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

 void calculate_trajectory(const double *parameters, double *trajectory, int size) {
     // parameters[0] = initial velocity, parameters[1] = angle in degrees double
     velocity = parameters[0];
     double angle_deg = parameters[1];
     double angle_rad = angle_deg * (M_PI / 180.0);  // Convert angle to radians

     double time_of_flight = (2 * velocity * sin(angle_rad)) / GRAVITY;
     double time_interval = time_of_flight / (size - 1);  // Divide flight time into ’size’ intervals

     // Calculate trajectory points (x, y, z) for
     (int i = 0; i < size; ++i) {
         double t = i * time_interval;  // Current time
         double x = velocity * cos(angle_rad) * t;  // Horizontal distance
         double y = velocity * sin(angle_rad) * t - 0.5 * GRAVITY * t * t;  // Vertical distance

         trajectory[2 * i] = x;  // x-coordinate
         trajectory[2 * i + 1] = y; // y-coordinate
     }

 }


 void print_trajectory(const double *trajectory, int size) { for
     (int i = 0; i < size; ++i) {
         printf("Point %d: (x = %.2f, y = %.2f)\n", i, trajectory[2 * i], trajectory[2 * i + 1]);
     }

 }


 int main() {
     double parameters[2] = {50.0, 45.0};  // Initial velocity (m/s), angle (degrees)
     int size = 10;  // Number of trajectory points

     double trajectory[2 * size];  // Array to store x, y points for each time step
```

```
    // Calculate the trajectory
    calculate_trajectory(parameters, trajectory, size);

    // Print the trajectory
    print_trajectory(trajectory, size);

    return 0;
}
```

2. 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 to update the position based on velocity
void update_position(const double *velocity, double *position, int size) {
    // Assuming size is 3, for a 3D vector (x, y, z) for
    (int i = 0; i < size; i++) {
        position[i] += velocity[i]; // Update the position: position = position + velocity
    }

}


// Function to simulate the orbit over a specified number of steps
void simulate_orbit(const double *initial_conditions, double *positions, int steps) {
    // Initial conditions: {x, y, z, vx, vy, vz}
    double position[3] = {initial_conditions[0], initial_conditions[1], initial_conditions[2]}; double
    velocity[3] = {initial_conditions[3], initial_conditions[4], initial_conditions[5]};

    // Array to store the positions over time (output) for
    (int step = 0; step < steps; step++) {
        // Store the current position in the positions array (flattens the 3D vector) positions[step * 3
        + 0] = position[0];
        positions[step * 3 + 1] = position[1];
        positions[step * 3 + 2] = position[2];

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

}


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

    // Number of steps for simulation
```

```
    int steps = 10;

    // Array to store the positions at each step (flattens the 3D vector over time) double
    positions[steps * 3];

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

    // Print the positions over time
    printf("Positions over time:\n");
    for (int step = 0; step < steps; step++) {
        printf("Step %d: (%f, %f, %f)\n", step, positions[step * 3 + 0], positions[step * 3 + 1], positions[step * 3 + 2]);
    }

    return 0;
}
```

3. 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>

#define PID_HISTORY_SIZE 3  // For storing the last 3 errors (Proportional, Integral, Derivative)

// Function to compute PID control output based on error values and gains
double compute_pid(const double *errors, int size, const double *gains) {
    if (size != PID_HISTORY_SIZE) {
        printf("Error: Array size mismatch.\n"); return -
        1;
    }


    // Proportional, Integral, and Derivative calculations double
    proportional = errors[0];  // P error (current error) double
    integral = 0;  // I error (sum of previous errors) double
    derivative = 0;  // D error (change in error)

    for (int i = 1; i < size; i++) {
        integral += errors[i];  // Accumulate error for integral term
        derivative = errors[i] - errors[i - 1];  // Difference between current and previous error for derivative
term
    }
```

```
    // PID formula: P + I + D
    double pid_output = gains[0] * proportional + gains[1] * integral + gains[2] * derivative;

    return pid_output;
}

// Function to update the error array with the latest error value void
update_errors(double *errors, double new_error) {
    // Shift the historical error values to the right for
    (int i = PID_HISTORY_SIZE - 1; i > 0; i--) {
        errors[i] = errors[i - 1];
    }

