

پیاده سازی  
الگوریتم  
ژنتیک حرفی  
با کنترلر فازی

# مشخصات

درس : مبانی هوش محاسباتی

زمان : ترم اول 1402 - 1403

مدرس : دکتر عباس بحرالعلوم

اعضای گروه : امیرحسین ابوالحسنی, حمیدرضا بازیار, صدرا کوچک زاده

موضوع : پیاده سازی الگوریتم و راثتی حقیقی

# هدف اصلی پیاده سازی چیست؟؟؟

1. مدیریت بهتر از تغییرات و ابهامات: الگوریتم‌های ژنتیک ممکن است در مواجهه با داده‌های ناقص یا مسائل نامعین دچار مشکل شوند. اضافه کردن عناصر فازی به الگوریتم ژنتیک می‌تواند در تصمیم‌گیری‌هایی که نیاز به انطباق با شرایط نامعین دارند، کمک کند.
2. افزایش کارایی الگوریتم: سیستم‌های فازی معمولاً قابلیت تطبیق با محیط را دارند. این قابلیت تطبیق در الگوریتم ژنتیک می‌تواند به معنای بهبود سرعت یا دقیقت در رسیدن به حلول بهینه باشد.
3. تصمیم‌گیری بهتر بر اساس ابهامات: معمولاً فازی‌سازی پارامترها و مقادیر در الگوریتم ژنتیک می‌تواند در تعیین اهمیت متغیرها و تعیین وزن برای هر یک از آن‌ها در فرآیند تکامل، کمک کند. این موارد می‌توانند به تصمیمات بهتر و بهینه‌تر در فرآیند تکامل منجر شوند.
4. افزایش امکانات تکاملی: سیستم فازی می‌تواند به الگوریتم ژنتیک کمک کند تا به روشن‌تر کردن راهنمایی‌ها و محدوده‌های تکاملی برای ایجاد نسل‌های جدید و بهبود آن‌ها بپردازد.

# انواع کتابخانه های به کار رفته:

```
from RGA_Class import RGA
import theorem
import random
from GA_Fuzzy_Controller import Fuzzy_Controller
import numpy as np
```

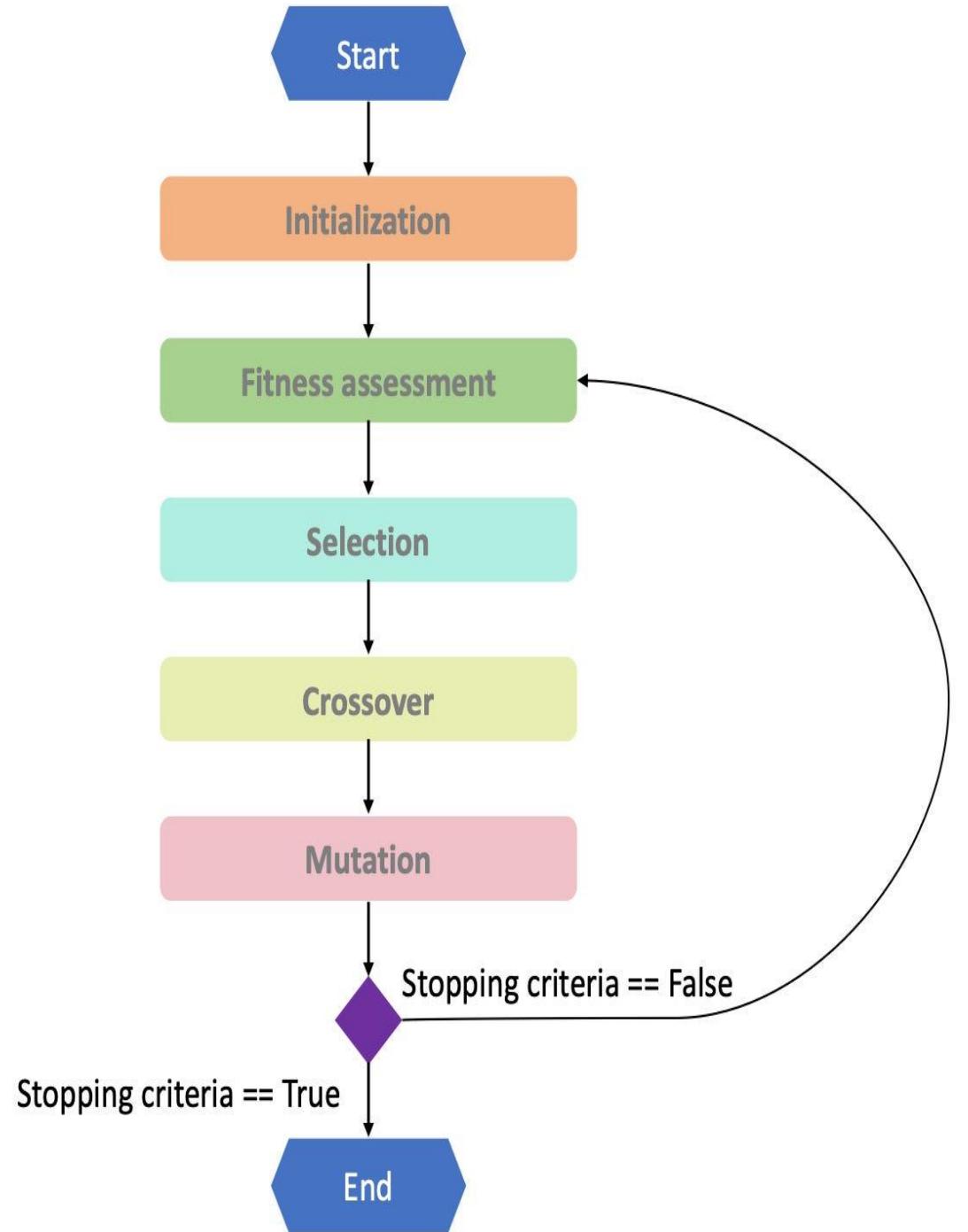
RGA\_Class : این کتابخانه شامل کلاس‌ها و توابع مربوط به الگوریتم بهینه‌سازی ژنتیک است. اینجا مایک کلاس بهینه‌سازی ژنتیک به نام RGA را از این کتابخانه استفاده کرده‌ایم.

theorem : این ماثول شامل توابع ریاضیاتی مورد استفاده در توابع تست مربوط به بهینه‌سازی ژنتیک است. مثلاً توابع Rastrigin و Griewank که اینجا به آنها اشاره شده است.

GA\_Fuzzy\_Controller : این کتابخانه شامل کنترل‌کننده‌ها و توابع مرتبط با الگوریتم‌های بهینه‌سازی ژنتیک و کنترل فازی است.

Numpy : این کتابخانه برای عملیات عددی و محاسبات علمی استفاده می‌شود.

random : این کتابخانه برای تولید اعداد تصادفی استفاده می‌شود.

