МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ
КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ

Звіт до лабораторної роботи №1 на тему «Метод роя часток»

Виконав студент групи ОМ-3
Скибицький Нікіта

Київ — 2019

Зміст

1 Неформальний опис алгоритму

1.1 Власне опис

Алгоритм працює за допомогою популяції (рою) кандидатних рішень або частинок. Кожна з них рухається у обмеженій частині простору, керуючись двома факторами. Перший — це особиста найкраща позиція частинки під час усього попереднього пошуку. Другий — найкраще положенням по всьому рою. Тим самим у якийсь момент виникає загальна тенденція до оптимуму і рій сходиться на оптимальному розв'язку.

1.2 Історія винаходу методу

Метод виник в середині 90-х років XX сторіччя, авторами вважають психолога Джеймса Кеннеді (eng. *Kennedy*) та інженера Рассела Еберхарта (eng. *Eberhart*). В подальшому численні дослідники запропонували різні модифікації цього методу.

1.3 Сфери застосування

Взагалі кажучи, метод можна використовувати для пошуку екстремуму будь-якої функції, яка може бути обчислена на заданій множині вхідних даних і не обов'язково задана в аналітичному вигляді.

1.4 Можливі модифікації

1.4.1 Збурення рою

Однією з можливих модифікацій, яка допомагає рою виходити з локальних екстремумів є збурення рою. Це означає, що якщо більша ($> 80\%$) частина часток рою опинилася у маленькому околі якоїсь однієї точки, то у цьому околі можна лишилити лише малу ($\approx 10\%$) частину часток, а решту заново розподілити по області пошуку, наприклад рівномірним випадковим чином.

Цьому присвячені роботи [1–4].

1.4.2 Багатокритеріальна оптимізація

Метод рою часто також може бути застосований до задач багато критеріальної оптимізації, тоді функція порівняння відображає поняття парето-оптимальності, з лексикографічним способом вирішення конфліктів.

Цьому присвячені роботи [5–7].

2 Формалізований опис алгоритму

Рій частинок представляє собою множину $\left\{P_j, j = \overline{1, L}\right\}$. Кожна частинка і весь рій в цілому характеризується набором параметрів, що визначає їх стан в конкретний момент часу k:

- $X_j^{(k)} = \left(x_{j,1}^{(k)}, x_{j,2}^{(k)}, \ldots, x_{j,n}^{(k)}\right)$, $j = \overline{1, L}$ — положення частинки P_j в n-вимірному просторі.
- Для кожної частинки P_j в кожний момент часу k може бути обчислене значення цільової функції, часто її називають фітнес-функцією (або функцією пристосваності):

$$F_j^{(k)} = F\left(X_j^{(k)}\right) = F\left(x_{j,1}^{(k)}, x_{j,2}^{(k)}, \ldots, x_{j,n}^{(k)}\right). \tag{2.1}$$

- $V_j^{(k)} = \left(v_{j,1}^{(k)}, v_{j,2}^{(k)}, \ldots, v_{j,n}^{(k)}\right)$, $j = \overline{1, L}$ — швидкості частинки P_j в кожному напрямку.
- $XL_j = (xl_{j,1}, xl_{j,2}, \ldots, xl_{j,n})$, $j = \overline{1, L}$ — найкраще положення частинки P_j за весь час.
- $XG = (xg_1, xg_2, \ldots, xg_n)$ — найкраще положення всіх частинок за весь час.

Алгоритм представляє собою ітераційний процес з дискретним часом.

На кожній ітерації кожна частинка переміщується з попереднього положення в своє нове положення за певним законом, при цьому закон переміщення кожної частинки рою враховує своє найкраще (екстремальне положення, локальний екстремум) і найкраще положення найкращої частинки рою (глобальний екстремум).

Для ініціалізації ітераційного процесу початковий стан $X_j^{(0)}$ кожної частинки P_j рою визначається рівномірно розподіленою випадковою величиною заданою на вхідній множині D.

Початкові швидкості руху кожної частинки також визначаються як випадкові величини рівномірно розподілені на n-вимірному паралелепіпеді $\Pi = [-\varepsilon, \varepsilon]^n$, де ε — деяке мале число. Допускається також і нульове значення швидкості.

