# گزارش کار

## پردازش تکاملی

## تمرین 3

**آرمین خیاطی**

**9931153**

## مقدمه

در این تمرین قصد داریم تا مسئله System reliability optimization را با استفاده از یک الگوریتم Evolutionary Strategy حل کنیم. برای طراحی یک سیستم قابل اطمینان دو روش داریم : 1- افزایش تعداد قطعات 2- افزایش قابل اطمینان بودن قعات

در روش اول با افزایش تعداد قطعات، با از کار افتادن قطعه ای، قطعات دیگر جایگزین آن میشوند و سیستم هنوز میتواند به کار خود ادامه دهد. در روش دوم نیز با استفاده از قطعاتی با کیفیت بالاتر میتوان اطمینان پذیری سیستم را بالاتر برد. فرض کنید سیستم ما یک خودرو باشد. این خودرو خود شامل چند زیر سیستم میشود که هر زیر سیستم شامل قطعاتی می باشد. اگر ما تنها از روش یک استفاده کنیم و هر چقدر که میخواهیم قطعات بیشتری استفاده کنیم، وزن خودرو زیاد شده و حرکت نمیکند و اگر از روش دوم و از بهترین قطعات موجود استفاده کنیم، هزینه نهایی بسیار بالا میرود. هدف ما پیدا کردن تعادلی بین استفاده از قطعات بیشتر و قطعات با کیفیت تر می باشد. به این مسئله System reliability optimization می گویند.

اما الگوریتم Evolutionary Strategy چیست؟ این نوع الگوریتم ها با کمی تفاوت با الگوریتم های ژنتیک که برای مسائل گسسته بودند، برای مسائلی که دامنه مقادیر پیوسته دارند مطرح میشود. اوپراتور اصلی آن Mutation یا جهش است و عمل کراس اور تقریبا یا در مواردی کاملا بی اثر است. عمل انتخاب والد بصورت Random Uniform می باشد و دو نوع Survival Selection تحت عنوان (mu + landa) و (mu,landa) در آن مطرح میشود که mu جمعیت اصلی و landa فرزندان تولید شده می باشد. معمولا بیشتر مواقع (mu,landa) بهتر از دیگری عمل میکند و نتایج بهتری میدهد حتی با اینکه در اولی ما نخبه گرایی داریم. در روش (mu,landa) اگر جمعیت ما شامل 100 فرد باشد، 700 فرزند از آن ها تولید میکنیم و صد تا از بهترین فرزندان را بعنوان جمعیت نسل بعدی انتخاب میکنیم. در روش (mu+landa) نیز با ترکیب 100 فرد جمعیت و 700 فرزند تولید شده ما 800 فرد داریم، از این 800 نفر ما بهترین 100 نفر را بر اساس فیتنس انتخاب میکنیم.

عملگر جهش نیز بحث مفصلی دارد که همه ی آن در اینجا نمیگنجد و علاقه مندان میتوانند به کتاب Introduction to Evolutionary Computing مراجعه کنند. اگر بخواهیم عملگر جهشی که در اینجا استفاده میشود را توضیح دهیم، باید یک فرد، یا کروموزوم را در نظر بگیریم که شامل برای مثال 5 ژن با مقادیر پیوسته هست. برای هر ژن یک متغیر سیگما را اختصاص میدهیم و مقدار جدید هر ژن با جمع مقدار قدیمی آن با حاصل ضرب سیگمایش در یک مقدار تصادفی از توزیع گاوسی با میانگین صفر و واریانس یک حساب میشود.

$$x'_i = x_i + \sigma'_i \cdot N_i(0,1)$$

این سیگما جهت و شدت حرکت مقدار ژن در فضا را تعیین میکند. حالا اینکه خود چطور بدست می آید، این نیز بسیار ساده است. سیگمایی که برای جهش و تعیین مقدار جهش ژن استفاده میشود، خود باید قبل از استفاده از آن جهش یابد. یعنی سیگمای جدید را حساب کنیم، و با این سیگمای جدید، مقدار جدید ژن را محاسبه کنیم.

