Стохастические методы оптимизации

Аксенов Антон

1 Алгоритм имитации отжига

1.1 Описание алгоритма имитации отжига

Приведем описание алгоритма имитации отжига, который используется для минимизации некоторой функции $H : S \to \mathbb{R}$.

1. Фиксируется начальное значение температуры T_0, а также счетчик $k = 0$.

2. Выбирается случайный элемент $s_0 \in S$.

3. Понижается температура $T_{k+1} = \alpha T_k, \alpha \in (0, 1)$.

4. Строится новый элемент $\tilde{s_k} = G(s_k)$, где $G : S \to S$ - фиксированная исследователем случайная функция.

5. Вычисляется $\Delta = H(\tilde{s_k}) - H(s_k)$.

6. Если $\Delta < 0$, то $s_{k+1} = \tilde{s_k}$. В противном случае с вероятностью $p = \exp(-\frac{\Delta}{T_k})$ полагается, что $s_{k+1} = \tilde{s_k}$. С вероятностью $1 - p$ $s_{k+1} = s_k$.

7. Полагается $k := k + 1$, переход к шагу 3, если не выполнен критерий останова (например, по достижению максимального количества итераций).

1.2 Задача о расстановке ферзей

В данной главе будет решаться задача расстановки n ферзей на шахматной доске размера $n \times n$ так, чтобы они друг друга не били. Для решения задачи будет минимизироваться количество различных упорядоченных пар бьющих друг друга ферзей с использованием алгоритма имитации отжига.

В качестве кодировки расстановки ферзей будет использован одномерный массив из n элементов, каждый из которых отвечает за номер горизонтали позиции соответствующего ферзя. Заметим, что для того, чтобы ферзи не били друг друга, необходимо, чтобы они стояли на разных вертикалях и горизонталях, поэтому искомая расстановка является перестановкой элементов множества $\{1, \ldots, n\}$.

Сначала требуется написать функцию, которая будет по расстановке ферзей считать количество упорядоченных пар бьющих друг друга ферзей. Идея алгоритма подсчета в том, что мы будем проходиться по всем парам ферзей и считать эту пару 'бьющей', если модуль разности номеров строк и столбцов позиций ферзей одинаковы. Таким образом, код этой функции в MATLAB имеет следующий вид:

```
function [cnt] = h(q)
n = size(q, 2);
cnt = 0;
for i = 1:n
  for j = i+1:n
    if j - i == abs(q(j) - q(i))
          cnt = cnt + 2;
    end
   end
end
end
```

Также требуется написать функцию, которая будет менять местами два случайных элемента массива, ее код будет иметь вид:

```
function [q_old] = swap(q_old)
n = size(q_old, 2);
idx = randi([1, n], 1, 2);
q_old([idx(1), idx(2)]) = q_old([idx(2), idx(1)]);
end
```

Теперь реализуем алгоритм имитации отжига в виде функции, которая на вход будет принимать три параметра: n - размер доски и одновременно количество ферзей, α - множитель, на который умножается температура, t_0 - начальная температура. В качестве выхода будет выступать итоговая расстановка ферзей q, количество шагов алгоритма k и общее количество итераций.

Основная идея алгоритма состоит в том, что мы будем менять номера горизонталей двух случайных ферзей местами - при уменьшении количества бьющих ферзей выберем новую расстановку, в противном случае

выберем ее с некоторой вероятностью, которая будет зависеть от прироста целевой функции и постепенно уменьшающейся температуры. Когда будет получена искомая расстановка, процесс остановится. Код функции в MATLAB:

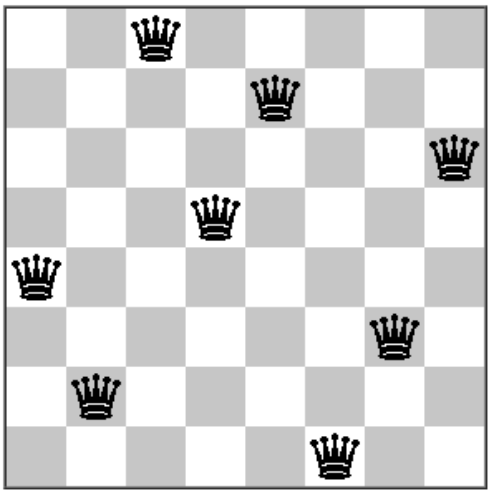
```
function [q, k] = annealing(n, alpha, t_0)
t = t_0;
q = 1:n;
strikes = h(q);
k = 0;
while strikes ~= 0
  t_new = alpha * t;
  q_new = swap(q);
  strikes_new = h(q_new);
  delta_strikes = strikes_new - strikes;
  p = exp(-delta_strikes / t);
  xi = binornd(1, p);
  if or(delta_strikes < 0, xi == 1)
    q = q_new;
    strikes = strikes_new;
  end
  t = t_new;
  k = k + 1;
end
```

