

حسابگری زیستی؛

یافتن کمینه و بیشینه توابع ریاضی به کمک روش بهینه‌سازی از دحام ذرات؛

کامیار میرزاوزیری؛ ۶۱۰۳۹۶۱۵۲

۱ مقدمات و ابزارها

۱.۱ ترسیم‌گر

کلاسی به اسم Visualizer پیاده‌سازی می‌کنیم که وظیفه ترسیم توابع و ذرات را دارد. این کلاس از کتابخانه matplotlib برای ترسیم استفاده می‌کند و کد آن در فایل utils.py موجود است اما چون مربوط به موضوع پروژه نمی‌باشد به آن نمی‌پردازیم.

۲.۱ ذره

کلاسی به اسم Particle به نمایندگی از ذرات ایجاد می‌کنیم که هر instance آن نماینده یک ذره باشد. هر ذره باید مختصات فعلی خود یعنی x و y را به خاطر داشته باشد و محاسبه مقدار تابع یعنی z با داشتن این مقادیر امکان‌پذیر است اما برای تسريع الگوریتم این مقدار را نیز در اشیاء نگه می‌داریم. همچنین هر ذره باید بهترین وضعیتی که تا کنون داشته را به خاطر داشته باشد، و به پارامترهایی نظیر ω و c_i نیز دسترسی داشته باشد. پس این کلاس را به صورت زیر پیاده‌سازی می‌کنیم.

```
def __init__(self, x, y, func, x_range, y_range, better, omega, c_1, c_2, c_3, t, cooling_ratio):
    self.func = func
    self.x_range = x_range
    self.y_range = y_range
    self.better = better
    self.omega = omega
    self.c_1 = c_1
    self.c_2 = c_2
    self.c_3 = c_3
    self.t = t
    self.cooling_ratio = cooling_ratio
    self.x = x
    self.y = y
    self.z = self.func(self.x, self.y)
    self.best = (self.x, self.y, self.z)
    self.v_x = 0
    self.v_y = 0
```

- پارامتر x نشان‌دهنده مختصات ذره در محور x ‌ها می‌باشد.
 - پارامتر y نشان‌دهنده مختصات ذره در محور y ‌ها می‌باشد.
 - پارامتر $func$ تابعی است که z بر اساس آن از روی x و y محاسبه می‌شود و می‌تواند f یا g باشد.
 - پارامتر x_range یک لیست دو تایی شامل حداقل و حداکثر مقدار x است که در تابع f مقادیر $[10, -10]$ و در g مقادیر $[-100, 100]$ را دارد.
 - پارامتر y_range مشابه x_range اما برای y می‌باشد.
 - پارامتر $better$ تابعی با دو ورودی می‌باشد که اگر ورودی اول بهتر از دوم باشد `True` برمی‌گرداند. برای f که به دنبال ماکسیمم هستیم این تابع باید بزرگ‌تری باشد و برای g کوچک‌تری.
 - پارامترهای c_omega , c_1 و c_2 همان پارامترهای الگوریتم PSO می‌باشند.
 - برای پارامتر c_3 لازم است اشاره کنیم که در این تمرین تصمیم گرفتیم تا برای آزمایش مقداری نویز به ذرات اضافه کنیم تا قابلیت پویش را بالا ببریم. به این صورت که در هر تکرار الگوریتم در زمان تعیین سرعت، عددی تصادفی را نیز به سرعت اضافه می‌کنیم که می‌تواند مثبت یا منفی باشد و این عدد در ضربی c_3 ضرب شده و نتیجه به بردار سرعت اضافه می‌شود.
 - پارامتر t برای کنترل نویز به کار می‌رود. نویزی که توضیح دادیم را به کمک الگوریتم تبرید تدریجی کنترل می‌کنیم پس نیاز به پارامتری به عنوان دمای اولیه داریم.
 - پارامتر $cooling_ratio$ مربوط به همان الگوریتم تبرید تدریجی نویز ذرات می‌باشد.
- در هر تکرار الگوریتم PSO لازم است ذره را بروزرسانی کنیم. برای این منظور متدهای `update` را روی کلاس ذره تعریف می‌کنیم. در این کلاس ابتدا بررسی می‌کنیم و اگر ذره‌ای در نقطه بهینه سراسری قرار داشت آن را حرکت نمی‌دهیم تا از از دست دادن بهینه سراسری جلوگیری کنیم و به قابلیت انتفاع کمک کنیم. اما اگر چنین نبود بردار سرعت را مطابق با روش PSO و افروزن نویز محاسبه می‌کنیم و سپس مکان جدید ذره را محاسبه می‌کنیم. در این مرحله امکان دارد ذره از محدوده مقادیر معتبر خود خارج شود. در این صورت بردار سرعت آن را منفی می‌کنیم و دوباره روی آن اثر می‌دهیم (مشابه اثری که برخورد با دیواره برای یک ذره دارد و باعث می‌شود در جهت عکس بازگردد)، اما اگر بردار سرعت بزرگ بود و در جهت عکس هم از محدوده خارج می‌شد ناچاریم آن را در انتهای محدوده قرار دهیم. در نهایت مقدار z را بر اساس x و y جدید محاسبه کرده و در صورتی که از بهترین خود ذره بهتر بود آن را جایگزین می‌کنیم.

```

def update(self, global_best):
    self.t *= self.cooling_ratio
    if self.z == global_best[2]:
        return

    self.v_x = (
        self.omega * self.v_x +
        self.c_1 * random() * (self.best[0] - self.x) +
        self.c_2 * random() * (global_best[0] - self.x) +
        self.c_3 * self.t * (.5 - random())
    )
    self.v_y = (
        self.omega * self.v_y +
        self.c_1 * random() * (self.best[1] - self.y) +
        self.c_2 * random() * (global_best[1] - self.y) +
        self.c_3 * self.t * (.5 - random())
    )

    new_x = self.x + self.v_x
    if new_x < self.x_range[0] or new_x > self.x_range[1]:
        self.v_x *= -.5
    self.x += self.v_x
    self.x = min(max(self.x, self.x_range[0]), self.x_range[1])

    new_y = self.y + self.v_y
    if new_y < self.y_range[0] or new_y > self.y_range[1]:
        self.v_y *= -.5
    self.y += self.v_y
    self.y = min(max(self.y, self.y_range[0]), self.y_range[1])

    self.z = self.func(self.x, self.y)
    if self.better(self.z, self.best[2]):
        self.best = (self.x, self.y, self.z)

