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

#define MAX_DATA_SIZE 1000
#define EPSILON 1e-6

// 算法S所需的系数表
typedef struct {
    int nu;      // 自由度
    double alpha; // 限系数
    double beta;  // 修正系数
} AlgorithmSFactor;

// 算法S系数表（表A1）
static const AlgorithmSFactor s_factors[] = {
    {1, 1.645, 1.097},
    {2, 1.517, 1.054},
    {3, 1.444, 1.039},
    {4, 1.395, 1.032},
    {5, 1.359, 1.027},
    {6, 1.332, 1.024},
    {7, 1.310, 1.021},
    {8, 1.292, 1.019},
    {9, 1.277, 1.018},
    {10, 1.264, 1.017}
};

// 比较函数，用于qsort排序
int compare_double(const void *a, const void *b) {
    double da = *(const double*)a;
    double db = *(const double*)b;
    return (da > db) - (da < db);
}

// 计算中位数
double median(double data[], int n) {
    if (n == 0) return 0.0;

    if (n % 2 == 1) {
        return data[n/2];
    } else {
        return (data[n/2-1] + data[n/2]) / 2.0;
    }
}

// 计算四分位数
void calculate_quartiles(double data[], int n, double *q1, double *q3) {
    if (n < 4) {
        *q1 = data[0];
        *q3 = data[n-1];
        return;
    }
```

```
    // 计算四分位数位置
    double q1_pos = (n + 1) / 4.0;
    double q3_pos = 3 * (n + 1) / 4.0;

    // 下四分位数Q1
    int q1_low = (int)floor(q1_pos) - 1;
    int q1_high = (int)ceil(q1_pos) - 1;
    if (q1_low < 0) q1_low = 0;
    if (q1_high >= n) q1_high = n - 1;

    if (q1_low == q1_high) {
        *q1 = data[q1_low];
    } else {
        double weight = q1_pos - floor(q1_pos);
        *q1 = data[q1_low] + weight * (data[q1_high] - data[q1_low]);
    }

    // 上四分位数Q3
    int q3_low = (int)floor(q3_pos) - 1;
    int q3_high = (int)ceil(q3_pos) - 1;
    if (q3_low < 0) q3_low = 0;
    if (q3_high >= n) q3_high = n - 1;

    if (q3_low == q3_high) {
        *q3 = data[q3_low];
    } else {
        double weight = q3_pos - floor(q3_pos);
        *q3 = data[q3_low] + weight * (data[q3_high] - data[q3_low]);
    }
}

/**
 * 算法A：计算稳健平均值和稳健标准差
 * 参数：
 *   data: 输入数据数组
 *   n: 数据个数
 *   robust_mean: 输出的稳健平均值
 *   robust_std: 输出的稳健标准差
 */
void algorithm_a(double data[], int n, double *robust_mean, double *robust_std)
{
    if (n <= 0) {
        *robust_mean = 0.0;
        *robust_std = 0.0;
        return;
    }

    // 复制并排序数据
    double *sorted_data = malloc(n * sizeof(double));
    memcpy(sorted_data, data, n * sizeof(double));
    qsort(sorted_data, n, sizeof(double), compare_double);

    // 计算初始值 (A.1) 和 (A.2)
    double x_star = median(sorted_data, n);  // 中位数作为初始稳健平均值

    // 计算初始稳健标准差
    double *abs_deviations = malloc(n * sizeof(double));
    for (int i = 0; i < n; i++) {
```

```
        abs_deviations[i] = fabs(sorted_data[i] - x_star);
    }
    qsort(abs_deviations, n, sizeof(double), compare_double);
    double s_star = 1.483 * median(abs_deviations, n);  // 公式(A.2)

    // 迭代更新
    double prev_x_star, prev_s_star;
    int max_iterations = 100;

    for (int iter = 0; iter < max_iterations; iter++) {
        prev_x_star = x_star;
        prev_s_star = s_star;

        // 计算δ = 1.5 * s* (A.3)
        double delta = 1.5 * s_star;

