1. Flight Trajectory Calculation

- Pointers: Use to traverse the trajectory array.
- Arrays: Store trajectory points (x, y, z) at discrete time intervals.
- Functions:
  - void calculate_trajectory(const double *parameters, double *trajectory, int size): Takes the initial velocity, angle, and an array to store trajectory points.
  - void print_trajectory(const double *trajectory, int size): Prints the stored trajectory points.
- Pass Arrays as Pointers: Pass the trajectory array as a pointer to the calculation function.

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

#define GRAVITY 9.8
#define PI 3.141

// Function prototypes
void calculate_trajectory(const double *parameters, double *trajectory, int size);
void print_trajectory(const double *trajectory, int size);

int main()
{
    double parameters[3];
    int size = 100;
```

```
    double trajectory[3 * size];

    // Input the initial velocity, angle, and time interval
    printf("Enter the initial velocity (m/s): ");
    scanf("%lf", &parameters[0]);

    double angle_degrees;
    printf("Enter the angle of projection (degrees): ");
    scanf("%lf", &angle_degrees);
    parameters[1] = angle_degrees * PI / 180.0;

    printf("Enter the time interval between points (seconds): ");
    scanf("%lf", &parameters[2]);

    // Call the function to calculate trajectory
    calculate_trajectory(parameters, trajectory, size);

    // Call the function to print trajectory
    print_trajectory(trajectory, size);

    return 0;
}

/*
Name: calculate_trajectory()
Return Type: void
Parameter:(data type of each parameter): const double*, double* and int
```

Short description: it is used to calculate the trajectory

*/

```
// Function to calculate the trajectory
void calculate_trajectory(const double *parameters, double *trajectory, int size)
{
    double initial_velocity = parameters[0];
    double angle_rad = parameters[1];
    double time_interval = parameters[2];

    for (int i = 0; i < size; i++)
    {
        double time = i * time_interval;
        double x = initial_velocity * cos(angle_rad) * time;
        double y = (initial_velocity * sin(angle_rad) * time) - (0.5 * GRAVITY * time * time);
        trajectory[3 * i] = x;
        trajectory[3 * i + 1] = y;
        trajectory[3 * i + 2] = time;
        if (y < 0)
            break;
    }
}
```

/*

Name: print_trajectory()

Return Type: void

Parameter:(data type of each parameter): const double* and int

Short description: it is used to print the trajectory

*/

```
// Function to print the trajectory
void print_trajectory(const double *trajectory, int size)
{
   printf("Trajectory Points\n");
   printf("Time (s)\tX (m)\t\tY (m)\n");
   for (int i = 0; i < size; i++)
   {
      if (trajectory[3 * i + 1] < 0)  // Stop printing if Y is below ground
         break;
      printf("%.2f\t\t%.2f\t\t%.2f\n", trajectory[3 * i + 2], trajectory[3 * i],
trajectory[3 * i + 1]);
   }
}
```

O/P:

```
Enter the initial velocity (m/s): 60
Enter the angle of projection (degrees): 90
Enter the time interval between points (seconds): 5
Trajectory Points
Time (s)      X (m)         Y (m)
0.00          0.00          0.00
5.00          0.09          177.50
10.00         0.18          110.00
```

2. Satellite Orbit Simulation

- Pointers: Manipulate position and velocity vectors.
- Arrays: Represent the satellite's position over time as an array of 3D vectors.
- Functions:
    - void update_position(const double *velocity, double *position, int size): Updates the position based on velocity.
    - void simulate_orbit(const double *initial_conditions, double *positions, int steps): Simulates orbit over a specified number of steps.
- Pass Arrays as Pointers: Use pointers for both velocity and position arrays.

```
#include <stdio.h>

// Function prototypes
void update_position(const double *velocity, double *position, int size);
void simulate_orbit(const double *initial_conditions, double *positions, int steps);

int main()
{
    int steps;

    // Input the number of simulation steps
    printf("Enter the number of simulation steps: ");
    scanf("%d", &steps);

    if (steps <= 0)
    {
```

```
        printf("Invalid number of steps\n");
        return 1;
    }

    double initial_conditions[6];
    double positions[steps * 3];

    // Input the initial position and velocity
    printf("Enter initial position (x, y, z in km): ");
    scanf("%lf %lf %lf", &initial_conditions[0], &initial_conditions[1],
&initial_conditions[2]);

    printf("Enter initial velocity (vx, vy, vz in km/s): ");
    scanf("%lf %lf %lf", &initial_conditions[3], &initial_conditions[4],
&initial_conditions[5]);

    // Call the function to simulate orbit
    simulate_orbit(initial_conditions, positions, steps);

    printf("Step\tX\t\tY\t\tZ\n");
    for (int step = 0; step < steps; step++)
    {
        printf("%d\t%.2f\t\t%.2f\t\t%.2f\n", step, positions[step * 3],
               positions[step * 3 + 1], positions[step * 3 + 2]);
    }

    return 0;
}
```

```
/*
Name: update_position()
Return Type: void
Parameter:(data type of each parameter): const double*, double* and int
Short description: it is used to update the position based on velocity
*/

// Function to update position based on velocity
void update_position(const double *velocity, double *position, int size)
{
    for (int i = 0; i < size; i++)
        position[i] += velocity[i];
}

/*
Name: simulate_orbit()
Return Type: void
Parameter:(data type of each parameter): const double*, double* and int
Short description: it is used to simulate orbit over a specified number of steps
*/

// Function to simulate orbit over a specified number of steps
void simulate_orbit(const double *initial_conditions, double *positions, int steps)
{
```

```
    double position[3] = {initial_conditions[0], initial_conditions[1], initial_conditions[2]};
    double velocity[3] = {initial_conditions[3], initial_conditions[4], initial_conditions[5]};

  for (int step = 0; step < steps; step++) // Store current position
  {
     positions[step * 3] = position[0];
     positions[step * 3 + 1] = position[1];
     positions[step * 3 + 2] = position[2];