سیگمای جدید برابر است با ضرب سیگمای قدیمی در e عدد نپر به توان مقدار تاو پرایم ضرب در یک عدد تصادفی توزیع گاوسی با میانگین صفر و واریانس یک که فقط یکبار تولید و برای همه ژن ها استفاده میشود بعلاوه مقدار تاو ضرب در یک عدد تصادفی توزیع گاوسی با همان میانگین و واریانس با این تفاوت که برای هر ژن جداگانه تولید میشود.

$$\sigma'_i = \sigma_i \cdot \exp(\tau' \cdot N(0,1) + \tau \cdot N_i\,(0,1))$$

متغیر تاو پرایم همان learning rate کلی است و متغیر تاو coordinate wise learning rate می باشد.

$$\tau' \propto 1/\sqrt{2n}$$

$$\tau \propto 1/\sqrt{2\sqrt{n}}$$

با اینکه در الگوریتم های ES چندین نوع عملگر کراس اور وجود دارد ولی در این تمرین هیچکدام استفاده نشده است و بدون این عملگر به نتایج بسیار خوبی میرسیم. میتوانید با مراجعه به بخش 4.4.3 کتاب ذکر شده، این عملگر ها را مطالعه کنید.

برای حل این مسئله ما چندین پارامتر داریم که در جدول زیر همه آن ها توضیح داده شده است. اصل مسئله در مقاله ای تحت عنوان An improved particle swarm optimization algorithm for reliability problems مطرح شده و با الگوریتم تکاملی به نام Particle Swarm Optimization (PSO) حل شده است اما در اینجا ما با یک الگوریتم Evolutionary Strategy قصد حل آن را داریم.

*Table 1: Notations*

| *Notation* | *Description* |
|---|---|
| $m$ | *number of subsystems in the system.* |
| $n_i$ | *the number of components in ith subsystem. 1≤ i ≤ m.* |
| $n$ | *n= [n₁,…, nₘ], the vector of the number of components in* |
| $r_i$ | *system.the reliability of each component in ith subsystem.* |
| $r$ | *= [r₁,…, rₘ], the vector of component reliabilities.* |
| $w_i$ | *the weight of each component in ith subsystem.* |
| $c_i$ | *the cost of each component in ith subsystem.* |
| $v_i$ | *the volume of each component in ith subsystem.* |
| $R_i$ | *the reliability of the ith subsystem.* $R_i = 1-(1-r_i)^{n_i}$. |
| $R_s$ | *the reliability of the system.* |
| $C$ | *the upper limit on the cost of the entire system.* |
| W | *the upper limit on the weight of the entire system.* |
| $V$ | *the upper limit on the volume of the entire system.* |

در این مسئله دو نوع سیستم پیچیده (ترکیب موازی و سری) و سیستم سری وجود دارد. که هر کدام تابع فیتنس خود را دارند. سایر پارامتر ها برای هر دو نوع سیستم ثابت است. در واقع باید یکبار الگوریتم را با تابع عضویت سیستم پیچیده و یکبار با تابع عضویت سیستم سری اجرا کنیم. سایر موارد دست نخورده باقی میمانند.

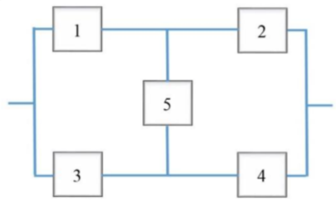



Figure 1. Complex (bridge) system (P1)

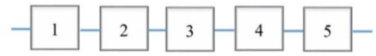



Figure 2. series system (P2)