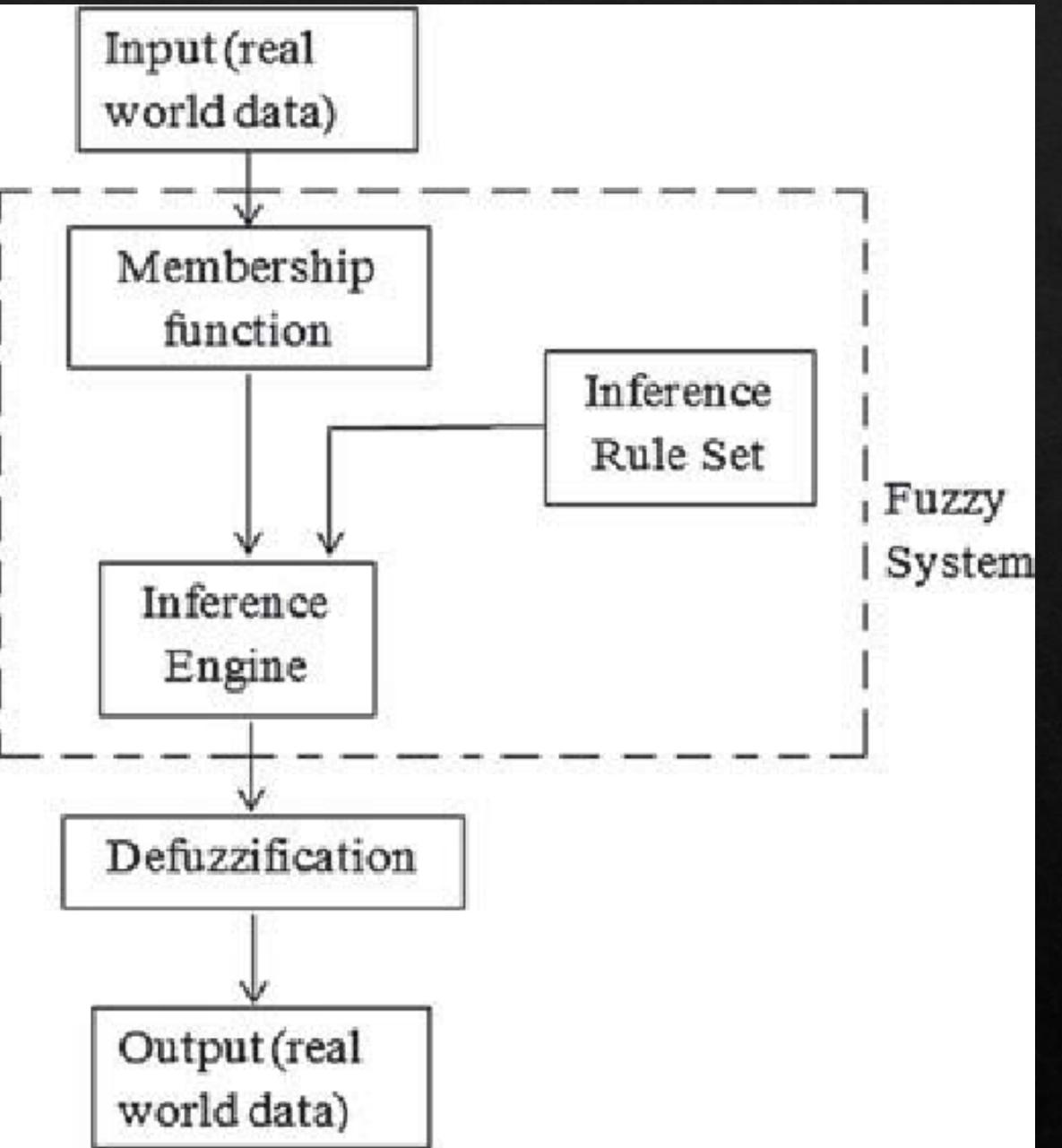


همانطور که مراحل طی شدن الگوریتم ژنتیک حقیقی را به پاد دارید، حال این الگوریتم بهینه سازی را بر روی سیستم فازی اعمال میکنیم

# سیستم فازی چیست؟

سیستم فازی، یک سیستم مبتنی بر قواعد است که می‌تواند با استفاده از اصول و مفاهیم فازی، منطق و قوانین غیردقیق را برای مدل‌سازی پدیده‌های پیچیده استفاده کند. در عین حال، این سیستم می‌تواند با دقت و اطمینان، به سطحی از تصمیم‌گیری برسد که در برخی موارد با استفاده از منطق دودویی سنتی قابل دستیابی نبوده است.

فازی بودن یک سیستم به این معناست که ورودی‌ها و خروجی‌های آن از مقادیر دودویی کلاسیک خارج شده و به صورت مقادیر فازی یا اعدادی بین مقادیر مشخص و دقیق قرار می‌گیرند. این مفهوم امکان مدل‌سازی و تعریف قوانینی را که در جهان واقعی معمولاً مبهم هستند، فراهم می‌کند.



نمایش ورودی و خروجی به صورت فازی : ورودی‌ها و خروجی‌های سیستم فازی به صورت مقادیر فازی (مقادیری که بین حدود مشخص نیستند) نمایش داده می‌شوند. برای این کار نیاز است که ورودی‌ها و خروجی‌ها به شکلی فازی تعریف و مدل شوند.

**Membership Functions** : این توابع برای تبدیل مقادیر ورودی به مقادیر فازی استفاده می‌شوند. آن‌ها توصیف می‌کنند که یک مقدار ورودی به چه اندازه عضو یک مجموعه فازی خاص است.

**Fuzzy Rules** : این قوانین به صورت "اگر-آنگاه" تعریف می‌شوند و وابستگی‌های فازی بین ورودی‌ها و خروجی‌ها را مدل می‌کنند.

**Fuzzy Inference Mechanism** (استفاده از مکانیزم استنتاج فازی) : این مرحله شامل اعمال قوانین فازی به داده‌های ورودی است و به دست آوردن خروجی‌های فازی متناظر با این ورودی‌ها با توجه به قوانین تعریف شده است.

**Defuzzification** : در این مرحله، مقادیر فازی حاصل از مرحله استنتاج به مقادیر عددی یا فازی تبدیل می‌شوند تا به عنوان خروجی‌های قابل استفاده در سیستم ارائه شوند.

این کد تابع **(main)** دارد که یک نمونه از الگوریتم بهینه‌سازی ژنتیک برای تابع Rastrigin را فراخوانی می‌کند. الگوریتم از تابع Rastrigin به عنوان تابع هدف استفاده می‌کند و با استفاده از **RGA** و تنظیمات مشخص، این الگوریتم بهینه‌سازی ژنتیک را با کنترل فازی اجرا می‌کند. **Fuzzy\_Controller**) دقت و تنظیمات الگوریتم نیز با پارامترهایی مانند حداقل تعداد نسل‌ها **population** (اندازه جمعیت) و **crossover\_rate** (نرخ تلاقي) تنظیم شده است.

```
def main():
    # # Griewank Function
    # rga_griewank = RGA(target_function=theorem.griewank, fitness_function=lambda x: 370 - theorem.griewank(x),
    #                      function_config=[{'low': -600, 'high': 600}, {'low': -600, 'high': 600},
    #                                     {'low': -600, 'high': 600}, {'low': -600, 'high': 600}, {'low': -600, 'high': 600}, {'low': -600, 'high': 600}],
    #                      crossover_rate=0.5, max_gen=100, population=80, run_nga=1, controller=Fuzzy_Controller, gpc=5)
    #
    # rga_griewank.Run()

    # Rastrigin Function
    rga_rastrigin = RGA(target_function=theorem.rastrigin, fitness_function=lambda x: 200 - theorem.rastrigin(x),
                          function_config=[{'low': -5.12, 'high': 5.12}, {'low': -5.12, 'high': 5.12},
                                           {'low': -5.12, 'high': 5.12}, {'low': -5.12, 'high': 5.12}],
                          crossover_rate=0.5, max_gen=300, population=300, run_nga=1, controller=Fuzzy_Controller, gpc=20)

    rga_rastrigin.Run()

if __name__ == "__main__":
    main()
```

کلاسی که در اینجا ارائه شده یک کنترل‌کننده فازی برای یک الگوریتم ژنتیک است. این الگوریتم از مفاهیم فازی استفاده می‌کند تا مقادیر دقیق ورودی‌های الگوریتم ژنتیک (مانند نرخ جهش و عملکرد کلونی) را ترکیب و به مقادیر دقیق برای نرخ جهش تبدیل کند.

مراحل اصلی الگوریتم شامل:

. فاز فازی‌سازی (**Fuzzification**) : تبدیل مقادیر دقیق به مقادیر فازی با استفاده از توابع عضویت فازی.

. فاز تطابق (**Matching**) : ارزیابی قدرت خروجی هر قانون فازی بر اساس ورودی‌های فازی.

. فاز استنتاج (**Inference**) : ترکیب خروجی‌های فاز تطابق به عنوان خروجی نهایی الگوریتم.

. فاز محاسبه نرخ جهش دقیق (**Defuzzification**) : تبدیل مقادیر فازی به یک مقدار دقیق برای نرخ جهش.

در هر مرحله، مفاهیمی از تئوری فازی مانند توابع عضویت و قوانین فازی استفاده شده‌اند تا فرآیند تصمیم‌گیری را بر اساس قوانین فازی و در نتیجه، مقدار دقیقی برای نرخ جهش ارائه دهند.

```
class Fuzzy_Controller:  
  
    def __init__(self, max_gen, k): ...  
  
    def control(self, cur_gen, p_m, cur_bsf): ...  
  
    def defuzzifier(self, fuzzy_output): ...  
  
    def inference(self, pm_fuzzy): ...  
  
    def matching(self, fuzzy_vars): ...  
  
    def fuzzifier(self, cur_gen, p_m, cur_bsf): ...  
  
    def fuzzify_generation(self, gen): ...  
  
    def fuzzify_cm(self, cm): ...  
  
    def fuzzify_pm(self, p_m): ...  
  
    def Reset(self): ...  
  
    def plot_pm(self, dir): ...
```

```

def control(self, cur_gen, p_m, cur_bsf):
    """
    This method runs the whole system.
    :param cur_gen: A number, the current iteration of the algorithm
    :param p_m: A number, current mutation rate
    :param cur_bsf: A number, the current best so far of the algorithm
    :return: a crisp number for mutation rate
    """
    self.pm_history.append(p_m)

    if (cur_gen % self.k != 0) or (cur_gen == 0):
        return p_m

    # Logging pm

    # Fuzzifying phase
    fuzzy_vars = self.fuzzifier(cur_gen=cur_gen, p_m=p_m, cur_bsf=cur_bsf)

    # Updating bsf and pm for the next control phase
    self.prev_bsf = cur_bsf
    self.prev_pm = p_m

    pm_strengths = self.matching(fuzzy_vars=fuzzy_vars)

    pm_fuzzy_output = self.inference(pm_strengths)

    crisp_pm = self.defuzzifier(pm_fuzzy_output)

    return crisp_pm

```

این یک متد در یک کلاس `Fuzzy_Controller` است که شامل چندین مرحله از فرایند بهینه‌سازی است. به طور خلاصه، این متد `control` در چرخه الگوریتم بهینه‌سازی، مقدار نرخ جهش (mutation rate) را بهروزرسانی می‌کند.

در این متد `self.pm_history.append(p_m)` : مقدار نرخ جهش فعلی را به تاریخچه نرخ‌های جهش اضافه می‌کند.

اجرای الگوریتم بررسی می‌شود. اگر شرط صحیح باشد، مقدار فعلی نرخ جهش را برمی‌گرداند و اگر شرط صحیح نباشد، به مراحل بعدی می‌رود.