     // Update position based on velocity
     update_position(velocity, position, 3);
  }
}
```

O/P:

```
Enter the number of simulation steps: 5
Enter initial position (x, y, z in km): 100 200 100
Enter initial velocity (vx, vy, vz in km/s): 20 50 30
Step  X        Y        Z
0     100.00   200.00   100.00
1     120.00   250.00   130.00
2     140.00   300.00   160.00
3     160.00   350.00   190.00
4     180.00   400.00   220.00
```

3. Weather Data Processing for Aviation

- Pointers: Traverse weather data arrays efficiently.
- Arrays: Store hourly temperature, wind speed, and pressure.
- Functions:
  - void calculate_daily_averages(const double *data, int size, double *averages): Computes daily averages for each parameter.
  - void display_weather_data(const double *data, int size): Displays data for monitoring purposes.
- Pass Arrays as Pointers: Pass weather data as pointers to processing functions.

```
#include <stdio.h>

// Function prototypes
void calculate_daily_averages(const double *data, int size, double *averages);
void display_weather_data(const double *data, int size);

int main()
{
    int hours = 4; // Consider for output purpose else hours = 24
    double weather_data[4 * 3];
    double daily_averages[3];

    // Input the weather data for 24 hours
    printf("Enter hourly weather data (Temperature °C, Wind Speed km/h, Pressure hPa)\n");
    for (int i = 0; i < hours; i++)
    {
```

```
        printf("Hour %d: ", i + 1);
        scanf("%lf %lf %lf", &weather_data[i * 3], &weather_data[i * 3 + 1],
&weather_data[i * 3 + 2]);
    }

    // Call the function to display weather data
    display_weather_data(weather_data, hours * 3);

    // Call the function to calculate daily averages
    calculate_daily_averages(weather_data, hours * 3, daily_averages);

    // Output the daily averages
    printf("\nDaily Averages\n");
    printf("Temperature: %.2f °C\n", daily_averages[0]);
    printf("Wind Speed: %.2f km/h\n", daily_averages[1]);
    printf("Pressure: %.2f hPa\n", daily_averages[2]);

    return 0;
}

/*
Name: calculate_daily_averages()
Return Type: void
Parameter:(data type of each parameter): const double*, int and double*
Short description: it is used to calculate daily averages for each parameter
*/

// Function to calculate daily averages for each parameter
```

```
void calculate_daily_averages(const double *data, int size, double *averages)
{
    for (int i = 0; i < 3; i++)
    {
        averages[i] = 0;
        for (int j = 0; j < size / 3; j++)
            averages[i] += data[j * 3 + i];

        averages[i] /= (size / 3);
    }
}

/*
Name: display_weather_data()
Return Type: void
Parameter:(data type of each parameter): const double* and int
Short description: it is used to display the data for monitoring purpose
*/

// Function to display weather data for monitoring purposes
void display_weather_data(const double *data, int size)
{
    printf("Hourly Weather Data\n");
        printf("Hour\tTemperature (°C)\tWind Speed (km/h)\tPressure (hPa)\n");
    for (int i = 0; i < size / 3; i++)
```

```
    printf("%d\t%.2f\t\t\t%.2f\t\t\t%.2f\n", i + 1, data[i * 3], data[i * 3 + 1], data[i * 3 + 2]);
}
```

O/P:

```
Enter hourly weather data (Temperature °C, Wind Speed km/h, Pressure hPa)
Hour 1: 27 11 1000
Hour 2: 32 8 950
Hour 3: 29 10 1125
Hour 4: 30 9 1500
Hourly Weather Data
Hour    Temperature (°C)    Wind Speed (km/h)    Pressure (hPa)
1       27.00               11.00                1000.00
2       32.00               8.00                 950.00
3       29.00               10.00                1125.00
4       30.00               9.00                 1500.00

Daily Averages
Temperature: 29.50 °C
Wind Speed: 9.50 km/h
Pressure: 1143.75 hPa
```

4. Flight Control System (PID Controller)

- Pointers: Traverse and manipulate error values in arrays.
- Arrays: Store historical error values for proportional, integral, and derivative calculations.

- Functions:
    - double compute_pid(const double *errors, int size, const double *gains): Calculates control output using PID logic.
    - void update_errors(double *errors, double new_error): Updates the error array with the latest value.
- Pass Arrays as Pointers: Use pointers for the errors array and the gains array.

```
#include <stdio.h>


// Function prototypes
double compute_pid(const double *errors, int size, const double *gains);
void update_errors(double *errors, double new_error);


int main()
{
   int size = 5;
   double errors[5] = {0};
   double gains[3]; // PID gains: Kp, Ki, Kd
   double new_error;
   double control_output;

   // Input the PID gains
   printf("Enter PID gains (Kp Ki Kd): ");
   scanf("%lf %lf %lf", &gains[0], &gains[1], &gains[2]);

   printf("***Enter -1 to stop the simulation***\n");
```

```
    while (1)
    {
        // Input the latest error
        printf("Enter the new error value: ");
        scanf("%lf", &new_error);

        if (new_error == -1)
            break;

        // Call the function to update error array
        update_errors(errors, new_error);

        // Call the function to compute PID output
        control_output = compute_pid(errors, size, gains);

        printf("Control Output: %.2f\n", control_output);
    }

    return 0;
}

/*
Name: compute_pid()
Return Type: double
Parameter:(data type of each parameter): const double*, int and double*
Short description: it is used to calculate control output using PID logic
*/
```

```
// Function to compute the PID control output
double compute_pid(const double *errors, int size, const double *gains)
{
    double proportional = errors[size - 1];
    double integral = 0.0;
    double derivative = 0.0;

    // Calculate integral
    for (int i = 0; i < size; i++)
        integral += errors[i];

    // Calculate derivative
    if (size > 1)
        derivative = errors[size - 1] - errors[size - 2];

    // PID formula: Kp * P + Ki * I + Kd * D
     return gains[0] * proportional + gains[1] * integral + gains[2] *
derivative;
}

