Assignment – 11

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.

Sol:
```
#include <stdio.h>

#include <math.h>

#define GRAVITY 9.81 // Acceleration due to gravity (m/s^2)

// Function to calculate the trajectory

void calculate_trajectory(const double *parameters, double *trajectory, int size) {

    double initial_velocity = parameters[0];

    double angle = parameters[1]; // Angle in degrees

    double time_interval = parameters[2];



    double angle_rad = angle * M_PI / 180.0; // Convert angle to radians

    double velocity_x = initial_velocity * cos(angle_rad);

    double velocity_y = initial_velocity * sin(angle_rad);



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

```
        double t = i * time_interval; // Current time
        double x = velocity_x * t;
        double y = velocity_y * t - 0.5 * GRAVITY * t * t;
        trajectory[3 * i] = x;        // Store x-coordinate
        trajectory[3 * i + 1] = y;    // Store y-coordinate
        trajectory[3 * i + 2] = 0.0;  // Assuming z = 0 for 2D trajectory

        if (y < 0) {
            trajectory[3 * i + 1] = 0.0; // Stop when projectile hits the ground
            break;
        }
    }
}

// Function to print the trajectory
void print_trajectory(const double *trajectory, int size) {
    printf("Trajectory points:\n");
    printf("Time\tX\t         Y\t     Z\n");
    for (int i = 0; i < size; i++) {
        double x = trajectory[3 * i];
        double y = trajectory[3 * i + 1];
```

```
        double z = trajectory[3 * i + 2];
        if (y < 0.01 && i > 0) // Stop printing after the projectile hits the ground
            break;
        printf("%d\t%.2f\t%.2f\t%.2f\n", i, x, y, z);
    }
}

int main() {
    const int size = 100; // Number of trajectory points
    double trajectory[3 * size]; // Array to store trajectory points (x, y, z)

    // Parameters: initial velocity (m/s), angle (degrees), time interval (s)
    double parameters[3] = {50.0, 45.0, 0.1};

    calculate_trajectory(parameters, trajectory, size);
    print_trajectory(trajectory, size);

    return 0;
}
```

O/p:

Trajectory points:

Time	X		Y	Z
0	0.00	0.00	0.00	
1	3.54	3.49	0.00	
2	7.07	6.87	0.00	
3	10.61	10.17	0.00	
4	14.14	13.36	0.00	
5	17.68	16.45	0.00	
6	21.21	19.45	0.00	
7	24.75	22.35	0.00	
8	28.28	25.15	0.00	
9	31.82	27.85	0.00	
10	35.36	30.45	0.00	
11	38.89	32.96	0.00	
12	42.43	35.36	0.00	
13	45.96	37.67	0.00	
14	49.50	39.88	0.00	
15	53.03	42.00	0.00	
16	56.57	44.01	0.00	
17	60.10	45.93	0.00	
18	63.64	47.75	0.00	
19	67.18	49.47	0.00	

20	70.71	51.09	0.00
21	74.25	52.62	0.00
22	77.78	54.04	0.00
23	81.32	55.37	0.00
24	84.85	56.60	0.00
25	88.39	57.73	0.00
26	91.92	58.77	0.00
27	95.46	59.70	0.00
28	98.99	60.54	0.00
29	102.53	61.28	0.00
30	106.07	61.92	0.00
31	109.60	62.46	0.00
32	113.14	62.91	0.00
33	116.67	63.26	0.00
34	120.21	63.51	0.00
35	123.74	63.66	0.00
36	127.28	63.71	0.00
37	130.81	63.67	0.00
38	134.35	63.52	0.00
39	137.89	63.28	0.00
40	141.42	62.94	0.00

41	144.96	62.50	0.00
42	148.49	61.97	0.00
43	152.03	61.33	0.00
44	155.56	60.60	0.00
45	159.10	59.77	0.00
46	162.63	58.84	0.00
47	166.17	57.82	0.00
48	169.71	56.69	0.00
49	173.24	55.47	0.00
50	176.78	54.15	0.00
51	180.31	52.73	0.00
52	183.85	51.22	0.00
53	187.38	49.60	0.00
54	190.92	47.89	0.00
55	194.45	46.08	0.00
56	197.99	44.17	0.00
57	201.53	42.16	0.00
58	205.06	40.06	0.00
59	208.60	37.85	0.00
60	212.13	35.55	0.00
61	215.67	33.15	0.00

62	219.20	30.65	0.00
63	222.74	28.06	0.00
64	226.27	25.37	0.00
65	229.81	22.57	0.00
66	233.35	19.68	0.00
67	236.88	16.70	0.00
68	240.42	13.61	0.00
69	243.95	10.42	0.00
70	247.49	7.14	0.00
71	251.02	3.76	0.00
72	254.56	0.28	0.00

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.

Sol: #include <stdio.h>

#define TIME_STEP 1.0 // Time step in seconds

