专题三：优化类模型

目录

I. 简单优化算法

1 线性优化

1.1 线性优化案例

某工厂生产两种产品：产品 A 和产品 B。生产这两种产品需要消耗两种原材料：原材料 1 和原材料 2。已知以下信息：

- 生产 1 单位产品 A 需要消耗 2 单位原材料 1 和 1 单位原材料 2
- 生产 1 单位产品 B 需要消耗 1 单位原材料 1 和 3 单位原材料 2
- 工厂每天可用的原材料 1 为 100 单位，原材料 2 为 90 单位
- 每单位产品 A 的利润为 40 元，每单位产品 B 的利润为 30 元

工厂希望制定一个生产计划，使得在原材料限制下，每日总利润最大。

1.2 线性优化建模

①决策变量：

x_1：每日生产产品 A 的数量

x_2：每日生产产品 B 的数量

②目标函数：最大化每日总利润 $\max z = 40x_1 + 30x_2$

③约束条件：

·原材料 1 的限制：$2x_1 + x_2 \le 100$；原材料 2 的限制：$x_1 + 3x_2 \le 90$

1.3 线性优化代码

```python
c = [-40, -30]  # 利润系数取负，因为linprog是求最小

# 不等式约束（原材料限制）
A = [
    [2, 1],  # 原材料1的消耗
    [1, 3]   # 原材料2的消耗
]
b = [100, 90]  # 可用资源上限

# 变量边界（生产数量不能为负）
x_bounds = [(0, None), (0, None)]  # 两种产品的生产数量都≥0

# 求解 - 使用默认的'interior-point'方法或'simplex'
try:
    # 首先尝试使用较新的highs方法（如果可用）
    res = linprog(c, A_ub=A, b_ub=b, bounds=x_bounds, method='highs')
except ValueError:
    # 如果highs不可用，则回退到simplex方法
    res = linprog(c, A_ub=A, b_ub=b, bounds=x_bounds, method='simplex')

# 输出结果
print(f"最优生产计划：生产产品A {res.x[0]:.2f} 单位，产品B {res.x[1]:.2f} 单位")
print(f"最大每日利润：{-res.fun:.2f} 元")

# 检查资源使用情况
used_raw1 = 2 * res.x[0] + res.x[1]
used_raw2 = res.x[0] + 3 * res.x[1]
print(f"原材料1使用量：{used_raw1:.2f}/{100}")
print(f"原材料2使用量：{used_raw2:.2f}/{90}")
```

2 非线性优化

2.1 非线性优化案例

某公司需要优化其供应链中的 库存管理，以最小化总成本（包括订购成本、库存持有成本和缺货成本）。由于需求波动、存储限制和供应商约束，该问题涉及非线性关系，适合用非线性优化建模。

注意：除了线性关系的变量关系事实上都可以认为是非线性关系

2.2 非线性优化建模

①决策变量：

Q: 每次的订货量（决策变量）；S: 安全库存量（决策变量）

②目标函数（最小化总成本）： $\min C(Q,S)=\frac{D}{Q}\cdot K+h\cdot(\frac{Q}{2}+S)+p\int_S^{\infty}(x-S)f(x)dx$