در اینجا هر سیستم (یا کروموزوم) پنج زیر سیستم (ژن) دارد که هر ژن نیز تعداد قطعات (n) و میزان قابل اطمینان بودن قطعات در هر زیر سیستم (r) خود را دارد. هدف ما بیشینه کردن تابع عضویت سیستم هاست که بر اساس این دو مقدار محاسبه میشود. برای سیستم های پیچیده، تابع عضویت زیر

$$Max\ f(r.nc) = R_1R_2 + R_3R_4 + R_1R_4R_5 + R_2R_3R_5 - R_1R_2R_3R_4 - R_1R_2R_3R_5 - R_1R_2R_4R_5 - R_1R_3R_4R_5 - R_2R_3R_4R_5 + 2R_1R_2R_3R_4R_5$$

و برای سیستم های سری نیز تابع عضویت زیر را داریم.

$$Max\ f(r.n) = \prod_{i=1}^{m} R_i = \prod_{i=1}^{5} (1 - (1 - r_i)^{n_i})$$

نکته قابل توجه این است که سیستم ها، چه سری و چه پیچیده، باید سه شرط زیر را داشته باشند و اگر یکی از آن ها نقض شود، مقدار تابع عضویت آن ها صفر میشود. شرط اول میگوید با این فرمول اگر حجم هر سیستم را به دست بیاوریم و از آستانه حجم V کم کنیم باید حاصل کمتر یا مساوی صفر شود یعنی حجم سیستم از آستانه V بیشتر نشود. شرط دوم نیز برای هزینه سیستم و آستانه C میباشد و شرط سوم نیز برای وزن سیستم و آستانه W می باشد.

$$g_1(\mathbf{r}, \mathbf{n}) = \sum_{i=1}^{m} w_i v_i^2 n_i^2 - V \le 0$$

$$g_2(\mathbf{r}, \mathbf{n}) = \sum_{i=1}^{m} \alpha_i \left( -\frac{1000}{\ln(r_i)} \right)^{\beta_i} [n_i + \exp(0.25 n_i)] - C \le 0$$

$$g_3(\mathbf{r}, \mathbf{n}) = \sum_{i=1}^{m} w_i n_i \exp(0.25 n_i) - W \le 0$$

$$0 \le r_i \le 1, \quad n_i \in Z^+, \ 1 \le i \le m.$$

مقدار C ،W و V برای همه سیستم ها و در کل مسئله یکسان است اما مقادیر آلفا، بتا، n ،r ،w و $wv^2$ برای هر زیر سیستم متفاوت است. دامنه مقادیر r ها بین صفر و یک و مقداری پیوسته است ولی برای مقادیر n شامل اعداد صحیح مثبت غیر صفر و گسسته است. جدول زیر پارامتر ها را به ازای هر زیر سیستم یک تا پنج مشاهده میکنید.

*Table 2. Parameter used for complex and series system*

| $i$ | $10^5\alpha_i$ | $\beta_i$ | $w_i v_i^2$ | $w_i$ | $V$ | $C$ | $W$ |
|---|---|---|---|---|---|---|---|
| 1 | 2.330 | 1.5 | 1 | 7 | 110 | 175 | 200 |
| 2 | 1.450 | 1.5 | 2 | 8 | 110 | 175 | 200 |
| 3 | 0.541 | 1.5 | 3 | 8 | 110 | 175 | 200 |
| 4 | 8.050 | 1.5 | 4 | 6 | 110 | 175 | 200 |
| 5 | 1.950 | 1.5 | 2 | 9 | 110 | 175 | 200 |

مقدار W، C و V برای همه سیستم ها و در کل مسئله یکسان است اما مقادیر آلفا، بتا، n، r، w و $wv$^2 برای هر زیر سیستم متفاوت است. در ستون دوم، مقداری که در آلفا ضرب شده به این معنی هست که الفا خیلی کوچک است و برای نمایش بهتر در جدول آن ها را در ده به توان پنج ضرب کرده ایم. در مسئله باید مقدار اصلی آلفا ها را یعنی تقسیم شده بر ده به توان پنج، استفاده کنیم.