`fuzzy_vars = self.fuzzifier(cur_gen=cur_gen, p_m=p_m, cur_bsf=cur_bsf)` : یک مرحله از فاز فازی‌سازی را اجرا می‌کند. این فاز اطلاعات مربوط به نرخ جهش و وضعیت‌های دیگر الگوریتم را به صورت فازی پردازش می‌کند. بهترین مقدار یافت شده تا این لحظه را ذخیره می‌کند تا در مراحل بعدی الگوریتم استفاده شود.

`pm_strengths = self.matching(fuzzy_vars=fuzzy_vars)` : اطلاعات فازی حاصل از مرحله قبلی را به یک مرحله مطابقت می‌دهد تا قوت نرخ جهش را محاسبه کند.

`pm_fuzzy_output = self.inference(pm_strengths)` : از اطلاعات فازی به دست آمده، نرخ جهش فازی را استنتاج می‌کند.

`crisp_pm = self.defuzzifier(pm_fuzzy_output)` : نرخ جهش فازی به دست آمده را به یک مقدار کریسپ (عدد دقیق) تبدیل می‌کند.

در نهایت، مقدار کریسپ نرخ جهش را برمی‌گرداند تا در مراحل بعدی الگوریتم استفاده شود.

```

def defuzzifier(self, fuzzy_output):
    """
    This method is defuzzify the fuzzy output with COG method
    :param fuzzy_output: A dict which represents a fuzzy output with items "fuzzy set: membership degree"
    :return: A crisp value that is the output of the controller
    """

    def COG(X, f_output):
        membership_degrees = []
        for x in X:
            f_var = self.fuzzify_pm(x)
            md = max(min(f_output['low'], f_var['low']), min(f_output['avg'], f_var['avg']),
                      min(f_output['high'], f_var['high']))
            membership_degrees.append(md)

        membership_degrees = np.array(membership_degrees)
        cog = np.dot(membership_degrees, X) / np.sum(membership_degrees)

        return cog

    X = np.arange(0, 2e-2, 1e-4)
    return COG(X, fuzzy_output)

```

این تابع **defuzzifier** در کلاس **Fuzzy\_controller** برای تبدیل خروجی فازی به یک مقدار کریسپ استفاده می‌شود. این تابع از روش COG (مرکز گرانش) یا Center of Gravity برای انجام عمل تفاضل‌دهی (defuzzification) استفاده می‌کند. در این تابع:

**COG** : یک تابع داخلی است که برای محاسبه مقدار COG استفاده می‌شود. این تابع از دو ورودی **X** (یک مجموعه از مقادیر ورودی) و **f\_output** (خروجی فازی) استفاده می‌کند. **X** : تعیین می‌کند که از چه بازه‌ای اعداد کریسپ برای محاسبه COG استفاده می‌شود.

**COG** : با استفاده از مقادیر ورودی **X** و **f\_output**، در هر مقدار از **X**، اعضایی را که با اعضای خروجی فازی مرتبط هستند، گرفته و از آن‌ها میانگین می‌گیرد تا COG را برای مقدار خروجی بدست آورد.

در نهایت، تابع اصلی **defuzzifier** از **COG** استفاده می‌کند و یک بازه از مقادیر کریسپ را به عنوان ورودی می‌دهد تا برای آن محاسبه شود و مقدار کریسپ خروجی نهایی را تولید کند.

این فرآیند از فاز فازی (fuzzy) به کریسپ (crisp)، که به آن تفاضل‌دهی یا تبدیل فازی می‌گویند، از طریق COG انجام می‌شود تا مقدار دقیق‌تری برای خروجی الگوریتم بهدست آید.

```

def inference(self, pm_fuzzy):
    """
    This method is for the inference and combination phase of the algorithm
    :param pm_fuzzy: A dict where keys represent the sets defined for the variable the fuzzy output from the matching phase.
    :return: A dict where its keys are the same as pm_fuzzy keys,
    this dict represents a fuzzy variable with its fuzzy sets defined on it,
    representing the fuzzy output of the inference engine.
    """
    fuzzy_sets = list(pm_fuzzy.keys())
    fuzzy_output = {f_set: max(pm_fuzzy[f_set]) for f_set in fuzzy_sets}
    return fuzzy_output

```

تابع **inference** یک مرحله مهم در فرایند الگوریتم فازی است که به آن "فاز استنتاج و ترکیب" می‌گویند.

پارامتر هایی که این تابع دارد و بر روی آنها کار انجام می دهد :

**pm\_fuzzy** : یک دیکشنری که کلیدهای آن نمایانگر مجموعه‌های فازی برای متغیرهای مورد بررسی است و مقادیرشان نیز مربوط به خروجی فازی از مرحله مطابقت هستند.  
عملیات درون این تابع:

((**fuzzy\_sets = list(pm\_fuzzy.keys())**) : این خط کلیدهای موجود در **pm\_fuzzy** را به یک لیست تبدیل می‌کند که مجموعه‌های فازی متناظر با متغیرهای مختلف را نمایش می‌دهد.  
**fuzzy\_output = {f\_set: max(pm\_fuzzy[f\_set]) for f\_set in fuzzy\_sets}** : در اینجا برای هر مجموعه فازی، مقدار بیشینه از آن مجموعه فازی به عنوان نتیجه استنتاج در نظر گرفته می‌شود و در **fuzzy\_output** ذخیره می‌شود.

این تابع یک دیکشنری را به عنوان خروجی بر می‌گرداند که کلیدهای آن با مجموعه‌های فازی اصلی متناظر دارای مقادیر بیشینه است. این دیکشنری نمایانگر متغیر فازی با مجموعه‌های فازی تعریف شده برای آن است که نمایانگر خروجی فازی مرحله استنتاج موتور است.  
به طور خلاصه، این تابع با دریافت خروجی فازی ، مقادیر بیشینه از هر مجموعه فازی را بر می‌گرداند که به عنوان خروجی نهایی فاز استنتاج و ترکیب محاسبه می‌شود.

```

def matching(self, fuzzy_vars):
    """
    This method implements the matching phase of the control,
    where we evaluate the strength the outcome of each rule.
    :param fuzzy_vars: A dict containing 3 other dicts which are fuzzified inputs
    :return: a dict,
    which is the pm strengths for each set defined on the variable
    """
    pm_strengths = {"low": [], "avg": [], "high": []}

    # Rule 1
    r1s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['low'], fuzzy_vars['gen']['start'])
    pm_strengths['avg'].append(r1s)

    # Rule 2
    r2s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['avg'], fuzzy_vars['gen']['start'])
    pm_strengths['high'].append(r2s)

    # Rule 3
    r3s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['high'], fuzzy_vars['gen']['start'])
    pm_strengths['low'].append(r3s)

    # Rule 4
    r4s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['low'], fuzzy_vars['gen']['start'])
    pm_strengths['low'].append(r4s)

    # Rule 5
    r5s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['avg'], fuzzy_vars['gen']['start'])
    pm_strengths['low'].append(r5s)

    # Rule 6
    r6s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['high'], fuzzy_vars['gen']['start'])
    pm_strengths['avg'].append(r6s)

    # Rule 7
    r7s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['low'], fuzzy_vars['gen']['middle'])
    pm_strengths['avg'].append(r7s)

    # Rule 8
    r8s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['avg'], fuzzy_vars['gen']['middle'])
    pm_strengths['avg'].append(r8s)

    # Rule 9
    r9s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['high'], fuzzy_vars['gen']['middle'])
    pm_strengths['low'].append(r9s)

    # Rule 10
    r10s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['low'], fuzzy_vars['gen']['middle'])
    pm_strengths['avg'].append(r10s)

    # Rule 11
    r11s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['avg'], fuzzy_vars['gen']['middle'])
    pm_strengths['avg'].append(r11s)

    # Rule 12
    r12s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['high'], fuzzy_vars['gen']['middle'])
    pm_strengths['high'].append(r12s)

    # Rule 13
    r13s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['low'], fuzzy_vars['gen']['end'])
    pm_strengths['avg'].append(r13s)

    # Rule 14
    r14s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['avg'], fuzzy_vars['gen']['end'])
    pm_strengths['high'].append(r14s)

    # Rule 15
    r15s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pm_prev']['high'], fuzzy_vars['gen']['end'])
    pm_strengths['low'].append(r15s)

    # Rule 16
    r16s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['low'], fuzzy_vars['gen']['end'])
    pm_strengths['avg'].append(r16s)

    # Rule 17
    r17s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['avg'], fuzzy_vars['gen']['end'])
    pm_strengths['high'].append(r17s)

    # Rule 18
    r18s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pm_prev']['high'], fuzzy_vars['gen']['end'])
    pm_strengths['avg'].append(r18s)

    return pm_strengths

```

تابع matching بخشی از فرایند فازی سازی و ارزیابی قدرت خروجی هر قانون (rule) در یک سیستم کنترل فازی را انجام می‌دهد.

اینجا ۱۸ قانون مختلف در نظر گرفته شده‌اند. این قوانین با استفاده از متغیرهای فازی (که از فازی‌سازی داده‌های ورودی ایجاد شده‌اند)، قوانین را مدل‌سازی کرده‌اند که میزان تأثیر و ورودی‌های مختلفی که در تصمیم‌گیری فازی دخیل هستند را ارزیابی می‌کنند.

به عنوان مثال، قانون ۱۳، از مقادیر مختلف متغیرهای فازی مانند cm، gen و pm\_prev استفاده می‌کند و با استفاده از تابع min، کمینه مقادیر این متغیرهای فازی را برای اعمال قانون به دست می‌آورد.