·D:年需求量（常量）；K:每次订货的固定成本（常量）

·h:单位库存持有成本（常量）；p:单位缺货成本（常量）

·$f(x)$:随机需求的概率密度函数（假设服从正态分布 $N(\mu,\sigma^2)$）

③约束条件

·库存容量限制：$Q+S\le Q_{\max}$

·服务水平约束（缺货概率不超过 5%）：$\Pr ob(x>S)\le 0.05$；非负约束：$Q\ge 0,S\ge 0$

2.3 非线性优化代码

```python
# 参数设置
D = 10000       # 年需求量
K = 100         # 每次订货固定成本
h = 0.5         # 单位持有成本
p = 10          # 单位缺货成本
mu = 500        # 平均需求
sigma = 100     # 需求标准差
Q_max = 2000    # 最大库存容量

# 目标函数
def total_cost(vars):
    Q, S = vars
    ordering_cost = (D / Q) * K
    holding_cost = h * (Q / 2 + S)

    # 计算缺货成本（积分形式）
    z = (S - mu) / sigma
    shortage_cost = p * sigma * (norm.pdf(z) - z * (1 - norm.cdf(z)))

    return ordering_cost + holding_cost + shortage_cost

# 约束条件
constraints = [
    {'type': 'ineq', 'fun': lambda vars: Q_max - vars[0] - vars[1]},  # Q + S ≤ Q_max
    {'type': 'ineq', 'fun': lambda vars: 1 - norm.cdf((vars[1] - mu)/sigma) - 0.05},  # Prob(x>S) ≤ 0.05
    {'type': 'ineq', 'fun': lambda vars: vars[0]},  # Q ≥ 0
    {'type': 'ineq', 'fun': lambda vars: vars[1]}   # S ≥ 0
]

# 初始猜测
initial_guess = [500, 100]

# 优化求解
result = minimize(
    total_cost,
    initial_guess,
    constraints=constraints,
    method='SLSQP',  # 适用于非线性约束
    options={'maxiter': 1000}
)
```

3 多目标优化

3.1 多目标优化案例

某投资者希望构建一个投资组合，选择若干股票进行投资。已知以下信息：

有 4 只股票可供选择，每只股票的年化收益率和风险（标准差）如下：

股票 A：收益率 8% ，风险 12%
股票 B：收益率 12% ，风险 18%
股票 C：收益率 6% ，风险 8%
股票 D：收益率 10% ，风险 15%

投资者的目标：

①最大化投资组合的年化收益率；最小化投资组合的风险（标准差） 投资组合的限制

②每只股票的投资比例在 0%到 50%之间；总投资比例为 100%

3.2 多目标优化建模

①决策变量：

x_1：投资股票 A 的比例；x_2：投资股票 B 的比例；

x_3：投资股票 C 的比例；x_4：投资股票 D 的比例。

②目标函数：

·最大化年化收益率：$\max z_1 = 0.08x_1 + 0.12x_2 + 0.06x_3 + 0.10x_4$

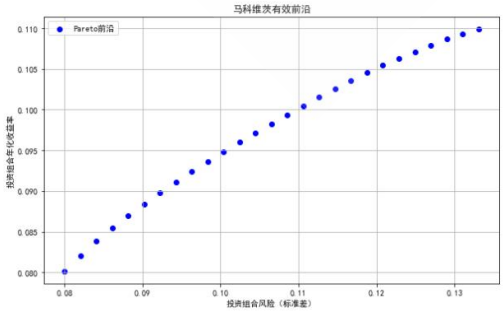
·最小化投资组合风险：$\min z_2 = \sqrt{(0.12x_1)^2 + (0.18x_2)^2 + (0.08x_3)^2 + (0.15x_4)^2}$

③约束条件：

·总投资比例：$x_1 + x_2 + x_3 + x_4 = 1$

·每只股票的投资比例限制：$0 \le x_1 \le 0.5, 0 \le x_2 \le 0.5, 0 \le x_3 \le 0.5, 0 \le x_4 \le 0.5$

3.3 多目标优化代码



风险最小化组合：
股票A：25.00%
股票B：25.00%
股票C：25.00%
股票D：25.00%
预期收益率：9.00%
组合风险：9.26%

收益最大化组合：
股票A：25.00%
股票B：25.00%
股票C：25.00%
股票D：25.00%
预期收益率：9.00%
组合风险：9.26%

```python
# 股票数据
returns = np.array([0.08, 0.12, 0.06, 0.10])  # 收益率
risks = np.array([0.12, 0.18, 0.08, 0.15])    # 风险（标准差）
n_assets = len(returns)

# 假设相关系数矩阵（简化，所有股票间相关系数为0.3）
corr_matrix = np.ones((n_assets, n_assets)) * 0.3
np.fill_diagonal(corr_matrix, 1)  # 对角线为1

# 协方差矩阵
cov_matrix = np.outer(risks, risks) * corr_matrix

# 目标函数1：最大化收益率（转换为最小化负收益率）
def target_return(weights):
    return -np.sum(returns * weights)

# 目标函数2：最小化风险
def target_risk(weights):
    return np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))

# 约束条件
constraints = (
    {'type': 'eq', 'fun': lambda w: np.sum(w) - 1},  # 总投资比例=100%
    {'type': 'ineq', 'fun': lambda w: w},            # 每只股票≥0
    {'type': 'ineq', 'fun': lambda w: 0.5 - w}       # 每只股票≤50%
)
```

