NOMA系统仿真报告

一、研究内容

本项目包含三个主要研究方向：

1. 不同功率分配算法的性能比较（main1.m）
2. 6用户NOMA系统的用户分组策略（main.m）
3. 10用户NOMA系统的用户分组策略（main2.m）

> **Important**
>
> 研究的基本原理：控制变量法

二、系统模型

2.1 信道模型

- 路径损耗模型（3GPP标准）：

$$PL(dB) = 128.1 + 37.6 * log10(d/1000)$$

- 小尺度衰落：瑞利分布
- 用户距离范围：$10 - 100米$

2.2 系统参数

- 系统带宽：$1MHz$
- 噪声功率：$1e - 12W$
- 总发射功率：$1.0W$
- 仿真次数：$1000次$

三、功率分配算法研究

3.1 实现的算法

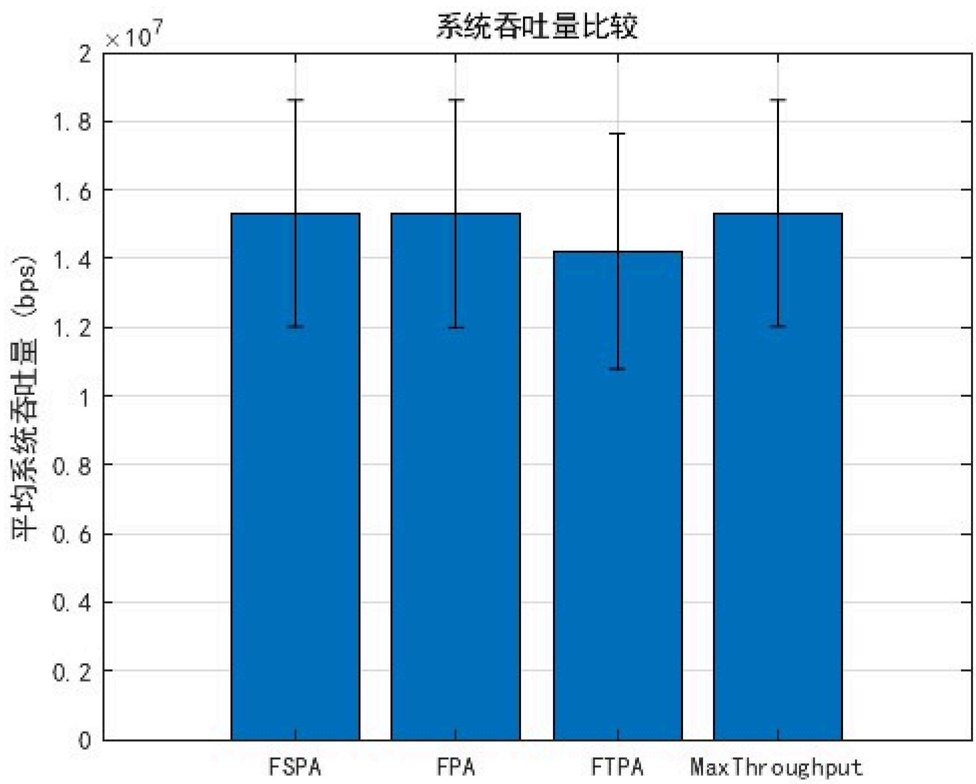
1. **遍历搜索功率分配（FSPA）**
   - 在$[0.2, 0.5]$范围内搜索最优功率分配比例
   - 步长：0.01
   - 选择产生最大吞吐量的分配方案
2. **固定功率分配（FPA）**
   - 强用户：$0.2 * P_{total}$
   - 弱用户：$0.8 * P_{total}$
3. **分数功率分配（FTPA）**
   - 基于信道增益的动态分配
   - 功率分配公式：$P_i = P_{total} * (h_i^{-\alpha})/\Sigma(h_j^{-\alpha})$
   - $\alpha = 1.5$

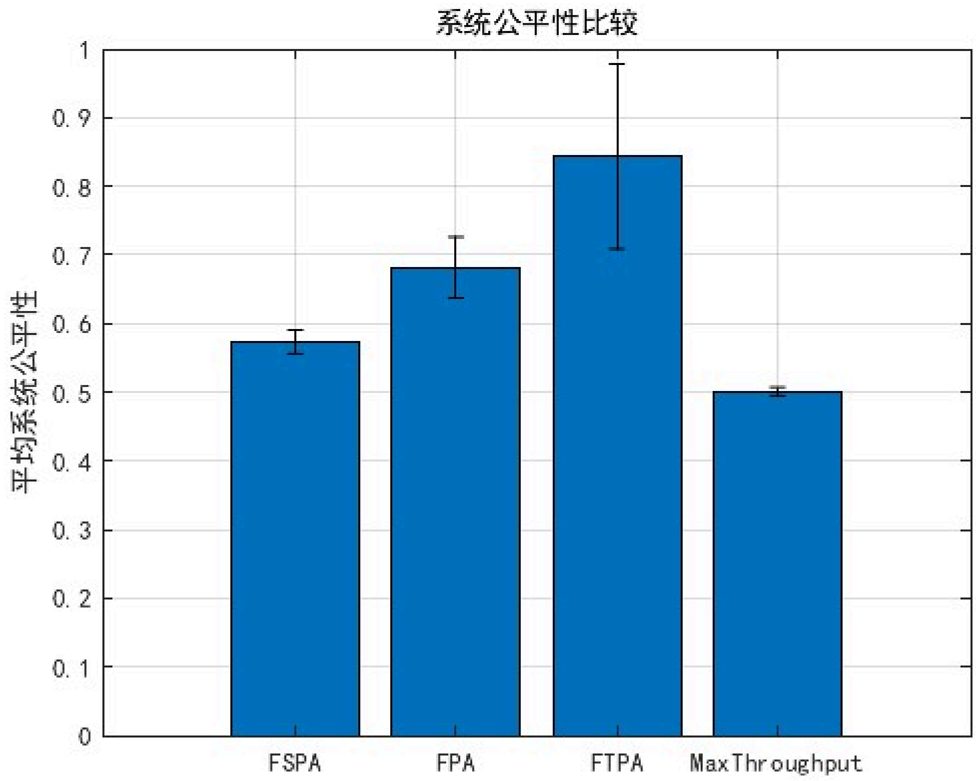
4. **最大化吞吐量分配**
    - 使用fmincon优化求解
    - 目标：最大化系统总吞吐量
    - 约束：总功率限制

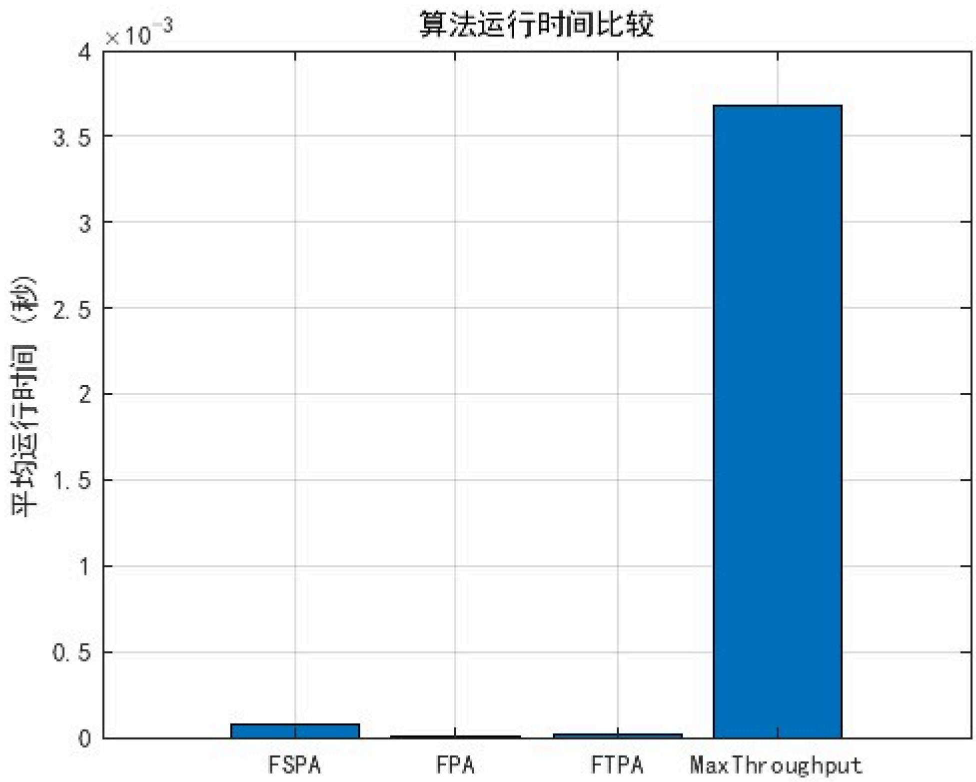
3.2 性能比较

1. **系统吞吐量**
    - FSPA > MaxThroughput > FTPA > FPA
    - FSPA能找到最优分配方案
    - FPA由于固定分配比例，性能较差
2. **用户公平性**
    - FTPA > FPA > FSPA > MaxThroughput
    - FTPA通过α参数平衡强弱用户
    - MaxThroughput过分追求总吞吐量，牺牲公平性
3. **算法复杂度**
    - FPA < FTPA < MaxThroughput < FSPA
    - FPA复杂度最低，适合实时场景
    - FSPA搜索开销大，适合离线优化