```
// Function to update the satellite's 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] * TIME_STEP;
    }
}


// Function to simulate the satellite's orbit
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 the current position in the positions array
        for (int i = 0; i < 3; i++) {
            positions[3 * step + i] = position[i];
        }


        // Update position using velocity
```

```c
        update_position(velocity, position, 3);
    }
}

// Function to print the simulated positions
void print_positions(const double *positions, int steps) {
    printf("Step\tX\tY\tZ\n");
    for (int step = 0; step < steps; step++) {
        printf("%d\t%.2f\t%.2f\t%.2f\n", step,
               positions[3 * step],
               positions[3 * step + 1],
               positions[3 * step + 2]);
    }
}

int main() {
    const int steps = 10; // Number of simulation steps
    double positions[3 * steps]; // Array to store positions over time

    // Initial conditions: position (x, y, z) and velocity (vx, vy, vz)
    double initial_conditions[6] = {0.0, 0.0, 0.0, 1.0, 1.0, 0.0};
```

```
    simulate_orbit(initial_conditions, positions, steps);

    print_positions(positions, steps);


    return 0;

}
```

O/p: Step X Y Z

0	0.00	0.00	0.00
1	1.00	1.00	0.00
2	2.00	2.00	0.00
3	3.00	3.00	0.00
4	4.00	4.00	0.00
5	5.00	5.00	0.00
6	6.00	6.00	0.00
7	7.00	7.00	0.00
8	8.00	8.00	0.00
9	9.00	9.00	0.00

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.

Sol:
```
#include <stdio.h>


// Function to calculate the daily average from hourly data

void calculate_daily_averages(const double *data, int size, double *average) {

    double sum = 0.0;

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

        sum += data[i];

    }

    *average = sum / size;  // Calculate the average and store it at the address of 
'average'

}


// Function to display the weather data

void display_weather_data(const double *data, int size) {

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

        printf("Hour %d: %.2f\n", i + 1, data[i]);  // Print hourly data

    }

}
```

```
int main() {
    // Hourly temperature data for 24 hours
    double temperature[24] = {20.5, 21.0, 22.0, 23.0, 24.5, 25.0, 26.0, 26.5, 27.0,
27.5,
                              28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0,
                              38.0, 39.0, 40.0, 41.0, 42.0, 43.0};

    // Calculate and display the daily average temperature
    double avg_temperature;
    calculate_daily_averages(temperature, 24, &avg_temperature);
    printf("Daily average temperature: %.2f\n", avg_temperature);