$$
\begin{aligned}
&\min \quad h = H - \sum_{i=1}^{N} l_i \cos\theta_i \\
&\min \quad \theta_6 = \arctan \frac{F_{5,6}\sin\beta_5}{0.5(T_6 - G_6) + F_{5,6}\cos\beta_5} \\
&\min \quad r = \sum_{i=1}^{N} l_i \sin\theta_i \\
&s1\begin{cases}
\begin{cases} F_{21} = \dfrac{\sqrt{25(l_1-h)^2 d_1 \gamma^4 + 4(\rho g \pi d_1^2 h - 4G)_1}^2}{8} \\ \beta_1 = \arctan \dfrac{5(l_1-h)d_1\gamma^2}{8G_1 + 2g\rho\pi d_1^2 h} \end{cases} \\
\begin{cases} \beta_6 = \arctan \dfrac{F_{5,6}\sin\beta_5}{F_{5,6}\cos\beta_5 + T_6 + T_q - G_6 - G_q} \\ F_{8,6} = \dfrac{F_{5,6}\sin\beta_5}{\sin(\arctan \dfrac{F_{5,6}\sin\beta_5}{F_{5,6}\cos\beta_5 + T_6 + T_q - G_6 - G_q})} \\ \theta_6 = \arctan \dfrac{F_{5,6}\cdot\sin\beta_5}{0.5(T_6 - G_6) + F_{5,6}\cdot\cos\beta_5} \end{cases} \\
\begin{cases} \beta_i = \arctan \dfrac{F_{i-1,i}\cdot\sin\beta_{i-1}}{T_i + F_{i-1,i}\cdot\cos\beta_{i-1} - G_i} \\ F_{i+1,i} = \dfrac{F_{i+1,i}\cdot\sin\beta_i}{\sin(\arctan \dfrac{F_{i-1,i}\cdot\sin\beta_{i-1}}{T_i + F_{i-1,i}\cdot\cos\beta_{i-1} - G_i})} \\ \theta_i = \arctan \dfrac{F_{i-1,i}\sin\beta_{i-1}}{0.5(T_i - G_i) + F_{i-1,i}\cos\beta_{i-1}} \\ i \in [2,5] \cup [7,216], \quad i \in z \end{cases} \\
h + \sum_{i=1}^{N} l_i \cos\theta_i = H \\
r = \sum_{i=1}^{N} l_i \sin\theta_i
\end{cases}
\end{aligned}
\tag{6-2-3}
$$

非线性优化模型实例【2016 年数模国赛 A 题】

对于问题一，本文首先对乡村耕地结构、农作物种植现状及亩均利润进行了可视化分析，为制定最佳种植策略提供数据支持。随后，构建了以**七年利润最大化**为目标的**种植策略优化模型**，结合 **12 个关键约束条件**，包括地块种植面积上限、各地块农作物种植面积下限、决策变量的约束关系、禁止连续重茬种植、豆类作物的种植要求，

线性优化模型实例【2024 年数模国赛 C 题】

针对问题一，为通过抽样检测判断次品率是否超过标称值，本文分别针对(1)(2)情形建立了**假设检验**问题，进而得到了是否接收零配件的条件。为确定合适的抽样样本量，本文利用检测花销与假设检验效果(运用假设检验**功效函数**衡量)之间相互制约的关系，建立**决策函数**，得到了**最优样本量**。

非线性优化模型实例（概率）【2024 年数模国赛 A 题】

针对问题二，本文提出了**单位镜面面积年平均输出热功率的优化模型**。问题二是在定日镜场的额定年平均输出热功率为 60MW 的条件下，设计定日镜场的参数，使得单位镜面面积年平均输出热功率尽量大。**决策变量**包括吸收塔的位置坐标、定日镜的尺寸（相同）、安装高度、数量和位置。**目标函数**是单位镜面面积年平均输出热功率的最大值，**约束条件**包括镜面边长在 2m 至 8m 之间、安装高度在 2m 至 6m 之间、相邻定日镜底座中心距离比镜面宽度多 5m 等。先取 w,v 相等，用**蜂窝排列法**确定圆环内能排列的定日镜数目和位置。然后将蜂窝进行绕原点旋转 μ，用**蒙特卡洛模拟法**对部分定日镜进行随机抽样，得到粗决策变量，从而进一步计算光学效率和单位面积输出热功率等。最后遍历 $\mu,w=v,\tilde{h},X_0,Y_0$ 寻找决策变量的最优解。得到该定日镜场的**年平均输出光学效率 0.591643667、年平均输出热功率 68.24427914MW、单位镜面面积年平均输出热功率 0.572538333kW/m²**。