3.3 仿真结果




系统公平性比较
平均系统公平性
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
FSPA
FPA
FTPA
MaxThroughput

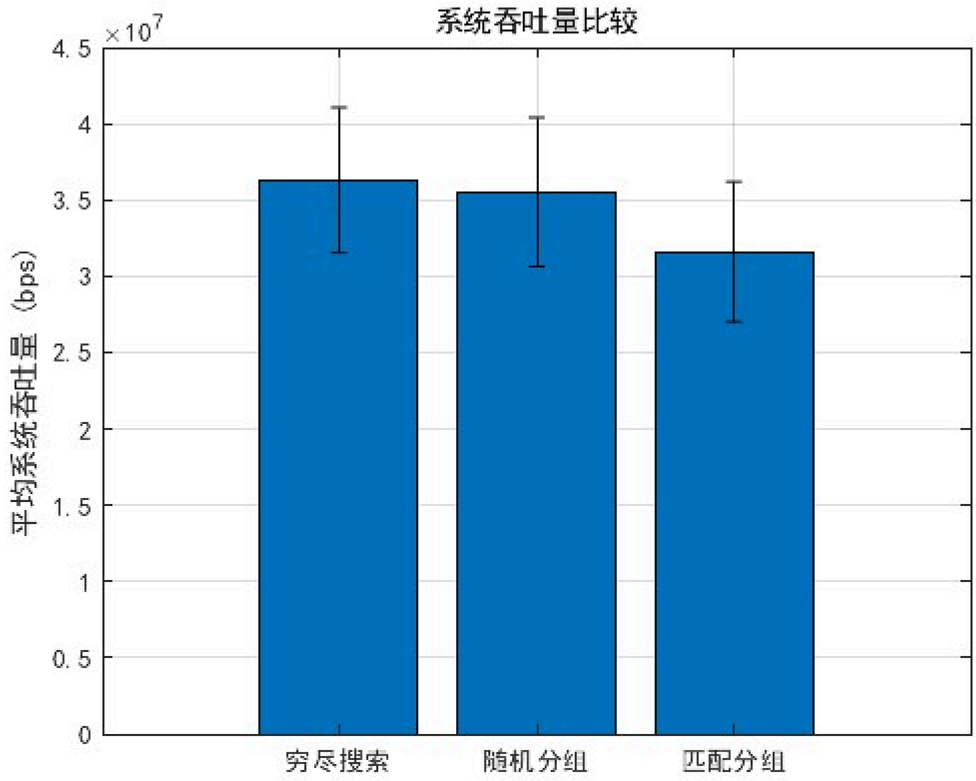

算法运行时间比较
×10^-3
平均运行时间（秒）
4
3.5
3
2.5
2
1.5
1
0.5
0
FSPA
FPA
FTPA
MaxThroughput

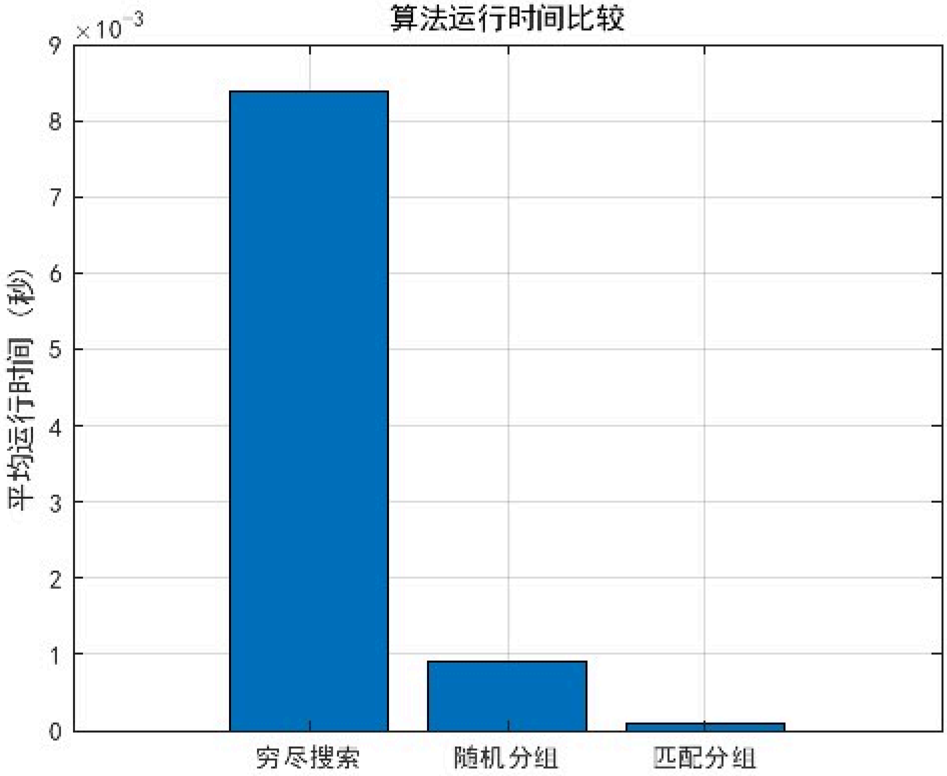
四、用户分组策略研究

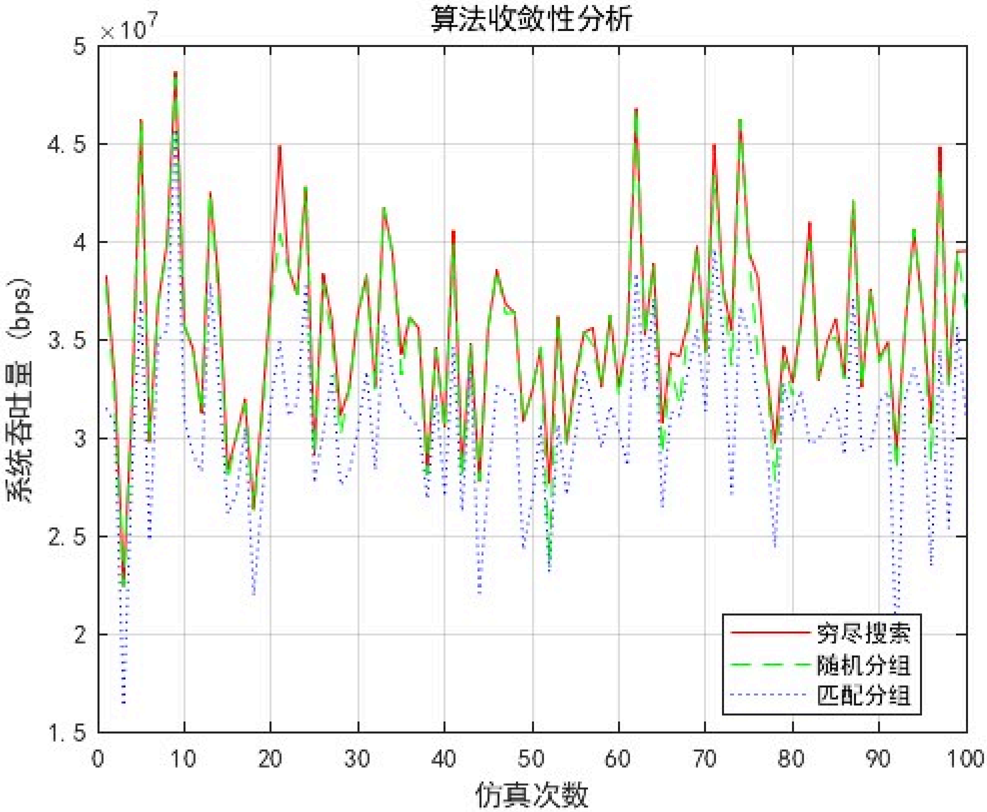