    // Insert the latest error value at the front errors[0]
    = new_error;
}


int main() {
    double errors[PID_HISTORY_SIZE] = {0.0, 0.0, 0.0};  // Array to store historical errors double
    gains[3] = {1.0, 0.1, 0.01};  // PID gains: P, I, D

    // Simulate receiving new errors update_errors(errors,
    0.5);  // New error: 0.5
    update_errors(errors, 0.4);  // New error: 0.4
    update_errors(errors, 0.3);  // New error: 0.3

    // Compute PID control output
    double output = compute_pid(errors, PID_HISTORY_SIZE, gains);

    // Output the result
    printf("PID Control Output: %f\n", output);

    return 0;
}
```

4. 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 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;  // Simple averaging of sensor data
```

```
    }

}


// Function to calibrate sensor data by applying some offset or scaling factor


void calibrate_data(double *data, int size) { for
    (int i = 0; i < size; i++) {
        // Applying a simple calibration factor, for example, subtracting an offset
        // In real cases, the calibration logic may involve more complex mathematical operations data[i] -=
        1.0;  // Example calibration: subtract 1.0 from each sensor reading
    }

}

int main() {
    int size = 5;  // Example array size

    // Example sensor readings (arrays)
    double sensor1[5] = {10.0, 20.0, 30.0, 40.0, 50.0};
    double sensor2[5] = {12.0, 22.0, 32.0, 42.0, 52.0};



    // Array to store fused data
    double result[5];

    // Fuse sensor data
    fuse_data(sensor1, sensor2, result, size);

    // Print fused data
    printf("Fused Data:\n");

    for (int i = 0; i < size; i++) {
        printf("Result[%d]: %.2f\n", i, result[i]);
    }


    // Calibrate sensor1 data
    calibrate_data(sensor1, size);

    // Print calibrated data printf("\nCalibrated
    Sensor1 Data:\n"); for (int i = 0; i < size;
    i++) {
        printf("Sensor1[%d]: %.2f\n", i, sensor1[i]);
    }


    return 0;
}
```

5. 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 *averages) { double
    temp_sum = 0.0, wind_sum = 0.0, pressure_sum = 0.0;

    for (int i = 0; i < size; i++) {
        temp_sum += data[i * 3]; wind_sum
        += data[i * 3 + 1]; pressure_sum +=
        data[i * 3 + 2];
    }

    averages[0] = temp_sum / size; averages[1]
    = wind_sum / size; averages[2] =
    pressure_sum / size;
}
```

```
// Main function int
main() {
    const int hours = 5; // Simplified to 5 hours double
    weather_data[hours * 3] = {
        15.0, 3.0, 1012.0, // Hour 1: Temp, Wind, Pressure
        16.0, 3.5, 1011.8, // Hour 2
        17.0, 4.0, 1011.6, // Hour 3
        18.0, 4.5, 1011.4, // Hour 4
        19.0, 5.0, 1011.2  // Hour 5
    };
    double daily_averages[3];

    // Calculate daily averages calculate_daily_averages(weather_data,
    hours, daily_averages);

    // Print daily averages printf("Daily
    Averages:\n");
    printf("Temperature: %.2f°C\n", daily_averages[0]);
    printf("Wind Speed: %.2f m/s\n", daily_averages[1]);



    printf("Pressure: %.2f hPa\n", daily_averages[2]);