    // Display hourly temperature data
    printf("Hourly temperature data:\n");
    display_weather_data(temperature, 24);

    return 0;
}
```

O/p: Daily average temperature: 30.25

Hourly temperature data:

Hour 1: 20.50

Hour 2: 21.00

Hour 3: 22.00

Hour 4: 23.00

Hour 5: 24.50

Hour 6: 25.00

Hour 7: 26.00

Hour 8: 26.50

Hour 9: 27.00

Hour 10: 27.50

Hour 11: 28.00

Hour 12: 29.00

Hour 13: 30.00

Hour 14: 31.00

Hour 15: 32.00

Hour 16: 33.00

Hour 17: 34.00

Hour 18: 35.00

Hour 19: 36.00

Hour 20: 37.00

Hour 21: 38.00

Hour 22: 39.00

Hour 23: 40.00

Hour 24: 41.00

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.

Sol:
```
#include <stdio.h>



#define ERROR_HISTORY_SIZE 3 // Store last 3 errors: [current, previous, pre-previous]



// Function to compute PID control output

double compute_pid(const double *errors, int size, const double *gains) {

    double proportional = gains[0] * errors[0];

    double integral = gains[1] * (errors[0] + errors[1] + errors[2]);

    double derivative = gains[2] * (errors[0] - errors[1]);



    return proportional + integral + derivative;
```

```
}


// Function to update the error array with the latest error value

void update_errors(double *errors, double new_error) {

    for (int i = ERROR_HISTORY_SIZE - 1; i > 0; i--) {

        errors[i] = errors[i - 1];

    }

    errors[0] = new_error;

}


int main() {

    double errors[ERROR_HISTORY_SIZE] = {0.0, 0.0, 0.0}; // Initialize error
history

    double gains[3] = {1.0, 0.1, 0.05}; // PID gains: [Kp, Ki, Kd]


    // Simulate errors and compute PID output

    double new_errors[] = {0.5, 0.2, -0.1, -0.3};

    int num_new_errors = sizeof(new_errors) / sizeof(new_errors[0]);


    for (int i = 0; i < num_new_errors; i++) {

        update_errors(errors, new_errors[i]);
```

```
        double control_output = compute_pid(errors, ERROR_HISTORY_SIZE,
gains);

        printf("Step %d: Error = %.2f, Control Output = %.2f\n", i + 1, new_errors[i],
control_output);

    }



    return 0;

}
```

O/p:

Step 1: Error = 0.50, Control Output = 0.58

Step 2: Error = 0.20, Control Output = 0.26

Step 3: Error = -0.10, Control Output = -0.06

Step 4: Error = -0.30, Control Output = -0.33

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.

Sol: #include <stdio.h>

```
// Function to fuse data from two sensors into one result array

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

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

        result[i] = (sensor1[i] + sensor2[i]) / 2.0;  // Average the readings from both
sensors

    }

}



// Function to calibrate sensor data by applying a calibration factor (for example)

void calibrate_data(double *data, int size) {

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

        data[i] = data[i] * 1.1;  // Example: Increase each sensor reading by 10% as
part of calibration

    }

}



int main() {

    // Example sensor data from two sensors (e.g., temperature readings)

    double sensor1[5] = {22.0, 23.5, 24.0, 25.0, 26.5};

    double sensor2[5] = {21.5, 23.0, 24.5, 25.5, 27.0};



    double fused_data[5];  // Array to store the fused data
```

```
double calibrated_data[5];  // Array to store calibrated data

// Fuse data from the two sensors
fuse_data(sensor1, sensor2, fused_data, 5);

// Display the fused data
printf("Fused Sensor Data:\n");
for (int i = 0; i < 5; i++) {
    printf("Fused Data[%d]: %.2f\n", i, fused_data[i]);
}

// Calibrate the fused data
for (int i = 0; i < 5; i++) {
    calibrated_data[i] = fused_data[i];
}
calibrate_data(calibrated_data, 5);

// Display the calibrated data
printf("\nCalibrated Sensor Data:\n");
for (int i = 0; i < 5; i++) {
    printf("Calibrated Data[%d]: %.2f\n", i, calibrated_data[i]);
```