/*
Name: update_errors()
Return Type: void
Parameter:(data type of each parameter): double* and double
Short description: it is used to update the error array with the latest value
*/
```

```
// Function to update the error array with the latest error value
void update_errors(double *errors, double new_error)
{
    int size = 5;
    // Shift errors to the left
    for (int i = 0; i < size - 1; i++)
        errors[i] = errors[i + 1];

    // Add the new error at the end
    errors[size - 1] = new_error;
}
```

O/P:

```
Enter PID gains (Kp Ki Kd): 1 1 1
***Enter -1 to stop the simulation***
Enter the new error value: 5
Control Output: 15.00
Enter the new error value: 4
Control Output: 12.00
Enter the new error value: 8
Control Output: 29.00
Enter the new error value: 9
Control Output: 36.00
Enter the new error value: 7
Control Output: 38.00
Enter the new error value: -1
```

5. Aircraft Sensor Data Fusion

- Pointers: Handle sensor readings and fusion results.
- Arrays: Store data from multiple sensors.
- Functions:
  - void fuse_data(const double *sensor1, const double *sensor2, double *result, int size): Merges two sensor datasets into a single result array.
  - void calibrate_data(double *data, int size): Adjusts sensor readings based on calibration data.
- Pass Arrays as Pointers: Pass sensor arrays as pointers to fusion and calibration functions.

```
#include <stdio.h>


// Function prototypes

void fuse_data(const double *sensor1, const double *sensor2, double *result, int size);

void calibrate_data(double *data, int size);


int main()

{

    int size;

    printf("Enter the number of sensor readings: ");

    scanf("%d", &size);


    double sensor1[size], sensor2[size], fused_result[size];


    // Input sensor data for sensor1 and sensor2

    printf("Enter readings for Sensor 1:\n");
```

```
for (int i = 0; i < size; i++)
{
    printf("Reading %d: ", i + 1);
    scanf("%lf", &sensor1[i]);
}

printf("Enter readings for Sensor 2:\n");
for (int i = 0; i < size; i++)
{
    printf("Reading %d: ", i + 1);
    scanf("%lf", &sensor2[i]);
}

// Call the function to fuse sensor data
fuse_data(sensor1, sensor2, fused_result, size);

printf("\nBefore Calibration: Fused Sensor Data:\n");
for (int i = 0; i < size; i++)
    printf("Data Point %d: %.2f\n", i + 1, fused_result[i]);

// Call the function to calibrate the fused data
calibrate_data(fused_result, size);

printf("\nAfter Calibration: Fused Sensor Data:\n");
for (int i = 0; i < size; i++)
    printf("Data Point %d: %.2f\n", i + 1, fused_result[i]);
return 0;
```

}

/*

Name: fuse_data()

Return Type: void

Parameter:(data type of each parameter): const double*, const double*, double* and int

Short description: it is used to merge two sensor datasets into single result array

*/

// Function to merge two sensor datasets into a single result array

void fuse_data(const double *sensor1, const double *sensor2, double *result, int size)

{

for (int i = 0; i < size; i++)

// Fusion logic: average the readings from both sensors

result[i] = (sensor1[i] + sensor2[i]) / 2.0;

}

/*

Name: calibrate_data()

Return Type: void

Parameter:(data type of each parameter): double* and int

Short description: it is used to adjust sensor readings based on calibration functions

*/

// Function to calibrate sensor readings based on a calibration factor

```
void calibrate_data(double *data, int size)
{
    double calibration_factor;
    printf("Enter calibration factor: ");
    scanf("%lf", &calibration_factor);

    for (int i = 0; i < size; i++)
        data[i] *= calibration_factor; // Adjust the data by the calibration factor
}
```

O/P:

```
Enter the number of sensor readings: 2
Enter readings for Sensor 1:
Reading 1: 18
Reading 2: 20
Enter readings for Sensor 2:
Reading 1: 15
Reading 2: 8

Before Calibration: Fused Sensor Data:
Data Point 1: 16.50
Data Point 2: 14.00
Enter calibration factor: 2.5

After Calibration: Fused Sensor Data:
Data Point 1: 41.25
Data Point 2: 35.00
```

6. Air Traffic Management

- Pointers: Traverse the array of flight structures.
- Arrays: Store details of active flights (e.g., ID, altitude, coordinates).
- Functions:
  - void add_flight(flight_t *flights, int *flight_count, const flight_t *new_flight): Adds a new flight to the system.
  - void remove_flight(flight_t *flights, int *flight_count, int flight_id): Removes a flight by ID.
- Pass Arrays as Pointers: Use pointers to manipulate the array of flight structures.

```
#include <stdio.h>

#include <string.h>


#define MAX_FLIGHTS 10


// Define the flight structure

typedef struct

{

    int flight_id;

    int altitude;

    float latitude;

    float longitude;

} flight_t;


// Function prototypes

void add_flight(flight_t *flights, int *flight_count, const flight_t *new_flight);

void remove_flight(flight_t *flights, int *flight_count, int flight_id);
```

```
int main()
{
    flight_t flights[MAX_FLIGHTS];
    int flight_count = 0;

    // Call the function to add some flights
    flight_t new_flight = {101, 35000.0, 40.7128, -74.0060}; // New York
    add_flight(flights, &flight_count, &new_flight);

     new_flight = (flight_t){102, 38000.0, 34.0522, -118.2437}; // Los
Angeles
    add_flight(flights, &flight_count, &new_flight);

    printf("Active Flights:\n");
    if (flight_count == 0)
        printf("No active flights.\n");
    else
    {
        for (int i = 0; i < flight_count; i++)
        {
            printf("Flight ID: %d\n", flights[i].flight_id);
            printf("Altitude: %d\n", flights[i].altitude);
                    printf("Coordinates: (%.2f, %.2f)\n", flights[i].latitude,
flights[i].longitude);
            printf("-------------------------\n");
        }
    }
```