در نهایت، این قوانین مختلف بر اساس ورودی‌های فازی، تأثیر هر یک از آن‌ها را با استفاده از متغیر pm\_strengths که شامل لیست‌های مربوط به مقادیر مختلف فازی low، average و high است، ارزیابی می‌کنند و نتایج را به عنوان خروجی بر می‌گردانند.

```

def fuzzifier(self, cur_gen, p_m, cur_bsf):
    """
    This method does the fuzzifying phase of the FCS.
    :param cur_gen: the current iteration of the genetic algorithm.
    :param p_m: the current mutation rate of the genetic algorithm.
    :param cur_bsf: the best so far up do this generation of the algoritm.
    :return: fuzzified input params as a dict.
    """

    membership_values = {"cm": None, "pm_prev": None, "gen": None}

    def CM(cur_bsf):
        """
        if cm gets closer to one, it means that algorithm is not making the desired amount of progress
        :param cur_bsf: a number, the best so far of the current generation
        :return: 0 < cm < inf
        """
        cm = self.prev_bsf / (cur_bsf + 1e-3)
        return cm

    cm = CM(cur_bsf)

    membership_values["pm_prev"] = self.fuzzify_pm(p_m)
    membership_values['cm'] = self.fuzzify_cm(cm)
    membership_values['gen'] = self.fuzzify_generation(cur_gen)

    return membership_values

```

این تابع **fuzzifier** یکی از مراحل مهم در فرایند کنترل فازی است که با استفاده از ورودی‌های عددی، این اطلاعات را به شکل متغیرهای فازی یا متغیرهایی که بیان کننده عضویت در مجموعه‌های فازی هستند، تبدیل می‌کند.

در این تابع:

**cur\_gen** : نمایانگر تعداد نسل فعلی الگوریتم ژنتیک است.  
**p\_m** : نرخ جهش فعلی الگوریتم ژنتیک.

**cur\_bsf** : بهترین عملکرد تا به حال در این نسل از الگوریتم ژنتیک.

به صورت خلاصه، این تابع از توابع داخلی مانند **CM** استفاده می‌کند تا مقادیر فازی مربوط به هریک از ورودی‌های عددی را ایجاد کند. به عنوان مثال:

- **CM** : این تابع یک مقدار **cm** (متغیر فازی) را بر اساس بهترین عملکرد تا آن لحظه محاسبه می‌کند. این مقدار **cm** به عنوان یک شاخص برای پیشرفت فازی در نسل جاری مورد استفاده قرار می‌گیرد.

- **membership\_values["pm\_prev"] = self.fuzzify\_pm(p\_m)** : این بخش با استفاده از تابع **fuzzify\_pm**، نرخ جهش فعلی را به یک متغیر فازی تبدیل می‌کند.

- **membership\_values['cm'] = self.fuzzify\_cm(cm)** : مقدار **cm** که از تابع **CM** بدست آمده است، به عنوان متغیر فازی **cm** در نظر گرفته می‌شود.

- **membership\_values['gen'] = self.fuzzify\_generation(cur\_gen)** : این بخش مربوط به تبدیل متغیر مربوط به نسل فعلی الگوریتم ژنتیک به یک متغیر فازی است.

در نهایت، این تابع یک دیکشنری با متغیرهای فازی مربوط به هریک از ورودی‌های عددی را بر می‌گرداند تا در مراحل بعدی الگوریتم کنترل فازی مورد استفاده قرار گیرد.

```

def fuzzify_generation(self, gen):
    """ this method gets the number of current generations
    and calculate the gen / max gen and fuzzify the result.
    it returns the result as a dictionary with three keys (start, middle, end)
    these keys are the labels of fuzzy sets """

    x = gen / self.N
    start_mf = self.mf_gen["start"]
    middle_mf = self.mf_gen["middle"]
    end_mf = self.mf_gen["end"]

    return {"start": start_mf(x), "middle": middle_mf(x), "end": end_mf(x)}

```

این تابع به نام **fuzzify\_generation**، یکی از مراحل فرایند فازی سازی را انجام می‌دهد. این تابع بر اساس تعداد نسل فعلی الگوریتم ژنتیک، مقداری عددی را می‌گیرد و آن را به شکل متغیر فازی با مجموعه‌های فازی مختلف تبدیل می‌کند.

بخش‌های مختلف این تابع:

- **x = gen / self.N** : ابتدا، مقدار **x** را بدست می‌آورد که نسبت تعداد نسل فعلی (**gen**) به بیشترین تعداد نسل ممکن (**self.N**) را نشان می‌دهد.
- **start\_mf = self.mf\_gen["start"], middle\_mf = self.mf\_gen["middle"], end\_mf = self.mf\_gen["end"]** : این قسمت از تابع از متغیرهایی مانند **mf\_gen** که مجموعه‌های فازی برای نسل را مشخص می‌کنند، استفاده می‌کند.
- **return {"start": start\_mf(x), "middle": middle\_mf(x), "end": end\_mf(x)}** : در نهایت، مقدار **x** (نسبت نسل فعلی به بیشترین تعداد نسل) را به هر یک از مجموعه‌های فازی تعریف شده برای نسل **start**، **middle** و **end** می‌دهد و مقادیر عضویت در هر یک از این مجموعه‌های فازی را به عنوان خروجی بر می‌گرداند.
- به طور خلاصه، این تابع نسبت تعداد نسل فعلی الگوریتم ژنتیک به بیشترین تعداد نسل ممکن را محاسبه کرده و این نسبت را به شکل متغیرهای فازی با مجموعه‌های فازی مختلف شامل **start**، **middle** و **end** بازمی‌گرداند.

این تابع با نام **fuzzify\_cm** وظیفه تبدیل یک مقدار عددی به متغیر فازی مربوط به cm را دارد.

درون این تابع:

```
low_mf = self.mf_cm["low"], high_mf =  
self.mf_cm["high"] : این بخش از تابع از متغیرهایی  
مانند mf_cm استفاده می‌کند که مجموعه های فازی برای  
متغیر cm را مشخص می‌کند.
```

```
return {"low": low_mf(cm), "high":  
high_mf(cm)} : در نهایت، این تابع مقدار cm (مقداری که  
به عنوان ورودی داده شده است) را به متغیرهای فازی low  
و high تبدیل می‌کند. این تبدیل به معنای محاسبه مقادیر  
عضویت در مجموعه های فازی متغیر cm بر اساس مقادیر  
داده شده است. با اعمال توابع مربوط به متغیرهای فازی  
high_mf و low_mf بر روی cm ، مقادیر عضویت در  
مجموعه های low و high بدست می‌آید و به صورت یک  
دیکشنری (که کلیدهایش low و high هستند) خروجی داده  
می‌شود.
```

بنابراین، این تابع دقیقاً مقادیر عضویت در مجموعه های  
فازی **low** و **high** را برای متغیر cm محاسبه و  
بازمی‌گرداند.

```
def fuzzify_cm(self, cm):  
  
    low_mf = self.mf_cm["low"]  
    high_mf = self.mf_cm["high"]  
  
    return {"low": low_mf(cm), "high": high_mf(cm)}
```

این تابع **fuzzify\_pm** مسئول تبدیل یک مقدار عددی به متغیر فازی مربوط به **p\_m** (نرخ جهش) است.  
توضیحات این تابع:

**low\_mf = self.mf\_pm["low"]**, **avg\_mf = self.mf\_pm["avg"]**, **high\_mf = self.mf\_pm["high"]**

بخش از تابع از متغیرهایی مانند **mf\_pm** استفاده می کند که مجموعه های فازی برای متغیر نرخ جهش (**p\_m**) را مشخص می کند. این متغیرهای **high\_mf** ، **low\_mf** و **avg\_mf** مربوط به مجموعه های فازی **low** ، **high** و **average** هستند.

**return {"low": low\_mf(p\_m), "avg": avg\_mf(p\_m), "high": high\_mf(p\_m)}**

عنوان ورودی داده شده است) به متغیرهای فازی مربوط به **low** ، **high** و **average** تبدیل می شود. این تبدیل به معنای محاسبه مقادیر عضویت در مجموعه های فازی متغیر **p\_m** بر اساس مقادیر داده شده است. با اعمال توابع مربوط به متغیرهای فازی **low\_mf** و **high\_mf** بر روی **p\_m** ، مقادیر عضویت در مجموعه های **low** ، **high** و **average** بدست می آید و به صورت یک دیکشنری خروجی داده می شود.

بنابراین، این تابع دقیقاً مقادیر عضویت در مجموعه های فازی **low** ، **high** و **average** را برای متغیر **p\_m** محاسبه و بازمی گرداند.

```
def fuzzify_pm(self, p_m):
    low_mf = self.mf_pm["low"]
    avg_mf = self.mf_pm["avg"]
    high_mf = self.mf_pm["high"]

    return {"low": low_mf(p_m), "avg": avg_mf(p_m), "high": high_mf(p_m)}
```

```
def Reset(self):
    """
    This method resets the pm and bsf of the controller,
    giving it the quality to be used multiple times for the multiple runs of the algorithm
    :return: Nothing
    """
    self.prev_pm = 0
    self.prev_bsf = 0

    def plot_pm(self, dir): ...
```

این تابع به نام **Reset** مسئول بازنشانی (**Reset**) مقادیر **prev\_pm** و **prev\_bsf** کنترل کننده است. این کار به کنترل کردن وضعیت اولیه کنترل کننده برای استفاده های بعدی الگوریتم یا فرایند مشابه کمک می کند.  
در این تابع:

**self.prev\_pm = 0** : مقدار **prev\_pm** را به صفر تنظیم می کند، بنابراین برای بار دیگری که این کنترل کننده استفاده می شود، مقدار قبلی نرخ جهش صفر می شود.