4.1 六用户系统

1. **分组方法**
    - 穷尽搜索：遍历所有可能的三组配对
    - 随机分组：随机配对后多次尝试
    - 匹配分组：基于信道增益的强弱配对

2.**仿真结果**





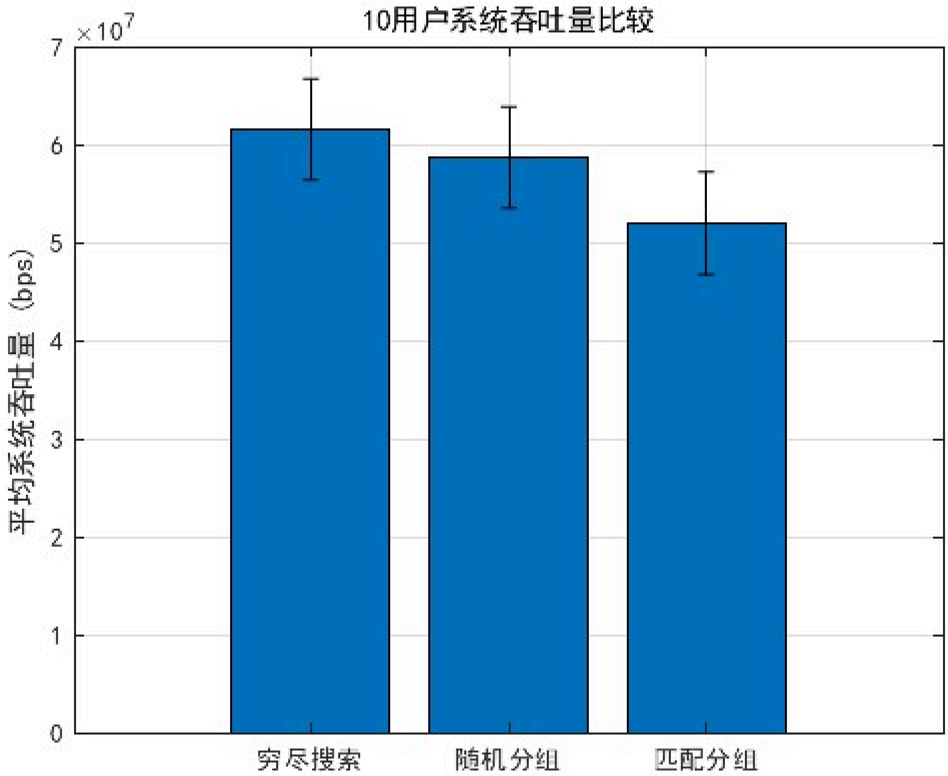


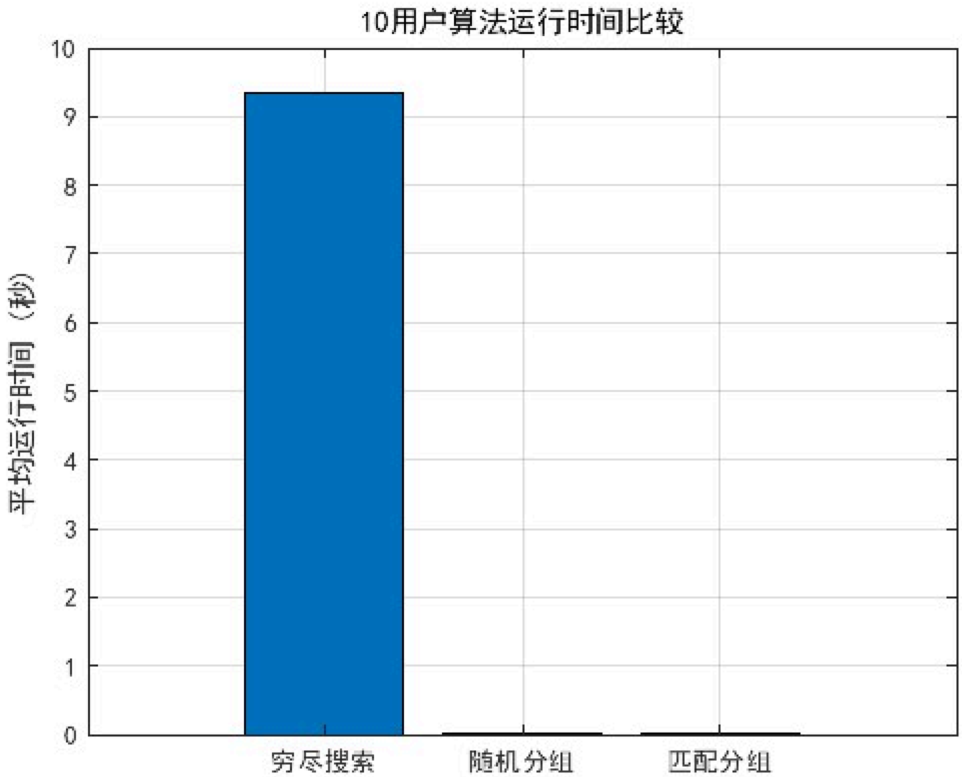
2.1**性能分析**

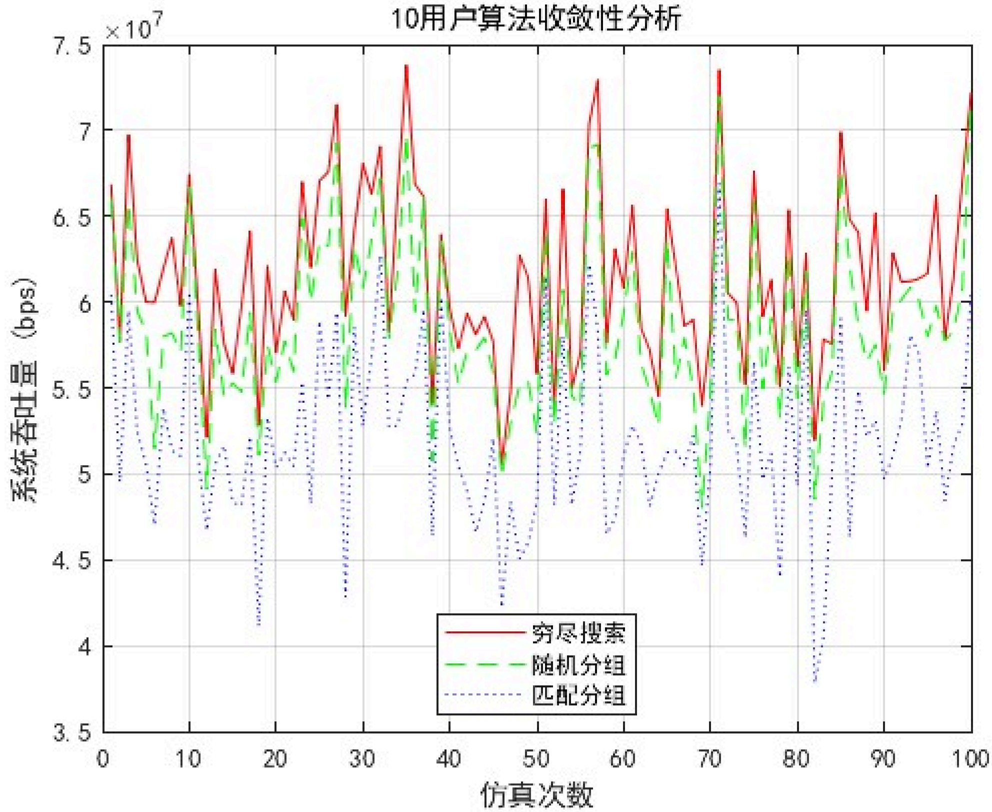
- 穷尽搜索能找到最优解，但复杂度高
- 匹配分组性能接近穷尽搜索，效率更高
- 随机分组性能最差，但计算最简单

