Pointers 03-01-2025

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>

// Constants

#define GRAVITY 9.81

// Function prototypes

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

void print_trajectory(const double *trajectory, int size);

int main() {

    // Parameters: [initial_velocity, launch_angle]

    double parameters[2] = {50.0, 45.0};  // 50 m/s at a 45° angle

    int size = 100;  // Number of trajectory points

    double trajectory[size * 3];  // Array to store x, y, z trajectory points (3D)



    // Calculate and print trajectory

    calculate_trajectory(parameters, trajectory, size);

    print_trajectory(trajectory, size);



    return 0;
```

```
}


// Function to calculate trajectory
void calculate_trajectory(const double *parameters, double *trajectory, int size) {
    double initial_velocity = parameters[0];
    double angle = parameters[1] * (M_PI / 180.0);  // Convert angle to radians
    double time_interval = 2 * initial_velocity * sin(angle) / (GRAVITY * (size - 1));


    for (int i = 0; i < size; ++i) {
        double t = i * time_interval;  // Time at this interval
        double x = initial_velocity * cos(angle) * t;
        double y = initial_velocity * sin(angle) * t - 0.5 * GRAVITY * t * t;
        double z = 0;  // Assuming no vertical deviation


        trajectory[i * 3 + 0] = x;
        trajectory[i * 3 + 1] = y > 0 ? y : 0;  // Ensure y doesn't go negative
        trajectory[i * 3 + 2] = z;
    }
}


// Function to print trajectory
void print_trajectory(const double *trajectory, int size) {
    printf("Trajectory Points:\n");
    for (int i = 0; i < size; ++i) {
        printf("Point %d: x = %.2f, y = %.2f, z = %.2f\n",
```

```
                i,
                trajectory[i * 3 + 0],
                trajectory[i * 3 + 1],
                trajectory[i * 3 + 2]);
        }
    }
```

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>


void update_position(const double *velocity, double *position, int size) {
    for (int i = 0; i < size; i++) {
        position[i] += velocity[i];
    }
}


void simulate_orbit(const double *initial_conditions, double *positions, int steps) {
    double velocity[3] = {initial_conditions[3], initial_conditions[4], initial_conditions[5]};
    double position[3] = {initial_conditions[0], initial_conditions[1], initial_conditions[2]};
    for (int step = 0; step < steps; step++) {
        update_position(velocity, position, 3);
```

```
        positions[step * 3] = position[0];

        positions[step * 3 + 1] = position[1];

        positions[step * 3 + 2] = position[2];

    }

}


int main() {

    double initial_conditions[6] = {0, 0, 0, 1, 1, 1}; // {px, py, pz, vx, vy, vz}

    int steps = 10;

    double positions[steps * 3]; // Array to store positions


    simulate_orbit(initial_conditions, positions, steps);


    // Print the results

    for (int step = 0; step < steps; step++) {

        printf("Step %d: Position = (%lf, %lf, %lf)\n", step, positions[step * 3], positions[step * 3 + 1], positions[step * 3 + 2]);

    }


    return 0;

}
```

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 to calculate daily averages

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); // Use pointer arithmetic

    }

    *average = sum / size; // Store result in average

}


// Function to display weather data

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

    printf("Hourly Weather Data:\n");

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

        printf("%.2f ", *(data + i)); // Access elements using pointers

    }

    printf("\n");

}


int main() {

    // Example hourly data for 24 hours

    double temperature[4] = {15.0, 16.5, 18.0,18.6};

    double wind_speed[4] = {5.2, 6.1, 7.0, 8.5};
```

```
    double pressure[4] = {1012.0, 1013.5, 1014.1, 1012.35};

    double avg_temperature, avg_wind_speed, avg_pressure;

    // Calculate averages
    calculate_daily_averages(temperature, 4, &avg_temperature);
    calculate_daily_averages(wind_speed, 4, &avg_wind_speed);
    calculate_daily_averages(pressure, 4, &avg_pressure);