```
    }

    return 0;
}
```

O/p: Fused Sensor Data:

Fused Data[0]: 21.75

Fused Data[1]: 23.25

Fused Data[2]: 24.25

Fused Data[3]: 25.25

Fused Data[4]: 26.75

Calibrated Sensor Data:

Calibrated Data[0]: 23.93

Calibrated Data[1]: 25.58

Calibrated Data[2]: 26.68

Calibrated Data[3]: 27.78

Calibrated Data[4]: 29.43

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.

Sol:
```
#include <stdio.h>

#include <string.h>



#define MAX_FLIGHTS 100



// Flight structure

typedef struct {

    int id;

    int altitude;

    float latitude;

    float longitude;

} flight_t;



// Function to add a flight

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

    if (*flight_count < MAX_FLIGHTS) {

        flights[*flight_count] = *new_flight;

        (*flight_count)++;
```

```
    } else {
        printf("Error: Maximum flight capacity reached.\n");
    }
}

// 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].id == flight_id) {
            // Shift flights down to remove the flight
            for (int j = i; j < *flight_count - 1; j++) {
                flights[j] = flights[j + 1];
            }
            (*flight_count)--;
            printf("Flight ID %d removed.\n", flight_id);
            return;
        }
    }
    printf("Error: Flight ID %d not found.\n", flight_id);
}
```

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

    // Adding some flights
    flight_t flight1 = {1, 30000, 40.7128, -74.0060};
    flight_t flight2 = {2, 35000, 34.0522, -118.2437};
    add_flight(flights, &flight_count, &flight1);
    add_flight(flights, &flight_count, &flight2);

    // Display flights
    printf("Active Flights:\n");
    for (int i = 0; i < flight_count; i++) {
        printf("ID: %d, Altitude: %d, Coordinates: (%.4f, %.4f)\n",
               flights[i].id, flights[i].altitude,
               flights[i].latitude, flights[i].longitude);
    }

    // Removing a flight
    remove_flight(flights, &flight_count, 1);
```

```
    // Display flights after removal
    printf("Active Flights After Removal:\n");
    for (int i = 0; i < flight_count; i++) {
        printf("ID: %d, Altitude: %d, Coordinates: (%.4f, %.4f)\n",
               flights[i].id, flights[i].altitude,
               flights[i].latitude, flights[i].longitude);
    }

    return 0;
}
```

o/p: Active Flights:

ID: 1, Altitude: 30000, Coordinates: (40.7128, -74.0060)

ID: 2, Altitude: 35000, Coordinates: (34.0522, -118.2437)

Flight ID 1 removed.

Active Flights After Removal:

ID: 2, Altitude: 35000, Coordinates: (34.0522, -118.2437)

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.

Sol:
```
#include <stdio.h>

#include <math.h>



#define MAX_DATA 100



// 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, mean, variance = 0, stddev;



    // Calculate mean

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

        sum += data[i];

    }
```

```
    mean = sum / size;

    // Calculate variance
    for (int i = 0; i < size; i++) {
        variance += (data[i] - mean) * (data[i] - mean);
    }
    variance /= size;
    stddev = sqrt(variance);

    // Display results
    printf("Telemetry Analysis:\n");
    printf("Mean: %.2f\n", mean);
    printf("Standard Deviation: %.2f\n", stddev);
}

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

```
double sum = 0, mean, stddev, variance = 0;
int new_size = 0;

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

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

// Filter outliers (values outside mean ± 2 * stddev)
double filtered[MAX_DATA];
for (int i = 0; i < *size; i++) {
    if (fabs(data[i] - mean) <= 2 * stddev) {
```

```
            filtered[new_size++] = data[i];
        }
    }

    // Update the original array
    for (int i = 0; i < new_size; i++) {
        data[i] = filtered[i];
    }
    *size = new_size;

    printf("Outliers removed. New size: %d\n", *size);
}

// Main function
int main() {
    double telemetry_data[MAX_DATA] = {120.5, 125.3, 130.2, 1000.0, 126.7,
128.1};
    int size = 6;