非线性优化模型实例（概率）【2023 年数模国赛 A 题】

II. 图与网络优化部分

4 弗洛伊德算法

4.1 弗洛伊德算法的案例

某快递公司需要为全国 50 个主要城市的配送中心规划最优运输路线。每个城市之间可能有直达的运输路线，也可能需要中转。

为什么适合弗洛伊德算法？

①多节点需求：需要计算任意两个城市之间的最低成本路线

②混合运输方式：存在公路/铁路/航空多种路径选择

③成本波动：不同季节运输成本会变化（需要频繁重新计算）

④中转需求：小城市之间往往需要大城市中转

重点：构造权重邻接矩阵

城市	北京	上海	广州	成都
北京	0	1200	1800	∞
上海	1200	0	1300	∞
广州	1800	1300	0	∞
成都	∞	∞	∞	0

4.2 弗洛伊德的算法原理

（1）算法用途

• 用于计算图中所有顶点之间的最短路径，适用于带权有向图和无向图。

• 能够处理包含负权边的图（但不能存在负权环）。

（2）核心思想

• 采用动态规划方法，通过逐步考虑每个顶点作为中间节点来更新最短路径。

• 最终得到任意两个顶点之间的最短路径长度。

（3）基本概念

• 维护一个距离矩阵，记录所有顶点对之间的当前最短距离。

• 初始时，距离矩阵存储直接相连的边权值，不相连的顶点距离设为无穷大。

（4）执行过程

• 依次将每个顶点作为中间节点进行考虑。

• 对于每对顶点，比较经过当前中间节点和不经过的路径长度，取较小值更新距离矩阵。

• 重复上述过程直到所有顶点都被考虑过。

弗洛伊德算法伪代码

```
let dist be a |V| × |V| matrix initialized with ∞
for each edge (u, v) do
    dist[u][v] ← weight(u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V| do
    for i from 1 to |V| do
        for j from 1 to |V| do
            dist[i][j] ← min(dist[i][j], dist[i][k] + dist[k][j])
```

3.3 模型

以平台的工作量尽量均衡、最长出警时间最短为目标，以平台至少管辖自己所在的路口和每个路口都被平台管辖为约束条件，建立了管辖区域的划分模型[5]为

$$\min f_1 = \frac{\sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(W_j-\overline{W})^2}}{\overline{W}},$$

$$\min f_2 = \max_{1\le j\le n}(T_j)$$

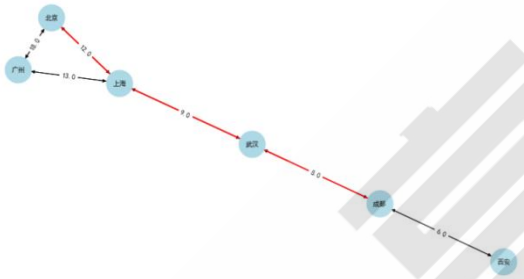
$$\text{s.t.}\begin{cases} x_{ij}=1, & i=j \\ \sum_{j=1}^{n} x_{ij}=1, & i\neq j \\ x_{ij}\in\{0,1\} \\ i=1,2,3,\cdots,m;\quad j=1,2,3,\cdots,n \end{cases}, \tag{3}$$

其中，W_j表示交巡警平台j的工作量，$\overline{W}$表示该区平台的平均工作量，T_j表示平台j的最长出警时间。

弗洛伊德算法事例【2011 年数模国赛 B 题】

4.3 弗洛伊德算法代码

```python
# ========== 弗洛伊德算法实现 ==========
def floyd_warshall(cost_matrix):
    n = len(cost_matrix)
    dist = cost_matrix.copy()
    path = np.zeros((n, n), dtype=int)  # 记录中转节点

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    path[i][j] = k + 1  # +1是为了区分0和未赋值
    return dist, path

# 执行算法
optimal_cost, path_matrix = floyd_warshall(cost_matrix)
```