    // Display data and results
    printf("Temperature Data:\n");
    display_weather_data(temperature, 4);
    printf("Average Temperature: %.2f°C\n\n", avg_temperature);

    printf("Wind Speed Data:\n");
    display_weather_data(wind_speed, 4);
    printf("Average Wind Speed: %.2f km/h\n\n", avg_wind_speed);

    printf("Pressure Data:\n");
    display_weather_data(pressure, 4);
    printf("Average Pressure: %.2f hPa\n", avg_pressure);

    return 0;
}
```

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 to compute PID output

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

    double proportional = errors[size - 1]; // Latest error

    double integral = 0.0;

    double derivative = 0.0;



    // Calculate integral (sum of all errors)

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

        integral += *(errors + i);

    }



    // Calculate derivative (difference between last two errors)

    if (size > 1) {

        derivative = errors[size - 1] - errors[size - 2];

    }



    // PID formula
```

```
    return (gains[0] * proportional) + (gains[1] * integral) + (gains[2] * derivative);
}

// Function to update the errors array
void update_errors(double *errors, double new_error) {
    // Shift all errors to the left
    for (int i = 0; i < 2; ++i) {
        errors[i] = errors[i + 1];
    }
    // Add the new error to the end of the array
    errors[2] = new_error;
}

int main() {
    // PID gains: Kp, Ki, Kd
    double gains[3] = {1.0, 0.5, 0.1};

    // Error history: stores the last 3 errors
    double errors[3] = {0.0, 0.0, 0.0};

    // Simulate new error values
    double new_errors[] = {2.5, 1.8, 1.2, 0.8, 0.4};
    int new_errors_size = sizeof(new_errors) / sizeof(new_errors[0]);

    printf("PID Controller Output:\n");
```

```
    for (int i = 0; i < new_errors_size; ++i) {
        // Update errors with the latest value
        update_errors(errors, new_errors[i]);

        // Compute the PID output
        double pid_output = compute_pid(errors, 3, gains);
        printf("New Error: %.2f, PID Output: %.2f\n", new_errors[i], pid_output);
    }

    return 0;
}
```

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>
void fuse_data(const double *sensor1, const double *sensor2, double *result, int size);
void calibrate_data(double *data, int size);
int main(){
    double sensor1[]={22.4,23.5,24.8,23,24.3};
    double sensor2[]={32.3,23.6,34.5,33.8,31.5};
    int size=sizeof(sensor1)/sizeof(sensor1[0]);
    double result[size];
```

```
    fuse_data(sensor1,sensor2,result,size);

    printf("Fused Data:\n");

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

        printf("%.2f ", result[i]);

    }

    printf("\n");


    calibrate_data(result,size);

    printf("Calibrated data:\n");

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

        printf("%0.2f",result[i]);

    }


    return 0;

}

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;