```c
    // Call the function to remove a flight
    remove_flight(flights, &flight_count, 101);

    printf("Active Flights after removal:\n");
    if (flight_count == 0)
        printf("No active flights.\n");
    else
    {
        for (int i = 0; i < flight_count; i++)
        {
            printf("Flight ID: %d\n", flights[i].flight_id);
            printf("Altitude: %d\n", flights[i].altitude);
            printf("Coordinates: (%.2f, %.2f)\n", flights[i].latitude, flights[i].longitude);
            printf("------------------------\n");
        }
    }

    return 0;
}

/*
Name: add_flight()
Return Type: void
Parameter:(data type of each parameter): flight_t*, int* and const flight_t*
Short description: it is used to add new flight to the system
*/
```

```
// Function to add a new flight
void add_flight(flight_t *flights, int *flight_count, const flight_t *new_flight)
{
   if (*flight_count >= MAX_FLIGHTS)
   {
      printf("Cannot add new flight\n");
      return;
   }
   flights[*flight_count] = *new_flight;
   (*flight_count)++;
   printf("Flight ID %d added successfully\n", new_flight->flight_id);
}

/*
Name: remove_flight()
Return Type: void
Parameter:(data type of each parameter): flight_t*, int* and int
Short description: it is used to remove a flight by ID
*/

// Function to remove a flight by ID
void remove_flight(flight_t *flights, int *flight_count, int flight_id)
{
   for (int i = 0; i < *flight_count; i++)
   {
      if (flights[i].flight_id == flight_id)
      {
```

```
            for (int j = i; j < *flight_count - 1; j++)
                flights[j] = flights[j + 1];
            (*flight_count)--;
            return;
        }
    }
    printf("Flight with ID %d not found.\n", flight_id);
}
```

O/P:

```
Flight ID 101 added successfully
Flight ID 102 added successfully
Active Flights:
Flight ID: 101
Altitude: 35000
Coordinates: (40.71, -74.01)
-------------------------
Flight ID: 102
Altitude: 38000
Coordinates: (34.05, -118.24)
-------------------------
Active Flights after removal:
Flight ID: 102
Altitude: 38000
Coordinates: (34.05, -118.24)
-------------------------
```

7. Satellite Telemetry Analysis

- Pointers: Traverse telemetry data arrays.
- Arrays: Store telemetry parameters (e.g., power, temperature, voltage).
- Functions:
  - void analyze_telemetry(const double *data, int size): Computes statistical metrics for telemetry data.
  - void filter_outliers(double *data, int size): Removes outliers from the telemetry data array.
- Pass Arrays as Pointers: Pass telemetry data arrays to both functions.

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

// Function prototypes
void analyze_telemetry(const double *data, int size);
void filter_outliers(double *data, int *size);

int main()
{
    double telemetry_data[] = {10.9, 18.8, 28.6, 15.8, 35.4, 16.4, 37.0, 86.1, 49.6, 72.6};
    int size = sizeof(telemetry_data) / sizeof(telemetry_data[0]);

    printf("Original telemetry data:\n");
    for (int i = 0; i < size; i++)
        printf("%.2f ", telemetry_data[i]);
    printf("\n");
```

```
    // Call the function to analyze telemetry data
    analyze_telemetry(telemetry_data, size);

    // Call the function to filter outliers and update telemetry data
    filter_outliers(telemetry_data, &size);

    printf("Telemetry data after filter outliers:\n");
    for (int i = 0; i < size; i++)
        printf("%.2f ", telemetry_data[i]);
    printf("\n");

    // Call the function again to analyze filtered telemetry data
    analyze_telemetry(telemetry_data, size);
    return 0;
}

/*
Name: analyze_telemetry()
Return Type: void
Parameter:(data type of each parameter): const double* and int
Short description: it is used to compute the statistical metrics
*/

// Function to compute statistical metrics for telemetry data
void analyze_telemetry(const double *data, int size)
{
    if (size <= 0)
```

```
    {
        printf("No data to analyze\n");
        return;
    }

    double sum = 0.0, mean, variance = 0.0, stddev;

    // Calculate sum for mean calculation
    for (int i = 0; i < size; i++)
        sum += data[i];
    mean = sum / size;

    // Calculate variance and standard deviation
    for (int i = 0; i < size; i++)
        variance += pow(data[i] - mean, 2);
    variance /= size;
    stddev = sqrt(variance);

    // Print statistical metrics
    printf("Analysis of telemetry data\n");
    printf("Mean: %.2f\n", mean);
    printf("Variance: %.2f\n", variance);
    printf("Standard Deviation: %.2f\n", stddev);
}

/*
Name: filter_outliers()
```

Return Type: void

Parameter:(data type of each parameter): double* and int

Short description: it is used to remove outliers from telemetry data

*/

```
// Function to filter outliers from the telemetry data array
void filter_outliers(double *data, int *size)
{
    if (*size <= 0)
    {
        printf("No data to filter\n");
        return;
    }
    double sum = 0.0, mean, variance = 0.0, stddev;

    // Calculate sum for mean calculation
    for (int i = 0; i < *size; i++)
        sum += data[i];
    mean = sum / *size;

    // Calculate variance and standard deviation
    for (int i = 0; i < *size; i++)
        variance += pow(data[i] - mean, 2);
    variance /= *size;
    stddev = sqrt(variance);

    // Define outlier threshold (2 standard deviations)
```

```
    double threshold = 2.0 * stddev;

    int new_size = 0;
    // Remove outliers
    for (int i = 0; i < *size; i++)
    {
        if (fabs(data[i] - mean) <= threshold)
            data[new_size++] = data[i];
    }
    *size = new_size;  // Update the size after removing outliers
    printf("New data size after removal of outliers: %d\n", *size);
}
```

O/P:

```
Original telemetry data:
10.90 18.80 28.60 15.80 35.40 16.40 37.00 86.10 49.60 72.60
Analysis of telemetry data
Mean: 37.12
Variance: 579.62
Standard Deviation: 24.08
New data size after removal of outliers: 9
Telemetry data after filter outliers:
10.90 18.80 28.60 15.80 35.40 16.40 37.00 49.60 72.60
Analysis of telemetry data
Mean: 31.68
Variance: 347.84
Standard Deviation: 18.65
```