**self.prev\_bsf = 0** : همین طور، مقدار **prev\_bsf** را به صفر تنظیم می کند، بنابراین برای بار دیگری که این کنترل کننده استفاده می شود، بهترین عملکرد تا به حال صفر می شود.

این عمل بازنشاندن مقادیر به وضعیت اولیه کنترل کننده کمک می کند تا بتواند برای بار های بعدی از اجرای الگوریتم یا فرایند مشابه با مقادیر اولیه و صفر شروع کند و به طور مستقیم از وضعیت قبلی مستقل شود.

تابع **plot\_pm** مسئول رسم نمودار نرخ جهش برای نسل‌های مختلف الگوریتم ژنتیک است.

در این تابع:

**plt.style.use('Solarize\_Light2')** : این بخش از کد استایل نمودار را به صورت 'Solarize\_Light2' تنظیم می‌کند که یک استایل ظاهری برای نمودار است.

```
def plot_pm(self, dir):
    plt.style.use('Solarize_Light2')

    plt.plot(np.arange(0, self.N, 1), self.pm_history, label="Mutation Rate")
    plt.title("Mutation Rate Plot")
    plt.xlabel("Generation")
    plt.ylabel("P_m")
    plt.legend()

    if dir is not None:
        plt.savefig(os.path.join(dir, "P_m_RGA_plot.png"))

    plt.show()
```

**plt.plot(np.arange(0, self.N, 1), self.pm\_history, label="Mutation Rate")** : این بخش از کد یک نمودار خطی برای نشان دادن نرخ جهش (**self.pm\_history**) بر حسب نسل‌های الگوریتم ژنتیک رسم می‌کند (X). محور افقی (X) نسل‌های الگوریتم ژنتیک را نشان می‌دهد و محور عمودی (Y) نرخ جهش را نشان می‌دهد.

**plt.title("Mutation Rate Plot")** : این بخش عنوان نمودار را تنظیم می‌کند.

**plt.xlabel("Generation")** : این بخش برچسب محور افقی را به عنوان "Generation" تنظیم می‌کند.

**plt.ylabel("P\_m")** : این بخش عنوانی به اسم "P\_m" (نرخ جهش) تنظیم می‌کند.

**plt.legend()** : این بخش نشانگر نرخ جهش را در نمودار نشان می‌دهد.

**if dir is not None: plt.savefig(os.path.join(dir, "P\_m\_RGA\_plot.png"))** : این بخش چک می‌کند که آیا مسیری برای ذخیره نمودار وجود دارد یا نه. اگر مسیر (dir) وجود داشته باشد، تصویر نمودار به فرمت png با نام "P\_m\_RGA\_plot.png" در آن مسیر ذخیره می‌شود.

**plt.show()** : این بخش نمودار را نشان می‌دهد.

با استفاده از این تابع، می‌توان نمودار نرخ جهش برای نسل‌های مختلف الگوریتم ژنتیک را رسم کرد.

نسخه اول (با کنترلر فازی) دارای یک پارامتر اضافی به نام **controller** است که مربوط به کنترل کردن  $p_m$  در طول اجرای الگوریتم است. این نرخ در نسخه اول به صورت پویا تغییر می‌کند در حالی که در نسخه دوم،  $p_m$  ثابت است.

همچنین نسخه اول دارای یک ویژگی به نام **gpc** (generation per control) است که تعداد نسل‌های هر فرمان کنترل کننده را تعیین می‌کند. این ویژگی در نسخه دوم وجود ندارد.

تغییرات در متدها:

- در نسخه اول، متدهای **Run** شامل قابلیت کنترل میوتیشن برای هر نسل است، در حالی که در نسخه دوم این ویژگی حذف شده است.
- توابع **roulette\_wheel**، **one\_run**، **reset**، **post\_process**، **Crossover**، **Random\_population**، **log\_algo**، **log\_gen**، **plot\_info** و **get\_fitness** در هر دو نسخه وجود دارند و عملکرد مشابهی دارند.

تفاوت اصلی میان دو نسخه، در متدهای **one\_gen** و ایجاد نسل جدید است. الگوریتم **Crossover** و **Mutation** در هر دو نسخه به همان شکل استفاده شده‌اند، اما در نسخه اول، تغییرات در جهش به صورت پویا و بر اساس کنترل کننده اعمال می‌شود در حالی که در نسخه دوم، جهش با نرخ ثابت اعمال می‌شود.

فرق بین RGA Class  
ای که با Fuzzy Controller  
پیاده سازی شده است و  
عادی در Class چیست؟

```

def trimf_maker(domain, start, peak, end):
    """
    This function returns a method that is a trimf on the domain of X.
    :param domain: domain of the variable we want to fuzzify (a 1d array of len 2)
    :param start: the starting point of mf
    :param peak: the point in the mf graph where mf(p) = 1
    :param end: the ending point of mf
    :return: a method for trimf in the given domain
    """

    # we should assert this so that the method would be logically valid
    assert start <= peak <= end, "Support is not valid"

    def trimf(x):
        assert domain[0] <= x <= domain[1], "x is not in domain range!"
        if x <= start or x >= end:
            return 0

        if x == peak:
            return 1

        before_peak_slope = 1 / (peak - start) if peak != start else 0
        after_peak_slope = 1 / (peak - end) if peak != end else 0

        if x < peak:
            before_bias = -(start - domain[0]) * before_peak_slope
            res = (before_peak_slope * x) + before_bias

            return res

        if x > peak:
            after_bias = -(end - domain[0]) * after_peak_slope
            res = (after_peak_slope * x) + after_bias

            return res

    return trimf

```

تابع **trimf\_maker** به ازای دامنه ای که به عنوان ورودی دریافت می کند، یک تابع عضویت سه گانه ایجاد می کند. این تابع عضویت سه گانه در فازی سازی برای تقریب مقادیر یک متغیر به صورت فازی استفاده می شود. وارد کردن دامنه متغیری که می خواهید فازی سازی کنید (به صورت یک آرایه یک بعدی با طول 2) و مشخص کردن سه نقطه کلیدی این توابع (شروع، اوج و پایان) به ترتیب باعث ایجاد یک تابع عضویت سه گانه می شود که برای هر مقدار  $X$ ، میزان عضویت آن در این مجموعه فازی را بین 0 و 1 تعیین می کند.

واضح است که عضویت در مجموعه فازی 0 خواهد بود در صورتی که  $X$  کمتر از شروع یا بیشتر از پایان باشد. و همچنین در این تابع، اگر  $X$  برابر با نقطه اوج باشد، مقدار عضویت 1 است.

برای مقادیر  $X$  بین شروع و اوج، یک خط به تناسب برای محاسبه مقدار عضویت استفاده می شود، که نشان دهنده افزایش تدریجی عضویت از 0 تا 1 است.

همچنین برای مقادیر  $X$  بین اوج و پایان، یک خط کاهشی برای محاسبه مقدار عضویت به کار می رود، که نشان دهنده کاهش تدریجی از 1 به 0 است. به طور خلاصه، این تابع ایجاد شده، برای هر مقدار  $X$  که در دامنه مشخص شده قرار دارد، مقدار عضویت در مجموعه فازی را تعیین می کند، به طوری که نشان دهنده ارتباط مقدار  $X$  با مجموعه ای فازی با درصدهای مختلف عضویت است.

```

def trapmf_maker(domain, start, peak, end):
    """
    :param domain: 1d array of len 2
    :param start: the starting point
    :param peak: 1d array of len 2, showing the len of the peak value
    :param end: the ending point
    :return: a method that gives you membership capacity for a given x
    """
    assert domain[0] <= start and domain[1] >= end, "Domain or start or end is not valid"
    assert start <= peak[0] <= peak[1] <= end, "parameters are not valid"

    def trapmf(x):
        if peak[0] <= x <= peak[1]:
            return 1

        if x <= start or x >= end:
            return 0

        before_peak_slope = 1 / (peak[0] - start) if peak[0] != start else 0
        after_peak_slope = 1 / (peak[1] - end) if peak[1] != end else 0

        if x < peak[0]:
            before_bias = -(start - domain[0]) * before_peak_slope
            res = (before_peak_slope * x) + before_bias

            return res

        if x > peak[1]:
            after_bias = -(end - domain[0]) * after_peak_slope
            res = (after_peak_slope * x) + after_bias

            return res

    return trapmf

```

تابع **trapmf\_maker** یک تابع را برای ایجاد توابع عضویت چهارگانه (trapezoidal membership function) در فازی سازی ایجاد می‌کند. این توابع عضویت چهارگانه معمولاً برای نمایش متغیرهای فازی در منطق فازی استفاده می‌شوند و شامل چهار نقطه برای تعریف منطقی متغیرهای فازی هستند.

در این تابع، ورودی‌ها به صورت زیر استفاده می‌شوند:  
**Domain**: یک آرایه یک بعدی با دو عنصر که نشان‌دهنده دامنه متغیر فازی است.

**Start**: نقطه شروع توابع عضویت.  
**Peak**: یک آرایه یک بعدی با دو عنصر که نشان‌دهنده نقطه قله (نقطه جایی که مقدار تابع عضویت یک است) در محدوده دامنه تعیین شده است.