## پیاده سازی

برای شبیه سازی سیستم و زیر سیستم ها دو کلاس به همین نام ها میسازیم. کلاس Subclass شامل دو متد، یکی برای تنظیم مقدار پارامتر ها بر اساس شماره زیر سیستم و دومی برای جهش زیر سیستم است.

```
class Subsystem:
    def __init__(self, index):
        self.index = index
        self.__init_params()
```

متد تنظیم پارامتر ها بر اساس شماره زیر سیستم :

```python
def __init_params(self):
    self.beta = 1.5
    self.t1 = 1 / sqrt(2 * 5)
    self.t2 = 1 / sqrt( 2 * sqrt(5))
    self.n = randint(1, 5) # generate random integer for n_i
    self.r = random()       # generate random float between 0 and 1 for r_i
    self.sigma = 1          # generate random value for sigma
    self.sigman = 1
    self.r_mutation_rate = 90
    self.n_mutation_rate = 70

    if self.index == 0:
        self.alpha = 2.330E-5
        self.wv2 = 1
        self.w = 7
    elif self.index == 1:
        self.alpha = 1.450E-5
        self.wv2 = 2
        self.w = 8
    elif self.index == 2:
        self.alpha = 0.541E-5
        self.wv2 = 3
        self.w = 8
    elif self.index == 3:
        self.alpha = 8.050E-5
        self.wv2 = 4
        self.w = 6
    elif self.index == 4:
        self.alpha = 1.950E-5
        self.wv2 = 2
        self.w = 9
```

متد جهش زیر سیستم ها:

```python
def update(self, random_num):
    '''Mutate r_i'''
    if randint(0,100) < self.r_mutation_rate:
        power =  (self.t1 * random_num) + (self.t2 * gauss(0, 1))
        new_sigma = self.sigma * exp(power)
        # A boundary rule is applied to prevent standard deviations very close to zero.
        new_sigma = new_sigma if new_sigma > 0.1E-4 else 0.1E-4
        new_r = self.r + (new_sigma * gauss(0, 1))
        if new_r >= 0 and new_r <= 1:
            self.r = new_r
            self.sigma = new_sigma
    '''Mutate n_i'''
    if randint(0,100) < self.n_mutation_rate:
        power =  (self.t1 * random_num) + (self.t2 * gauss(0, 1))
        new_sigman = self.sigman * exp(power)
        new_sigman = new_sigman if new_sigman > 0.1E-4 else 0.1E-4
        new_n = self.n + floor(new_sigman * gauss(0, 1))
        if new_n > 0:
            self.n = new_n
            self.sigman = new_sigman
```

کلاس سیستم نیز شامل لیست پنج تایی از زیر سیستم ها و مقدار پارامتر های W، C و V میباشد.

```python
class System():

    def  __init__(self):
        self.subsystems = []
        self.V = 110
        self.W = 200
        self.C = 175
        self.num = 5 # Number of subsystems
        self.__create_subsystems()

    def __create_subsystems(self):
        for i in range(self.num):
            self.subsystems.append(Subsystem(i))

    def update(self):
        random_num = gauss(0, 1)
        for i in range(len(self.subsystems)):
            self.subsystems[i].update(random_num)
```

پیاده سازی توابع g1 و g2 و g3 که شروط مسئله ما هستند نیز به صورت زیر است.

```python
def g1(system):
    res = 0
    for subsys in system.subsystems:
        res += subsys.wv2 * (subsys.n ** 2)
    return res - system.V <= 0

def g2(system):
    res = 0
    for subsys in system.subsystems:
        temp = subsys.alpha * ((-1000 / log(subsys.r)) ** subsys.beta)
        res += temp * subsys.n + exp(0.25 * subsys.n)
    return res - system.C <= 0

def g3(system):
    res = 0
    for subsys in system.subsystems:
        res += subsys.w * subsys.n * exp(0.25 * subsys.n)
    return res - system.W <= 0
```