8. Rocket Thrust Calculation

- Pointers: Traverse thrust arrays.
- Arrays: Store thrust values for each stage of the rocket.
- Functions:
  - double compute_total_thrust(const double *stages, int size): Calculates cumulative thrust across all stages.
  - void update_stage_thrust(double *stages, int stage, double new_thrust): Updates thrust for a specific stage.
- Pass Arrays as Pointers: Use pointers for thrust arrays.

```
#include <stdio.h>

// Function prototypes
double compute_total_thrust(const double *stages, int size);
void update_stage_thrust(double *stages, int stage, double new_thrust);

int main()
{
    double thrust_values[] = {1880.0, 815.0, 900.0};
    int num_stages = sizeof(thrust_values) / sizeof(thrust_values[0]);

    printf("Initial thrust values for each stage:\n");
    for (int i = 0; i < num_stages; i++)
        printf("Stage %d: %.2f N\n", i + 1, thrust_values[i]);

    // Compute the total thrust across all stages
    double total_thrust = compute_total_thrust(thrust_values, num_stages);
    printf("Total thrust of the rocket: %.2f N\n", total_thrust);
```

```c
    // Call the function to update the thrust of the second stage
    update_stage_thrust(thrust_values, 1, 580.0);

    printf("Updated thrust values for each stage:\n");
    for (int i = 0; i < num_stages; i++)
        printf("Stage %d: %.2f N\n", i + 1, thrust_values[i]);

    // Recompute the total thrust after the update
    total_thrust = compute_total_thrust(thrust_values, num_stages);
    printf("Updated total thrust of the rocket: %.2f N\n", total_thrust);

    return 0;
}

/*
Name: compute_total_thrust()
Return Type: double
Parameter:(data type of each parameter): const double* and int
Short description: it is used to compute the total thrust across all stages
*/

// Function to compute the total thrust across all stages
double compute_total_thrust(const double *stages, int size)
{
    double total_thrust = 0.0;
    for (int i = 0; i < size; i++)
```

```
        total_thrust += stages[i];
    return total_thrust;
}

/*
Name: update_stage_thrust()
Return Type: void
Parameter:(data type of each parameter): double*, int amd double
Short description: it is used to update the thrust of a specific stage
*/

// Function to update the thrust of a specific stage
void update_stage_thrust(double *stages, int stage, double new_thrust)
{
    if (stage >= 0)
    {
        stages[stage] = new_thrust;
        printf("Updated thrust for stage %d: %.2f\n", stage + 1, new_thrust);
    }
    else
        printf("Invalid thrust stage number\n");
}
```

O/P:

```
Initial thrust values for each stage:
Stage 1: 1880.00 N
Stage 2: 815.00 N
```

Stage 3: 900.00 N

Total thrust of the rocket: 3595.00 N

Updated thrust for stage 2: 580.00

Updated thrust values for each stage:

Stage 1: 1880.00 N

Stage 2: 580.00 N

Stage 3: 900.00 N

Updated total thrust of the rocket: 3360.00 N

9. Wing Stress Analysis

- Pointers: Access stress values at various points.
- Arrays: Store stress values for discrete wing sections.
- Functions:
  - void compute_stress_distribution(const double *forces, double *stress, int size): Computes stress values based on applied forces.
  - void display_stress(const double *stress, int size): Displays the stress distribution.
- Pass Arrays as Pointers: Pass stress arrays to computation functions.

```
#include <stdio.h>

// Function prototypes
void compute_stress_distribution(const double *forces, double *stress, int size);
void display_stress(const double *stress, int size);

int main()
```

```
{
    double applied_forces[] = {100.0, 300.0, 500.0, 750.0, 450.0};
    int num_sections = sizeof(applied_forces) / sizeof(applied_forces[0]);
    double stress_distribution[num_sections];

    // Call the function to compute the stress distribution based on applied forces
    compute_stress_distribution(applied_forces, stress_distribution, num_sections);

    // Call the function to display the computed stress distribution
    display_stress(stress_distribution, num_sections);
    return 0;
}

/*
Name: compute_stress_distribution()
Return Type: void
Parameter:(data type of each parameter): const double*, double* and int
Short description: it is used to compute stress distribution based on applied forces
*/

// Function to compute stress distribution based on applied forces
void compute_stress_distribution(const double *forces, double *stress, int size) {
    for (int i = 0; i < size; i++)
    {
        // stress = force / area
```

```c
        double area = 20.0;
        stress[i] = forces[i] / area;
    }
}

/*
Name: display_stress()
Return Type: void
Parameter:(data type of each parameter): const double* and int
Short description: it is used to display the stress distribution for each section
*/

// Function to display the stress distribution for each section
void display_stress(const double *stress, int size)
{
    printf("Stress distribution across wing sections:\n");
    for (int i = 0; i < size; i++)
        printf("Section %d: %.2f Pa\n", i + 1, stress[i]);
}
```

O/P:

```
Stress distribution across wing sections:
Section 1: 5.00 Pa
Section 2: 15.00 Pa
Section 3: 25.00 Pa
Section 4: 37.50 Pa
Section 5: 22.50 Pa
```

10. Drone Path Optimization

- Pointers: Traverse waypoint arrays.
- Arrays: Store coordinates of waypoints.
- Functions:
    - double optimize_path(const double *waypoints, int size): Reduces the total path length.
    - void add_waypoint(double *waypoints, int *size, double x, double y): Adds a new waypoint.
- Pass Arrays as Pointers: Use pointers to access and modify waypoints.