**End**: نقطه پایان توابع عضویت.  
 این تابع در نهایت یک تابع دیگر ایجاد می‌کند که عضویت چهارگانه را برای مقادیر ورودی محاسبه می‌کند. چنین توابعی به شکل چهارگوشه هایی در نمودار عضویت ظاهر می‌شوند که دارای مقادیر صفر و یک در نقاط خاص هستند و در محدوده بین دامنه تعیین شده عضویت دارند. همچنین، شبکه و ارتفاع‌های مختلف باعث می‌شوند که این توابع متفاوت باشند و به مقدار فازی متغیر وابسته باشند.

تابع `__init__` درون کلاس **Fuzzy\_Controller** به عنوان تابع مقداردهی اولیه استفاده می‌شود. در اینجا، ویژگی‌های اصلی کلاس مانند متغیرهای `N` و `k` (که مربوط به تعداد نسل‌ها و ضریب `k` برای الگوریتم بین‌مسازی هستند) و متغیرهای پیشین مانند `prev_pm` و `prev_bsf` مقداردهی اولیه می‌شوند.

علاوه بر این در این بخش از کد سه دسته اصلی از توابع عضویت فازی برای متغیرهای `pm` و `cm` ایجاد می‌شوند:

**1. mf\_pm.1** : این دیکشنری شامل توابع عضویت برای متغیر `pm` است که شامل سه حالت "low" ، "avg" و "high" هستند. هر یک از این حالت‌ها تابع عضویت فازی خود را برای مقادیر مختلف متغیر `pm` تولید می‌کند.

**2. mf\_cm.2** : این دیکشنری شامل توابع عضویت برای متغیر `cm` است که نیاز به دو حالت "low" و "high" دارد. همانند `mf_pm` ، هر حالت مقادیر مختلف عضویت را برای متغیر `cm` تولید می‌کند.

**3. mf\_gen.3** در این قسمت، سه تابع عضویت برای متغیر `gen` ایجاد شده است که شامل سه حالت "start" ، "middle" و "end" هستند. هر یک از این توابع عضویت، مقادیر عضویت را برای مقادیر مختلف متغیر `gen` تعیین می‌کنند.

این توابع عضویت فازی برای هر متغیر، با استفاده از متدهای `trimf_maker` و `trapmf_maker` از ماثول

**Membership\_functions** ساخته شده‌اند تا به ویژگی‌های خاص هر متغیر در کنترل فازی توجه کنند.

```
def __init__(self, max_gen, k):
    self.N = max_gen
    self.k = k
    self.prev_bsf = 0
    self.prev_pm = 0

    self.mf_pm = {"low": mfs.trapmf_maker([0, 1], 0, [0, 1e-3], 5e-3),
                  "avg": mfs.trimf_maker([0, 1], 1e-3, 5e-3, 1e-2),
                  "high": mfs.trapmf_maker([0, 1], 5e-3, [1e-2, 15e-3], 15e-3)}

    self.mf_cm = {"low": mfs.trapmf_maker([0, float("inf")], 0, [0, 7e-1], 99e-2),
                  "high": mfs.trapmf_maker([0, float("inf")], 7e-1, [1, float("inf")], float("inf"))}

    self.mf_gen = {"start": mfs.trapmf_maker([0, 1], 0, [0, 4e-1], 6e-1),
                   "middle": mfs.trimf_maker([0, 1], 0.4, 0.6, 0.8),
                   "end": mfs.trapmf_maker([0, 1], 6e-1, [8e-1, 1], 1)}
```

## تابع : control

```
def control(self, cur_gen, p_m, cur_bsf):  
    fuzzy_vars = self.fuzzifier(cur_gen=cur_gen, p_m=p_m, cur_bsf=cur_bsf)  
  
    # updating bsf and pm for the next control phase  
    self.prev_bsf = cur_bsf  
    self.prev_pm = p_m  
    print(f"_____cur gen is : {cur_gen}_____")  
    print(fuzzy_vars)
```

```
def fuzzify_variable(self, value, mf_dict):  
    fuzzified_values = {}  
    for label, mf_func in mf_dict.items():  
        fuzzified_values[label] = mf_func(value)  
    return fuzzified_values
```

## تابع : fuzzify\_variable

- این مت وظیفه اصلی اجرای کنترل فازی را دارد. با گرفتن مقادیر ورودی مانند **cur\_gen** (تعداد نسل فعلی)، **p\_m** (مقدار تابع منفعت) و **cur\_bsf** (بهترین جواب تاکنون)، ابتدا از تابع **fuzzifier** مقادیر فازی را دریافت می‌کند.
- سپس مقادیر **prev\_pm** و **prev\_bsf** را برای مرحله کنترل بعدی به روزرسانی می‌کند.
- در نهایت، اطلاعات مربوط به مقادیر فازی (که در **fuzzy\_vars** ذخیره شده‌اند) را چاپ می‌کند.

- این مت مقدار ورودی را با استفاده از تابع عضویت فازی (**mf\_dict**) تبدیل به مقادیر فازی مختلف می‌کند.
- با گرفتن یک مقدار **(value)** و دیکشنری توابع عضویت **(mf\_dict)**، برای هر تابع عضویت موجود در دیکشنری **(mf\_dict)**، مقدار فازی متناظر با آن تولید می‌شود و در یک دیکشنری **(fuzzified\_values)** ذخیره می‌شود.

این دو تابع به همراه با دیگر متدها و ویژگی‌های کلاس **Fuzzy\_Controller** در کنترل و مدیریت عملکرد الگوریتم بهینه‌سازی با استفاده از کنترل فازی موثر هستند.

```

# _____ Please write this part based on rulebase _____
pm_strengths = {"low": [], "avg": [], "high": []}

# Rule 1
r1s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['low'], fuzzy_vars['gen']['low'])
pm_strengths['avg'].append(r1s)

# Rule 2
r2s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['avg'], fuzzy_vars['gen']['low'])
pm_strengths['high'].append(r2s)

# Rule 3
r3s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['high'], fuzzy_vars['gen']['low'])
pm_strengths['low'].append(r3s)

# Rule 4
r4s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['low'], fuzzy_vars['gen']['low'])
pm_strengths['low'].append(r4s)

# Rule 5
r5s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['avg'], fuzzy_vars['gen']['low'])
pm_strengths['low'].append(r5s)

# Rule 6
r6s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['high'], fuzzy_vars['gen']['low'])
pm_strengths['avg'].append(r6s)

# Rule 7
r7s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['low'], fuzzy_vars['gen']['middle'])
pm_strengths['avg'].append(r7s)

# Rule 8
r8s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['avg'], fuzzy_vars['gen']['middle'])
pm_strengths['avg'].append(r8s)

# Rule 9
r9s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['high'], fuzzy_vars['gen']['middle'])
pm_strengths['low'].append(r9s)

# Rule 10
r10s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['low'], fuzzy_vars['gen']['middle'])
pm_strengths['avg'].append(r10s)

# Rule 11
r11s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['avg'], fuzzy_vars['gen']['middle'])
pm_strengths['avg'].append(r11s)

# Rule 12
r12s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['high'], fuzzy_vars['gen']['middle'])
pm_strengths['high'].append(r12s)

# Rule 13
r13s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['low'], fuzzy_vars['gen']['high'])
pm_strengths['avg'].append(r13s)

# Rule 14
r14s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['avg'], fuzzy_vars['gen']['high'])
pm_strengths['high'].append(r14s)

# Rule 15
r15s = min(fuzzy_vars['cm']['high'], fuzzy_vars['pe_prev']['high'], fuzzy_vars['gen']['high'])
pm_strengths['low'].append(r15s)

# Rule 16
r16s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['low'], fuzzy_vars['gen']['high'])
pm_strengths['avg'].append(r16s)

# Rule 17
r17s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['avg'], fuzzy_vars['gen']['high'])
pm_strengths['high'].append(r17s)

# Rule 18
r18s = min(fuzzy_vars['cm']['low'], fuzzy_vars['pe_prev']['high'], fuzzy_vars['gen']['high'])
pm_strengths['avg'].append(r18s)

return pm_strengths

```

تابع **matching** در کلاس **Fuzzy\_Controller** مسئولیت مطابقت مقادیر فازی با قوانین (rulebase) را دارد. در این تابع، با دریافت مقادیر فازی (fuzzy\_vars) برای متغیرهای **gen** و **pm\_prev** ، مقادیر قویت پیش‌بینی (pm\_strengths) بر اساس قوانین تعریف شده محاسبه می‌شود.

الگوریتم این تابع به این صورت است:

- ایجاد یک دیکشنری خالی به نام **pm\_strengths** که شامل سه حالت "avg" ، "low" و "high" است. این دیکشنری برای ذخیره کردن مقادیر قویت پیش‌بینی در هر حالت فازی ساخته شده است.

- محاسبه مقادیر قویت پیش‌بینی (pm\_strengths) براساس قوانین مشخصی که بر اساس مقادیر فازی متغیرهای **cm** و **pm\_prev** تعریف شده‌اند. برای هر یک از قوانین، مقدار **min** از مقادیر فازی متغیرها با استفاده از **min()** محاسبه می‌شود و به مجموعه مربوطه از **pm\_strengths** اضافه می‌شود.

- در نهایت، مقادیر **pm\_strengths** که شامل مقادیر قویت پیش‌بینی برای هر حالت فازی است، به عنوان خروجی تابع بازگردانده می‌شود.

