained from the matrix's top-left corner. The solution is to get the determinant of each leading primary minor by utilising the function from determinant calculation.

```
double determinant(double **matrix, int n) {
    double det = 0.0;
    if (n == 1) {
        det = matrix[0][0];
    }
    else if (n == 2) {
        det = matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
    }
    else if (n == 3) {
        det = matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1])
            - matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2] * matrix[2][0])
            + matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0]);
    }
    return det;
}
```

In order for there to be positive definiteness, each of these factors must be true. The determinants' signs must start with negative and alternate for negative definiteness to hold. Careful comparison of the signs of all determinants with these principles is an integral part of the procedure. Since dynamic allocation and deallocation are required for each minor to prevent memory leaks, efficient memory management is also critical in this case.

Approach. Create functions that successively verify the determinants of the leading main minors to determine positive definiteness, negative definiteness, and semidefiniteness. All determinants need to be positive for positive definiteness to hold. In order for these determinants to be negative definite, their signs must start with negative and then alternate. It is critical to methodically handle these tests and guarantee proper memory management throughout these activities.

```
int checkPositiveDefinite(double **matrix, int n) {
    // Loop through all sizes of leading principal submatrices from 1x1 up to nxn
    for (int i = 1; i <= n; i++) {
        // Allocate memory dynamically for the submatrix of dimension i x i
        double **subMatrix = (double **)malloc(i * sizeof(double *));
        for (int j = 0; j < i; j++) {
            subMatrix[j] = (double *)malloc(i * sizeof(double));

            // Copy the top-left i x i portion of the original matrix into subMatrix
            for (int k = 0; k < i; k++) {
                subMatrix[j][k] = matrix[j][k];
            }
        }

        // Calculate the determinant of the current i x i subMatrix
        double det = determinant(subMatrix, i);

        // Free the memory
        for (int j = 0; j < i; j++) free(subMatrix[j]);
        free(subMatrix);

        // Check if the determinant is non-positive. If so, the matrix is not positive definite
        if (det <= 0) return 0;
    }

    // If all determinants are positive, the matrix is positive definite, return 1
    return 1;
}
```

2 Question 2

2.1 Overview of the Problem and Scenario

The main challenge in this task is to find the root of the function $f(x) = \tan(x) - Kx$ where K is a parameter derived from physical constants of the system. Newton's method is used to iteratively converge on the root. Additionally, the program calculates a physical property (the dielectric constant, e_r), based on the found root. The overall program needs to handle input validation, unit conversions, tolerance handling, and iterative numerical computation.

2.2 Subproblem 1: Input Validation and Radian Conversion

The first subproblem focuses on capturing and validating user input to initiate the Newton's root-finding method effectively. An initial guess is pivotal for the convergence of Newton's method, and its accuracy directly influences the computational efficiency and reliability of the iterative process. Given that trigonometric calculations in the function $f(x)$ are involved, where $f(x) = \tan(x) - Kx$, the input degree must be converted into radians due to the mathematical requirements of the C programming language's standard library functions which operate in radians. Moreover, ensuring the input lies within a specified range (90 to 270 degrees) is critical to avoid asymptotic behaviors of the tangent function, which are undefined at $90°$ and $270°$ and their multiples.

The procedure consists of many consecutive phases, the first of which is asking the user to provide an angle in degrees. A while-loop checks the user's input for validity and requires them to re-enter data if their values are beyond the valid range of 90 to 270 degrees. Programme failure or incorrect output might be prevented from runtime issues during consecutive trigonometric evaluations by using this validation loop, which acts as a basic error management system.

Following successful input validation, the entered degree is converted into radians using the transformation formula:

$$\text{Radians} = \text{Degrees} \times \frac{\pi}{180}$$

Due to its role in laying the groundwork for later computational stages, this subproblem is fundamental. For subsequent stages to use Newton's technique with stability and convergence, accurate and correct beginning data are essential. The subproblem is significant in the entire algorithmic framework since errors at this early step might cause erroneous outputs or non-convergence when determining the root of $f(x)$. A small but crucial change is to change the units of measurement from degrees to radians. This makes the calculation space more accurate in representing the trigonometric features of the angle and complies with the mathematical requirements of the processing functions.

```c
#include <stdio.h>
#include <math.h> // Include for M_PI and trigonometric functions

// Function prototype for converting degrees to radians
double to_radians(double degrees);

int main() {
    double x0_degrees; // Variable to store the initial guess in degrees

    // Prompt user for the initial guess within the specific range
    printf("Initial Point in Degrees (Between 90 and 270): ");
    scanf("%lf", &x0_degrees);

    // Validate the input: it must be between 90 and 270 degrees
    while (x0_degrees <= 90 || x0_degrees >= 270) {
        printf("Invalid input. Please enter a degree between 90 and 270: ");
        scanf("%lf", &x0_degrees);
    }

    // Convert the validated degree input into radians
    double x0 = to_radians(x0_degrees);

    // The rest of the code to perform Newton's method would follow here...
    printf("Initial guess in radians: %f\n", x0); // This line is for demonstration

    return 0;
}

// Function to convert degrees to radians
double to_radians(double degrees) {
    return degrees * M_PI / 180.0;
}
```