    return 0;
}
```

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 MAX_FLIGHTS 5

// Function to add a new flight
void add_flight(int *flight_ids, double *altitudes, int *flight_count, int id, double altitude) {
    if (*flight_count < MAX_FLIGHTS) {
        flight_ids[*flight_count] = id;
        altitudes[*flight_count] = altitude;
        (*flight_count)++;
        printf("Flight %d added.\n", id);
    } else {
        printf("Error: Maximum flight limit reached.\n");
    }
}


// Function to display all flights
void display_flights(const int *flight_ids, const double *altitudes, int flight_count) {
    printf("Active Flights:\n");
    for (int i = 0; i < flight_count; i++) {
        printf("ID: %d, Altitude: %.2f\n", flight_ids[i], altitudes[i]);
```

```
}
```

```
}


int main() {
    int flight_ids[MAX_FLIGHTS];    // Array to store flight IDs
    double altitudes[MAX_FLIGHTS];  // Array to store altitudes
    int flight_count = 0;           // Number of flights

    // Adding flights
    add_flight(flight_ids, altitudes, &flight_count, 101, 35000.0);
    add_flight(flight_ids, altitudes, &flight_count, 102, 30000.0);

    // Display flights
    display_flights(flight_ids, altitudes, flight_count);

    return 0;
}
```

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>

#define DATA_SIZE 5
```

```c
// Function to analyze telemetry data (e.g., compute the average)
void analyze_telemetry(const double *data, int size) {
    double sum = 0.0;
    for (int i = 0; i < size; i++) {
        sum += data[i];
    }
    double average = sum / size;
    printf("Average telemetry value: %.2f\n", average);
}


// Function to filter out outliers (simple example, removing values outside 1.5 * average)
void filter_outliers(double *data, int *size) {
    double sum = 0.0;
    for (int i = 0; i < *size; i++) {
        sum += data[i];
    }
    double average = sum / *size;
    double threshold = 1.5 * average;

    int new_size = 0;
    for (int i = 0; i < *size; i++) {
        if (data[i] < threshold) {
            data[new_size++] = data[i];
        }
    }
    *size = new_size;  // Update size after removing outliers
    printf("Outliers filtered. New data size: %d\n", *size);
```

```
}

int main() {
    double telemetry_data[DATA_SIZE] = {10.0, 12.0, 100.0, 15.0, 9.0};  // Example telemetry data
    int data_size = DATA_SIZE;

    // Analyze the telemetry data
    analyze_telemetry(telemetry_data, data_size);

    // Filter out outliers
    filter_outliers(telemetry_data, &data_size);

    // Display the remaining data
    printf("Filtered telemetry data:\n");
    for (int i = 0; i < data_size; i++) {
        printf("%.2f ", telemetry_data[i]);
    }
    printf("\n");

    return 0;
}
```

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>


#define STAGES 3


// Function to compute total thrust from 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];  // Accumulate thrust for each stage
    }
    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 < STAGES) {
        stages[stage] = new_thrust;  // Update the thrust for the specified stage
        printf("Stage %d thrust updated to %.2f.\n", stage + 1, new_thrust);
    } else {
        printf("Error: Invalid stage number.\n");
    }
}



int main() {
    double thrust_stages[STAGES] = {500.0, 800.0, 1200.0};  // Example thrust values for 3 stages
```

```c
    // Display total thrust
    double total_thrust = compute_total_thrust(thrust_stages, STAGES);
    printf("Total thrust from all stages: %.2f\n", total_thrust);

    // Update thrust for stage 2
    update_stage_thrust(thrust_stages, 1, 900.0);  // Update 2nd stage thrust

    // Display total thrust after update
    total_thrust = compute_total_thrust(thrust_stages, STAGES);
    printf("Total thrust after update: %.2f\n", total_thrust);

    return 0;
}
```

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.

```c
#include <stdio.h>

#define WING_SECTIONS 5

// Function to compute stress distribution based on applied forces
```

```c
void compute_stress_distribution(const double *forces, double *stress, int size) {
    for (int i = 0; i < size; i++) {
        // Simple stress computation: Stress = Force / Area (Area is constant here for simplicity)
        stress[i] = forces[i] / 10.0;  // Example: Assume a constant area of 10
    }
}


// Function to display stress distribution
void display_stress(const double *stress, int size) {
    printf("Stress Distribution (N/m^2):\n");
    for (int i = 0; i < size; i++) {
        printf("Section %d: %.2f\n", i + 1, stress[i]);
    }
}


int main() {
    double forces[WING_SECTIONS] = {100.0, 200.0, 150.0, 250.0, 300.0};  // Applied forces at each wing section
    double stress[WING_SECTIONS];  // Array to store computed stress values

    // Compute stress distribution
    compute_stress_distribution(forces, stress, WING_SECTIONS);

    // Display stress distribution
    display_stress(stress, WING_SECTIONS);

    return 0;
}
```