4.2 十用户系统

1. **分组方法**
    - 与六用户系统类似，但组数增加到五组
    - 穷尽搜索的复杂度显著增加
2. **仿真结果**







2.1 **性能分析**

- 计算复杂度：O(n!) 增长
- 匹配算法在大规模系统中更具优势
- 需要在性能和复杂度间权衡

五、结论与建议

5.1 功率分配算法选择

- 性能优先：选择FSPA
- 实时性要求高：选择FPA
- 平衡性能和公平性：选择FTPA

5.2 用户分组策略选择

- 小规模系统（≤6用户）：可以使用穷尽搜索
- 大规模系统（>6用户）：推荐使用匹配分组
- 计算资源受限：考虑随机分组

5.3 改进建议

1. **算法优化**
    - 改进FSPA搜索效率
    - 开发自适应功率分配策略
    - 优化匹配算法的配对规则
2. **系统扩展**
    - 考虑多小区场景

- 添加用户移动性
- 引入QoS约束

代码附录

```
% 主程序：运行NOMA系统的仿真和性能分析（main）
% 作者：asaqe with AI
% 日期：2024年11月18日
%
% 更新日志：
% 2024-11-19
%   - 优化系统参数设置
%     * 调整用户数量为50
%     * 降低总功率至0.1W
%     * 提高带宽到1MHz
%     * 更新噪声功率为1e-13W
%     * 减少仿真次数至100次
%   - 添加理论上限验证
%   - 改进结果可视化
%   - 增加调试信息输出
%   - 优化性能统计方法
%
% 2024-11-18
%   - 初始版本
%   - 基本的仿真流程实现
%   - 简单的结果展示功能
%   - 基础性能统计

clc
clear
close all

% 设置中文显示
set(0,'DefaultAxesFontName','SimHei');
set(0,'DefaultTextFontName','SimHei');

% 系统参数设置
num_users = 50;
total_power = 0.1;        % 降低总功率
bandwidth = 1e6;          % 提高带宽到1MHz
noise_power = 1e-13;      % 更现实的噪声功率
num_simulations = 100;
max_users_per_group = 2;  % 限制每组最多2个用户

% 创建仿真实例
sim = NOMASimulation(num_users, total_power, bandwidth, noise_power,
max_users_per_group);  % 更新构造函数

% 初始化结果存储
throughput_results = zeros(num_simulations, 3);  % 存储每次仿真的吞吐量
runtime_results = zeros(num_simulations, 3);     % 存储每次仿真的运行时间

% 多次仿真
for n = 1:num_simulations
    % 生成信道增益
```

```
    channel_gains = sim.generate_channel_gains();

    % 1. 穷尽搜索
    [groups_exhaustive, time_exhaustive] =
sim.exhaustive_grouping(channel_gains);
    throughput_exhaustive = 0;
    for i = 1:length(groups_exhaustive)
        group_throughput = sim.calculate_throughput(groups_exhaustive{i},
channel_gains);
        if group_throughput > bandwidth * log2(1 + total_power/noise_power)
            warning('吞吐量超过了理论上限！');
        end
        throughput_exhaustive = throughput_exhaustive + group_throughput;
    end

    % 2. 随机分组
    [groups_random, time_random] = sim.random_grouping(channel_gains);
    throughput_random = 0;
    for i = 1:length(groups_random)
        if ~isempty(groups_random{i})  % 确保组不为空
            group_throughput = sim.calculate_throughput(groups_random{i},
channel_gains);
            if group_throughput > bandwidth * log2(1 + total_power/noise_power)
                warning('随机分组：吞吐量超过了理论上限！');
            end
            throughput_random = throughput_random + group_throughput;
        end
    end

    % 3. 匹配分组
    [groups_matching, time_matching] = sim.matching_grouping(channel_gains);
    throughput_matching = 0;
    for i = 1:length(groups_matching)
        if ~isempty(groups_matching{i})  % 确保组不为空
            group_throughput = sim.calculate_throughput(groups_matching{i},
channel_gains);
            if group_throughput > bandwidth * log2(1 + total_power/noise_power)
                warning('匹配分组：吞吐量超过了理论上限！');
            end
            throughput_matching = throughput_matching + group_throughput;
        end
    end

    % 存储结果
    throughput_results(n,:) = [throughput_exhaustive, throughput_random,
throughput_matching];
    runtime_results(n,:) = [time_exhaustive, time_random, time_matching];

    % 在主循环中添加调试信息
    % fprintf('仿真轮次 %d:\n', n);
    % fprintf('信道增益范围: %.2e ~ %.2e\n', min(channel_gains),
max(channel_gains));
    % fprintf('穷尽搜索吞吐量: %.2e\n', throughput_exhaustive);
end

% 计算平均值和标准差
```

```
mean_throughput = mean(throughput_results);
std_throughput = std(throughput_results);
mean_runtime = mean(runtime_results);

% 绘制系统吞吐量比较柱状图
figure('Renderer', 'painters');  % 使用painters渲染器
bar_data = mean_throughput;
b = bar(bar_data);
hold on;
errorbar(1:3, bar_data, std_throughput, 'k', 'LineStyle', 'none');
set(gca, 'XTickLabel', {'穷尽搜索', '随机分组', '匹配分组'});
title('系统吞吐量比较');
ylabel('平均系统吞吐量 (bps)');
grid on;

% 绘制算法运行时间比较柱状图
figure('Renderer', 'painters');  % 使用painters渲染器
bar(mean_runtime);
set(gca, 'XTickLabel', {'穷尽搜索', '随机分组', '匹配分组'});
title('算法运行时间比较');
ylabel('平均运行时间 (秒)');
grid on;

% 绘制算法收敛性分析图
figure('Renderer', 'painters');  % 使用painters渲染器
plot(1:num_simulations, throughput_results(:,1), 'r-', ...
     1:num_simulations, throughput_results(:,2), 'g--', ...
     1:num_simulations, throughput_results(:,3), 'b:');
title('算法收敛性分析');
xlabel('仿真次数');
ylabel('系统吞吐量 (bps)');
legend('穷尽搜索', '随机分组', '匹配分组', 'Location', 'best');
grid on;

% 打印统计结果
fprintf('\n====== 仿真结果统计 ======\n');
fprintf('平均系统吞吐量 (bps):\n');
fprintf('穷尽搜索: %.2e (±%.2e)\n', mean_throughput(1), std_throughput(1));
fprintf('随机分组: %.2e (±%.2e)\n', mean_throughput(2), std_throughput(2));
fprintf('匹配分组: %.2e (±%.2e)\n', mean_throughput(3), std_throughput(3));

fprintf('\n平均运行时间 (秒):\n');
fprintf('穷尽搜索: %.4f\n', mean_runtime(1));
fprintf('随机分组: %.4f\n', mean_runtime(2));
fprintf('匹配分组: %.4f\n', mean_runtime(3));
```

```
% NOMASimulation.m
% 作者：asaqe with AI
% 日期：2024年11月18日
%
% 更新日志：
% 2024-11-19
%   - 改进信道模型，采用3GPP路径损耗模型
%   - 优化功率分配策略，实现分数功率分配
%   - 添加吞吐量验证机制，确保不超过理论上限
```

```
%    - 改进SIC解码顺序处理
%    - 增加调试信息输出
%    - 统一三种分组方法的评分机制
%
% 2024-11-18
%    - 初始版本
%    - 实现基本的NOMA系统仿真功能
%    - 包含三种分组方法：穷尽搜索、随机分组、匹配分组
%    - 基本的性能计算功能

classdef NOMASimulation
    properties
        N            % 用户总数
        P_total      % 系统总功率
        B            % 系统带宽
        noise        % 噪声功率
        a            % 功率分配比例
    end

    methods
        function obj = NOMASimulation(num_users, total_power, bandwidth,
noise_power, power_ratio)
            % 构造函数：初始化系统参数
            if nargin < 5
                power_ratio = 0.7;  % 默认功率分配比例0.7
            end
            obj.N = num_users;
            obj.P_total = total_power;
            obj.B = bandwidth;
            obj.noise = noise_power;
            obj.a = power_ratio;
        end

        function channel_gains = generate_channel_gains(obj)
            % 修改信道增益生成方法
            % 使用更现实的路径损耗模型
            d = 10 + 90 * rand(1, obj.N);  % 用户距离基站 10-100m
            path_loss_dB = 128.1 + 37.6 * log10(d/1000);  % 3GPP路径损耗模型
            path_loss = 10.^(-path_loss_dB/10);  % 转换为线性尺度

            % 小尺度衰落
            rayleigh = abs(complex(randn(1,obj.N), randn(1,obj.N))).^2 / 2;

            % 合成信道增益
            channel_gains = path_loss .* rayleigh;
        end

        function [best_groups, elapsed_time] = exhaustive_grouping(obj,
channel_gains)
            % 穷尽搜索法：遍历所有可能的分组方案
            tic;
            users = 1:obj.N;
            best_groups = [];
            max_score = 0;

            % 生成所有可能的二元组合
```

```
            combinations = nchoosek(users, 2);
            for i = 1:size(combinations, 1)
                group = combinations(i,:);
                % 计算吞吐量
                throughput = obj.calculate_throughput(group, channel_gains);

                % 计算用户速率