2.3 Subproblem 2: Calculation of the Wave Number K

The calculation of the wave number K is pivotal in the context of Newton's method for root-finding, as K directly influences the functional form of $f(x) = \tan(x) - Kx$. K itself is derived from the physical parameters of the system, specifically the dimensions d and L, and the guide wavelength λ_g. Accurate computation of K is crucial because any error in its determination would propagate through to the calculations of $f(x)$ and thereby affect the accuracy and reliability of the root-finding process.

The wave number K is mathematically formulated as:

$$K = \frac{\tan\left(\frac{2\pi(d+L)}{\lambda_g}\right)}{2\pi\left(\frac{d}{\lambda_g}\right)}$$

This expression encapsulates the system's configuration by relating the geometric parameters d and L with the guide wavelength λ_g. The use of the tangent function hints at the periodic nature of the system, while the division by the product of 2π and a ratio emphasizes the scale adjustment based on the wavelength.

In the implemented C program, the function $K()$ is statically defined, encapsulating the computation of the wave number as a single, reusable function. This function is crucial for modularizing the code and ensuring that the wave number calculation can be independently verified and reused within the broader context of the program. The function does not take any arguments, as it utilizes global variables pre-defined in the code, thereby simplifying the function's interface and usage in $f(x)$.

```c
#include <math.h>

// Physical parameters as global constants
const double d = 0.5;      // Distance parameter in meters
const double L = 1.4;      // Additional length parameter in meters
const double lambda_g = 3.6; // Guide wavelength in meters

/**
 * Function to calculate the wave number K.
 * This function computes K based on the predefined physical constants of the system.
 * The use of the tan and pi functions requires including the math.h library.
 *
 * @return Computed value of K.
 */
double K() {
    return tan((2 * M_PI * (d + L)) / lambda_g) / (2 * M_PI * (d / lambda_g));
}
```

2.4 Subproblem 3: Implementation of Newton's Method

Newton's method, a cornerstone of numerical root-finding algorithms, is employed here to determine the root of the function $f(x) = \tan(x) - Kx$. Given its iterative nature, the method provides a potent approach to rapidly converge to a solution, provided an adequately close initial estimate and a suitable stopping criterion based on a tolerance threshold. This method's application is particularly apt for this problem due to the involvement of trigonometric functions, which are well-handled by Newton's iterative refinement.

Newton's method is mathematically represented as:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_n is the current estimate of the root, and $f'(x_n)$ is the derivative of $f(x)$ at x_n. The choice of $f(x)$ as $\tan(x) - Kx$ and its derivative $f'(x) = \sec^2(x) - K$ are critical as they dictate the behavior of the iterative process. The secant squared term, representing the derivative of the tangent function, introduces sensitivity to the input value, reflecting how slight deviations in x_n can significantly alter the outcome of subsequent iterations.

The implementation of Newton's method in the program is encapsulated in the function newtons_method(), which accepts the initial guess x_0, the tolerance for stopping, and a pointer to track the number of iterations. This function robustly applies the Newton-Raphson formula iteratively until the absolute difference between successive estimates x_n and x_{n+1} falls below the given tolerance. This approach not only ensures precision in finding the root but also provides insights into the algorithm's convergence characteristics through the iteration count. The dynamic interplay between the calculation of $f(x)$ and $f'(x)$ at each step forms the core computational loop, critically driving towards the algorithmic goal of minimizing root-finding errors.

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

// Prototype declarations for auxiliary functions
double f(double x);      // Function whose root is to be found
double f_prime(double x); // Derivative of the function

/**
 * Implementation of Newton's Method to find the root of a function.
 *
 * @param x0 Initial guess for the root.
 * @param tol Tolerance for the stopping criterion.
 * @param iterations Pointer to an integer to store the number of iterations performed.
 * @return The estimated root of the function.
 */
double newtons_method(double x0, double tol, int* iterations) {
    double x1;
    *iterations = 0; // Initialize the iteration counter

    do {
        double f_val = f(x0);
        double f_prime_val = f_prime(x0);
        if (fabs(f_prime_val) < 1e-10) { // Check to avoid division by zero
            fprintf(stderr, "Error: Derivative is too small, potential division by zero.\n");
            exit(EXIT_FAILURE);
        }

        x1 = x0 - f_val / f_prime_val;
        (*iterations)++; // Increment the iteration counter

        if (fabs(x1 - x0) < tol) { // Check if the current estimate is within the desired
             tolerance
            break;
        }
        x0 = x1; // Update the estimate for the next iteration
    } while (1);

    return x1; // Return the converged root estimate
}
```