	北京	上海	广州	成都	武汉	西安
北京	0.0	12.0	18.0	29.0	21.0	35.0
上海	12.0	0.0	13.0	17.0	9.0	23.0
广州	18.0	13.0	0.0	30.0	22.0	36.0
成都	29.0	17.0	30.0	0.0	8.0	6.0
武汉	21.0	9.0	22.0	8.0	0.0	14.0
西安	35.0	23.0	36.0	6.0	14.0	0.0

5 迪杰斯特拉法

5.1 迪杰斯特拉算法的案例

某快递公司需要为全国 50 个主要城市的配送中心规划最优运输路线。每个城市之间可能有直达的运输路线，也可能需要中转。

重点：构造权重邻接矩阵

城市	北京	上海	广州	成都
北京	0	1200	1800	∞
上海	1200	0	1300	∞
广州	1800	1300	0	∞
成都	∞	∞	∞	0

重要区别：贪心策略

5.2 迪杰斯特拉算法的原理

（1）核心用途

用于在带权有向图或无向图中，找到从单个起点到所有其他顶点的最短路径。

- 要求所有边的权重 非负（如距离、时间、成本等）
- 适合计算点对多点的最短路径（如导航中从一个地点到其他所有地点的最短路线）

（2）核心思想

通过**贪心策略**逐步扩展已知的最短路径：

初始化：

- 记录起点到所有顶点的 当前最短距离（起点设为 0，其他设为无穷大）
- 维护一个 未处理顶点集合（初始包含所有顶点）

迭代过程：

- 每次从 未处理集合 中选择 当前距离起点最近的顶点
- 对该顶点的所有邻居进行 松弛操作（即检查是否通过当前顶点能缩短到邻居的路径）
- 将当前顶点移出未处理集合

终止条件：

- 当所有顶点都被处理时，算法结束

迪杰斯特拉算法伪代码

```
function Dijkstra(图, 起点):
    初始化距离字典: 起点到自身距离为 0，其他顶点为∞
    初始化未处理集合: 包含所有顶点
    while 未处理集合非空:
        当前顶点 = 从未处理集合中选取距离起点最近的顶点
        从未处理集合中移除当前顶点
        for 每个邻居 in 当前顶点的所有邻居:
            新距离 = 起点到当前顶点的距离 + 当前顶点到邻居的距离
            if 新距离 < 当前记录的邻居距离:
                更新邻居的最短距离
    返回所有顶点的最短距离
```

标函数的最优目标，针对该问题我们采用的优化模型是图论中解决最短路问题的 Dijkstra 算法。

V 和 E 分别是图的顶点的集合 $V=\{v_1,v_2,\cdots,v_n\ v_1,v_2,\cdots,v_n\}$；

边的集合 $E=\{e_1,e_2,\cdots,e_m\}$

弧的集合，$A=\{a_1,a_2,\cdots,a_m\}$

链：在无向图中，点与边的交错序列（$v_{i_1},e_{i_1},v_{i_2},\cdots,v_{i_{k-1}},e_{i_{k-1}},v_{i_k}$）

称为连结 v_{i_1}和v_{i_k}的链。（e_i为连接v_i和v_{i+1}的边）

路径：（$v_{i_1},a_{i_1},v_{i_2},\cdots,v_{i_{k-1}},a_{i_{k-1}},v_{i_k}$）是有向图中一条链（$a_i$为连接$v_i$和$v_{i+1}$的弧），

称之为从 v_{i_1}到 v_{i_k}的路径。

圈：闭合的（无向）链称为圈。
回路：闭合的路径称为回路。
连通图：图 G 中任何两个点之间至少有一条链，称 G 为连通图。
树：一个无圈的连通图称为树。

生成树（支撑树）：若 $G_1=(V_1, E_1)$是连通图 $G_2=(V_2, E_2)$的生成子图 （即

$V_1=V_2$，$E_1\subseteq E_2$），且 G_1本身是树，则称 G_1为 G_2的生成树。

迪杰斯特拉算法事例【2016 年数模国赛 B 题】

5.3 迪杰斯特拉算法的代码

```python
def dijkstra(graph, start):
    """迪杰斯特拉算法实现"""
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    heap = [(0, start)]
    visited = set()

    while heap:
        current_dist, current_node = heapq.heappop(heap)
        if current_node in visited:
            continue
        visited.add(current_node)

        for neighbor, weight in graph[current_node].items():
            distance = current_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))

    return distances
```