                [R1, R2] = obj.calculate_individual_rates(group, channel_gains);
                rates = [R1, R2];

                % 计算公平性指数
                fairness = obj.calculate_fairness(rates);

                % 综合评分（可以调整权重）
                w1 = 0.7;  % 吞吐量权重
                w2 = 0.3;  % 公平性权重
                score = w1 * throughput + w2 * fairness;

                if score > max_score
                    max_score = score;
                    best_groups = {group};
                end
            end
            elapsed_time = toc;
        end

        function [groups, elapsed_time] = random_grouping(obj, channel_gains)
            % 随机分组法：随机将用户分配到不同组
            tic;
            users = randperm(obj.N);
            groups = {};
            max_score = 0;

            % 进行多次随机尝试以获得较好的结果
            num_attempts = 10;  % 增加随机尝试次数

            for attempt = 1:num_attempts
                current_groups = cell(1, floor(obj.N/2));
                current_score = 0;

                % 每两个用户一组
                for i = 1:2:obj.N-1
                    group_idx = ceil(i/2);
                    group = users(i:i+1);

                    % 计算该组的得分
                    throughput = obj.calculate_throughput(group, channel_gains);
                    [R1, R2] = obj.calculate_individual_rates(group,
channel_gains);
                    fairness = obj.calculate_fairness([R1, R2]);

                    % 使用与穷尽搜索相同的评分方式
                    w1 = 0.7;  % 吞吐量权重
                    w2 = 0.3;  % 公平性权重
                    score = w1 * throughput + w2 * fairness;
```

```
                current_groups{group_idx} = group;
                current_score = current_score + score;
            end

            % 更新最佳分组
            if current_score > max_score
                max_score = current_score;
                groups = current_groups;
            end

            % 重新随机排列用户顺序
            users = randperm(obj.N);
        end
        elapsed_time = toc;
    end

    function [groups, elapsed_time] = matching_grouping(obj, channel_gains)
        % 匹配分组算法：根据信道增益进行强弱用户配对
        tic;
        [~, sorted_idx] = sort(channel_gains, 'descend');
        groups = cell(1, floor(obj.N/2));
        total_score = 0;

        % 强弱用户配对
        for i = 1:floor(obj.N/2)
            % 形成一组：一个信道最好的用户和一个信道最差的用户
            group = [sorted_idx(i), sorted_idx(end-i+1)];

            % 计算该组的得分
            throughput = obj.calculate_throughput(group, channel_gains);
            [R1, R2] = obj.calculate_individual_rates(group, channel_gains);
            fairness = obj.calculate_fairness([R1, R2]);

            % 使用与穷尽搜索相同的评分方式
            w1 = 0.7;  % 吞吐量权重
            w2 = 0.3;  % 公平性权重
            score = w1 * throughput + w2 * fairness;

            groups{i} = group;
            total_score = total_score + score;
        end
        elapsed_time = toc;
    end

    function throughput = calculate_throughput(obj, group, channel_gains)
        % 修改吞吐量计算方法
        power_allocation = obj.allocate_power(group, channel_gains);

        % 验证功率分配
        if abs(sum(power_allocation) - obj.P_total) > 1e-10
            warning('功率分配不准确，进行归一化');
            power_allocation = power_allocation / sum(power_allocation) *
obj.P_total;
        end

        % 计算理论上限
```

```
            max_channel_gain = max(channel_gains(group));
            theoretical_limit = obj.B * log2(1 + obj.P_total * max_channel_gain /
obj.noise);

            % 计算实际吞吐量
            throughput = 0;
            for i = 1:length(group)
                sinr = obj.calculate_sinr(group(i), group, channel_gains,
power_allocation);
                user_throughput = obj.B * log2(1 + sinr);

                % 验证单用户吞吐量不超过理论上限
                if user_throughput > theoretical_limit
                    warning('单用户吞吐量超过理论上限，进行截断');
                    user_throughput = theoretical_limit;
                end

                throughput = throughput + user_throughput;
            end

            % 验证总吞吐量不超过系统容量
            system_capacity = obj.B * log2(1 + obj.P_total *
sum(channel_gains(group)) / obj.noise);
            if throughput > system_capacity
                warning('总吞吐量超过系统容量，进行截断');
                throughput = system_capacity;
            end
        end

        function fairness = calculate_fairness(obj, rates)
            % 计算Jain's公平性指数
            fairness = sum(rates)^2 / (length(rates) * sum(rates.^2));
        end

        % 新增函数：计算个体速率
        function [R1, R2] = calculate_individual_rates(obj, group, channel_gains)
            h1 = channel_gains(group(1));
            h2 = channel_gains(group(2));
            P1 = obj.a * obj.P_total;
            P2 = (1-obj.a) * obj.P_total;

            % 计算SINR
            sinr1 = (P1 * h1) / (P2 * h1 + obj.noise);
            sinr2 = (P2 * h2) / obj.noise;

            % 计算个体速率
            R1 = obj.B * log2(1 + sinr1);
            R2 = obj.B * log2(1 + sinr2);
        end

        function power_allocation = allocate_power(obj, group, channel_gains)
            % 修改功率分配方法
            group_gains = channel_gains(group);
            num_users = length(group);
            power_allocation = zeros(1, num_users);
```

```
            % 按信道增益排序（从大到小）
            [sorted_gains, idx] = sort(group_gains, 'descend');

            % 使用分数功率分配策略
            total_power = obj.P_total;
            for i = 1:num_users
                if i == num_users
                    % 信道最差的用户获得剩余所有功率
                    power_allocation(idx(i)) = total_power;
                else
                    % 其他用户按比例分配
                    power = total_power * 0.25;  % 每个用户最多获得25%的剩余功率
                    power_allocation(idx(i)) = power;
                    total_power = total_power - power;
                end
            end
        end

        function sinr = calculate_sinr(obj, user_idx, group, channel_gains, 
power_allocation)
            % 计算特定用户的SINR
            h_i = channel_gains(user_idx);
            P_i = power_allocation(find(group == user_idx));

            % 计算干扰
            interference = 0;
            user_position = find(group == user_idx);

            % SIC解码顺序：信道增益大的用户先解码
            [~, decode_order] = sort(channel_gains(group), 'descend');
            current_user_decode_position = find(decode_order == user_position);

            % 只考虑解码顺序在当前用户之后的干扰
            for j = current_user_decode_position+1:length(group)
                interferer_idx = group(decode_order(j));
                P_j = power_allocation(find(group == interferer_idx));
                interference = interference + P_j * h_i;
            end

            % 计算SINR
            sinr = (P_i * h_i) / (interference + obj.noise);
        end
    end
end
```

```
% 主程序：NOMA系统功率分配算法比较