برای محاسبه $R_i$ ها نیز که میزان قابل اطمینان بودن هر زیر سیستم هستند و با توجه به مقدار قابل اطمینان بودن قطعات در یک زیر سیستم یعنی $r_i$ به دست می آیند نیز با تابع زیر محاسبه میشوند.

```python
def subsys_reliability(system):
    R = []
    for subsys in system.subsystems:
        temp = 1 - ((1 - subsys.r) ** subsys.n)
        R.append(temp)
    return R
```

تابع محاسبه فیتنس هر سیستم پیچیده را نیز در زیر میبینید.

```python
def fitness_func_complex(system):
    if g1(system) and g2(system) and g3(system):
        R = subsys_reliability(system)
        fitness = ((R[0]*R[1]) + (R[2]*R[3]) + (R[0]*R[3]*R[4])+
                   (R[1]*R[2]*R[4]) - (R[0]*R[1]*R[2]*R[3]) - (R[0]*R[1]*R[2]*R[4])-
                   (R[0]*R[1]*R[3]*R[4]) - (R[0]*R[2]*R[3]*R[4]) - (R[1]*R[2]*R[3]*R[4])+
                   (2*R[0]*R[1]*R[2]*R[3]*R[4]))
        return fitness
    else:
        return 0
```

برای فیتنس سیستم های سری نیز تابع زیر را داریم

```python
def fitness_func_series(system):
    if g1(system) and g2(system) and g3(system):
        R = subsys_reliability(system)
        fitness = 1
        for r in R:
            fitness *= r
        return fitness
    else:
        return 0
```

برای تولید جمعیت اولیه که همه افراد آن valid باشند و شروط مسئله را رعایت کنند نیز تابع زیر را داریم.

```python
def init_population(size):
    population = []
    while len(population) < size:
        system = System()
        if g1(system) and g2(system) and g3(system):
            population.append(system)
        else:
            continue
    return population
```

برای انتخاب والد برای تولید فرزند نیز تابع زیر را استفاده میکنیم.

```
def select(population):
    index = randint(0, len(population) - 1)
    return deepcopy(population[index])
```

برای تولید فرزندان نیز تابع زیر را داریم.

```
def generate_childs(population, size):
    childs = []
    while len(childs) < size:
        system = select(population)
        system.update()
        childs.append(system)
    return childs
```

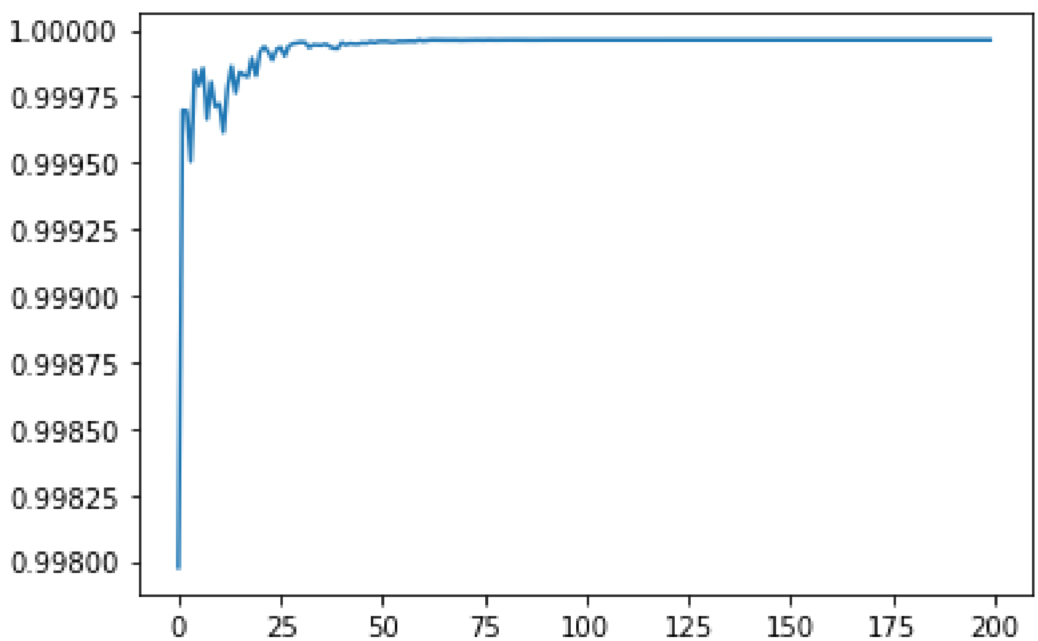
برای اجرای الگوریتم در 200 حلقه نیز توابع زیر را داریم.

```
def fit_complex(population):
    epochs = 200
    size = len(population) * 7
    for i in range(epochs):
        childs = generate_childs(population, size)
        childs_fitness = list(map(fitness_func_complex, childs))
        sorted_indexes = np.array(childs_fitness).argsort()[::-1]
        mean_fit_log.append(np.mean(np.array(childs_fitness)[sorted_indexes[:len(population)]]))
        best_fit_log.append(childs_fitness[sorted_indexes[0]])
        population = np.array(childs)[sorted_indexes[:len(population)]]
```