    }

}

void calibrate_data(double *data, int size){

    const double calibration_offset = -0.5; // Example offset

    const double calibration_scale = 1.1;  // Example scale factor

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

        data[i]=(data[i]+calibration_offset)*calibration_scale;
```

```
    }
}
```

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>


// Define a structure to store flight details

typedef struct {

    int flight_id;

    double altitude;

    double latitude;

    double longitude;

} flight_t;


// Function declarations

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);

void display_flights(const flight_t *flights, int flight_count);


int main() {

    int flight_count = 0; // Number of active flights

    const int max_flights = 10; // Maximum number of flights that can be stored
```

```
    // Declare an array of flight structures
    flight_t flights[max_flights];

    // Adding new flights
    flight_t flight1 = {101, 30000, 37.7749, -122.4194}; // Flight 101
    add_flight(flights, &flight_count, &flight1);

    flight_t flight2 = {102, 32000, 34.0522, -118.2437}; // Flight 102
    add_flight(flights, &flight_count, &flight2);

    // Display the active flights
    printf("Active Flights:\n");
    display_flights(flights, flight_count);

    // Remove a flight by ID
    remove_flight(flights, &flight_count, 101);

    // Display the active flights after removal
    printf("\nActive Flights after removal:\n");
    display_flights(flights, flight_count);

    return 0;
}
```

```
// Function to add a new flight to the array
void add_flight(flight_t *flights, int *flight_count, const flight_t *new_flight) {
    if (*flight_count < 10) {
        flights[*flight_count] = *new_flight;  // Add the flight to the array
        (*flight_count)++;  // Increment the flight count
    } else {
        printf("Error: Maximum flight capacity reached.\n");
    }
}
// Function to remove a flight from the array by its flight_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) {
            // Shift the subsequent flights to fill the gap
            for (int j = i; j < *flight_count - 1; j++) {
                flights[j] = flights[j + 1];
            }
            (*flight_count)--;  // Decrement the flight count
            return;
        }
    }
    printf("Error: Flight with ID %d not found.\n", flight_id);
}

// Function to display all active flights
```

```
void display_flights(const flight_t *flights, int flight_count) {

    if (flight_count == 0) {

        printf("No active flights.\n");

        return;

    }



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

        printf("Flight ID: %d, Altitude: %.2f, Latitude: %.4f, Longitude: %.4f\n",

               flights[i].flight_id, flights[i].altitude,

               flights[i].latitude, flights[i].longitude);

    }

}
```

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. (abnormal values that don't fit the expected pattern)
- **Pass Arrays as Pointers**: Pass telemetry data arrays to both functions.

Mean=∑Data Points/Number of Data Points

Standard Deviation=(∑(Data Point−Mean)2/Number of data Points)^1/2

```
#include <stdio.h>

#include <math.h>



#define SAMPLE_SIZE 10
```

```
int main() {
    // Example telemetry data (power, temperature, voltage, etc.)
    double telemetry_data[SAMPLE_SIZE] = {100.5, 101.2, 102.0, 99.5, 110.5, 98.0, 
105.5, 102.5, 50.0, 103.0};

    // Analyze telemetry data (compute mean and standard deviation)
    analyze_telemetry(telemetry_data, SAMPLE_SIZE);

    // Print original data
    printf("Original telemetry data: ");
    print_data(telemetry_data, SAMPLE_SIZE);

    // Filter outliers
    filter_outliers(telemetry_data, SAMPLE_SIZE);

    // Print telemetry data after removing outliers
    printf("Telemetry data after filtering outliers: ");
    print_data(telemetry_data, SAMPLE_SIZE);

    return 0;
}

// Function to compute the mean of the telemetry data
double compute_mean(const double *data, int size) {
    double sum = 0.0;
    for (int i = 0; i < size; i++) {
```

```
        sum += data[i];
    }
    return sum / size;
}

// Function to compute the standard deviation of the telemetry data
double compute_stddev(const double *data, int size, double mean) {
    double sum = 0.0;
    for (int i = 0; i < size; i++) {
        sum += (data[i] - mean) * (data[i] - mean);
    }
    return sqrt(sum / size);
}

// Function to analyze telemetry data and compute statistical metrics
void analyze_telemetry(const double *data, int size) {
    double mean = compute_mean(data, size);
    double stddev = compute_stddev(data, size, mean);

    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) {
    double mean = compute_mean(data, size);
    double stddev = compute_stddev(data, size, mean);
    for (int i = 0; i < size; i++) {
        if (data[i] < (mean - 2 * stddev) || data[i] > (mean + 2 * stddev)) {
            data[i] = 0.0;  // Remove outlier (set it to 0 or some other marker)
        }
    }
}


// Function to print telemetry data
void print_data(const double *data, int size) {
    for (int i = 0; i < size; i++) {
        printf("%.2f ", data[i]);
    }
    printf("\n");
}
```

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 declarations
```

```c
double compute_total_thrust(const double *stages, int size);
void update_stage_thrust(double *stages, int stage, double new_thrust);


int main() {
    int size = 5;  // Number of stages
    double stages[] = {150.0, 200.0, 250.0, 300.0, 350.0};  // Initial thrust for each stage


    // Compute total thrust
    double total_thrust = compute_total_thrust(stages, size);
    printf("Total Thrust: %.2f N\n", total_thrust);