Протестируем полученную программу, для чего рассмотрим случай 8 ферзей и обычной шахматной доски 8×8, в качестве α возьмем 0.8 (значение было подобрано на основании результатов нескольких запусков, минимизируя количество итераций алгоритма), а начальную температуру положим равной 100. Тогда получим следующее:

```
>> [q, k] = annealing(8, 0.8, 100)

q =

5     7     1     4     2     8     6     3


k =

60

```

```
>>
```

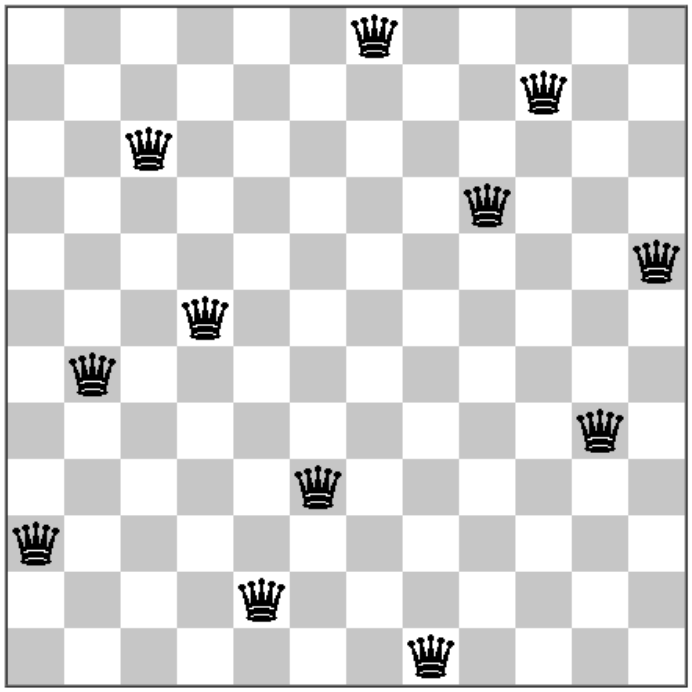
Нетрудно заметить, что в полученой расстановке ферзи не бьют друг друга, продемонстрируем это визуально.



Сделаем то же самое, но с 12 ферзями:

```
>> [q, k]  = annealing(12, 0.8, 100)

q =

10      7      3      6     11      9      1     12      4
        2      8      5


k =

628

>>
```

На этот раз ферзи также не бьют друг друга:



Попробуем увеличить размерность задачи и расставим 20 ферзей.

```
>> [q, k_0] = annealing(20, 0.8, 100);
[q, k_1] = annealing(20, 0.8, 100);
[q, k_2] = annealing(20, 0.8, 100);
[q, k_3] = annealing(20, 0.8, 100);
[q, k_4] = annealing(20, 0.8, 100);
[k_0, k_1, k_2, k_3, k_4]

ans =

499   654   425   467   219

>>
```

Видно, что количество шагов и итераций достаточно низкое, поэтому можно сказать, что алгоритм достаточно быстро справился с поставленной задачей.

Даже если увеличить количество ферзей до 100, то итераций будет по-прежнему немного.

```
>> [q, k_0] = annealing(100, 0.8, 100);
[q, k_1] = annealing(100, 0.8, 100);
```

```
[q, k_2] = annealing(100, 0.8, 100);
[q, k_3] = annealing(100, 0.8, 100);
[q, k_4] = annealing(100, 0.8, 100);
[k_0, k_1, k_2, k_3, k_4]

ans =

6036         3693         5115         8328        12685

>>
```

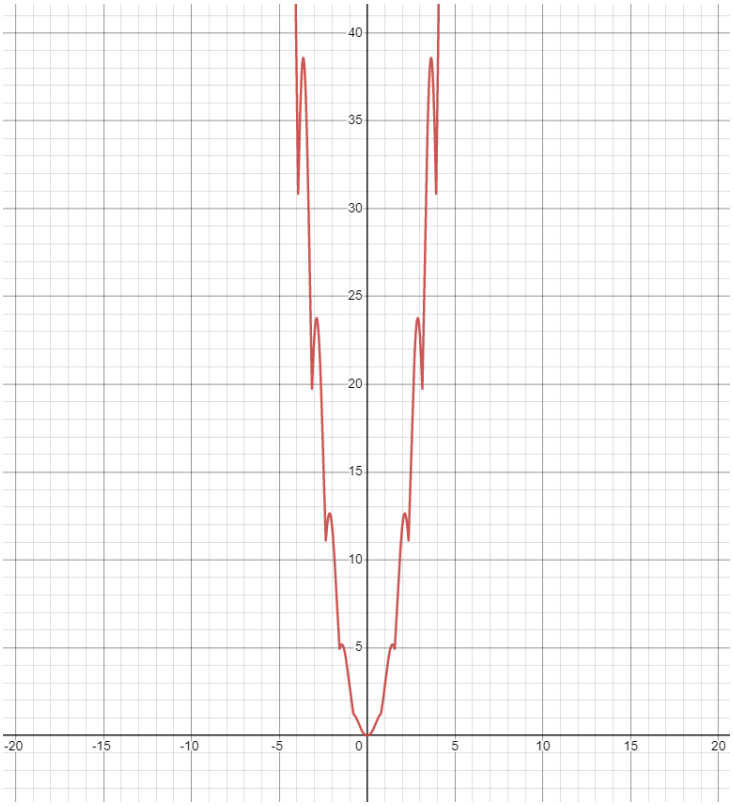
Таким образом, задача по применению метода имитации отжига к решению задачи о расстановке ферзей на шахматной доске так, чтобы они друг друга не били, успешно решена.