این تابع در محاسبه و تعیین مقادیر قویت پیش‌بینی بر اساس قوانین فازی که برای کنترل کننده الگوریتم بهینه‌سازی مشخص شده‌اند دست دارد.

تابع **fuzzifier** در کلاس **Fuzzy\_Controller** مسئول تبدیل مقادیر ورودی الگوریتم به مقادیر فازی است. این تابع ورودی‌هایی از نوع **cur\_gen** (تعداد نسل فعلی)، **p\_m** (مقدار جدید تابع منفعت) و **cur\_bsf** (بهترین جواب تاکنون در نسل فعلی) دریافت می‌کند و مقادیر فازی مربوط به این ورودی‌ها را تولید می‌کند.

در این تابع:

یک دیکشنری است که ابتدا مقداردهی اولیه شده و شامل سه پارامتر "cm" ، "pm\_prev" و "gen" است که به ترتیب مربوط به مقادیر فازی متغیرهای "cm" (میزان تغییر بهبود بهترین جواب)، "pm\_prev" (مقدار جدید تابع منفعت) و "gen" (تعداد نسل) هستند.

در بخش شرطی این تابع (**if cur\_gen % self.k != 0:**) اگر **cur\_gen** بر **k** (یکی از ویژگی‌های اولیه کلاس) قابل قسمت پذیر نباشد، تابع بدون انجام هیچ محاسبه‌ای متوقف می‌شود.

یک تابع داخلی است که با دریافت بهترین جواب تاکنون در نسل فعلی **CM** (مقدار **cm** (بهترین جواب) را محاسبه می‌کند. این مقدار به عنوان یک معیار برای ارزیابی پیشرفت الگوریتم استفاده می‌شود.

در این تابع، مقادیر فازی مربوط به متغیرهای "cm" ، "gen" و "pm\_prev" با استفاده از توابع **fuzzify\_cm** ، **fuzzify\_pm** و **fuzzify\_generation** تولید می‌شوند و در دیکشنری **membership\_values** ذخیره می‌شوند.

در نهایت، مقادیر فازی حاصله برای متغیرهای مختلف به عنوان خروجی از تابع برگشت داده می‌شود.

```
def fuzzifier(self, cur_gen, p_m, cur_bsf):
    membership_values = {"cm": None, "pm_prev": None, "gen": None}
    if cur_gen % self.k != 0:
        return

    def CM(cur_bsf):
        """
        if cm gets closer to one, it means that algorithm is not making the desired amount of progress
        :param cur_bsf: a number, the best so far of the current generation
        :return: 0 < cm < inf
        """
        cm = self.prev_bsf / (cur_bsf + 1e-3)
        return cm

    cm = CM(cur_bsf)

    membership_values["pm_prev"] = self.fuzzify_pm(p_m)
    membership_values['cm'] = self.fuzzify_cm(cm)
    membership_values['gen'] = self.fuzzify_generation(cur_gen)

    return membership_values
```

این تابع **fuzify\_generation** در کلاس **Fuzzy\_Controller** فازی برای متغیر **gen** استفاده می‌شود. این تابع تعداد نسل فعلی الگوریتم بهینه سازی را به عنوان ورودی می‌گیرد و مقادیر "middle" ، "start" و "end" را محاسبه می‌کند.

توضیحات تابع:

• **gen** : عدد نسل فعلی الگوریتم بهینه سازی که به عنوان ورودی به تابع داده می‌شود.

• **x** : این متغیر محاسبه می‌شود تا نسبت عدد نسل فعلی به تعداد نسل‌های کلی **self.N** بدست آید.

• **start\_mf, middle\_mf, end\_mf** : این متغیرها توابع عضویت فازی برای سه حالت "start" ، "middle" و "end" از متغیر **self.mf\_gen** را به دست می‌آورند.

تابع بر می‌گرداند یک دیکشنری که هر کلید آن ، **start** ، **middle** و **end** نشان دهنده برچسب‌های مجموعه‌های فازی است و مقادیر متناظر با هر برچسب، نشان دهنده میزان عضویت عدد نرمال شده **x** در هر مجموعه فازی مربوطه است.

```
def fuzify_generation(self, gen):
    """ this method gets the number of current generations
    and calculate the gen / max gen and fuzzify the result.
    it returns the result as a dictionary with three keys (start, middle, end)
    these keys are the labels of fuzzy sets """

    x = gen / self.N
    start_mf = self.mf_gen["start"]
    middle_mf = self.mf_gen["middle"]
    end_mf = self.mf_gen["end"]

    return {"start": start_mf(x), "middle": middle_mf(x), "end": end_mf(x)}
```

```

def fuzzify_cm(self, cm):
    """ the set for cm in minimize and maximize is diffrent.
    to slve this problem and use uniq sets, we use diffrent formula
    for cm.
    minimize parameter shows that we want to maximize or not.
    """
    low_mf = self.mf_cm["low"]
    high_mf = self.mf_cm["high"]

    return {"low": low_mf(cm), "high": high_mf(cm)}

def fuzzify_pm(self, p_m):
    low_mf = self.mf_pm["low"]
    avg_mf = self.mf_pm["avg"]
    high_mf = self.mf_pm["high"]

    return {"low": low_mf(p_m), "avg": avg_mf(p_m), "high": high_mf(p_m)}

def Reset(self):
    """
    This method resets the pm and bsf of the controller,
    giving it the quality to be used multiple times for the multiple runs of the algorithm
    :return: Noting
    """
    self.prev_pm = 0
    self.prev_bsf = 0

```

در کلاس **Fuzzy\_Controller** ، این سه متد عده وظیفه های مختلفی را انجام می دهند:

#### : fuzzify\_cm.1

- این تابع برای تبدیل مقادیر ورودی مربوط به متغیر **cm** به مقادیر فازی استفاده می شود.

با توجه به مقادیر **cm** دریافتی، از توابع عضویت **low** و **high** که قبلاً در **self.mf\_cm** مقداردهی شده‌اند، استفاده می‌کند.

در نهایت، مقادیر فازی متناظر با مقادیر **cm** در حالت‌های **low** و **high** برگردانده می‌شود.

#### : fuzzify\_pm.2

- این تابع برای تبدیل مقادیر ورودی مربوط به متغیر **pm** به مقادیر فازی استفاده می شود.

با استفاده از مقادیر **p\_m** دریافتی، از توابع عضویت **low** ، **avg** و **high** که قبلاً در **self.mf\_pm** مقداردهی شده‌اند، استفاده می‌شود.

در نهایت، مقادیر فازی متناظر با مقادیر **p\_m** در حالت‌های **low** ، **avg** و **high** برگردانده می‌شود.

#### : Reset.3

- این متد یک عملیات بازنشاندن (reset) برای کنترل‌کننده انجام می‌دهد.

وظیفه آن اعمال بازنشانی بر روی مقادیر **prev\_bsf** و **prev\_pm** است که مربوط به متغیرهای پیشین **pm** و بهترین جواب تا کنون می‌باشد. این کار باعث می‌شود که کنترل‌کننده برای اجرای چندین دور از الگوریتم بهینه‌سازی، آماده شود و از مقادیر قبلی خود برای محاسبات جدید استفاده نکند.

```

def main():
    fcs = Fuzzy_Controller(100, 5)
    bsf = np.arange(0, 202, 2)
    pm = np.arange(1e-3, 1e-2, 9e-5)
    for i in range(100):
        fcs.control(cur_gen=i, p_m=pm[i], cur_bsf=bsf[i])

if __name__ == "__main__":
    main()

```

این تابع `main()` در واقع یک تابع آزمایشی است که یک نمونه از کلاس **Fuzzy\_Controller** ایجاد می‌کند و سپس یک حلقه اجرا می‌کند. در این حلقه، برای هر زدرا بازه ۰ تا ۹۹، متدهای کنترل (`fcs.control()`) از کلاس **Fuzzy\_Controller** را با پارامترهای `cur_bsf=bsf[i]` و `p_m=pm[i]` و `cur_gen=i` فراخوانی می‌کند.

این تابع کنترل کننده فازی را با داده‌های ورودی مختلف تست و عملکرد آن را بررسی می‌کند. با فراخوانی این تابع `main()`، الگوریتمی که توسط کلاس **Fuzzy\_Controller** پیاده‌سازی شده است اجرا می‌شود و عملکرد آن بررسی می‌شود.

نتیجه در صورت رسیدن به یکی از شروط پایان الگوریتم... .