        // 修正数据 (A.4)
        double *x_modified = malloc(n * sizeof(double));
        for (int i = 0; i < n; i++) {
            if (sorted_data[i] < x_star - delta) {
                x_modified[i] = x_star - delta;
            } else if (sorted_data[i] > x_star + delta) {
                x_modified[i] = x_star + delta;
            } else {
                x_modified[i] = sorted_data[i];
            }
        }

        // 计算新的x* (A.5)
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            sum += x_modified[i];
        }
        x_star = sum / n;

        // 计算新的s* (A.6)
        double sum_sq = 0.0;
        for (int i = 0; i < n; i++) {
            double diff = x_modified[i] - x_star;
            sum_sq += diff * diff;
        }
        s_star = 1.134 * sqrt(sum_sq / (n - 1));

        free(x_modified);

        // 检查收敛性
        if (fabs(x_star - prev_x_star) < EPSILON && fabs(s_star - prev_s_star) <
EPSILON) {
            break;
        }
    }

    *robust_mean = x_star;
    *robust_std = s_star;

    free(sorted_data);
    free(abs_deviations);
}
```

```
/**
 * 算法S：计算标准差的稳健联合值
 * 参数：
 *   data: 输入数据数组（标准差或极差）
 *   n: 数据个数
 *   nu: 自由度
 *   robust_joint_value: 输出的稳健联合值
 */
void algorithm_s(double data[], int n, int nu, double *robust_joint_value) {
    if (n <= 0) {
        *robust_joint_value = 0.0;
        return;
    }

    // 复制并排序数据
    double *sorted_data = malloc(n * sizeof(double));
    memcpy(sorted_data, data, n * sizeof(double));
    qsort(sorted_data, n, sizeof(double), compare_double);

    // 查找对应自由度的系数
    double alpha = 1.645, beta = 1.097;  // 默认值
    int factor_count = sizeof(s_factors) / sizeof(s_factors[0]);

    for (int i = 0; i < factor_count; i++) {
        if (s_factors[i].nu == nu) {
            alpha = s_factors[i].alpha;
            beta = s_factors[i].beta;
            break;
        }
    }

    // 计算初始值 w* = med(wi) (A.7)
    double w_star = median(sorted_data, n);

    // 迭代更新
    double prev_w_star;
    int max_iterations = 100;

    for (int iter = 0; iter < max_iterations; iter++) {
        prev_w_star = w_star;

        // 计算γ = α * w* (A.8)
        double gamma = alpha * w_star;

        // 修正数据 (A.9)
        double *w_modified = malloc(n * sizeof(double));
        for (int i = 0; i < n; i++) {
            if (sorted_data[i] < gamma) {
                w_modified[i] = sorted_data[i];
            } else {
                w_modified[i] = gamma;
            }
        }

        // 计算新的w* (A.10)
        double sum_sq = 0.0;
        for (int i = 0; i < n; i++) {
```

```
            sum_sq += w_modified[i] * w_modified[i];
        }
        w_star = beta * sqrt(sum_sq / n);

        free(w_modified);

        // 检查收敛性
        if (fabs(w_star - prev_w_star) < EPSILON) {
            break;
        }
    }

    *robust_joint_value = w_star;
    free(sorted_data);
}

/**
 * 中位值和标准化四分位距法
 * 参数:
 *   data: 输入数据数组
 *   n: 数据个数
 *   median_val: 输出的中位值
 *   niqr: 输出的标准化四分位距
 */
void median_and_niqr(double data[], int n, double *median_val, double *niqr) {
    if (n <= 0) {
        *median_val = 0.0;
        *niqr = 0.0;
        return;
    }

    // 复制并排序数据
    double *sorted_data = malloc(n * sizeof(double));
    memcpy(sorted_data, data, n * sizeof(double));
    qsort(sorted_data, n, sizeof(double), compare_double);

    // 计算中位值 (A.11)
    *median_val = median(sorted_data, n);

    // 计算四分位数
    double q1, q3;
    calculate_quartiles(sorted_data, n, &q1, &q3);