    // Update thrust for the second stage (index 1)
    update_stage_thrust(stages, 1, 220.0);


    // Compute total thrust after updating the second stage
    total_thrust = compute_total_thrust(stages, size);
    printf("Total Thrust after update: %.2f N\n", total_thrust);


    return 0;
}


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

```
        total += stages[i];  // Add the thrust for each stage
    }
    return total;
}


// Function to update thrust for a specific stage
void update_stage_thrust(double *stages, int stage, double new_thrust) {
    if (stage >= 0 && stage < 5) {  // Ensure the stage index is valid
        stages[stage] = new_thrust;  // Update the thrust for the specified stage
    } else {
        printf("Error: Invalid stage index.\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 declarations
void compute_stress_distribution(const double *forces, double *stress, int size);
void display_stress(const double *stress, int size);


int main() {
```

```
    int size = 5;  // Number of sections on the wing

    double forces[] = {1000.0, 1200.0, 1100.0, 900.0, 850.0};  // Forces applied at each wing
section (in Newtons)

    double stress[size];  // Array to store computed stress values



    // Compute stress distribution based on forces

    compute_stress_distribution(forces, stress, size);



    // Display the stress values

    display_stress(stress, size);



    return 0;

}



// Function to compute stress distribution based on forces

void compute_stress_distribution(const double *forces, double *stress, int size) {

    const double wing_area = 50.0;  // Assume a constant area for simplicity (in square meters)

    const double stress_factor = 0.1;  // Example stress factor (in Pa per Newton)



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

        // Stress at each section is calculated using the formula: stress = force / area *
stress_factor

        stress[i] = (forces[i] / wing_area) * stress_factor;

    }

}
```

```
// Function to display the computed stress distribution

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

    printf("Stress Distribution (Pa) for each wing section:\n");

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

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

    }

}
```

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>

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

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

int main() {

    int size = 5;  // Number of initial waypoints

    double waypoints[10][2] = {  // 10 waypoints (x, y) coordinates

        {0.0, 0.0},

        {1.0, 2.0},

        {2.0, 4.0},

        {4.0, 5.0},

        {6.0, 7.0}

    };
```

```
    // Optimize the initial path
    double total_distance = optimize_path((const double *)waypoints, size);
    printf("Total path length before optimization: %.2f units\n", total_distance);

    // Adding a new waypoint
    add_waypoint((double *)waypoints, &size, 7.0, 8.0);
    total_distance = optimize_path((const double *)waypoints, size);
    printf("Total path length after adding a waypoint: %.2f units\n", total_distance);

    return 0;
}

// Function to calculate the total path length between waypoints
double optimize_path(const double *waypoints, int size) {
    double total_distance = 0.0;
    for (int i = 1; i < size; i++) {
        // Calculate distance between consecutive waypoints using Euclidean distance
        double x1 = waypoints[(i - 1) * 2];
        double y1 = waypoints[(i - 1) * 2 + 1];
        double x2 = waypoints[i * 2];
        double y2 = waypoints[i * 2 + 1];

        double distance = sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
        total_distance += distance;
```

```
    }

    return total_distance;