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

// Function prototypes
double optimize_path(const double *waypoints, int size);
void add_waypoint(double *waypoints, int *size, double x, double y);

int main()
{
    double waypoints[20];
    int size = 0;

    // Call the function to add some initial waypoints
    add_waypoint(waypoints, &size, 0.0, 0.0);
    add_waypoint(waypoints, &size, 4.0, 5.0);
    add_waypoint(waypoints, &size, 8.0, 9.0);

    printf("Waypoints:\n");
```

```
    for (int i = 0; i < size; i++)
        printf("Waypoint %d: (%.2f, %.2f)\n", i + 1, waypoints[2 * i], waypoints[2 * i + 1]);

    // Call the function to compute the optimized path length
    double total_distance = optimize_path(waypoints, size);
    printf("Total path length: %.2f units\n", total_distance);

    // Call the function again to add a new waypoint and recompute the path length
    add_waypoint(waypoints, &size, 5.0, 7.0);

    printf("Waypoints:\n");
    for (int i = 0; i < size; i++)
        printf("Waypoint %d: (%.2f, %.2f)\n", i + 1, waypoints[2 * i], waypoints[2 * i + 1]);

    // Recompute the optimized path length after adding the new waypoint
    total_distance = optimize_path(waypoints, size);
    printf("Updated total path length: %.2f units\n", total_distance);
    return 0;
}

/*
Name: add_waypoint()
Return Type: void
Parameter:(data type of each parameter): double*, int*, double and double
Short description: it is used to add a new waypoint to the array of waypoints
```

```
*/

// Function to add a new waypoint to the array of waypoints
void add_waypoint(double *waypoints, int *size, double x, double y)
{
    waypoints[2 * (*size)] = x;
    waypoints[2 * (*size) + 1] = y;
    (*size)++;
    printf("Waypoint added: (%.2f, %.2f)\n", x, y);
}

/*
Name: optimize_path()
Return Type: double
Parameter:(data type of each parameter): const double* and int
Short description: it is used to compute the total path length for the waypoints
*/

// Function to compute the total path length for the waypoints
double optimize_path(const double *waypoints, int size)
{
    double total_distance = 0.0;

    // Traverse the array of waypoints and compute the distance between consecutive points
    for (int i = 0; i < size - 1; i++)
    {
```

```
        double x1 = waypoints[2 * i];
        double y1 = waypoints[2 * i + 1];
        double x2 = waypoints[2 * (i + 1)];
        double y2 = waypoints[2 * (i + 1) + 1];

        // Calculate the distance between (x1, y1) and (x2, y2)
        total_distance += sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
    }
    return total_distance;
}
```

O/P:

```
Waypoint added: (0.00, 0.00)
Waypoint added: (4.00, 5.00)
Waypoint added: (8.00, 9.00)
Waypoints:
Waypoint 1: (0.00, 0.00)
Waypoint 2: (4.00, 5.00)
Waypoint 3: (8.00, 9.00)
Total path length: 12.06 units
Waypoint added: (5.00, 7.00)
Waypoints:
Waypoint 1: (0.00, 0.00)
Waypoint 2: (4.00, 5.00)
Waypoint 3: (8.00, 9.00)
Waypoint 4: (5.00, 7.00)
Updated total path length: 15.67 units
```

11. Satellite Attitude Control

- Pointers: Manipulate quaternion arrays.
- Arrays: Store quaternion values for attitude control.
- Functions:
    - void update_attitude(const double *quaternion, double *new_attitude): Updates the satellite's attitude.
    - void normalize_quaternion(double *quaternion): Ensures quaternion normalization.
- Pass Arrays as Pointers: Pass quaternion arrays as pointers.

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

// Function prototypes
void update_attitude(const double *quaternion, double *new_attitude);
void normalize_quaternion(double *quaternion);

int main()
{
    // Example quaternion representing the attitude (w, x, y, z)
    double quaternion[4] = {0.1580, 0.8180, 2.5, 1.1};

    // Print initial quaternion
     printf("Initial quaternion: (%.2f, %.2f, %.2f, %.2f)\n", quaternion[0],
quaternion[1], quaternion[2], quaternion[3]);

    // Call the function to normalize the quaternion
    normalize_quaternion(quaternion);
```

```
    // Call the function to update the satellite attitude based on the normalized quaternion
    double new_attitude[4];
    update_attitude(quaternion, new_attitude);

    printf("Updated attitude: (%.2f, %.2f, %.2f, %.2f)\n", new_attitude[0], new_attitude[1], new_attitude[2], new_attitude[3]);

    return 0;
}

/*
Name: normalize_quaternion()
Return Type: void
Parameter:(data type of each parameter): double*
Short description: it is used to normalize the quaternion
*/

// Function to normalize the quaternion
void normalize_quaternion(double *quaternion)
{
    double norm = 0.0;
    for (int i = 0; i < 4; i++)
        norm += quaternion[i] * quaternion[i];

    norm = sqrt(norm);

    // Normalize the quaternion by dividing each component by the norm
```

```
    if (norm > 0.0)
    {
        for (int i = 0; i < 4; i++)
            quaternion[i] /= norm;
    }
        printf("Quaternion normalized to: (%.2f, %.2f, %.2f, %.2f)\n", quaternion[0], quaternion[1], quaternion[2], quaternion[3]);
}


/*
Name: update_attitude()
Return Type: void
Parameter:(data type of each parameter): const double* and double*
Short description: it is used to update the satellite's attitude based on a quaternion
*/


// Function to update the satellite's attitude based on a quaternion
void update_attitude(const double *quaternion, double *new_attitude)
{
    for (int i = 0; i < 4; i++)
        new_attitude[i] = quaternion[i];
}
```

O/P:

```
Initial quaternion: (0.16, 0.82, 2.50, 1.10)
Quaternion normalized to: (0.06, 0.29, 0.88, 0.39)
Updated attitude: (0.06, 0.29, 0.88, 0.39)
```