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

```
    }
    printf("\n");

    // Analyze telemetry data
    analyze_telemetry(telemetry_data, size);

    // Filter outliers
    filter_outliers(telemetry_data, &size);

    // Display filtered data
    printf("Filtered Telemetry Data:\n");
    for (int i = 0; i < size; i++) {
        printf("%.2f ", telemetry_data[i]);
    }
    printf("\n");

    return 0;
}
```

O/p: Original Telemetry Data:

120.50 125.30 130.20 1000.00 126.70 128.10

Telemetry Analysis:

Mean: 271.80

Standard Deviation: 325.67

Outliers removed. New size: 5

Filtered Telemetry Data:

120.50 125.30 130.20 126.70 128.10

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.

Sol:
```
#include <stdio.h>


#define MAX_STAGES 5


// Function to compute total thrust across all stages

double compute_total_thrust(const double *stages, int size) {

    double total_thrust = 0;

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

        total_thrust += stages[i];

    }
```

```
    return total_thrust;
}

// Function to update thrust for a specific stage
void update_stage_thrust(double *stages, int stage, double new_thrust) {
    if (stage >= 0 && stage < MAX_STAGES) {
        stages[stage] = new_thrust;
        printf("Thrust for stage %d updated to %.2f\n", stage, new_thrust);
    } else {
        printf("Error: Invalid stage number.\n");
    }
}

// Main function
int main() {
    double rocket_thrust[MAX_STAGES] = {150.5, 200.3, 250.7, 180.2, 220.0};
    int num_stages = 5;

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

```
    // Update thrust for the second stage (stage 1)

    update_stage_thrust(rocket_thrust, 1, 210.5);


    // Recompute total thrust after update

    total_thrust = compute_total_thrust(rocket_thrust, num_stages);

    printf("Total thrust after update: %.2f\n", total_thrust);


    return 0;

}
```

O/p: Total thrust of the rocket: 1001.70

Thrust for stage 1 updated to 210.50

Total thrust after update: 1011.90

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.

Sol: #include <stdio.h>

```
#define MAX_SECTIONS 5


// 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 is calculated as force divided by the area of the section (assumed area
= 10 for simplicity)

        stress[i] = forces[i] / 10.0;  // Example stress calculation: force/area

    }

}


// Function to display stress distribution across sections

void display_stress(const double *stress, int size) {

    printf("Stress distribution across the wing sections:\n");

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

        printf("Section %d: %.2f MPa\n", i + 1, stress[i]);

    }

}


// Main function

int main() {
```

```
    double applied_forces[MAX_SECTIONS] = {5000.0, 6000.0, 5500.0, 4500.0,
4000.0};  // Forces in Newtons

    double wing_stress[MAX_SECTIONS];  // Stress values for each section

    int num_sections = 5;


    // Compute the stress distribution

    compute_stress_distribution(applied_forces, wing_stress, num_sections);


    // Display the stress distribution

    display_stress(wing_stress, num_sections);


    return 0;

}
```

O/p: Stress distribution across the wing sections:

Section 1: 500.00 MPa

Section 2: 600.00 MPa

Section 3: 550.00 MPa

Section 4: 450.00 MPa

Section 5: 400.00 MPa

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.

Sol:
```
#include <stdio.h>

#include <math.h>



#define MAX_WAYPOINTS 10



// Function to calculate the Euclidean distance between two points (x1, y1) and (x2, y2)

double calculate_distance(double x1, double y1, double x2, double y2) {

    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));