```

۲ ازدحام

کلاس Swarm را به نمایندگی از ازدحام ذرات پیاده‌سازی می‌کنیم. در constructor این کلاس نیز مشابه کلاس ذره تعدادی پارامتر دریافت می‌کنیم.

```
def __init__(self, func, x_range, y_range, iterations_count, visualizer,
            better, best, sqrt_count, omega, c_1, c_2, c_3, t, cooling_ratio):
    self.func = func
    self.x_range = x_range
    self.y_range = y_range
    self.iterations_count = iterations_count
    self.visualizer = visualizer
    self.better = better
    self.best = best
    self.count = sqrt_count ** 2
    self.omega = omega
    self.c_1 = c_1
    self.c_2 = c_2
    self.c_3 = c_3
    self.t = t
    self.cooling_ratio = cooling_ratio

    self.particles = []
    for x in np.linspace(x_range[0] + .1, x_range[1] - .1, sqrt_count):
        for y in np.linspace(y_range[0] + .1, y_range[1] - .1, sqrt_count):
            self.particles.append(Particle(
                x + (.5 - random()) / 5, y + (.5 - random()) / 5,
                func, x_range, y_range, better, omega, c_1, c_2, c_3, t,
                cooling_ratio
            ))
```

- پارامتر iterations_count تعداد تکرارهای الگوریتم را نشان می‌دهد. در این تمرین شرط توقف را تعداد در نظر گرفتیم.
- پارامتر visualizer یک ترسیم‌کننده در اختیار ازدحام قرار می‌دهد تا بتواند تابع و ذرات را در فضای سه‌بعدی رسم کند.
- پارامتر best مشابه پارامتر better که در بخش قبل توضیح دادیم یک تابع است که با دریافت لیستی از نقاط بهترین را خروجی می‌دهد. این تابع برای f برابر \max و برای g برابر \min می‌باشد.
- پارامتر sqrt_count نشان‌دهنده ریشه دوم تعداد ذرات می‌باشد. علت این است که در این تمرین تصمیم گرفتیم به جای انتخاب تصادفی ذرات، آن‌ها به طور کاملاً منظم و با فواصل برابر از محدوده مجاز انتخاب

کنیم. پس برای مثال اگر این پارامتر برابر 10° باشد، محدوده مجاز برای x ها را به 10° و y ها را نیز به 10° قسمت تقسیم می کنیم و از این طریق 100° نقطه انتخاب می کنیم و ذرات را ایجاد می کنیم.