1.3 Минимизация функции вещественного аргумента

В данной главе будет использован метод имитиации отжига для минимизации следующей функции

$$f(x) = x^2(2 + |\sin 4x|), x \in \mathbb{R}$$

Построим ее график:



Очевидно, что ее глобальный минимум равен нулю и достигается он при $x = 0$.

Теперь опишем детали применения алгоритма имитации отжига к данной задаче. Преобразование точки x для получения новой точки x' будет иметь следующий вид:

$$x' = x + \xi, \text{ где } \xi \sim U[-1, 1]$$

Начальное приближение x_0 будет браться из равномерного распределения на отрезке $[-10, 10]$, то есть

$$x_0 \sim U[-10, 10]$$

В качестве критерия останова будет выступать достижение температурой близкого к нулю значения.

Тогда код алгоритма имитации отжига будет иметь следующий вид:

```
1 function [x, y] = annealing_alg(alpha, t_0, t_thr)
2 t = t_0;
3 x = unifrnd(-10, 10);
4 y = x^2 * (2 + abs(sin(4 * x)));
```

```
while t > t_thr
  x_new = x + unifrnd(-1, 1);
  y_new = x_new^2 * (2 + abs(sin(4 * x_new)));
  delta_y = y_new - y;
  p = exp(-delta_y / t);
  xi = binornd(1, p);
  if or(delta_y < 0, xi == 1)
    x = x_new;
    y = y_new;
  end
  t = t * alpha;
end
```

Начальную температуру положим равной 100, нижнюю границу температуры - 10^{-10}, а также пусть $\alpha = 0.95$. Запустим несколько раз наш алгоритм:

```
[x, y] = annealing_alg(0.95, 100, 10^(-10))

x =

0.0022


y =

9.6014e-06

>> [x, y] = annealing_alg(0.95, 100, 10^(-10))

x =

-1.2132e-04


y =

2.9446e-08

>> [x, y] = annealing_alg(0.95, 100, 10^(-10))

x =
```

```
26
27 -0.0177
28
29
30 y =
31
32 6.4701e-04
33
34 >>
```

Во всех тестах оптимальная точка колеблется около нуля, который является точкой глобального минимума целевой функции. Также каждая из итераций была выполнена достаточно быстро.

Таким образом, несмотря на то, что алгоритм имитации отжига и не нашел точный глобальный минимум, он подобрался к нему достаточно близко, при этом сделал это довольно быстро.

2 Метод роения частиц

2.1 Описание метода роения частиц

Суть метода роения частиц состоит в создании некоторого сообщества точек, которые будут постепенно эволюционировать, стараясь достичь как можно меньшего значения некоторой функции $f : \mathbb{R}^n \to \mathbb{R}$. Далее кратко опишем данный метод:

1. Случайно генерируется сообщество $G = \{x^1, \ldots, x^M\}$, состоящее из M точек, где $x^m \in \mathbb{R}^n,\ m = 1, 2, \ldots M$.

   Для каждой из них генерируется случайная скорость $v^m(0) \in \mathbb{R}^n, m = 1, 2, \ldots, M$.

   Пусть

   $x^m(t)$ – значение точки m на итерации с номером t, где $t = 0, 1, \ldots$

   $\tilde{x}^m(t) = \underset{s\in\{1,2,\ldots,t\}}{\arg\min} F(x^m(s))$ – наилучшая версия точки m за всю ее историю эволюции до итерации с номером t.

   $\tilde{x}(t) = \underset{m\in\{1,2,\ldots,M\}}{\arg\min} F(x^m(t))$ – лучшая точка популяции итерации t.

   $v^m(t)$ – значение скорости точки m на итерации с номером t.

2. Для каждой точки с номером m пересчитывается ее скорость по следующей формуле

$$v^m(t) = \alpha v^m(t-1) + \beta\xi_1(\tilde{x}^m(t) - x^m(t)) + \gamma\xi_2(\tilde{x}(t) - x^m(t)),$$

   где $\alpha, \beta, \gamma \in (0, 1)$ – настраиваемые параметры, ξ_1, ξ_2 – независимые случайные величины, равномерно распределенные на отрезке $[0, 1]$.

3. Каждая точка делает шаг в соответствии со своей скоростью.

$$x^m(t+1) = x^m(t) + v^m(t)$$

4. Возврат к шагу 2, если не выполнен критерий останова (например, по достижению максимального количества итераций).

2.2 Применение метода для минимизации функций вещественных переменных

В данной главе будет использован метод роения частиц для минимизации нескольких трудных для оптимизации функций.