III. 智能优化算法部分

6 遗传算法

6.1 遗传算法的原理

Step1 **初始化种群：**

随机生成一组初始解（称为“种群”），每个解称为一个“个体”。 个体通常用二进制串、实数向量或其他编码方式表示。

Step2 **计算适应度：**

根据问题的目标函数，计算每个个体的适应度值。适应度值越高，表示个体越优。

Step3 **是否满足终止条件？：**

检查是否满足终止条件（如达到最大迭代次数、适应度值收敛等），如果满足，输出当前最优解；否则，继续下一步。

Step4 **选择（Selection）：**

根据个体的适应度值，选择一部分优秀个体作为父代。

常用的选择方法：轮盘赌选择、锦标赛选择等。

Step5 **交叉（Crossover）：**

对选出的父代个体进行交叉操作，生成新的子代个体。

交叉操作模拟生物遗传中的基因重组，常用的方法：单点交叉、多点交叉、均匀交叉等。

Step6 **变异（Mutation）：**

对子代个体进行变异操作，引入随机性以避免陷入局部最优。

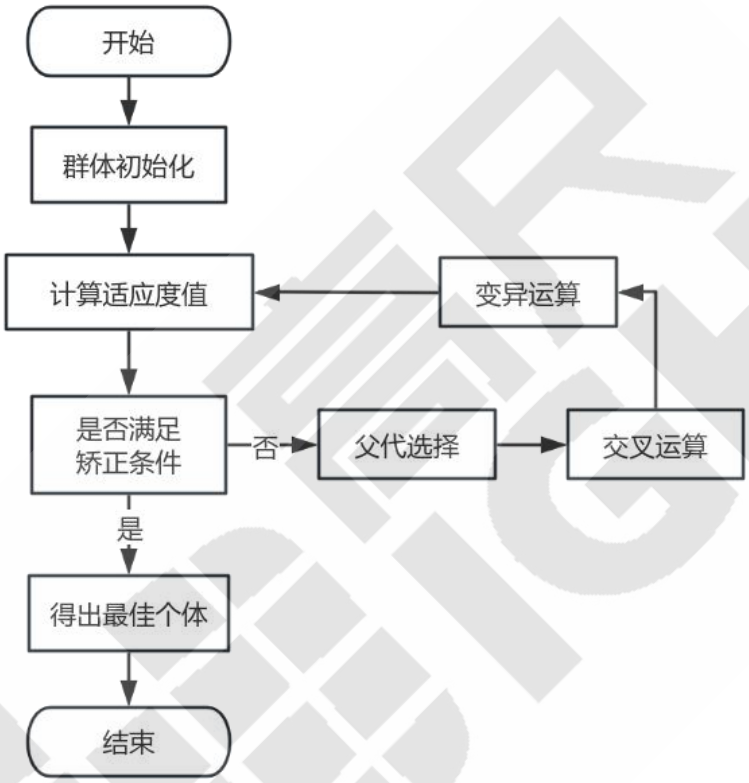
变异操作模拟生物遗传中的基因突变，常用的方法：位翻转、随机扰动等。

Step7 **生成新一代种群：**

将父代和子代个体合并，生成新一代种群。

Step8 **返回步骤 2：**

重复计算适应度、选择、交叉、变异等操作，直到满足终止条件。



4.5 基于差分进化算法的改进遗传算法的运用

优化算法在解决复杂问题时发挥了重要作用，其中遗传算法与差分进化算法作为两种经典的进化算法，广泛应用于多领域的全局优化问题。本文将遗传算法与差分进化算法相结合，旨在充分发挥二者的优势，提高算法的收敛速度与精度，从而为复杂优化问题的求解提供更有效的解决方案。