    // 计算IQR和NIQR (A.12) 和 (A.13)
    double iqr = q3 - q1;
    *niqr = 0.7413 * iqr;

    free(sorted_data);
}

// 打印数组（用于调试）
void print_array(const char* name, double data[], int n) {
    printf("%s: ", name);
    for (int i = 0; i < n; i++) {
        printf("%.4f ", data[i]);
    }
    printf("\n");
}
```

```
// 测试函数
void test_algorithms() {
    printf("=== 稳健统计算法测试 ===\n\n");

    // 测试数据
    double test_data[] = {12.5, 13.1, 12.8, 15.2, 12.9, 13.0, 12.7, 14.8, 13.2,
12.6, 20.0};
    int n = sizeof(test_data) / sizeof(test_data[0]);

    printf("原始数据：\n");
    print_array("数据", test_data, n);
    printf("数据个数：%d\n\n", n);

    // 测试算法A
    printf("1. 算法A（稳健平均值和稳健标准差）：\n");
    double robust_mean, robust_std;
    algorithm_a(test_data, n, &robust_mean, &robust_std);
    printf("   稳健平均值 x*: %.6f\n", robust_mean);
    printf("   稳健标准差 s*: %.6f\n\n", robust_std);

    // 测试中位值和标准化四分位距法
    printf("2. 中位值和标准化四分位距法：\n");
    double median_val, niqr;
    median_and_niqr(test_data, n, &median_val, &niqr);
    printf("   中位值: %.6f\n", median_val);
    printf("   标准化四分位距(NIQR): %.6f\n\n", niqr);

    // 测试算法S（用标准差数据）
    printf("3. 算法S（标准差的稳健联合值）：\n");
    double std_data[] = {0.5, 0.8, 0.6, 1.2, 0.7, 0.9, 0.4, 1.5, 0.8, 0.6};
    int std_n = sizeof(std_data) / sizeof(std_data[0]);
    print_array("标准差数据", std_data, std_n);

    double robust_joint;
    int nu = 5;  // 假设自由度为5
    algorithm_s(std_data, std_n, nu, &robust_joint);
    printf("   自由度 ν = %d\n", nu);
    printf("   稳健联合值 w*: %.6f\n\n", robust_joint);
}

int main() {
    test_algorithms();
    return 0;
}
```

```
#ifndef ROBUST_STATISTICS_H
#define ROBUST_STATISTICS_H

// 函数声明

/**
 * 算法A：计算稳健平均值和稳健标准差
 * 参数：
 *   data: 输入数据数组
 *   n: 数据个数
 *   robust_mean: 输出的稳健平均值
```

```
 *   robust_std: 输出的稳健标准差
 */
void algorithm_a(double data[], int n, double *robust_mean, double *robust_std);

/**
 * 算法S：计算标准差的稳健联合值
 * 参数：
 *   data: 输入数据数组（标准差或极差）
 *   n: 数据个数
 *   nu: 自由度
 *   robust_joint_value: 输出的稳健联合值
 */
void algorithm_s(double data[], int n, int nu, double *robust_joint_value);

/**
 * 中位值和标准化四分位距法
 * 参数：
 *   data: 输入数据数组
 *   n: 数据个数
 *   median_val: 输出的中位值
 *   niqr: 输出的标准化四分位距
 */
void median_and_niqr(double data[], int n, double *median_val, double *niqr);

// 辅助函数
double median(double data[], int n);
void calculate_quartiles(double data[], int n, double *q1, double *q3);
int compare_double(const void *a, const void *b);
void print_array(const char* name, double data[], int n);

#endif // ROBUST_STATISTICS_H
```

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -lm
TARGET = robust_statistics
SOURCES = robust_statistics.c

all: $(TARGET)

$(TARGET): $(SOURCES)
    $(CC) -o $(TARGET) $(SOURCES) $(CFLAGS)

clean:
    rm -f $(TARGET)

run: $(TARGET)
    ./$(TARGET)

.PHONY: all clean run
```