Для начала реализуем метод роения частиц, причем для ускорения работы алгоритма постараемся избежать использования циклов там, где это возможно. Вместо этого будем стараться выполнять операции в матричном виде.

Также стоит отметить, что начальное приближение вектора положения частиц и их скоростей будет генерироваться из равномерного распределения на отрезке $[\underline{x}, \bar{x}]$ и $[\underline{x} - \bar{x}, \bar{x} - \underline{x}]$ соответственно. Параметры $\underline{x}, \bar{x}$ будут задаваться исследователем.

В таком случае код метода роения частиц будет иметь следующий вид:

```
function [argvalue_global, argmin_global] =
   particle_swarm(f, dim, x_lower, x_upper, alpha,
   beta, gamma, M, L)
X = unifrnd(x_lower, x_upper, M, dim);
V = unifrnd(x_lower - x_upper, x_upper - x_lower, M,
    dim);
X = X + V;
y = f(X);
argvalue_local = y;
argmin_local = X;

[argvalue_global, idxmin_global] = min(
   argvalue_local);
argmin_global = argmin_local(idxmin_global, :);

for t = 1:L
  xi_1 = repmat(unifrnd(0, 1, M, 1), 1, dim);
  xi_2 = repmat(unifrnd(0, 1, M, 1), 1, dim);
   V = alpha * V + beta * xi_1 .* (argmin_local - X)
       +  gamma * xi_2 .* (argmin_global - X);
  X = X + V;
  y = f(X);
  [argvalue_local, idxmin_local] = min([
     argvalue_local, y], [], 2);
  mask_idx = find(idxmin_local == 2);
```

```
  argmin_local(mask_idx, :) = X(mask_idx, :);
  [argvalue_global, idxmin_global] = min(
     argvalue_local);
  argmin_global = argmin_local(idxmin_global, :);
end
```

Протестируем данный метод на функции Розенброка, которая имеет следующий вид:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Видно, что глобальный минимум функции достигается в точке $(x, y) = (1, 1)$

Положим $\underline{x} = -10, \bar{x} = 10, \alpha = 0.95, \beta = 0.2, \gamma = 0.2$, а также количество частиц зафиксируем на уровне 100, количество итераций будет равным 1000. В результате применения метода получим следующее:

```
>> F = @(X)(1 - X(:, 1)).^2 + 100 * (X(:, 2) - X(:,
   1).^2).^2

F =

function_handle with value:

@(X)(1-X(:,1)).^2+100*(X(:,2)-X(:,1).^2).^2

>> [y, x] = particle_swarm(F, 2, -10, 10, 0.95, 0.2,
    0.2, 100, 1000)

y =

5.3524e-22


x =

1.0000    1.0000
```

Таким образом, глобальный минимум был найден, причем алгоритм отработал очень быстро, чему поспособствовало использование матричных вычислений.

Теперь протестируем метод роения частиц на другой трудной для оптимизации функции, имеющей следующий вид:

$$f(x,y,z) = 0.01(x-0.5)^2 + |x^2 - y| + |x^2 - z|$$
$$-0.2 \leqslant x \leqslant 1, \ -0.3 \leqslant y \leqslant 1, \ -0.5 \leqslant z \leqslant 1$$

Известно, что глобальный минимум этой функции достигается в точке $(x, y, z) = (0.5, 0.25, 0.25)$

Применим метод роения частиц к этой функции, пусть $\underline{x} = -1, \bar{x} = 1$, а количество итераций возьмем равным 10^6 (оно было подобрано на основании нескольких запусков). В результате получим следующее:

```
>> F = @(X)0.01 * (X(:, 1) - 0.5).^2 + abs(X(:, 1)
   .^2 - X(:, 2)) + abs(X(:, 1).^2 - X(:, 3))

F =

function_handle with value:

@(X)0.01*(X(:,1)-0.5).^2+abs(X(:,1).^2-X(:,2))+abs(X
   (:,1).^2-X(:,3))

>> [y, x] = particle_swarm(F, 3, -1, 1, 0.95, 0.2,
   0.2, 100, 10^6)

y =

0


x =

0.5000    0.2500    0.2500

>>
```

Видно, что глобальный минимум был найден. Алгоритм работал примерно 10-12 секунд, что можно считать неплохим результатом.

Таким образом, в данной главе был реализован метод роения частиц в MATLAB и с помощью него были успешно найдены глобальные минимумы двух непростых для оптимизации функций.

3 Генетический алгоритм

В данной главе будет реализована одна вариация генетического алгоритма, которая затем будет протестирована на нескольких задачах.

3.1 Описание генетического алгоритма

Для начала кратко опишем генетический алгоритм: данная процедура используется для решения задачи минимизации некоторой функции $f : \mathbb{R}^n \to \mathbb{R}$ на некотором множестве $D \subseteq \mathbb{R}^n$. Также предполагается, что она неотрицательная, иначе говоря $f(x) \geqslant 0 \ \forall x \in D$.

В рамках алгоритма генами будут называться вектора переменных, то есть точки $x \in D$. Хромосомами будут называться компоненты данного вектора.