На кожній ітерації (в кожній момент часу) обчислюється нова швидкість кожної частинки рою за формулою:

$$V_{j,i}^{(k+1)} = \omega \cdot V_{j,i}^{(k)} + a_1 \cdot \text{rand} \cdot \left(xl_{j,i} - x_{j,i}^{(k)} \right) + a_2 \cdot \text{rand} \cdot \left(xg_i - x_{j,i}^{(k)} \right), \tag{2.2}$$

де $\omega = 1 - \varepsilon$ — коефіцієнт інерції ($\varepsilon \approx 10^{-2}$ тут вже інше), a_1, a_2 — постійні значення прискорень, rand — випадкова величина рівномірно розподілена на відрізку $[0, 1]$.

Після обчислення значення швидкості обчислюється нове положення кожної частинки

$$X_j^{(k+1)} = X_j^{(k)} + V_j^{(k)}. \tag{2.3}$$

Критерієм зупинки може бути досягнення заданого числа ітерацій або будь-який інший критерій.

Окремо зауважимо, що якщо якась частинка P_j рою виходить за межі допустимої області D то її можна і варто повернути, наприклад у положення XL_j.

3 Код програмного продукту

3.1 Використані бібліотеки

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from random import uniform, seed
from math import cos, pi, exp, sqrt
from typing import List
```

3.2 Параметри алгоритму

```
seed(65537)

_omega, step, max_iter, grid_sz = .99, .1, 1 << 7, 100 + 1


def sign(x: float) -> float:
        return (x > 0) - (x < 0)


def a(Xj: List[float], XLj: List[float], XG: List[float]) -> float:
        b1, b2 = f(Xj) - f(XLj), f(Xj) - f(XG)

        return sign(b1) * (.5 + b1**2) / (1 + b1**2 + b2**2), \
                sign(b2) * (.5 + b2**2) / (1 + b1**2 + b2**2)


def omega(k):
        return _omega * (1 - k / (2 * max_iter))
```

3.3 Початковий стан рою

```
X = [[uniform(b_lo, b_up) for i in range(n)] for j in range(l)]
V = [[uniform(-step, step) for i in range(n)] for j in range(l)]
F = [f(X[j]) for j in range(l)]
XL, XG = X[:], X[F.index(min(F))]
```

3.4 Побудова графіків у $\mathbb{R}^2$

```
def plot():
        fig = plt.figure(figsize=(10, 10))
        plt.axis([b_lo, b_up, b_lo, b_up])
        xs = [b_lo + (b_up - b_lo) * i / (grid_sz - 1) for i in range(grid_sz)]
        x1s, x2s = [[xs[i] for i in range(grid_sz)] for j in range(grid_sz)], \
                [[xs[j] for i in range(grid_sz)] for j in range(grid_sz)]
        fs = [[f([x1s[i][j], x2s[i][j]]) for j in range(grid_sz)]
                for i in range(grid_sz)]
        fs_raw = [fs[i][j] for i in range(grid_sz) for j in range(grid_sz)]
        levels = MaxNLocator(nbins=10).tick_values(min(fs_raw), max(fs_raw))
        plt.contourf(x1s, x2s, fs, levels=levels, cmap="RdBu_r")
        plt.colorbar()
        plt.contour(x1s, x2s, fs, levels=levels, colors='k')
        px1s, px2s = [X[j][0] for j in range(l)], [X[j][1] for j in range(l)]
        plt.scatter(px1s, px2s, c='k')
        plt.quiver(px1s, px2s, [V[j][0] for j in range(l)], [V[j][1]
                for j in range(l)], scale=10)
        plt.draw()
        plt.pause(.1)
        plt.savefig(f'img/{k}.png')
        plt.close()


def plot_err():
        ks = list(range(max_iter + 1))
        fig = plt.figure(figsize=(10, 10))
        plt.plot(ks, err_best, 'r.', label='fitness of the best particle')
        plt.plot(ks, err_local, 'g--', label='average fitness of xl_j')
        plt.plot(ks, err_mean, 'b-', label='average fitness of x_j')
        plt.legend(loc='upper right')
        plt.draw()
        plt.pause(5)
        plt.savefig(f'img/err_plot.png')
        plt.close()
```

3.5 Логування значень цільової функції