12. Aerospace Material Thermal Analysis

- Pointers: Access temperature arrays for computation.
- Arrays: Store temperature values at discrete points.
- Functions:
  - void simulate_heat_transfer(const double *material_properties, double *temperatures, int size): Simulates heat transfer across the material.
  - void display_temperatures(const double *temperatures, int size): Outputs temperature distribution.
- Pass Arrays as Pointers: Use pointers for temperature arrays.

```
#include <stdio.h>

//Function prototypes
void simulate_heat_transfer(const double *material_properties, double *temperatures, int size);
void display_temperatures(const double *temperatures, int size);

int main()
{

    double material_properties[] = {1.5};

    double temperatures[] = {10.0, 20.0, 30.0, 40.0, 50.0};
    int size = sizeof(temperatures) / sizeof(temperatures[0]);

    // Call the function to display initial temperature distribution
    display_temperatures(temperatures, size);
```

```
    // Call the function to simulate heat transfer
    simulate_heat_transfer(material_properties, temperatures, size);

    // Call the function again to display updated temperature distribution after heat transfer simulation
    display_temperatures(temperatures, size);

    return 0;
}

/*
Name: display_temperatures()
Return Type: void
Parameter:(data type of each parameter): const double* and int
Short description: it is used to display the temperature distribution
*/

// Function to display the temperature distribution
void display_temperatures(const double *temperatures, int size)
{
    printf("Temperature distribution:\n");
    for (int i = 0; i < size; i++)
        printf("Point %d: %.2f °C\n", i + 1, temperatures[i]);
}

/*
Name: simulate_heat_transfer()
Return Type: void
```

Parameter:(data type of each parameter): const double*, double* and int

Short description: it is used to simulate heat transfer

*/

```
// Function to simulate heat transfer
void simulate_heat_transfer(const double *material_properties, double *temperatures, int size)
{
    double thermal_conductivity = material_properties[0];

    for (int i = 1; i < size - 1; i++)
        // Apply heat transfer from neighboring points
            temperatures[i] = temperatures[i] + thermal_conductivity * (temperatures[i - 1] + temperatures[i + 1] - 2 * temperatures[i]);

        temperatures[0] = temperatures[0] + thermal_conductivity * (temperatures[1] - temperatures[0]);  // Left boundary
    temperatures[size - 1] = temperatures[size - 1] + thermal_conductivity * (temperatures[size - 2] - temperatures[size - 1]);  // Right boundary

    printf("Heat transfer simulation complete.\n");
}
```

O/P:

```
Temperature distribution:
Point 1: 10.00 °C
Point 2: 20.00 °C
Point 3: 30.00 °C
Point 4: 40.00 °C
```

Point 5: 50.00 °C

Heat transfer simulation complete.

Temperature distribution:

Point 1: 25.00 °C

Point 2: 20.00 °C

Point 3: 30.00 °C

Point 4: 40.00 °C

Point 5: 35.00 °C

13. Aircraft Fuel Efficiency

- Pointers: Traverse fuel consumption arrays.
- Arrays: Store fuel consumption at different time intervals.
- Functions:
    - double compute_efficiency(const double *fuel_data, int size): Calculates overall fuel efficiency.
    - void update_fuel_data(double *fuel_data, int interval, double consumption): Updates fuel data for a specific interval.
- Pass Arrays as Pointers: Pass fuel data arrays as pointers.

```
#include <stdio.h>

// Function prototypes
double compute_efficiency(const double *fuel_data, int size);
void update_fuel_data(double *fuel_data, int interval, double consumption);

int main()
```

```
{
    double fuel_data[5] = {100.0, 130.0, 110.0, 140.0, 150.0};
    int size = sizeof(fuel_data) / sizeof(fuel_data[0]);

    printf("Initial fuel consumption data\n");
    for (int i = 0; i < size; i++)
        printf("Interval %d: %.2f units\n", i + 1, fuel_data[i]);

    // Call the function to compute and display the overall fuel efficiency
    double efficiency = compute_efficiency(fuel_data, size);
    printf("Overall fuel efficiency: %.2f units per interval\n", efficiency);

    // Call the function to update fuel consumption for the 3rd interval
    update_fuel_data(fuel_data, 2, 120.0);

    // Call the function to recompute and display the updated fuel efficiency
    efficiency = compute_efficiency(fuel_data, size);
    printf("Updated fuel efficiency: %.2f units per interval\n", efficiency);

    printf("Updated fuel consumption data\n");
    for (int i = 0; i < size; i++)
        printf("Interval %d: %.2f units\n", i + 1, fuel_data[i]);
    return 0;
}

/*
Name: compute_efficiency()
```

```
Return Type: double
Parameter:(data type of each parameter): const double* and int
Short description: it is used to compute the overall fuel efficiency
*/

// Function to compute the overall fuel efficiency
double compute_efficiency(const double *fuel_data, int size)
{
    double total_fuel_consumed = 0.0;

    for (int i = 0; i < size; i++)
        total_fuel_consumed += fuel_data[i];

    return total_fuel_consumed / size;
}

/*
Name: update_fuel_data()
Return Type: void
Parameter:(data type of each parameter): double*, int and double
Short description: it is used to update the fuel data for a specific time interval
*/

// Function to update the fuel data for a specific time interval
void update_fuel_data(double *fuel_data, int interval, double consumption)
{
```

```
    if (interval >= 0)
    {
        fuel_data[interval] = consumption;
        printf("Fuel data updated at interval %d: %.2f units\n", interval + 1,
consumption);
    }
    else
        printf("Invalid interval\n");
}
```

O/P:

```
Initial fuel consumption data
Interval 1: 100.00 units
Interval 2: 130.00 units
Interval 3: 110.00 units
Interval 4: 140.00 units
Interval 5: 150.00 units
Overall fuel efficiency: 126.00 units per interval
Fuel data updated at interval 3: 120.00 units
Updated fuel efficiency: 128.00 units per interval
Updated fuel consumption data
Interval 1: 100.00 units
Interval 2: 130.00 units
Interval 3: 120.00 units
Interval 4: 140.00 units
Interval 5: 150.00 units
```