Идея алгоритма состоит в создании некоторой популяции генов, которая будет постепенно эволюционировать и находить все лучшее приближение минимума $f(x)$. Размер популяции M, количество итераций L и заменяемых генов M_c задается исследователем. Опишем основные шаги рассматриваемой вариации генетического алгоритма:

1. Генерация популяции из M генов случайным образом.

2. Подсчет функции приспособленности гена $fit(x) = \frac{1}{1+f(x)}$ и сортировка популяции по убыванию значений приспособленности.

3. Замена каждой особи с номером $m \in [2, M - M_c]$ на результат ее скрещивания со случайной особью в пуле.

   Скрещивание двух генов $x, y \in \mathbb{R}^n$ подразумевает создание нового гена z, где каждая хромосома z_i будет взята случайно из множества $\{x_i, y_i\}$, причем $\mathbb{P}(z_i = x_i) = \frac{fit(x)}{fit(x)+fit(y)}$.

4. Замена M_c наименее приспособленных особей случайно сгенерированными генами.

5. Возврат к шагу 2, если не выполнен критерий останова (например, по достижению максимального числа итераций L).

Заметим, что лучшая особь на каждой итерации не изменяется, поэтому алгоритм гарантированно не ухудшает уже найденное приближение. Иначе говоря, последовательность значений функции будет монотонно(нестрого) убывать. Если $f(x)$ – ограничена на множестве D, то по теореме Вейерштрасса вышеописанная последовательность будет иметь предел при устремлении количества итераций к бесконечности.

3.2 Минимизация числовой функции

Пусть

$$f(x) = \sum_{k=1}^{n} x_k^2(1 + |\sin(100x_k)|)$$

Требуется минимизировать данную функцию на множестве $D = [-10, 10]^n$. Нетрудно видеть, что глобальным минимумом является нулевой вектор размера n. Однако градиентными методами данную задачу решить представляется очень сложным ввиду большого количества локальных минимумов. Подтверждением этого тезиса является график функции для $n = 1$.

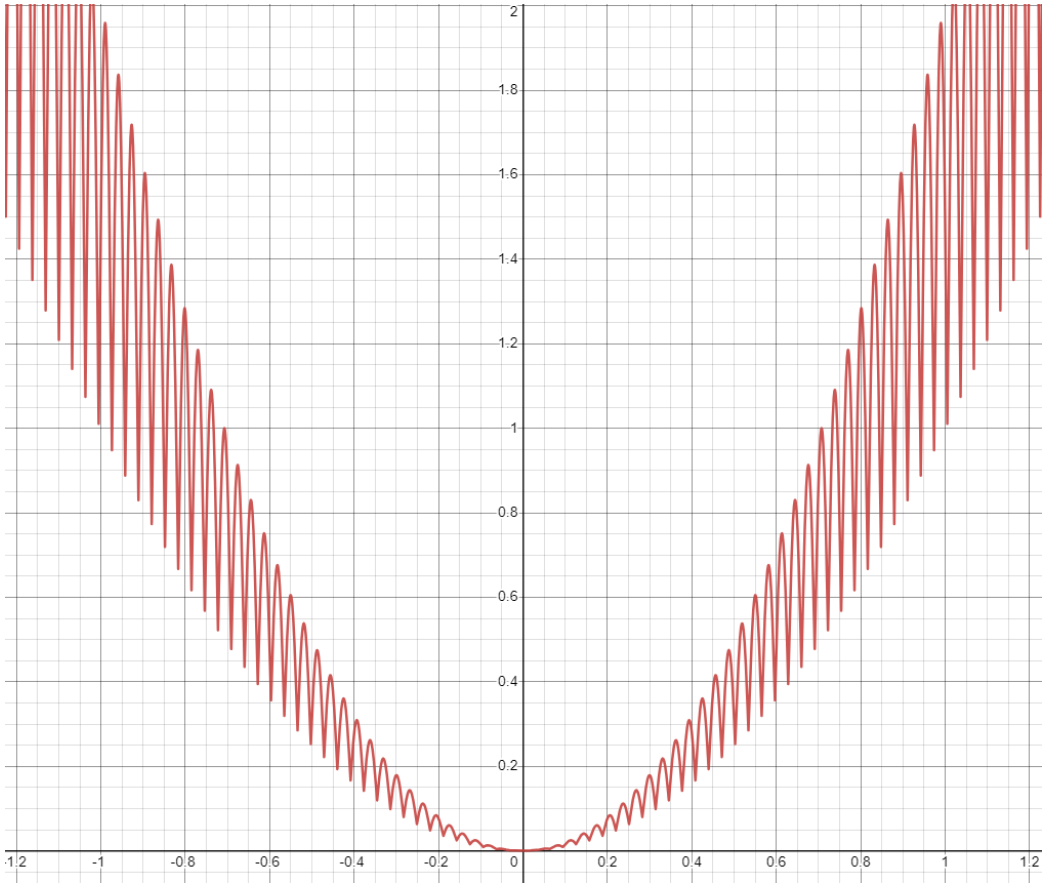


Рис. 1: График функции $f(x) = x^2(1 + |\sin(100x)|)$

Рассмотрим случай $n = 5$. Сначала реализуем расчет значений функции $f(x)$ в MATLAB:

```
function [y] = gen_alg_test_1(X)
dim = size(X, 2);
y = 0;
for i = 1:dim
  y = y + X(:, i).^2 .* (1 + abs(sin(100 * X(:, i))));
end
```