```
err_best = [f(XG)]
xy_best = [XG]
err_local = [sum(f(XL[j]) for j in range(l)) / l]
err_mean = [sum(f(X[j]) for j in range(l)) / l]


def err():
        global err_mean, err_local, err_best
        xy_best.append(XG)
        err_best.append(f(XG))
        err_local.append(sum(f(XL[j]) for j in range(l)) / l)
        err_mean.append(sum(f(X[j]) for j in range(l)) / l)


def log_err():
        with open(f'err.log', 'w') as out:
                for k in range(max_iter):
                        out.write(f'iteration number {k}:\n'
                                f'\txy_best   = {xy_best[k]},\n'
                                f'\terr_best  = {err_best[k]},\n'
                                f'\terr_local = {err_local[k]},\n'
                                f'\terr_mean  = {err_mean[k]}.\n\n')
```

3.6 Власне алгоритм

```
for k in range(max_iter):
        rand = [uniform(0, step) for j in range(l)]

        for j in range(l):
                a1, a2 = a(X[j], XL[j], XG)
                V[j] = [omega(k) * V[j][i] + rand[j] * (a1 * (XL[j][i] - X[j][i]) + \
                        a2 * (XG[i] - X[j][i])) for i in range(n)]

        if n == 2:
                plot()

        err()

        for j in range(l):
                X[j] = [X[j][i] + V[j][i] for i in range(n)]

        X = [X[j] if all(b_lo < X[j][i] < b_up for i in range(n)) else XL[j]
                for j in range(l)]
        XL = [XL[j] if f(XL[j]) < f(X[j]) else X[j] for j in range(l)]
```

```
        F = [f(X[j]) for j in range(l)]
        XG = XG if f(XG) < min(F) else X[F.index(min(F))]
```

4 Тестування програмного продукту

Було розглянути три класичні тестові функції для задач оптимізації, у тому числі оптимізації з обмеженнями.

4.1 Функція Растрігіна

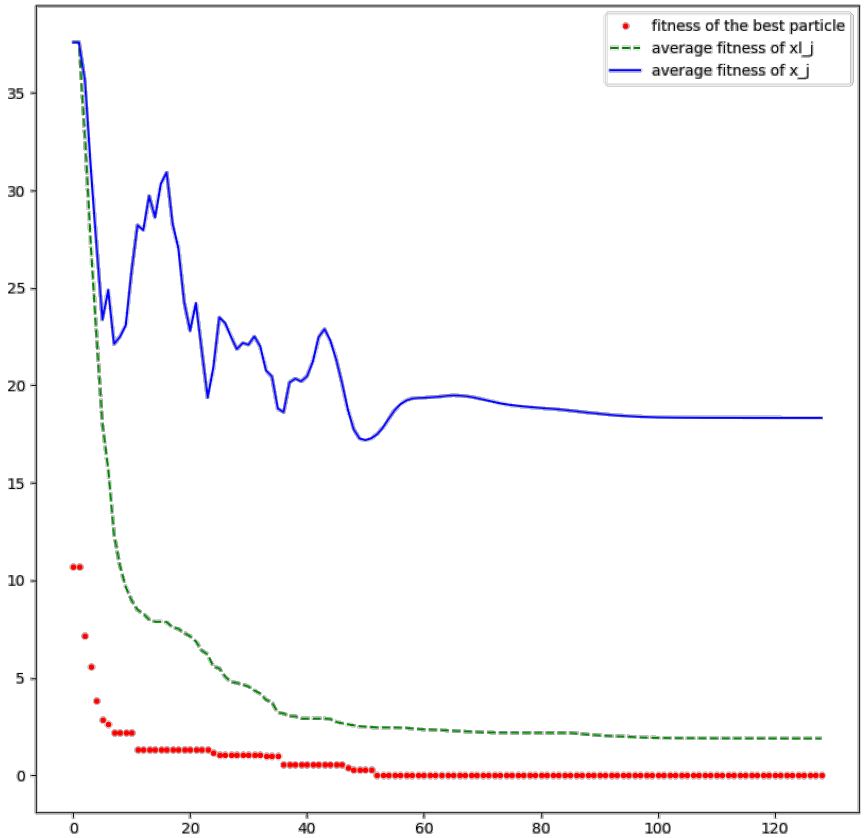
Аналітичний вигляд:

$$f(x_1, \ldots, x_n) = 10n + \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i). \tag{4.1}$$

У коді задається наступним чином:

```
def rastrigin(Xj: List[float]) -> float:
        """ Rastrigin function """
        return 10 * n + sum(Xj[i]**2 - 10 * cos(2 * pi * Xj[i])
                for i in range(n))
```

При тестуванні на $\mathbb{R}^2$ з розміром рою у 50 часток було отримано наступний графік похибки від часу:



4.2 Функція Розенброка

Аналітичний вигляд:

$$f(x_1, \ldots, x_n) = 100 \sum_{i=1}^{n-1} (x_{i+1} - x_i)^2 + (1 - x_i)^2. \tag{4.2}$$

У коді задається наступним чином:

```
def rosenbrock(Xj: List[float]) -> float:
        """ Rosenbrock's function """
        return sum(100 * (Xj[i + 1] - Xj[i]**2)**2 + (1 - Xj[i])**2
                for i in range(n - 1))
```

4.3 Функція Розенброка з обмеженнями

Аналітичний вигляд:

$$L(x_1, \ldots, x_n) = f(x) + 100((x_i - 1)^3 - x_i + 1)_+^2 + 100(x_i + x_{i+1}^3 - 2)_+^2, \tag{4.3}$$

де $(\cdot)_+ = \max\{\cdot, 0\}$ — невідє'мна частина числа.

У коді задається наступним чином:

```
def rosenbrock_penalty(Xj: List[float]) -> float:
        """ Rosenbrock's function with penalty """
        p1 = 100 * max((Xj[0] - 1)**3 - Xj[1] + 1, 0)**2
        p2 = 100 * max(Xj[0] + Xj[1] - 2, 0)**2
        return sum(100 * (Xj[i + 1] - Xj[i]**2)**2 + (1 - Xj[i])**2
                for i in range(n - 1)) + p1 + p2
```

Література

[1] G. Evers, "An automatic regrouping mechanism to deal with stagnation in particle swarm optimization," Master's thesis, The University of Texas, Department of Electrical Engineering, 2009. http://www.georgeevers.org/thesis.pdf.

[2] M. Lovbjerg and T. Krink, "Proceedings of the fourth congress on evolutionary computation (cec)," in *Extending Particle Swarm Optimisers with Self-Organized Criticality*, vol. 2, pp. 1588–1593, 2002.

[3] Z. Xinchao, "A perturbed particle swarm algorithm for numerical optimization," *Applied Soft Computing*, vol. 10, no. 1, pp. 119–124, 2010. https://www.sciencedirect.com/science/article/pii/S1568494609000830.

[4] X.-F. Xie, W.-J. Zhang, and Z.-L. Yang, "Congress on evolutionary computation (cec)," in *A dissipative particle swarm optimization*, pp. 1456–1461, 2002. http://www.wiomax.com/team/xie/paper/CEC02.pdf.

[5] K. Parsopoulos and M. Vrahatis, "Proceedings of the acm symposium on applied computing (sac)," in *Particle swarm optimization method in multiobjective problems*, pp. 603–607, 2002. https://dl.acm.org/citation.cfm?doid=508791.508907.

[6] C. Coello Coello and M. Salazar Lechuga, "Congress on evolutionary computation (cec'2002)," in *MOPSO: A Proposal for Multiple Objective Particle Swarm Optimization*, pp. 1051–1056, 2002. https://dl.acm.org/citation.cfm?id=1252327.

[7] K. Mason, J. Duggan, and E. Howley, "Multi-objective dynamic economic emission dispatch using particle swarm optimisation variants," *Neurocomputing*, vol. 270, pp. 188–197, 2017. https://www.sciencedirect.com/science/article/pii/S0925231217310767.