در نهایت برای اجرای الگوریتم کافی است مطابق آنچه در کلاس بیان شده پیش برویم.

```
def run(self, printing_iterations=[], visualizing_iterations=[], output=None):
    for iteration in range(self.iterations_count + 1):
        values = [particle.z for particle in self.particles]
        global_best_index = values.index(self.best(values))
        global_best = (self.particles[global_best_index].x,
                       self.particles[global_best_index].y, self.particles
[global_best_index].z)

        if iteration in printing_iterations:
            print(f'Iteration: {iteration}\tBest: {global_best}')
        if iteration in visualizing_iterations:
            self.visualizer.add(self.x_range, self.y_range, self.func, self
.particles,
                               title=f'Iteration {iteration}\nBest: {round
(global_best[2], 2)}')

        for particle in self.particles:
            particle.update(global_best)

    self.visualizer.show(output)
```

در این قطعه کد نکته خاصی وجود ندارد، تنها مراحل ساده PSO پیاده سازی شده اند.

۳ نتایج

۱۰۳ تابع f

برای این تابع قطعه کد زیر را نوشتیم

```
import math
import numpy as np

import utils

f = lambda x, y: abs(np.sin(x) * np.cos(y) * np.exp(np.abs(1 - (np.sqrt(np.
    power(x, 2) + np.power(y, 2)) / math.pi))))
x_range = [-10, 10]
y_range = [-10, 10]

ITERATIONS_COUNT = 20
VISUALIZER = utils.Visualizer(2, 3)

BETTER = lambda x, y: x > y
BEST = max
SQRT_COUNT = 40
OMEGA = .3
C_1 = .3
C_2 = .3
C_3 = 1
T = 1
COOLING_RATIO = .99

PRINTING_ITERATIONS = range(0, ITERATIONS_COUNT + 1, ITERATIONS_COUNT // 10
    or 1)
VISUALIZING_ITERATIONS = range(0, ITERATIONS_COUNT + 1, ITERATIONS_COUNT //
    5 or 1)

swarm = utils.Swarm(
    f, x_range, y_range,
    ITERATIONS_COUNT, VISUALIZER,
    BETTER, BEST, SQRT_COUNT, OMEGA, C_1, C_2, C_3, T, COOLING_RATIO
)
swarm.run(PRINTING_ITERATIONS, VISUALIZING_ITERATIONS, 'f.png')
```

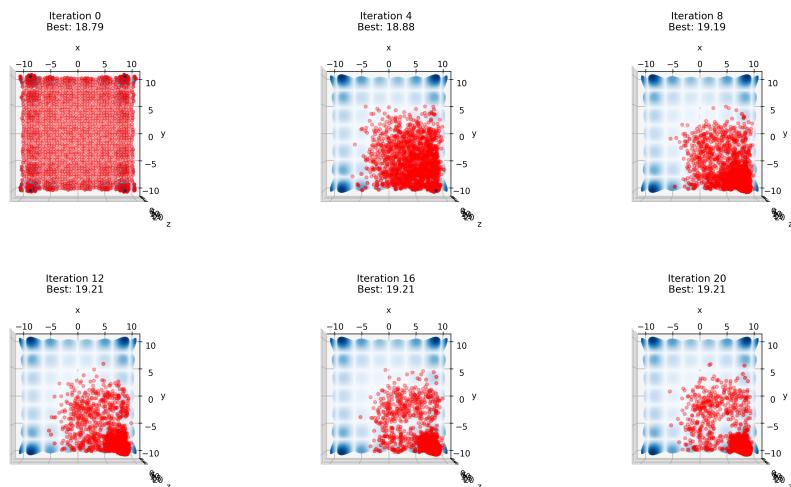
برای رسیدن به پاسخ مورد نظر ۲۰ تکرار روی ۱۶۰۰ ذره انجام دادیم، که نتیجه زیر حاصل شد.

```

Iteration: 0      Best: (7.947600056760501, -9.487163093588515,
                  18.786426719534973)
Iteration: 2      Best: (7.947600056760501, -9.487163093588515,
                  18.786426719534973)
Iteration: 4      Best: (8.225229978380645, -9.72601793689017,
                  18.876424873854354)
Iteration: 6      Best: (8.076344700994468, -9.705532543685338,
                  19.186795657811704)
Iteration: 8      Best: (8.012787397277569, -9.657418813289324,
                  19.190429288939153)
Iteration: 10     Best: (8.027794348548882, -9.65095237996594,
                  19.19926187892689)
Iteration: 12     Best: (8.05469486077135, -9.653163543268402,
                  19.207187159143487)
Iteration: 14     Best: (8.05469486077135, -9.653163543268402,
                  19.207187159143487)
Iteration: 16     Best: (8.056979252942076, -9.675544105678043,
                  19.207248164069608)
Iteration: 18     Best: (8.056979252942076, -9.675544105678043,
                  19.207248164069608)
Iteration: 20     Best: (8.056979252942076, -9.675544105678043,
                  19.207248164069608)