Далее напишем код генетического алгоритма в соответствии с вышеописанными шагами для случая $n = 5$. Генерация гена будет происходить из равномерного распределения на множестве $[-10, 10]^5$.

```
function [y_min, x_min] = gen_alg_unif(f, D, M, M_c, L)
dim = size(D, 1);
X = zeros(M, dim);
for i = 1:dim
  x_lower = D(i, 1);
  x_upper = D(i, 2);
  X(:, i) = unifrnd(x_lower, x_upper, M, 1);
end
for t = 1:L
  fit = ones(M, 1) ./ (1 + f(X));
  X = [fit X];
  X = sortrows(X, 1, "descend");
  fit = X(:, 1);
  X = X(:, 2:(dim + 1));
  male_idx = 2:M - M_c;
  female_idx = randsample(1:length(M), M - M_c - 1, true);
  male = X(male_idx, :);
  female = X(female_idx, :);
  child_male_gen_ind = zeros(M - M_c - 1, dim);
  male_proba = fit(male_idx) / (fit(male_idx) + fit(female_idx));
  for i = 1:M - M_c - 1
          child_male_gen_ind(i, :) = binornd(1, male_proba(i), 1, dim);
  end
  X = male .* child_male_gen_ind + female .* (1 - child_male_gen_ind);
```

```
  replace_ind = (M - M_c + 1):M;
  for i = 1:dim
          x_lower = D(i, 1);
          x_upper = D(i, 2);
          X(replace_ind, i) = unifrnd(x_lower,
             x_upper, M_c, 1);
  end
end
x_min = X(1, :);
y_min = f(x_min);
```

Зафиксируем параметры алгоритма: $M = 1000, M_c = 200, L = 100$. В результате запуска получим:

```
>> [y_min, x_min] = gen_alg_unif(@gen_alg_test_1,
   repmat([-10, 10], 5, 1), 1000, 300, 100)

y_min =

0


x_min =

0     0     0     0     0
```

Видно, что алгоритм нашел глобальный минимум, причем время исполнения не превышает одной секунды.

3.3 Дискретная задача

Перейдем к следующей задаче – имеется множество $A = \{1, 2, \ldots, n\}$, которое необходимо разбить на два непересекающихся множества K_1 и K_2 так, чтобы

$$H(K_1, K_2) = \left| \sum_{a_k \in K_1} a_k - \sum_{a_m \in K_2} a_m \right|$$

была минимальна. Причем $K_1 \cup K_2 = A, K_1 \cap K_2 = \varnothing$.

Известно, что данная задача не решается точно за полиномиальное время – воспользуемся генетическим алгоритмом для нахождения приближенного решения.

Заметим, что разбиение A взаимно-однозначно задается характеристическим вектором подмножества K_1, то есть вектором $x = \{\mathbb{I}\{i \in K_1\}\}_{i=1}^n$. Тогда вышеописанная задача сводится к минимизации следующей функции

$$f(x) = \left| \sum_{k=1}^{n} k \cdot \mathbb{I}\{x_k = 1\} - \sum_{k=1}^{n} k \cdot \mathbb{I}\{x_k = 0\} \right| = \left| \sum_{k=1}^{n} k \cdot (\mathbb{I}\{x_k = 1\} - \mathbb{I}\{x_k = 0\}) \right|,$$

где $x \in \{0, 1\}^n$

Решим эту задачу, используя MATLAB. Реализация функции $f(x)$ будет иметь вид:

```
function [y] = gen_alg_test_2(X)
dim = size(X, 2);
start = 1;
stop = start + dim - 1;
A = repmat(start:stop, size(X, 1), 1);

Z = zeros(size(X, 1), dim);
mask_0 = X == 0;
Z(mask_0) = A(mask_0);
sum_0 = sum(Z, 2);

Z = zeros(size(X, 1), dim);
mask_1 = X == 1;
Z(mask_1) = A(mask_1);
sum_1 = sum(Z, 2);

y = abs(sum_1 - sum_0);
```