}



// Function to add a new waypoint to the path

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

    waypoints[*size * 2] = x;       // x-coordinate

    waypoints[*size * 2 + 1] = y;   // y-coordinate

    (*size)++;                      // Increment the size of the path

}
```

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 declarations

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

void normalize_quaternion(double *quaternion);



int main() {

    // Quaternion representation: [w, x, y, z]
```

```
    double quaternion[4] = {1.0, 0.5, 0.5, 0.5}; // Example quaternion for attitude
control

    double new_attitude[4];  // Array to store the updated attitude (quaternion)



    // Normalize the quaternion

    normalize_quaternion(quaternion);



    // Update the attitude using the quaternion

    update_attitude(quaternion, new_attitude);



    // Display the updated attitude (quaternion)

    printf("Updated Attitude (Quaternion): [%.2f, %.2f, %.2f, %.2f]\n",

           new_attitude[0], new_attitude[1], new_attitude[2], new_attitude[3]);



    return 0;

}



// Function to normalize a quaternion (make sure its magnitude is 1)

void normalize_quaternion(double *quaternion) {

    double magnitude = sqrt(quaternion[0] * quaternion[0] + quaternion[1] *
quaternion[1] +

                            quaternion[2] * quaternion[2] + quaternion[3] * quaternion[3]);



    // Normalize by dividing each component by the magnitude

    quaternion[0] /= magnitude;

    quaternion[1] /= magnitude;
```

```
    quaternion[2] /= magnitude;

    quaternion[3] /= magnitude;

}


// Function to update the satellite's attitude based on the current quaternion

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

    // Example function: here we simply copy the quaternion to the new attitude

    // In a real application, this function would involve applying the quaternion rotation

    // to the satellite's current orientation or attitude.


    for (int i = 0; i < 4; i++) {

        new_attitude[i] = quaternion[i];  // For simplicity, we just copy the quaternion

    }

}
```

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>

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

void display_temperatures(const double *temperatures, int size);
```

```
int main() {

    // Example material properties (e.g., thermal conductivity, ambient temperature)

    double material_properties[2] = {50.0, 300.0}; // Thermal conductivity = 50,
Ambient temperature = 300 K

    int size = 5; // Number of temperature points along the material

    double temperatures[5] = {350.0, 340.0, 330.0, 320.0, 310.0}; // Initial
temperatures at each point



    // Simulate heat transfer

    simulate_heat_transfer(material_properties, temperatures, size);



    // Display the updated temperature distribution

    display_temperatures(temperatures, size);



    return 0;

}



// Function to simulate heat transfer across the material

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

    double conductivity = material_properties[0]; // Thermal conductivity

    double ambient_temp = material_properties[1]; // Ambient temperature



    // Simulate heat transfer: Here, we apply a simple model for heat diffusion

    // In reality, you would use a more complex model such as finite difference method
(FDM)

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

```
        temperatures[i] = (temperatures[i-1] + temperatures[i+1]) / 2.0 * conductivity / (ambient_temp);

    }



    // Boundary conditions (assuming the ends are exposed to ambient temperature)

    temperatures[0] = ambient_temp;  // Leftmost temperature point

    temperatures[size - 1] = ambient_temp;  // Rightmost temperature point

}



// 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 K\n", i, temperatures[i]);

    }

}
```

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>

double compute_efficiency(const double *fuel_data, int size);

void update_fuel_data(double *fuel_data, int interval, double consumption);
```

```
int main() {
    // Example fuel consumption data (in liters per time interval)
    int size = 5; // Number of intervals
    double fuel_data[5] = {100.0, 150.0, 120.0, 130.0, 110.0}; // Initial fuel consumption values (in liters)

    // Calculate the overall fuel efficiency
    double efficiency = compute_efficiency(fuel_data, size);
    printf("Overall fuel efficiency: %.2f liters/interval\n", efficiency);

    // Update the fuel consumption data for a specific interval
    update_fuel_data(fuel_data, 2, 140.0); // Update fuel consumption at interval 2
    printf("Updated fuel data:\n");
    for (int i = 0; i < size; i++) {
        printf("Interval %d: %.2f liters\n", i + 1, fuel_data[i]);
    }

    return 0;
}

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

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

```
        total_fuel += fuel_data[i]; // Sum up the fuel consumption over all intervals
    }

    return total_fuel / size; // Return the average fuel consumption as efficiency
}

// Function to update fuel data for a specific time interval
void update_fuel_data(double *fuel_data, int interval, double consumption) {
    if (interval >= 0 && interval < 5) {
        fuel_data[interval] = consumption; // Update fuel consumption at the specified interval
    } else {
        printf("Invalid interval\n");
    }
}
```