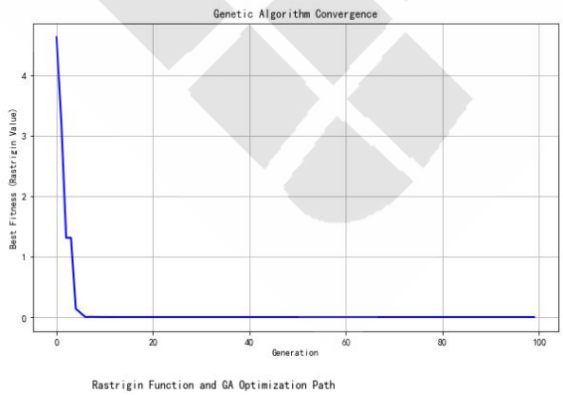
遗传算法（Genetic Algorithm, GA）源于对生物进化过程的模拟，是一种全局优化方法。其核心思想是通过模拟自然选择、遗传和变异等生物进化现象，从初始种群中进行随机选择、交叉和变异，逐步产生更适应环境的个体，使种群在搜索空间中进化到更优区域[1]。遗传算法的主要步骤包括染色体编码、种群初始化、选择、交叉和变异。它通过群体搜索策略及群体内基因的信息交换，具备较强的全局搜索能力[2]。然而，遗传算法也存在陷入局部最优解的风险，限制了其在某些复杂优化问题中的效果。

遗传算法实例【2024 年数模国赛 C 题】

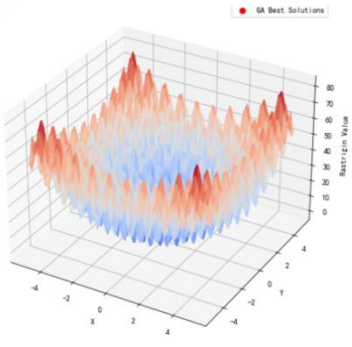
6.2 遗传算法的事例与代码

```python
# 目标函数：Rastrigin函数
def rastrigin(x):
    A = 10
    n = len(x)
    return A * n + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# 初始化种群
def initialize_population(pop_size, n_variables, bounds):
    population = np.zeros((pop_size, n_variables))
    for i in range(pop_size):
        for j in range(n_variables):
            population[i][j] = np.random.uniform(bounds[j][0], bounds[j][1])
    return population

# 选择操作（锦标赛选择）
def selection(population, fitness, num_parents):
    parents = np.zeros((num_parents, population.shape[1]))
    for i in range(num_parents):
        candidates = np.random.choice(range(len(population)), size=3, replace=False)
        best_candidate = candidates[np.argmin(fitness[candidates])]
        parents[i] = population[best_candidate]
    return parents
```

```python
# 选择操作（锦标赛选择）
def selection(population, fitness, num_parents):
    parents = np.zeros((num_parents, population.shape[1]))
    for i in range(num_parents):
        candidates = np.random.choice(range(len(population)), size=3, replace=False)
        best_candidate = candidates[np.argmin(fitness[candidates])]
        parents[i] = population[best_candidate]
    return parents

# 交叉操作（模拟二进制交叉）
def crossover(parents, offspring_size, crossover_rate):
    offspring = np.zeros(offspring_size)
    for i in range(offspring_size[0]):
        if np.random.rand() < crossover_rate:
            parent1_idx = i % parents.shape[0]
            parent2_idx = (i + 1) % parents.shape[0]
            beta = np.random.rand()
            offspring[i] = parents[parent1_idx] + beta * (parents[parent2_idx] - parents[parent1_idx])
        else:
            offspring[i] = parents[i % parents.shape[0]]
    return offspring

# 变异操作（高斯变异）
def mutation(offspring, bounds, mutation_rate):
    for i in range(offspring.shape[0]):
        if np.random.rand() < mutation_rate:
            for j in range(offspring.shape[1]):
                offspring[i][j] += np.random.normal(0, 0.5)
                # 确保变异后仍在边界内
                offspring[i][j] = np.clip(offspring[i][j], bounds[j][0], bounds[j][1])
    return offspring
```