## Griewank Function

Formula :  $f(X) = \sum_{i=1}^d \frac{x_i^2}{4000} - \prod_{i=1}^d \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$

Global Minimum :  $f(X^*) = 0$  Where  $X^* = (0, \dots, 0)$

Domain :  $x_i \in [-600, 600]$

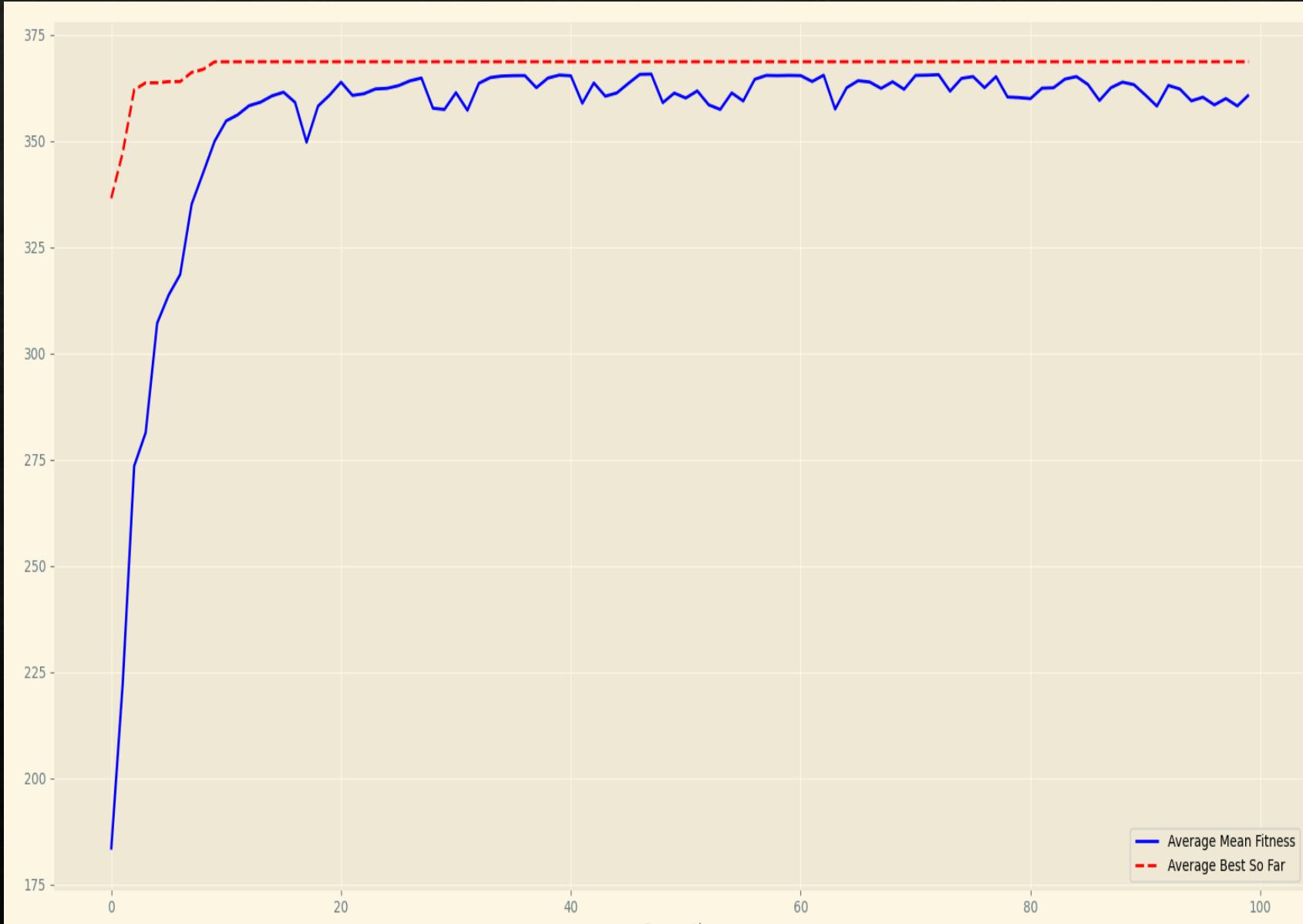
Running Real-Valued Genetic Algorithm	
Parameter	Value
Algorithm Runs	1
Maximum Generation	100
Population Size	80
CrossOver Rate	0.5
Mutation Rate	0.001
Function Input Shape	6
Controller	FCS

# Griewank Function's Parameters

```
Results
-----
Average Best Solution   : (-4.075598821417893, -12.440435493319585, -54.39755872315799, 62.23701590024873, -84.41090695938637, -29.912495740699136)
Optimum Value          : 4.325145663606861
Last Average Best So Far : 368.63792203077355
Last Average Mean Fitness : 360.84500241002047
```

# Griewank Function's Results

# Average Mean Fitness and Average Best So Far of Griewank Function



# Griewank Function's Mutation Rate



# Rastrigin Function

Formula :  $f_{(x)} = 10d + \sum_{i=1}^d [x_i^2 - 10 \cos(2\pi x_i)]$

Global Minimum :  $f(x^*) = 0$  Where  $x^* = (0, \dots, 0)$

Domain :  $x_i \in [-5.12, 5.12]$

# Rastrigin Function's Parameters

Parameter	Value
Algorithm Runs	5
Maximum Generation	300
Population Size	300
CrossOver Rate	0.5
Mutation Rate	0.001
Function Input Shape	4
Controller	FCS

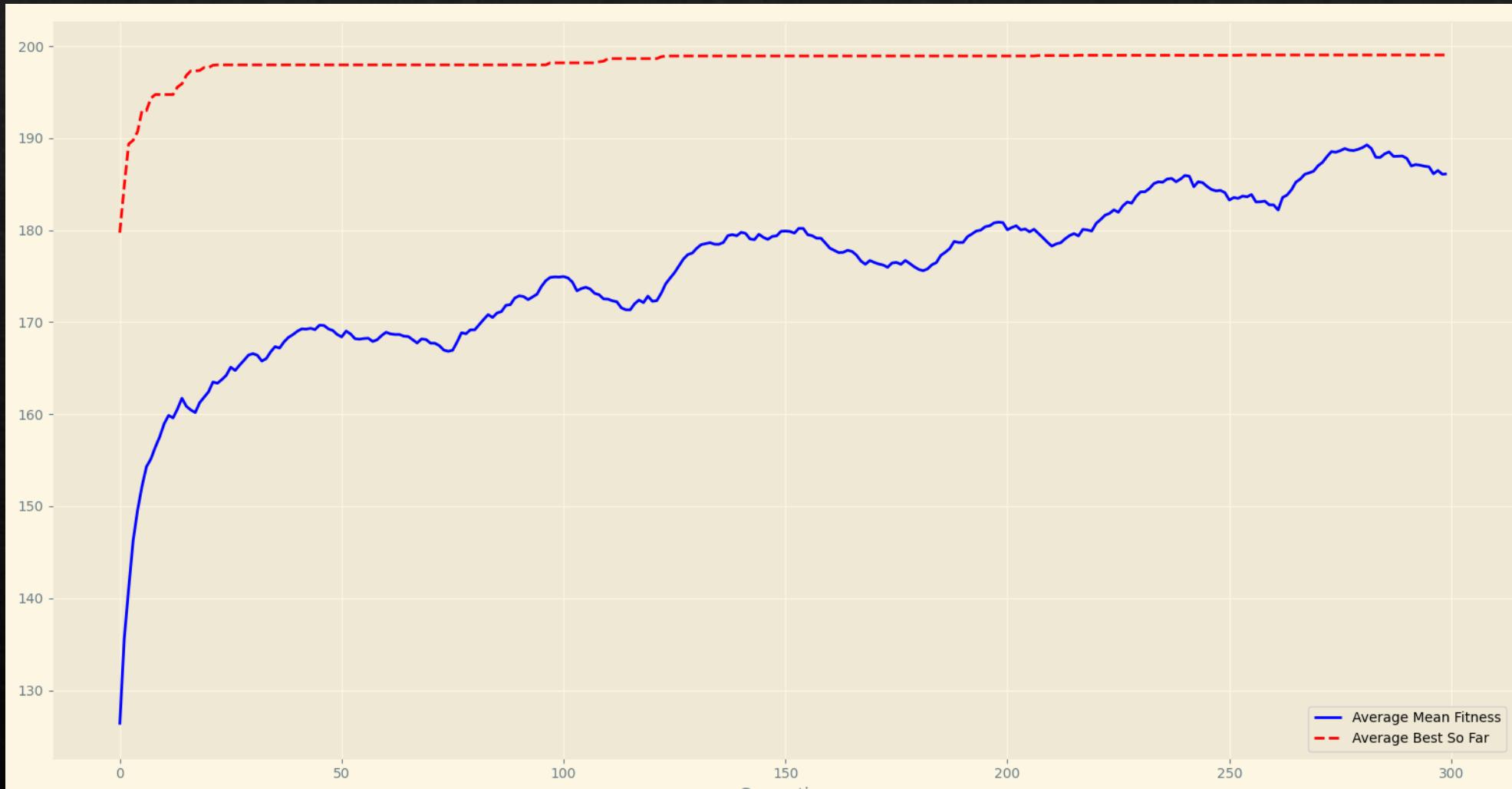
```
# Rastrigin Function
rga_rastrigin = RGA(target_function=theorem.rastrigin, fitness_function=lambda x: 200 - theorem.rastrigin(x),
                      function_config=[{'low': -5.12, 'high': 5.12}, {'low': -5.12, 'high': 5.12},
                                       {'low': -5.12, 'high': 5.12}, {'low': -5.12, 'high': 5.12}],
                      crossover_rate=0.5, max_gen=300, population=300, run_rga=5, controller=Fuzzy_Controller, gpc=20)
```

# Rastrigin Function's Runs

# Rastrigin Function's Results

```
----- Results -----  
Average Best Solution   : (-0.02943562349284689, 0.008132433138281887, -0.005966885713978293, -0.013503304631166335)  
Optimum Value           : 0.22774488920414626  
Last Average Best So Far : 199.03199588770025  
Last Average Mean Fitness : 186.11626515206262
```

# Average Mean Fitness and Average Best So Far of Rastrigin Function



# Mutation Rate of Rastrigin Function