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.

```c
#include <stdio.h>

#include <math.h>


#define MAX_WAYPOINTS 10


// Function to calculate distance between two points (waypoints)

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

    return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));

}



// Function to optimize the path (simplified version)

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

    double total_distance = 0.0;

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

        total_distance += calculate_distance(waypoints[2*i], waypoints[2*i+1], waypoints[2*(i+1)], waypoints[2*(i+1)+1]);

    }

    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[2 * (*size)] = x;
        waypoints[2 * (*size) + 1] = y;
        (*size)++;
        printf("Added waypoint (%.2f, %.2f)\n", x, y);
    } else {
        printf("Error: Maximum waypoints reached.\n");
    }
}


int main() {
    double waypoints[2 * MAX_WAYPOINTS];  // Array to store waypoints (x, y) for each point
    int size = 0;  // Number of waypoints added


    // Add some initial waypoints
    add_waypoint(waypoints, &size, 0.0, 0.0);
    add_waypoint(waypoints, &size, 1.0, 1.0);
    add_waypoint(waypoints, &size, 4.0, 5.0);
    add_waypoint(waypoints, &size, 7.0, 8.0);


    // Calculate and display optimized path distance
    double total_distance = optimize_path(waypoints, size);
    printf("Total path length: %.2f\n", total_distance);


    return 0;
}
```

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>


#define QUATERNION_SIZE 4


// Function to normalize a quaternion

void normalize_quaternion(double *quaternion) {

    double magnitude = 0.0;

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

        magnitude += quaternion[i] * quaternion[i];

    }

    magnitude = sqrt(magnitude);


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

        quaternion[i] /= magnitude;  // Normalize each component

    }

}


// Function to update satellite's attitude
```

```
void update_attitude(double *quaternion, double *new_attitude) {
    for (int i = 0; i < QUATERNION_SIZE; i++) {
        new_attitude[i] = quaternion[i];  // Simply copy quaternion to new attitude
    }
}


int main() {
    double quaternion[QUATERNION_SIZE] = {1.0, 2.0, 3.0, 4.0};  // Example
quaternion
    double new_attitude[QUATERNION_SIZE];  // Array to store updated attitude

    // Normalize the quaternion
    normalize_quaternion(quaternion);

    // Display the normalized quaternion
    printf("Normalized quaternion: ");
    for (int i = 0; i < QUATERNION_SIZE; i++) {
        printf("%.2f ", quaternion[i]);
    }
    printf("\n");

    // Update attitude based on quaternion
    update_attitude(quaternion, new_attitude);

    // Display the updated attitude
    printf("Updated attitude: ");
    for (int i = 0; i < QUATERNION_SIZE; i++) {
        printf("%.2f ", new_attitude[i]);
    }
```

```
    printf("\n");

    return 0;
}
```

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>

#define SIZE 5  // Number of temperature points

// Function to simulate heat transfer across the material
void simulate_heat_transfer(const double *material_properties, double *temperatures, int size) {
    for (int i = 0; i < size; i++) {
        temperatures[i] += material_properties[i] * 0.1;  // Apply heat transfer based on properties (simplified)
    }
}

// Function to display temperature distribution
void display_temperatures(const double *temperatures, int size) {
```

```c
    printf("Temperature distribution:\n");
    for (int i = 0; i < size; i++) {
        printf("Point %d: %.2f\n", i + 1, temperatures[i]);
    }
}

int main() {
    double material_properties[SIZE] = {2.0, 1.5, 3.0, 2.5, 4.0};  // Simplified material properties
    double temperatures[SIZE] = {300.0, 310.0, 320.0, 330.0, 340.0};  // Initial temperatures

    // Simulate heat transfer
    simulate_heat_transfer(material_properties, temperatures, SIZE);

    // Display the updated temperature distribution
    display_temperatures(temperatures, SIZE);

    return 0;
}
```