Генетический алгоритм будет применяться аналогично предыдущей задаче, однако теперь для создания случайного гена каждая хромосома будет независимо генерироваться из распределения Бернулли с параметром $p = \frac{1}{2}$. Тогда код генетического алгоритма будет следующим:

```
function [y_min, x_min] = gen_alg_bern(f, dim, M,
   M_c, L)
X = zeros(M, dim);
X = binornd(1, 0.5, M, dim);
for t = 1:L
  fit = ones(M, 1) ./ (1 + f(X));
  X = [fit X];
```

```
  X = sortrows(X, 1, "descend");
  fit = X(:, 1);
  X = X(:, 2:(dim + 1));
  male_idx = 2:M - M_c;
  female_idx = randsample(1:length(M), M - M_c - 1,
     true);
  male = X(male_idx, :);
  female = X(female_idx, :);
  child_male_gen_ind = zeros(M - M_c - 1, dim);
  male_proba = fit(male_idx) / (fit(male_idx) + fit(
     female_idx));
  for i = 1:M - M_c - 1
    child_male_gen_ind(i, :) = binornd(1, male_proba
       (i), 1, dim);
  end
  X = male .* child_male_gen_ind + female .* (1 -
     child_male_gen_ind);
  replace_ind = (M - M_c + 1):M;
  X(replace_ind, :) = binornd(1, 0.5, M_c, dim);
end
x_min = X(1, :);
y_min = f(x_min);
```

Оставим параметры алгоритма фиксированными на прежнем уровне и запустим оптимизацию для $n = 100, n = 1000, n = 10000$.

```
>> [y_min, ~] = gen_alg_bern(@gen_alg_test_2, 100,
   1000, 300, 100)

y_min =

0

>> [y_min, ~] = gen_alg_bern(@gen_alg_test_2, 1000,
   1000, 300, 100)

y_min =

0


>> [y_min, ~] = gen_alg_bern(@gen_alg_test_2, 10000,
```

```
    1000, 300, 100)

y_min =

22
```

Заметим, что при $n = 100$ и $n = 1000$ генетический алгоритм нашел оптимальное разбиение исходного множества на два подмножества с одинаковой суммой. При $n = 10000$ суммы уже получились не одинаковыми, однако значение целевой функции оказалось относительно близким к нулю.

3.4 Линейное программирование

Задача линейного программирования заключается в минимизации линейной функции при условии ограничений-равенств и ограничений-неравенств, где функции-ограничения имеют линейный вид. Кроме того, на некоторые переменные может быть наложено ограничение целочисленности и нахождение на некотором отрезке. Таким образом, данная задача будет иметь следующий вид

$$\begin{cases} f(x) = c^T x \to \min\limits_{x \in \mathbb{R}^n} \\ Ax \leqslant b \\ A_{eq}x = b_{eq} \\ lb_i \leqslant x_i \leqslant ub_i,\ i \in J \\ x_i \in \mathbb{Z},\ i \in Z \end{cases},$$

где $A, A_{eq} \in Mat_{m\,n}(\mathbb{R})$; $b, b_{eq} \in \mathbb{R}^m$; $c \in \mathbb{R}^n$; $J, Z \subseteq \{1, 2, \ldots, n\}$;
$lb_i, ub_i \in \mathbb{R}$, $i \in J$

Теперь от данной общей постановки перейдем к более узкой, где нет ограничений-равенств и верхних границ значений переменных. Также будем рассматривать только неотрицательные значения переменных и параметров задачи. Кроме того, теперь задача будет заключаться в максимизации целевой функции. То есть

$$\begin{cases} f(x) = c^T x \to \max\limits_{x \in \mathbb{R}^n} \\ Ax \leqslant b \\ x_i \geqslant 0,\ i \in \{1, 2, \ldots, n\} \\ x_i \in \mathbb{Z},\ i \in Z \end{cases},$$

где $A \in Mat_{m\,n}(\mathbb{R}_+)$; $b \in \mathbb{R}^m_+$; $c \in \mathbb{R}^n_+$; $Z \subseteq \{1, 2, \ldots, n\}$;

Данная задача чаще всего решается симплекс-методом, однако генетический алгоритм к ней также применим – реализуем его в MATLAB.

Заметим, что основная часть этого алгоритма уже была реализована в вышеописанных задачах, поэтому необходимо лишь адаптировать его под задачу линейного программирования. В частности, необходимо поменять генерацию генов, а также подсчет функции приспособленности.

В силу того что все переменные и параметры неотрицательны, для каждой переменной может быть вычислен отрезок, в который она точно должна попасть в соответствии с ограничениями-неравенствами. Легко видеть, что

$$x_j \in [0, v_j] \text{ , где } v_j = \min_{i:a_{ij}\neq 0} \frac{b_i}{a_{ij}}$$

Последнее неравенство получается, если обнулить значения всех переменных, кроме той, что имеет номер j, далее решить систему $Ax \leqslant b$. В силу вышеупомянутой неотрицательности всех параметров и переменных полученное неравенство обязательно соблюдается для допустимых точек.

Тогда каждую веществennозначную хромосому для j - ой компоненты гена можно генерировать из равномерного распределения на отрезке $[0, v_j]$, а каждую целочисленную - на множестве $\{0, 1, \dots \lfloor v_j \rfloor\}$. Если правая граница равна нулю, то соответствующая переменная всегда будет равна нулю.