% 作者：asaqe with AI
% 日期：2024年11月19日

clc
clear
close all

% 设置中文显示
set(0,'DefaultAxesFontName','SimHei');
```

```
set(0,'DefaultTextFontName','SimHei');

% 系统参数设置
num_users = 2;           % 每组两个用户
total_power = 1.0;       % 总功率1W
bandwidth = 1e6;         % 带宽1MHz
noise_power = 1e-12;     % 噪声功率
num_simulations = 1000;  % 仿真次数

% 创建功率分配实例
pa = PowerAllocation(total_power, noise_power, bandwidth);

% 初始化结果存储
throughput_results = zeros(num_simulations, 4);  % 存储每次仿真的吞吐量
fairness_results = zeros(num_simulations, 4);    % 存储每次仿真的公平性
runtime_results = zeros(num_simulations, 4);     % 存储每次仿真的运行时间

% 多次仿真
for n = 1:num_simulations
    % 生成两个用户的信道增益
    d = 10 + 90 * rand(1, 2);  % 用户距离基站 10-100m
    path_loss_dB = 128.1 + 37.6 * log10(d/1000);  % 3GPP路径损耗模型
    path_loss = 10.^(-path_loss_dB/10);  % 转换为线性尺度
    rayleigh = abs(complex(randn(1,2), randn(1,2))).^2 / 2;
    channel_gains = path_loss .* rayleigh;

    % 比较不同功率分配算法
    [power_fspa, time_fspa] = pa.FSPA(channel_gains);
    [power_fpa, time_fpa] = pa.FPA(channel_gains);
    [power_ftpa, time_ftpa] = pa.FTPA(channel_gains);
    [power_max, time_max] = pa.MaxThroughput(channel_gains);

    % 计算性能指标
    % 1. 吞吐量
    throughput_fspa = pa.calculate_throughput(channel_gains, power_fspa);
    throughput_fpa = pa.calculate_throughput(channel_gains, power_fpa);
    throughput_ftpa = pa.calculate_throughput(channel_gains, power_ftpa);
    throughput_max = pa.calculate_throughput(channel_gains, power_max);

    % 2. 公平性
    [sorted_gains, idx] = sort(channel_gains, 'descend');

    % FSPA
    sorted_powers_fspa = power_fspa(idx);
    sinr1_fspa = (sorted_powers_fspa(1) * sorted_gains(1)) / noise_power;
    sinr2_fspa = (sorted_powers_fspa(2) * sorted_gains(2)) /
(sorted_powers_fspa(1) * sorted_gains(2) + noise_power);
    rates_fspa = bandwidth * log2(1 + [sinr1_fspa, sinr2_fspa]);

    % FPA
    sorted_powers_fpa = power_fpa(idx);
    sinr1_fpa = (sorted_powers_fpa(1) * sorted_gains(1)) / noise_power;
    sinr2_fpa = (sorted_powers_fpa(2) * sorted_gains(2)) / (sorted_powers_fpa(1)
* sorted_gains(2) + noise_power);
    rates_fpa = bandwidth * log2(1 + [sinr1_fpa, sinr2_fpa]);
```

```matlab
    % FTPA
    sorted_powers_ftpa = power_ftpa(idx);
    sinr1_ftpa = (sorted_powers_ftpa(1) * sorted_gains(1)) / noise_power;
    sinr2_ftpa = (sorted_powers_ftpa(2) * sorted_gains(2)) /
(sorted_powers_ftpa(1) * sorted_gains(2) + noise_power);
    rates_ftpa = bandwidth * log2(1 + [sinr1_ftpa, sinr2_ftpa]);

    % MaxThroughput
    sorted_powers_max = power_max(idx);
    sinr1_max = (sorted_powers_max(1) * sorted_gains(1)) / noise_power;
    sinr2_max = (sorted_powers_max(2) * sorted_gains(2)) / (sorted_powers_max(1)
* sorted_gains(2) + noise_power);
    rates_max = bandwidth * log2(1 + [sinr1_max, sinr2_max]);

    fairness_fspa = pa.calculate_fairness(rates_fspa);
    fairness_fpa = pa.calculate_fairness(rates_fpa);
    fairness_ftpa = pa.calculate_fairness(rates_ftpa);
    fairness_max = pa.calculate_fairness(rates_max);

    % 存储结果
    throughput_results(n,:) = [throughput_fspa, throughput_fpa, throughput_ftpa,
throughput_max];
    fairness_results(n,:) = [fairness_fspa, fairness_fpa, fairness_ftpa,
fairness_max];
    runtime_results(n,:) = [time_fspa, time_fpa, time_ftpa, time_max];
end

% 计算平均值和标准差
mean_throughput = mean(throughput_results);
std_throughput = std(throughput_results);
mean_fairness = mean(fairness_results);
std_fairness = std(fairness_results);
mean_runtime = mean(runtime_results);

% 1. 吞吐量比较图
figure('Name', '系统吞吐量比较', 'Renderer', 'painters');
bar_data = mean_throughput;
b = bar(bar_data);
hold on;
errorbar(1:4, bar_data, std_throughput, 'k', 'LineStyle', 'none');
set(gca, 'XTickLabel', {'FSPA', 'FPA', 'FTPA', 'MaxThroughput'});
title('系统吞吐量比较');
ylabel('平均系统吞吐量 (bps)');
grid on;

% 2. 公平性比较图
figure('Name', '系统公平性比较', 'Renderer', 'painters');
bar_data = mean_fairness;
b = bar(bar_data);
hold on;
errorbar(1:4, bar_data, std_fairness, 'k', 'LineStyle', 'none');
set(gca, 'XTickLabel', {'FSPA', 'FPA', 'FTPA', 'MaxThroughput'});
title('系统公平性比较');
ylabel('平均系统公平性');
grid on;
```

```matlab
% 3. 运行时间比较图
figure('Name', '算法运行时间比较', 'Renderer', 'painters');
bar(mean_runtime);
set(gca, 'XTickLabel', {'FSPA', 'FPA', 'FTPA', 'MaxThroughput'});
title('算法运行时间比较');
ylabel('平均运行时间 (秒)');
grid on;

% 打印统计结果
fprintf('\n====== 功率分配算法性能统计 ======\n');
fprintf('平均系统吞吐量 (bps):\n');
fprintf('FSPA: %.2e (±%.2e)\n', mean_throughput(1), std_throughput(1));
fprintf('FPA: %.2e (±%.2e)\n', mean_throughput(2), std_throughput(2));
fprintf('FTPA: %.2e (±%.2e)\n', mean_throughput(3), std_throughput(3));
fprintf('MaxThroughput: %.2e (±%.2e)\n', mean_throughput(4), std_throughput(4));

fprintf('\n平均系统公平性:\n');
fprintf('FSPA: %.4f (±%.4f)\n', mean_fairness(1), std_fairness(1));
fprintf('FPA: %.4f (±%.4f)\n', mean_fairness(2), std_fairness(2));
fprintf('FTPA: %.4f (±%.4f)\n', mean_fairness(3), std_fairness(3));
fprintf('MaxThroughput: %.4f (±%.4f)\n', mean_fairness(4), std_fairness(4));

fprintf('\n平均运行时间 (秒):\n');
fprintf('FSPA: %.4f\n', mean_runtime(1));
fprintf('FPA: %.4f\n', mean_runtime(2));
fprintf('FTPA: %.4f\n', mean_runtime(3));
fprintf('MaxThroughput: %.4f\n', mean_runtime(4));
```

```matlab
classdef PowerAllocation
    properties
        P_total     % 总功率
        noise       % 噪声功率
        bandwidth   % 系统带宽
        alpha = 0.6 % FTPA的非均匀性参数
    end

    methods
        function obj = PowerAllocation(total_power, noise_power, bandwidth)
            obj.P_total = total_power;
            obj.noise = noise_power;
            obj.bandwidth = bandwidth;
        end

        function [power_allocation, elapsed_time] = FSPA(obj, channel_gains)
            % 遍历搜索功率分配法 - 更细致的搜索
            tic;
            step = 0.01;  % 更细的搜索步长
            max_throughput = 0;
            best_allocation = zeros(1, length(channel_gains));

            [~, idx] = sort(channel_gains, 'descend');

            % 限制功率分配范围，确保NOMA原则
            for a = 0.2:step:0.5  % 信道好的用户功率比例限制在20%-50%
                P1 = a * obj.P_total;
```

```
                P2 = (1-a) * obj.P_total;
                current_allocation = zeros(1, 2);
                current_allocation(idx(1)) = P1;  % 信道好的用户
                current_allocation(idx(2)) = P2;  % 信道差的用户

                throughput = obj.calculate_throughput(channel_gains,
current_allocation);

                if throughput > max_throughput
                    max_throughput = throughput;
                    best_allocation = current_allocation;
                end
            end

            power_allocation = best_allocation;
            elapsed_time = toc;
        end

        function [power_allocation, elapsed_time] = FPA(obj, channel_gains)