Rastrigin Function and GA Optimization Path



最优解：x = 0.0024, y = 0.0002
最优适应度：0.0012

7 模拟退火算法

7.1 模拟退火算法的原理

模拟退火算法是一种基于概率的全局优化算法，灵感来源于物理退火过程。其核心思想是通过引入“温度”参数和控制接受劣解的概率，逐步逼近全局最优解。

①物理退火过程：

·在物理退火中，固体材料被加热到高温，然后缓慢冷却，使其达到低能态的稳定晶体结构

·高温时，分子运动剧烈，可以跳出局部低能态；低温时，分子运动减缓，逐渐趋于稳定

②模拟退火的核心思想：

将优化问题类比为物理退火过程：

·**目标函数值** 对应 **能量**

·**解** 对应 **分子状态**

·**温度** 是一个控制参数，决定**接受劣解的概率**

③接受劣解的机制：

在搜索过程中，算法不仅接受更优的解，还以一定概率接受劣解接受劣解的概率由 Metropolis 准则决定

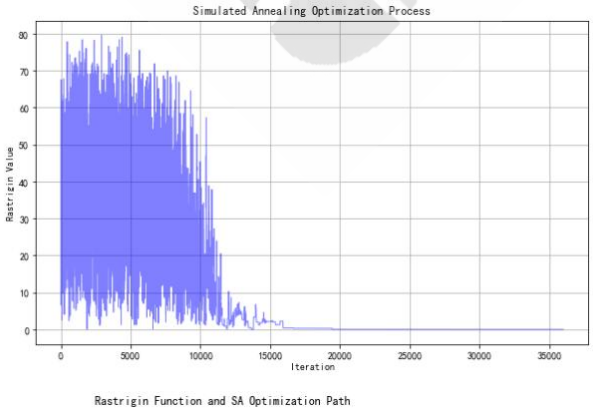
摘　要：将自适应遗传算法与模拟退火算法相结合，形成一种自适应模拟退火遗传算法。该算法不但具备了自适应遗传算法的强大全局搜索能力，也拥有模拟退火算法的强大局部搜索能力。针对月球软着陆轨道优化的特点，利用一种新的参数化方法将轨道优化问题转换为非线性规划问题，并应用提出的自适应模拟退火遗传算法进行优化。数值结果表明：该算法的收敛速度快，优化精度高，且避免了初值敏感、病态梯度和局部收敛等问题，能够搜索到全局最优轨道。
关键词：轨道优化；自适应模拟退火遗传算法；模拟退火算法；遗传算法；月球软着陆；参数化方法
中图分类号：V412.4+1　　文献标识码：A

模拟退火算法实例【论文来源：北京航空航天大学】

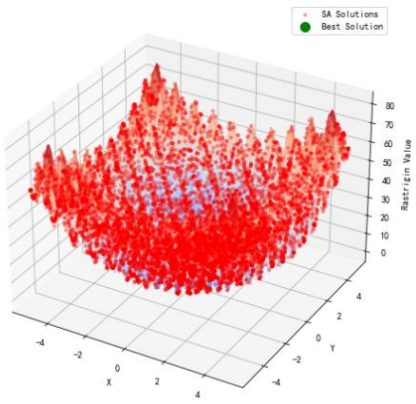
对于多变量决策问题，可以采用模拟退火算法来优化变量矩阵 X。对于某个 X，每一轮循环都会得到三个指标值，即直接盈利值、调换费用、环保性，将这三个值累加到最终决策指标 m_1、m_2、m_3 上。这三个指标中，m_1 的重要性最大，而且数值比重也最大，故将中止循环的条件设为

模拟退火算法实例【2024 年数模国赛 B 题】

7.2 模拟退火算法的事例与代码



Rastrigin Function and SA Optimization Path



```
    while temp > final_temp:
        for _ in range(iterations_per_temp):
            # 生成新解（邻域搜索）
            new_solution = current_solution + np.random.normal(0, 1, size=len(bounds))
            # 确保新解在边界内
            new_solution = np.clip(new_solution, [b[0] for b in bounds], [b[1] for b in bounds])
            new_energy = rastrigin(new_solution)

            # 计算能量差
            delta_energy = new_energy - current_energy

            # 决定是否接受新解
            if delta_energy < 0 or np.random.rand() < np.exp(-delta_energy / temp):
                current_solution = new_solution.copy()
                current_energy = new_energy

                # 更新最优解
                if current_energy < best_energy:
                    best_solution = current_solution.copy()
                    best_energy = current_energy

            # 记录当前状态
            history.append((current_solution.copy(), current_energy))

        # 降温
        temp *= cooling_rate

    return best_solution, best_energy, history
```