}



// Function to optimize the path by reducing the total path length

double optimize_path(const double *waypoints, int size) {

    double total_distance = 0;



    for (int i = 0; i < size - 1; i++) {

        double x1 = waypoints[i * 2];

        double y1 = waypoints[i * 2 + 1];
```

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

        double y2 = waypoints[(i + 1) * 2 + 1];


        total_distance += calculate_distance(x1, y1, x2, y2);

    }


    return total_distance;

}


// Function to add a new waypoint to the array

void add_waypoint(double *waypoints, int *size, double x, double y) {

    if (*size < MAX_WAYPOINTS) {

        waypoints[*size * 2] = x;

        waypoints[*size * 2 + 1] = y;

        (*size)++;

    } else {

        printf("Error: Cannot add more waypoints, maximum limit reached.\n");

    }

}


// Main function
```

```
int main() {

    double waypoints[MAX_WAYPOINTS * 2] = {0, 0, 3, 4, 6, 8};  // Example
waypoints: (0,0), (3,4), (6,8)

    int size = 3;  // Number of waypoints



    // Display initial waypoints

    printf("Initial waypoints:\n");

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

        printf("Waypoint %d: (%.2f, %.2f)\n", i + 1, waypoints[i * 2], waypoints[i * 2
+ 1]);

    }



    // Calculate and display the total path length

    double total_distance = optimize_path(waypoints, size);

    printf("Total path length: %.2f\n", total_distance);



    // Add a new waypoint

    add_waypoint(waypoints, &size, 9, 12);  // Add a new waypoint at (9,12)



    // Display updated waypoints

    printf("\nUpdated waypoints:\n");

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

```
        printf("Waypoint %d: (%.2f, %.2f)\n", i + 1, waypoints[i * 2], waypoints[i * 2 + 1]);
    }

    // Calculate and display the new total path length
    total_distance = optimize_path(waypoints, size);
    printf("Total path length after adding a waypoint: %.2f\n", total_distance);

    return 0;
}
```

O/p: Initial waypoints:

Waypoint 1: (0.00, 0.00)

Waypoint 2: (3.00, 4.00)

Waypoint 3: (6.00, 8.00)

Total path length: 10.00

Updated waypoints:

Waypoint 1: (0.00, 0.00)

Waypoint 2: (3.00, 4.00)

Waypoint 3: (6.00, 8.00)

Waypoint 4: (9.00, 12.00)

Total path length after adding a waypoint: 15.00

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.

Sol:
```
#include <stdio.h>

#include <math.h>



#define QUATERNION_SIZE 4  // Quaternion has 4 components: w, x, y, z



// Function to update the satellite's attitude using a quaternion

void update_attitude(const double *quaternion, double *new_attitude) {

    // Assuming quaternion is in the form (w, x, y, z)

    // Here, we simply copy the quaternion to the new attitude array as an example.

    // In practice, this would involve applying the quaternion to the current
orientation.

    for (int i = 0; i < QUATERNION_SIZE; i++) {

        new_attitude[i] = quaternion[i];

    }

}
```

```
// Function to normalize the quaternion
void normalize_quaternion(double *quaternion) {
    // Calculate the magnitude of the quaternion
    double magnitude = 0;
    for (int i = 0; i < QUATERNION_SIZE; i++) {
        magnitude += quaternion[i] * quaternion[i];
    }
    magnitude = sqrt(magnitude);

    // Normalize each component of the quaternion
    for (int i = 0; i < QUATERNION_SIZE; i++) {
        quaternion[i] /= magnitude;
    }
}

// Main function
int main() {
    // Example quaternion: (w, x, y, z)
    double quaternion[QUATERNION_SIZE] = {1.0, 2.0, 3.0, 4.0};
    double new_attitude[QUATERNION_SIZE];
```

```
    // Display the initial quaternion

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



    // Normalize the quaternion

    normalize_quaternion(quaternion);



    // Display the normalized quaternion

    printf("Normalized quaternion: (%.2f, %.2f, %.2f, %.2f)\n", quaternion[0],
quaternion[1], quaternion[2], quaternion[3]);



    // Update the satellite's attitude with the normalized quaternion

    update_attitude(quaternion, new_attitude);



    // Display the updated attitude (new quaternion)

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



    return 0;

}
```

O/p: Initial quaternion: (1.00, 2.00, 3.00, 4.00)

Normalized quaternion: (0.18, 0.37, 0.55, 0.73)

Updated attitude: (0.18, 0.37, 0.55, 0.73)

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.

Sol:

```
#include <stdio.h>



#define MAX_POINTS 10



// Function to simulate heat transfer across the material

void simulate_heat_transfer(const double *material_properties, double *temperatures, int size) {

    // Assume material_properties contains a coefficient for heat conduction (just a simple example)

    double conduction_coefficient = material_properties[0];  // Example: heat conductivity coefficient



    // Update temperature at each point based on a simple heat transfer formula

    for (int i = 1; i < size - 1; i++) {  // Avoid first and last points for simplicity
```

```
        temperatures[i] += conduction_coefficient * (temperatures[i - 1] - 2 *
temperatures[i] + temperatures[i + 1]);

    }

}



// Function to display the temperature distribution across the material

void display_temperatures(const double *temperatures, int size) {

    printf("Temperature distribution across the material:\n");

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

        printf("Point %d: %.2f°C\n", i + 1, temperatures[i]);

    }

}



// Main function

int main() {

    double material_properties[1] = {0.5};  // Heat conductivity coefficient (example
value)

    double temperatures[MAX_POINTS] = {100.0, 150.0, 200.0, 250.0, 300.0,
350.0, 400.0, 450.0, 500.0, 550.0};  // Initial temperatures at each point

    int size = 10;  // Number of points



    // Display initial temperatures

    printf("Initial temperatures:\n");
```

```
    display_temperatures(temperatures, size);

    // Simulate heat transfer across the material
    simulate_heat_transfer(material_properties, temperatures, size);

    // Display updated temperatures after simulation
    printf("\nUpdated temperatures after heat transfer:\n");
    display_temperatures(temperatures, size);

    return 0;
}
```

O/p: Initial temperatures:

Temperature distribution across the material:

Point 1: 100.00°C

Point 2: 150.00°C

Point 3: 200.00°C

Point 4: 250.00°C

Point 5: 300.00°C

Point 6: 350.00°C

Point 7: 400.00°C

Point 8: 450.00°C

Point 9: 500.00°C

Point 10: 550.00°C

Updated temperatures after heat transfer:

Temperature distribution across the material:

Point 1: 100.00°C

Point 2: 150.00°C

Point 3: 200.00°C

Point 4: 250.00°C

Point 5: 300.00°C

Point 6: 350.00°C

Point 7: 400.00°C

Point 8: 450.00°C

Point 9: 500.00°C

Point 10: 550.00°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.

Sol:
```
#include <stdio.h>

#define MAX_INTERVALS 10

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

    // Assuming each time interval corresponds to a fixed distance (e.g., 100 km per interval for simplicity)
    double distance_per_interval = 100.0;

    for (int i = 0; i < size; i++) {
        total_fuel += fuel_data[i];
        total_distance += distance_per_interval;
    }

    // Fuel efficiency: distance per unit of fuel
    return total_distance / total_fuel;
}
```

```
// Function to update fuel consumption data for a specific interval

void update_fuel_data(double *fuel_data, int interval, double consumption) {

    if (interval >= 0 && interval < MAX_INTERVALS) {

        fuel_data[interval] = consumption;

        printf("Fuel consumption at interval %d updated to %.2f\n", interval,
consumption);

    } else {

        printf("Error: Invalid interval.\n");

    }

}



// Main function

int main() {

    // Initial fuel consumption data (for 10 intervals, in liters)

    double fuel_data[MAX_INTERVALS] = {50.0, 55.0, 53.0, 60.0, 58.0, 55.0,
52.0, 57.0, 59.0, 61.0};

    int size = 10;



    // Calculate and display the initial fuel efficiency

    double efficiency = compute_efficiency(fuel_data, size);

    printf("Initial fuel efficiency: %.2f km per liter\n", efficiency);
```

```
    // Update the fuel consumption at a specific interval (e.g., interval 2)
    update_fuel_data(fuel_data, 2, 54.0);  // Update fuel consumption at interval 2