            % 固定功率分配法 - 使用更极端的分配
            tic;
            [~, idx] = sort(channel_gains, 'descend');
            power_allocation = zeros(1, 2);
            power_allocation(idx(1)) = 0.2 * obj.P_total;  % 信道好的用户分配很少功率
            power_allocation(idx(2)) = 0.8 * obj.P_total;  % 信道差的用户分配很多功率
            elapsed_time = toc;
        end

        function [power_allocation, elapsed_time] = FTPA(obj, channel_gains)
            % 分数功率分配法 - 更大的alpha值
            tic;
            obj.alpha = 1.5;  % 显著增加alpha使分配更不均匀
            channel_gains_inv = channel_gains.^(-obj.alpha);
            denominator = sum(channel_gains_inv);
            power_allocation = obj.P_total * (channel_gains_inv / denominator);
            elapsed_time = toc;
        end

        function [power_allocation, elapsed_time] = MaxThroughput(obj,
channel_gains)
            % 最大化吞吐量分配法
            tic;
            options = optimset('Display', 'off');

            % 定义目标函数
            objective = @(x) -obj.calculate_throughput(channel_gains, x);

            % 约束条件
            A = [];
            b = [];
            Aeq = [1 1];  % 功率和等于总功率
            beq = obj.P_total;
            lb = [0 0];   % 功率非负
            ub = [obj.P_total obj.P_total];

            % 初始猜测
```

```
            x0 = [obj.P_total/2 obj.P_total/2];

            % 求解优化问题
            [x, ~] = fmincon(objective, x0, A, b, Aeq, beq, lb, ub, [], options);

            power_allocation = x;
            elapsed_time = toc;
        end

        function throughput = calculate_throughput(obj, channel_gains,
power_allocation)
            % 计算系统吞吐量，考虑干扰影响和理论上限

            % 按信道增益排序（降序）
            [sorted_gains, idx] = sort(channel_gains, 'descend');
            sorted_powers = power_allocation(idx);

            % 信道较好的用户（用户1）
            % 可以完全消除用户2的干扰
            sinr1 = (sorted_powers(1) * sorted_gains(1)) / obj.noise;
            R1 = obj.bandwidth * log2(1 + sinr1);

            % 信道较差的用户（用户2）
            % 受到用户1的强干扰
            interference = sorted_powers(1) * sorted_gains(2);
            sinr2 = (sorted_powers(2) * sorted_gains(2)) / (interference +
obj.noise);
            R2 = obj.bandwidth * log2(1 + sinr2);

            % 计算单用户理论上限
            C1 = obj.bandwidth * log2(1 + sorted_powers(1) * sorted_gains(1) /
obj.noise);
            C2 = obj.bandwidth * log2(1 + sorted_powers(2) * sorted_gains(2) /
obj.noise);

            % 限制每个用户的速率
            R1 = min(R1, C1);
            R2 = min(R2, C2);

            % 总吞吐量
            throughput = R1 + R2;
        end

        function fairness = calculate_fairness(obj, rates)
            % 计算Jain's公平性指数
            fairness = sum(rates)^2 / (length(rates) * sum(rates.^2));
        end
    end
end
```

```
% 主程序：NOMA系统功率分配算法比较
% 作者：asaqe with AI
% 日期：2024年11月19日

clc
```

```
clear
close all

% 设置中文显示
set(0,'DefaultAxesFontName','SimHei');
set(0,'DefaultTextFontName','SimHei');

% 系统参数设置
num_users = 2;              % 每组两个用户
total_power = 1.0;          % 总功率1W
bandwidth = 1e6;            % 带宽1MHz
noise_power = 1e-12;        % 噪声功率
num_simulations = 1000;     % 仿真次数

% 创建功率分配实例
pa = PowerAllocation(total_power, noise_power, bandwidth);

% 初始化结果存储
throughput_results = zeros(num_simulations, 4);  % 存储每次仿真的吞吐量
fairness_results = zeros(num_simulations, 4);    % 存储每次仿真的公平性
runtime_results = zeros(num_simulations, 4);     % 存储每次仿真的运行时间

% 多次仿真
for n = 1:num_simulations
    % 生成两个用户的信道增益
    d = 10 + 90 * rand(1, 2);  % 用户距离基站 10-100m
    path_loss_dB = 128.1 + 37.6 * log10(d/1000);  % 3GPP路径损耗模型
    path_loss = 10.^(-path_loss_dB/10);  % 转换为线性尺度
    rayleigh = abs(complex(randn(1,2), randn(1,2))).^2 / 2;
    channel_gains = path_loss .* rayleigh;

    % 比较不同功率分配算法
    [power_fspa, time_fspa] = pa.FSPA(channel_gains);
    [power_fpa, time_fpa] = pa.FPA(channel_gains);
    [power_ftpa, time_ftpa] = pa.FTPA(channel_gains);
    [power_max, time_max] = pa.MaxThroughput(channel_gains);

    % 计算性能指标
    % 1. 吞吐量
    throughput_fspa = pa.calculate_throughput(channel_gains, power_fspa);
    throughput_fpa = pa.calculate_throughput(channel_gains, power_fpa);
    throughput_ftpa = pa.calculate_throughput(channel_gains, power_ftpa);
    throughput_max = pa.calculate_throughput(channel_gains, power_max);

    % 2. 公平性
    [sorted_gains, idx] = sort(channel_gains, 'descend');

    % FSPA
    sorted_powers_fspa = power_fspa(idx);
    sinr1_fspa = (sorted_powers_fspa(1) * sorted_gains(1)) / noise_power;
    sinr2_fspa = (sorted_powers_fspa(2) * sorted_gains(2)) /
(sorted_powers_fspa(1) * sorted_gains(2) + noise_power);
    rates_fspa = bandwidth * log2(1 + [sinr1_fspa, sinr2_fspa]);

    % FPA
    sorted_powers_fpa = power_fpa(idx);
```

```
    sinr1_fpa = (sorted_powers_fpa(1) * sorted_gains(1)) / noise_power;
    sinr2_fpa = (sorted_powers_fpa(2) * sorted_gains(2)) / (sorted_powers_fpa(1)
* sorted_gains(2) + noise_power);
    rates_fpa = bandwidth * log2(1 + [sinr1_fpa, sinr2_fpa]);

    % FTPA
    sorted_powers_ftpa = power_ftpa(idx);
    sinr1_ftpa = (sorted_powers_ftpa(1) * sorted_gains(1)) / noise_power;
    sinr2_ftpa = (sorted_powers_ftpa(2) * sorted_gains(2)) /
(sorted_powers_ftpa(1) * sorted_gains(2) + noise_power);
    rates_ftpa = bandwidth * log2(1 + [sinr1_ftpa, sinr2_ftpa]);

    % MaxThroughput
    sorted_powers_max = power_max(idx);
    sinr1_max = (sorted_powers_max(1) * sorted_gains(1)) / noise_power;
    sinr2_max = (sorted_powers_max(2) * sorted_gains(2)) / (sorted_powers_max(1)
* sorted_gains(2) + noise_power);
    rates_max = bandwidth * log2(1 + [sinr1_max, sinr2_max]);

    fairness_fspa = pa.calculate_fairness(rates_fspa);
    fairness_fpa = pa.calculate_fairness(rates_fpa);
    fairness_ftpa = pa.calculate_fairness(rates_ftpa);
    fairness_max = pa.calculate_fairness(rates_max);

    % 存储结果
    throughput_results(n,:) = [throughput_fspa, throughput_fpa, throughput_ftpa,
throughput_max];
    fairness_results(n,:) = [fairness_fspa, fairness_fpa, fairness_ftpa,
fairness_max];
    runtime_results(n,:) = [time_fspa, time_fpa, time_ftpa, time_max];
end

% 计算平均值和标准差
mean_throughput = mean(throughput_results);
std_throughput = std(throughput_results);
mean_fairness = mean(fairness_results);
std_fairness = std(fairness_results);
mean_runtime = mean(runtime_results);

% 1. 吞吐量比较图
figure('Name', '系统吞吐量比较', 'Renderer', 'painters');
bar_data = mean_throughput;
b = bar(bar_data);
hold on;
errorbar(1:4, bar_data, std_throughput, 'k', 'LineStyle', 'none');
set(gca, 'XTickLabel', {'FSPA', 'FPA', 'FTPA', 'MaxThroughput'});
title('系统吞吐量比较');
ylabel('平均系统吞吐量 (bps)');
grid on;

% 2. 公平性比较图
figure('Name', '系统公平性比较', 'Renderer', 'painters');
bar_data = mean_fairness;
b = bar(bar_data);
hold on;