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>
double compute_link_budget(const double *parameters, int size);
void update_parameters(double *parameters, int index, double value);

int main() {
    // Parameters: {P_t, G_t, G_r, L_fs, L_other}
```

```
    // Example values in dB (e.g., transmit power, antenna gains, losses)
    int size = 5; // Number of parameters
    double parameters[5] = {30.0, 15.0, 15.0, 100.0, 2.0}; // Transmit power, gains, losses

    // Calculate the 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., changing the transmit power)
    update_parameters(parameters, 0, 32.0); // Update P_t (transmit power)
    printf("Updated Parameters:\n");
    for (int i = 0; i < size; i++) {
        printf("Parameter %d: %.2f dB\n", i, parameters[i]);
    }

    // Recalculate the link budget after the update
    link_budget = compute_link_budget(parameters, size);
    printf("Updated Link Budget: %.2f dB\n", link_budget);

    return 0;
}

// Function to compute the total link budget (in dB)
double compute_link_budget(const double *parameters, int size) {
    // Assuming parameters are in order: [P_t, G_t, G_r, L_fs, L_other]
```

```
    double P_t = parameters[0]; // Transmit power
    double G_t = parameters[1]; // Transmit antenna gain
    double G_r = parameters[2]; // Receive antenna gain
    double L_fs = parameters[3]; // Free space loss
    double L_other = parameters[4]; // Other losses

    // Link budget formula: Link Budget = P_t + G_t + G_r - L_fs - L_other
    return P_t + G_t + G_r - L_fs - L_other;
}

// Function to update a specific parameter in the array
void update_parameters(double *parameters, int index, double value) {
    if (index >= 0 && index < 5) {
        parameters[index] = value; // Update the specified parameter
    } else {
        printf("Invalid parameter index\n");
    }
}
```

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>
```

```
#define SAMPLE_SIZE 100  // Number of acceleration data points

#define TURBULENCE_THRESHOLD 1.0  // Threshold for detecting turbulence (in m/s^2 or any suitable unit)

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

void log_turbulence(double *turbulence_log, const double *detection_output, int size);

void print_turbulence_log(const double *log, int size);



int main() {

    // Example acceleration data (in m/s^2 or any suitable unit)

    double accelerations[SAMPLE_SIZE] = {1.3,-2.9,-3,5.6,5.7,5.9,-1.5,4,5,6,6.2};

    double detection_output[SAMPLE_SIZE] = {0};  // Store turbulence detection results (1.0 for turbulence)

    double turbulence_log[SAMPLE_SIZE] = {0};  // Store indices of turbulence events



    // Detect turbulence in the acceleration data

    detect_turbulence(accelerations, SAMPLE_SIZE, detection_output);



    // Log detected turbulence events

    log_turbulence(turbulence_log, detection_output, SAMPLE_SIZE);



    // Print the log of turbulence events (indices)

    print_turbulence_log(turbulence_log, SAMPLE_SIZE);



    return 0;
```

```
}
// Function to detect turbulence based on acceleration data
void detect_turbulence(const double *accelerations, int size, double *output) {
    for (int i = 1; i < size - 1; i++) {
        // Detect sudden changes or spikes in acceleration
        if (fabs(accelerations[i] - accelerations[i - 1]) > TURBULENCE_THRESHOLD ||
            fabs(accelerations[i] - accelerations[i + 1]) > TURBULENCE_THRESHOLD) {
            output[i] = 1.0;  // Mark as turbulence
        } else {
            output[i] = 0.0;  // No turbulence
        }
    }
}

// Function to log turbulence events
void log_turbulence(double *turbulence_log, const double *detection_output, int size) {
    int log_index = 0;
    for (int i = 0; i < size; i++) {
        if (detection_output[i] == 1.0) {
            turbulence_log[log_index++] = i;  // Store the index of turbulence event
        }
    }
}

// Function to print the detected turbulence indices
```

```
void print_turbulence_log(const double *log, int size) {
    printf("Turbulence detected at indices: ");
    for (int i = 0; i < size; i++) {
        if (log[i] != 0) {
            printf("%d ", (int)log[i]);
        }
    }
    printf("\n");
}
```