Однако стоит отметить, что вышеописанное условие является лишь необходимым, но не достаточным – оно гарантирует неотрицаетльность всех переменных и целочисленность соответствующих компонент гена, однако условие $Ax \leqslant b$ нужно проверять дополнительно. Чтобы учесть это в рамках генетического алгоритма, присвоим нулевые значения приспособленности тем генам, для которых не выполняется $Ax \leqslant b$. Для остальных особей приспособленность будет считаться как

$$fit(x) = c^T x$$

Тогда код генетического алгоритма будет иметь следующий вид

```
function [y_best, x_best] = gen_alg_lp(A, c, b,
   int_idx, M, M_c, L)
% get continious variables indexes
n = size(A, 2);
cont_idx = 1:n;
cont_idx(int_idx) = [];
% determine bounds of variables
```

```
B = repmat(b, 1, n);
x_max = min(B ./ (A + 1e-8))';
x_min = zeros(n, 1);

% generate population
X = zeros(M, n);
% continious variables
for i = cont_idx
  x_lower = x_min(i);
  x_upper = x_max(i);
  X(:, i) = unifrnd(x_lower, x_upper, M, 1);
end
% integer variables
for i = int_idx
  x_lower = x_min(i);
  x_upper = floor(x_max(i)) + 1;
    X(:, i) = randi([x_lower, x_upper], M, 1);
end

% main loop
for t = 1:L
  % profit calc
  y = X * c;
  % fit func calc and checking constraint
  res_usage = A * X';
  res_cond_mask = all(res_usage <= repmat(b, 1, M),
     1);
  fit = y;
  fit(~res_cond_mask) = 0;
  % sort gens
  X = [fit X];
  X = sortrows(X, 1, "descend");
  fit = X(:, 1);
  X = X(:, 2:(n + 1));
  % crossing
  male_idx = 2:M - M_c;
  female_idx = randsample(1:length(M), M - M_c - 1,
     true);
  male = X(male_idx, :);
  female = X(female_idx, :);
  child_male_gen_ind = zeros(M - M_c - 1, n);
```

```
  male_proba = fit(male_idx) ./ (fit(male_idx) + fit
     (female_idx) + 1e-8);
  for i = 1:M - M_c - 1
    child_male_gen_ind(i, :) = binornd(1, male_proba
       (i), 1, n);
  end
  X = male .* child_male_gen_ind + female .* (1 -
     child_male_gen_ind);
  replace_ind = (M - M_c + 1):M;
  % replacing M_c gens
  % cont variables
  for i = cont_idx
    x_lower = x_min(i);
    x_upper = x_max(i);
    X(replace_ind, i) = unifrnd(x_lower, x_upper,
       M_c, 1);
  end
  % integer variables
  for i = int_idx
    x_lower = x_min(i);
    x_upper = floor(x_max(i)) + 1;
    X(replace_ind, i) = randi([x_lower, x_upper],
       M_c, 1);
  end
end
x_best = X(1, :);
y_best = x_best * c;
```

Теперь протестируем данный алгоритм на конкретной задаче и сравним полученный результат с симплекс-методом.

Пусть

$$A = \begin{pmatrix} 10 & 20 & 30 \\ 60 & 20 & 10 \\ 35.4 & 40.2 & 50 \end{pmatrix} \quad b = \begin{pmatrix} 1000 \\ 750 \\ 830.5 \end{pmatrix} c = \begin{pmatrix} 30 \\ 15 \\ 22.5 \end{pmatrix} x_2, x_3 \in \mathbb{Z}$$

Параметры алгоритма возьмем следующими: $M = 1000, M_c = 200, L = 300$. Тогда выполним максимизацию исходной функции с помощью генетического алгоритма и встроенного в MATLAB симплекс-метода.

```
>> M = 1000;
M_c = 200;
```

```
3  L = 300;
4  >> A = [[10, 20, 30]; [60, 20, 10]; [35.4, 40.2,
       50]];
5  b = [1000, 750, 830.5]';
6  c = [30, 15, 22.5]';
7  int_idx = [2, 3];
8  >> simp_meth = intlinprog(-c, int_idx, A, b ,[], []
       , zeros(3, 1))
9  LP:                 Optimal objective value is
       -528.968254.
10
11 Heuristics:         Found 2 solutions using ZI round.
12 Upper bound is -511.009070.
13 Relative gap is 2.72%.
14
15 Cut Generation:     Applied 1 mir cut.
16 Lower bound is -524.957627.
17 Relative gap is 0.00%.
18
19
20 Optimal solution found.
21
22 Intlinprog stopped at the root node because the
       objective value is within a gap tolerance of the
       optimal value,
23 options.AbsoluteGapTolerance = 0 (the default value)
       . The intcon variables are integer within
       tolerance,
24 options.IntegerTolerance = 1e-05 (the default value)
       .
25
26
27 simp_meth =
28
29 10.7486
30 0
31 9.0000
32
33 >> [~, x_best] = gen_alg_lp(A, c, b, int_idx, M, M_c
       , L)
34
```

```
x_best =

10.7450         0    9.0000
```

Видно, что найденные приближения решения задачи линейного программирования, построенные симплекс-методом и генетическим алгоритмом, равны с точностью до 3 знака после запятой, что свидетельствует о том, что генетический алгоритм успешно справился с данной задачей. Причем его время работы не превышает 1-2 секунд.