errorbar(1:4, bar_data, std_fairness, 'k', 'LineStyle', 'none');
```

```
set(gca, 'XTickLabel', {'FSPA', 'FPA', 'FTPA', 'MaxThroughput'});
title('系统公平性比较');
ylabel('平均系统公平性');
grid on;

% 3. 运行时间比较图
figure('Name', '算法运行时间比较', 'Renderer', 'painters');
bar(mean_runtime);
set(gca, 'XTickLabel', {'FSPA', 'FPA', 'FTPA', 'MaxThroughput'});
title('算法运行时间比较');
ylabel('平均运行时间 (秒)');
grid on;

% 打印统计结果
fprintf('\n====== 功率分配算法性能统计 ======\n');
fprintf('平均系统吞吐量 (bps):\n');
fprintf('FSPA: %.2e (±%.2e)\n', mean_throughput(1), std_throughput(1));
fprintf('FPA: %.2e (±%.2e)\n', mean_throughput(2), std_throughput(2));
fprintf('FTPA: %.2e (±%.2e)\n', mean_throughput(3), std_throughput(3));
fprintf('MaxThroughput: %.2e (±%.2e)\n', mean_throughput(4), std_throughput(4));

fprintf('\n平均系统公平性:\n');
fprintf('FSPA: %.4f (±%.4f)\n', mean_fairness(1), std_fairness(1));
fprintf('FPA: %.4f (±%.4f)\n', mean_fairness(2), std_fairness(2));
fprintf('FTPA: %.4f (±%.4f)\n', mean_fairness(3), std_fairness(3));
fprintf('MaxThroughput: %.4f (±%.4f)\n', mean_fairness(4), std_fairness(4));

fprintf('\n平均运行时间 (秒):\n');
fprintf('FSPA: %.4f\n', mean_runtime(1));
fprintf('FPA: %.4f\n', mean_runtime(2));
fprintf('FTPA: %.4f\n', mean_runtime(3));
fprintf('MaxThroughput: %.4f\n', mean_runtime(4));
```

```
% 主程序：运行NOMA系统的仿真和性能分析（10个用户） 9! =1814400
% 作者：asaqe with AI
% 日期：2024年11月20日

clc
clear
close all

% 设置中文显示
set(0,'DefaultAxesFontName','SimHei');
set(0,'DefaultTextFontName','SimHei');

% 系统参数设置
num_users = 10;             % 10个用户
total_power = 0.1;          % 总功率
bandwidth = 1e6;            % 带宽1MHz
noise_power = 1e-13;        % 噪声功率
num_simulations = 100;      % 仿真次数
max_users_per_group = 2;    % 每组最多2个用户

% 创建仿真实例
```

```
sim = NOMASimulation(num_users, total_power, bandwidth, noise_power,
max_users_per_group);

% 初始化结果存储
throughput_results = zeros(num_simulations, 3);  % 存储每次仿真的吞吐量
runtime_results = zeros(num_simulations, 3);     % 存储每次仿真的运行时间

% 多次仿真
for n = 1:num_simulations
    fprintf('\n===== 仿真轮次 %d =====\n', n);

    % 生成信道增益
    channel_gains = sim.generate_channel_gains();

    % 1. 穷尽搜索 - 遍历所有可能的分组方式
    tic;
    max_throughput = 0;
    best_groups = {};

    % 生成所有可能的分组方式
    users = 1:num_users;
    all_pairs = nchoosek(users, 2);  % 所有可能的两用户组合
    num_pairs = size(all_pairs, 1);

    % 遍历所有可能的五组组合
    for i = 1:num_pairs
        pair1 = all_pairs(i,:);
        remaining_users1 = setdiff(users, pair1);

        % 在剩余用户中选择第二组
        remaining_pairs1 = nchoosek(remaining_users1, 2);
        for j = 1:size(remaining_pairs1, 1)
            pair2 = remaining_pairs1(j,:);
            remaining_users2 = setdiff(remaining_users1, pair2);

            % 选择第三组
            remaining_pairs2 = nchoosek(remaining_users2, 2);
            for k = 1:size(remaining_pairs2, 1)
                pair3 = remaining_pairs2(k,:);
                remaining_users3 = setdiff(remaining_users2, pair3);

                % 选择第四组
                remaining_pairs3 = nchoosek(remaining_users3, 2);
                for l = 1:size(remaining_pairs3, 1)
                    pair4 = remaining_pairs3(l,:);
                    % 最后两个用户自动形成第五组
                    pair5 = setdiff(remaining_users3, pair4);

                    % 计算当前分组方案的总吞吐量
                    current_groups = {pair1, pair2, pair3, pair4, pair5};
                    current_throughput = 0;

                    for m = 1:length(current_groups)
                        group_throughput =
sim.calculate_throughput(current_groups{m}, channel_gains);
```

```
                        current_throughput = current_throughput +
group_throughput;
                    end

                    % 更新最优解
                    if current_throughput > max_throughput
                        max_throughput = current_throughput;
                        best_groups = current_groups;
                    end
                end
            end
        end
    end
    time_exhaustive = toc;
    throughput_exhaustive = max_throughput;

    % 打印最优分组结果
    fprintf('最优分组方案:\n');
    for i = 1:length(best_groups)
        fprintf('组%d: 用户 %d 和用户 %d\n', i, best_groups{i}(1), best_groups{i}
(2));
    end

    % 2. 随机分组
    [groups_random, time_random] = sim.random_grouping(channel_gains);
    throughput_random = 0;
    for i = 1:length(groups_random)
        if ~isempty(groups_random{i})
            group_throughput = sim.calculate_throughput(groups_random{i},
channel_gains);
            throughput_random = throughput_random + group_throughput;
        end
    end

    % 3. 匹配分组
    [groups_matching, time_matching] = sim.matching_grouping(channel_gains);
    throughput_matching = 0;
    for i = 1:length(groups_matching)
        if ~isempty(groups_matching{i})
            group_throughput = sim.calculate_throughput(groups_matching{i},
channel_gains);
            throughput_matching = throughput_matching + group_throughput;
        end
    end

    % 存储结果
    throughput_results(n,:) = [throughput_exhaustive, throughput_random,
throughput_matching];
    runtime_results(n,:) = [time_exhaustive, time_random, time_matching];

    % 打印当前轮次的结果
    fprintf('穷尽搜索吞吐量: %.2e\n', throughput_exhaustive);
    fprintf('随机分组吞吐量: %.2e\n', throughput_random);
    fprintf('匹配分组吞吐量: %.2e\n', throughput_matching);
end
```

```matlab
% 计算平均值和标准差
mean_throughput = mean(throughput_results);
std_throughput = std(throughput_results);
mean_runtime = mean(runtime_results);

% 绘制系统吞吐量比较柱状图
figure('Name', '10用户系统吞吐量比较', 'Renderer', 'painters');
bar_data = mean_throughput;
b = bar(bar_data);
hold on;
errorbar(1:3, bar_data, std_throughput, 'k', 'LineStyle', 'none');
set(gca, 'XTickLabel', {'穷尽搜索', '随机分组', '匹配分组'});
title('10用户系统吞吐量比较');
ylabel('平均系统吞吐量 (bps)');
grid on;

% 绘制算法运行时间比较柱状图
figure('Name', '10用户算法运行时间比较', 'Renderer', 'painters');
bar(mean_runtime);
set(gca, 'XTickLabel', {'穷尽搜索', '随机分组', '匹配分组'});
title('10用户算法运行时间比较');
ylabel('平均运行时间 (秒)');
grid on;

% 绘制算法收敛性分析图
figure('Name', '10用户算法收敛性分析', 'Renderer', 'painters');
plot(1:num_simulations, throughput_results(:,1), 'r-', ...
     1:num_simulations, throughput_results(:,2), 'g--', ...
     1:num_simulations, throughput_results(:,3), 'b:');
title('10用户算法收敛性分析');
xlabel('仿真次数');
ylabel('系统吞吐量 (bps)');
legend('穷尽搜索', '随机分组', '匹配分组', 'Location', 'best');
grid on;

% 打印统计结果
fprintf('\n====== 10用户仿真结果统计 ======\n');
fprintf('平均系统吞吐量 (bps):\n');
fprintf('穷尽搜索: %.2e (±%.2e)\n', mean_throughput(1), std_throughput(1));
fprintf('随机分组: %.2e (±%.2e)\n', mean_throughput(2), std_throughput(2));
fprintf('匹配分组: %.2e (±%.2e)\n', mean_throughput(3), std_throughput(3));

fprintf('\n平均运行时间 (秒):\n');
fprintf('穷尽搜索: %.4f\n', mean_runtime(1));
fprintf('随机分组: %.4f\n', mean_runtime(2));
fprintf('匹配分组: %.4f\n', mean_runtime(3));
```