    // Recalculate and display the updated fuel efficiency
    efficiency = compute_efficiency(fuel_data, size);
    printf("Updated fuel efficiency: %.2f km per liter\n", efficiency);


    return 0;
}
```

O/p: Initial fuel efficiency: 1.79 km per liter

Fuel consumption at interval 2 updated to 54.00

Updated fuel efficiency: 1.78 km per liter

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.

Sol:
```
#include <stdio.h>


#define MAX_PARAMETERS 10
```

```
// Function to compute the total link budget
double compute_link_budget(const double *parameters, int size) {
    double link_budget = 0.0;

    // Sum all parameters to compute the link budget
    for (int i = 0; i < size; i++) {
        link_budget += parameters[i];
    }

    return link_budget;
}

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

```
}


// Main function
int main() {
    // Initial communication parameters (e.g., power, losses, gains, etc.)
    double parameters[MAX_PARAMETERS] = {50.0, -3.0, 10.0, 5.0, -2.0, 6.0, 1.0, -1.0, 4.0, 2.0};
    int size = 10;

    // Calculate and display the initial link budget
    double link_budget = compute_link_budget(parameters, size);
    printf("Initial link budget: %.2f dB\n", link_budget);

    // Update a specific parameter (e.g., parameter at index 3)
    update_parameters(parameters, 3, 7.0);  // Update parameter at index 3 to 7.0

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

    return 0;
}
```

O/p: Initial link budget: 72.00 dB

Parameter at index 3 updated to 7.00

Updated link budget: 74.00 dB

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.\

Sol:
```
#include <stdio.h>

#include <math.h>



#define MAX_DATA_POINTS 10



// Function to detect turbulence based on acceleration data (simplified frequency analysis)

void detect_turbulence(const double *accelerations, int size, double *output) {

    // A simple approach: detect turbulence when the acceleration exceeds a threshold

    double threshold = 2.0;  // Example threshold for turbulence detection (in m/s^2)



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

```
        // Mark as turbulence if acceleration exceeds the threshold
        if (fabs(accelerations[i]) > threshold) {
            output[i] = 1.0;  // Indicate turbulence detected
        } else {
            output[i] = 0.0;  // No turbulence
        }
    }
}

// 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;  // Log turbulence event
            printf("Turbulence detected at index %d\n", i);
        } else {
            turbulence_log[i] = 0.0;  // No turbulence event to log
        }
    }
}
```

```c
// Main function

int main() {

    // Example acceleration data from sensors (in m/s^2)

    double accelerations[MAX_DATA_POINTS] = {1.5, 3.2, 0.8, 2.5, 1.9, 3.1, 1.3,
2.8, 0.6, 3.5};

    double detection_output[MAX_DATA_POINTS] = {0};  // To store detection
results (1 for turbulence, 0 for no turbulence)

    double turbulence_log[MAX_DATA_POINTS] = {0};    // To store logged
turbulence events



    int size = MAX_DATA_POINTS;



    // Detect turbulence in the acceleration data

    detect_turbulence(accelerations, size, detection_output);



    // Log detected turbulence events

    log_turbulence(turbulence_log, detection_output, size);



    // Display the turbulence log

    printf("\nTurbulence Log:\n");

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

        printf("Index %d: %s\n", i, turbulence_log[i] == 1.0 ? "Turbulence Detected"
: "No Turbulence");
```

```
    }

    return 0;
}
```

O/p: Turbulence detected at index 1

Turbulence detected at index 3

Turbulence detected at index 5

Turbulence detected at index 7

Turbulence detected at index 9

Turbulence Log:

Index 0: No Turbulence

Index 1: Turbulence Detected

Index 2: No Turbulence

Index 3: Turbulence Detected

Index 4: No Turbulence

Index 5: Turbulence Detected

Index 6: No Turbulence

Index 7: Turbulence Detected

Index 8: No Turbulence

Index 9: Turbulence Detected