```

که از چیزی که در تمرین خواسته شده بهتر است.



این تصاویر از زاویه بالا رسم شده‌اند.

۲۰۳ تابع g

برای این تابع کار کمی سنگین‌تر شد. قطعه کد زیر را نوشتیم

```
import math
import numpy as np

import utils

g = lambda x, y: x * np.sin(math.pi * np.cos(x) * np.tan(y)) * (np.sin(y /
    x) / (1 + np.cos(y / x)))
x_range = [-100, 100]
y_range = [-100, 100]

ITERATIONS_COUNT = 4000
VISUALIZER = utils.Visualizer(2, 3)

BETTER = lambda x, y: x < y
BEST = min
SQRT_COUNT = 40
OMEGA = 2
C_1 = 3
C_2 = 4
C_3 = 1
T = 1
COOLING_RATIO = .99

PRINTING_ITERATIONS = range(0, ITERATIONS_COUNT + 1, ITERATIONS_COUNT // 10
    or 1)
VISUALIZING_ITERATIONS = range(0, ITERATIONS_COUNT + 1, ITERATIONS_COUNT //
    5 or 1)

swarm = utils.Swarm(
    g, x_range, y_range,
    ITERATIONS_COUNT, VISUALIZER,
    BETTER, BEST, SQRT_COUNT, OMEGA, C_1, C_2, C_3, T, COOLING_RATIO
)
swarm.run(PRINTING_ITERATIONS, VISUALIZING_ITERATIONS, 'g.png')
```

برای رسیدن به پاسخ مورد نظر ۴۰۰۰ تکرار روی ۱۶۰۰ ذره انجام دادیم، که نتیجه زیر حاصل شد.

Iteration: 0 Best: (28.2082418755946, 89.6334235683545,
-693.2995631503871)

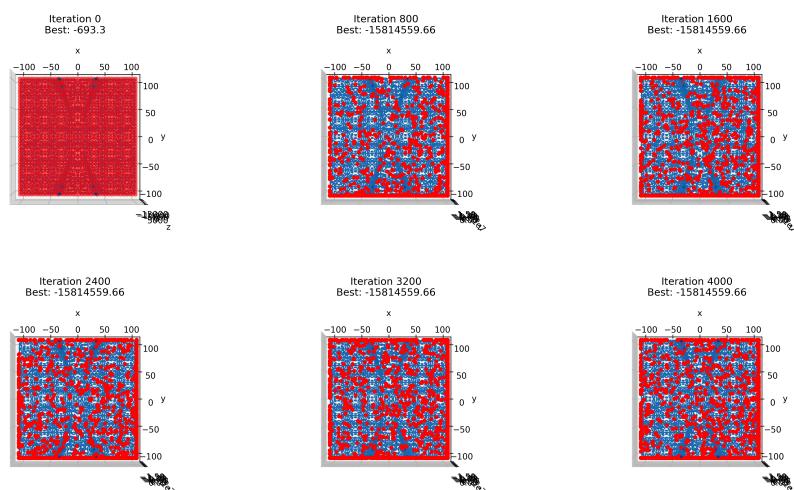
Iteration: 400 Best: (31.831119959341926, -100, -4877436.573243496)

```

Iteration: 800 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 1200 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 1600 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 2000 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 2400 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 2800 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 3200 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 3600 Best: (-31.831029125477045, 100, -15814559.656371178)
Iteration: 4000 Best: (-31.831029125477045, 100, -15814559.656371178)

```

که با اختلاف از چیزی که در تمرین خواسته شده بهتر است.



همانطور که دیده می‌شود ذرات هنوز همگرا نشده بودند اما با این وجود پاسخی در حدود $10^6 \times 15$ - به دست آوردیم که چیزی حدود ۹ برابر بهتر از خواسته تمرین می‌باشد.