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>

#define TIME_INTERVALS 5  // Number of time intervals

// Function to compute 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];  // Add fuel consumption at each interval
    }
    return total_fuel / size;  // Simple efficiency: average fuel consumption
}


// Function to update fuel consumption for a specific interval
void update_fuel_data(double *fuel_data, int interval, double consumption) {
    if (interval >= 0 && interval < TIME_INTERVALS) {
        fuel_data[interval] = consumption;  // Update the fuel consumption for the
interval
        printf("Fuel data updated for interval %d: %.2f\n", interval + 1, consumption);
    } else {
        printf("Invalid interval.\n");
    }
}


int main() {
    double fuel_data[TIME_INTERVALS] = {50.0, 60.0, 55.0, 65.0, 70.0};  // Fuel
consumption at different intervals

    // Compute and display the overall fuel efficiency
```

```c
    double efficiency = compute_efficiency(fuel_data, TIME_INTERVALS);
    printf("Overall fuel efficiency: %.2f\n", efficiency);

    // Update fuel consumption for a specific interval (example: interval 2)
    update_fuel_data(fuel_data, 2, 58.0);

    // Recompute and display the updated fuel efficiency
    efficiency = compute_efficiency(fuel_data, TIME_INTERVALS);
    printf("Updated fuel efficiency: %.2f\n", efficiency);

    return 0;
}
```

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.

```c
#include <stdio.h>

#define PARAMETER_COUNT 5  // Number of parameters (e.g., power, losses, etc.)

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

```c
    for (int i = 0; i < size; i++) {
        total_link_budget += parameters[i];  // Sum up the parameters
    }
    return total_link_budget;  // Return the total link budget
}


// Function to update a specific parameter in the array
void update_parameters(double *parameters, int index, double value) {
    if (index >= 0 && index < PARAMETER_COUNT) {
        parameters[index] = value;  // Update the specified parameter
        printf("Parameter %d updated to %.2f\n", index + 1, value);
    } else {
        printf("Invalid index. Parameter not updated.\n");
    }
}


int main() {
    double parameters[PARAMETER_COUNT] = {100.0, -30.0, -10.0, 5.0, -20.0};  //
Example communication parameters

    // Compute the total link budget
    double link_budget = compute_link_budget(parameters, PARAMETER_COUNT);
    printf("Total link budget: %.2f dB\n", link_budget);

    // Update a specific parameter (example: index 2, value -15.0)
    update_parameters(parameters, 2, -15.0);

    // Recompute the total link budget after the update
    link_budget = compute_link_budget(parameters, PARAMETER_COUNT);
```

```
    printf("Updated total link budget: %.2f dB\n", link_budget);

    return 0;
}
```

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 SIZE 5  // Number of acceleration data points

// Function to detect turbulence based on frequency analysis
void detect_turbulence(const double *accelerations, int size, double *output) {
    for (int i = 0; i < size; i++) {
        // Simplified turbulence detection: if acceleration exceeds a threshold,
turbulence detected
        if (accelerations[i] > 2.0) {
            output[i] = 1.0;  // Turbulence detected
        } else {
            output[i] = 0.0;  // No turbulence
        }
```

```c
    }
}


// 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
        } else {
            turbulence_log[i] = 0.0;  // No turbulence logged
        }
    }
}


// Function to display the turbulence log
void display_log(const double *log, int size) {
    printf("Turbulence Log:\n");
    for (int i = 0; i < size; i++) {
        printf("Time Interval %d: %s\n", i + 1, log[i] == 1.0 ? "Turbulence Detected" : "No Turbulence");
    }
}


int main() {
    double accelerations[SIZE] = {1.5, 2.5, 1.2, 3.1, 0.9};  // Example acceleration data (m/s^2)
    double detection_output[SIZE];  // Array to store turbulence detection results
    double turbulence_log[SIZE];  // Array to store logged turbulence events
```

```
    // Detect turbulence
    detect_turbulence(accelerations, SIZE, detection_output);

    // Log detected turbulence events
    log_turbulence(turbulence_log, detection_output, SIZE);

    // Display the turbulence log
    display_log(turbulence_log, SIZE);

    return 0;
}
```