14. Satellite Communication Link Budget

- Pointers: Handle parameter arrays for computation.
- Arrays: Store communication parameters like power and losses.
- Functions:
  - double compute_link_budget(const double *parameters, int size): Calculates the total link budget.
  - void update_parameters(double *parameters, int index, double value): Updates a specific parameter.
- Pass Arrays as Pointers: Pass parameter arrays as pointers.

```
#include <stdio.h>

// Function prototypes
double compute_link_budget(const double *parameters, int size);
void update_parameters(double *parameters, int index, double value);

int main()
{
    double parameters[] = {50.0, 12.5, 74.6, 40.7, 48.6};
    int size = sizeof(parameters) / sizeof(parameters[0]);

    printf("Initial communication parameters:\n");
    for (int i = 0; i < size; i++)
        printf("Parameter %d: %.2f\n", i + 1, parameters[i]);

    // Call the function to compute and display the total link budget
    double link_budget = compute_link_budget(parameters, size);
    printf("Total link budget: %.2f dB\n", link_budget);
```

```
    // Call the function to update a parameter
    update_parameters(parameters, 2, -80.8);

    // Call the function to recompute and display the updated link budget
    link_budget = compute_link_budget(parameters, size);
    printf("Updated link budget: %.2f dB\n", link_budget);

    printf("Updated communication parameters:\n");
    for (int i = 0; i < size; i++)
        printf("Parameter %d: %.2f\n", i + 1, parameters[i]);
    return 0;
}

/*
Name: compute_link_budget()
Return Type: double
Parameter:(data type of each parameter): const double* and int
Short description: it is used to compute the total link budget
*/

// Function to compute the total link budget
double compute_link_budget(const double *parameters, int size)
{
    double total_link_budget = 0.0;
    for (int i = 0; i < size; i++)
        total_link_budget += parameters[i];
```

```
    return total_link_budget;
}

/*
Name: update_parameters()
Return Type: void
Parameter:(data type of each parameter): double*, int and double
Short description: it is used to update a specific parameter
*/

// Function to update a specific parameter
void update_parameters(double *parameters, int index, double value)
{
    if (index >= 0)
    {
        parameters[index] = value;
        printf("Parameter at index %d updated to: %.2f\n", index, value);
    }
    else
        printf("Invalid index\n");
}
```

O/P:

```
Initial communication parameters:
Parameter 1: 50.00
Parameter 2: 12.50
```

Parameter 3: 74.60

Parameter 4: 40.70

Parameter 5: 48.60

Total link budget: 226.40 dB

Parameter at index 2 updated to: -80.80

Updated link budget: 71.00 dB

Updated communication parameters:

Parameter 1: 50.00

Parameter 2: 12.50

Parameter 3: -80.80

Parameter 4: 40.70

Parameter 5: 48.60

15. Turbulence Detection in Aircraft

- Pointers: Traverse acceleration arrays.
- Arrays: Store acceleration data from sensors.
- Functions:
  - void detect_turbulence(const double *accelerations, int size, double *output): Detects turbulence based on frequency analysis.
  - void log_turbulence(double *turbulence_log, const double *detection_output, int size): Logs detected turbulence events.
- Pass Arrays as Pointers: Pass acceleration and log arrays to functions.

```
#include <stdio.h>

#include <math.h>


//Function prototypes
```

```c
void detect_turbulence(const double *accelerations, int size, double *output);
void log_turbulence(double *turbulence_log, const double *detection_output, int size);


int main()
{
    double accelerations[] = {1.5, 1.0, 2.5, 3.2, -4.1};
    int size = sizeof(accelerations) / sizeof(accelerations[0]);


    double detection_output[size];
    double turbulence_log[size];


    // Call the function to detect turbulence
    detect_turbulence(accelerations, size, detection_output);


    // Call the function to log detected turbulence events
    log_turbulence(turbulence_log, detection_output, size);


    printf("Acceleration data:\n");
    for (int i = 0; i < size; i++)
        printf("Sensor %d: %.2f\n", i + 1, accelerations[i]);


    printf("Turbulence detection output (1 = Turbulence, 0 = No turbulence)\n");
    for (int i = 0; i < size; i++)
        printf("Sensor %d: %.2f\n", i + 1, detection_output[i]);
```

```
    printf("Turbulence Log (1 = Logged Event, 0 = No Event)\n");
    for (int i = 0; i < size; i++)
        printf("Sensor %d: %.2f\n", i + 1, turbulence_log[i]);
    return 0;
}

/*
Name: detect_turbulence()
Return Type: void
Parameter:(data type of each parameter): const double*, int and double*
Short description: it is used to detect turbulence based on frequency analysis
*/

// Function to detect turbulence based on frequency analysis
void detect_turbulence(const double *accelerations, int size, double *output)
{
    double turbulence_threshold = 2.5;

    for (int i = 0; i < size; i++)
    {
        if (fabs(accelerations[i]) > turbulence_threshold)
            output[i] = 1.0;
        else
            output[i] = 0.0;
    }
}
```

```
/*
Name: log_turbulence()
Return Type: void
Parameter:(data type of each parameter): double*, const double* and int
Short description: it is used to log detected turbulence events
*/

// Function to log detected turbulence events
void log_turbulence(double *turbulence_log, const double *detection_output, int size)
{
   for (int i = 0; i < size; i++)
   {
      if (detection_output[i] == 1.0)
         turbulence_log[i] = 1.0;
      else
         turbulence_log[i] = 0.0;
   }
}
```

O/P:

```
Acceleration data:
Sensor 1: 1.50
Sensor 2: 1.00
Sensor 3: 2.50
Sensor 4: 3.20
Sensor 5: -4.10
Turbulence detection output (1 = Turbulence, 0 = No turbulence)
```

Sensor 1: 0.00

Sensor 2: 0.00

Sensor 3: 0.00

Sensor 4: 1.00

Sensor 5: 1.00

Turbulence Log (1 = Logged Event, 0 = No Event)

Sensor 1: 0.00

Sensor 2: 0.00

Sensor 3: 0.00

Sensor 4: 1.00

Sensor 5: 1.00