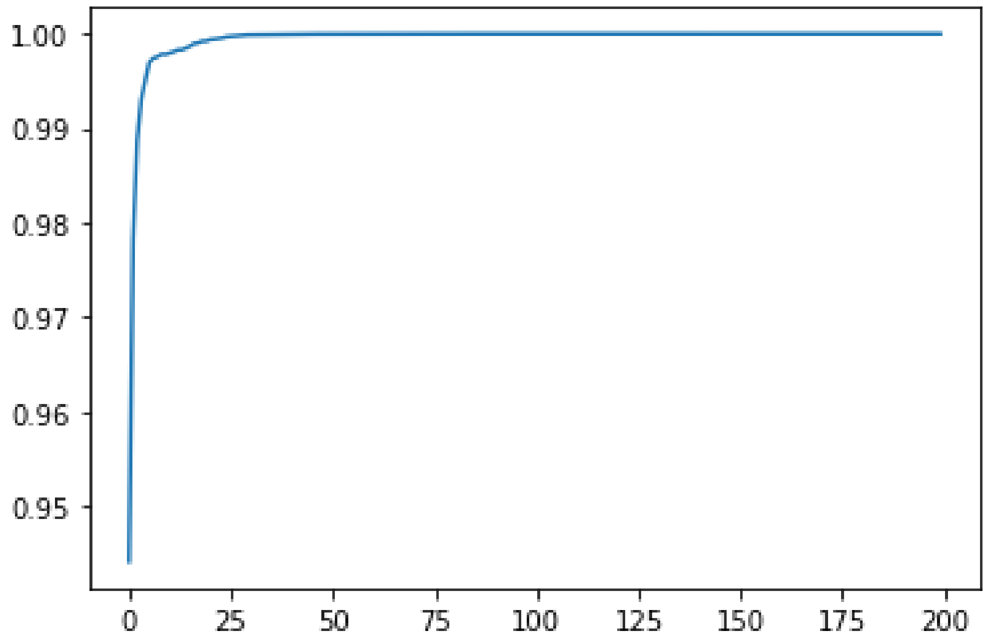
```
def fit_series(population):
    epochs = 200
    size = len(population) * 7
    for i in range(epochs):
        childs = generate_childs(population, size)
        childs_fitness = list(map(fitness_func_series, childs))
        sorted_indexes = np.array(childs_fitness).argsort()[::-1]
        mean_fit_log.append(np.mean(np.array(childs_fitness)[sorted_indexes[:len(population)]]))
        best_fit_log.append(childs_fitness[sorted_indexes[0]])
        population = np.array(childs)[sorted_indexes[:len(population)]]
```

## نتیجه

بعد از اجرای الگوریتم مشاهده میکنیم که بهترین فیتنس های سیستم های پیچیده در هر نسل رو به افزایش است و در نهایت به یک میرسید. همین روند نیز برای میانگین فیتنس های هر نسل در سیستم های پیچیده نیز مشاهده میشود.

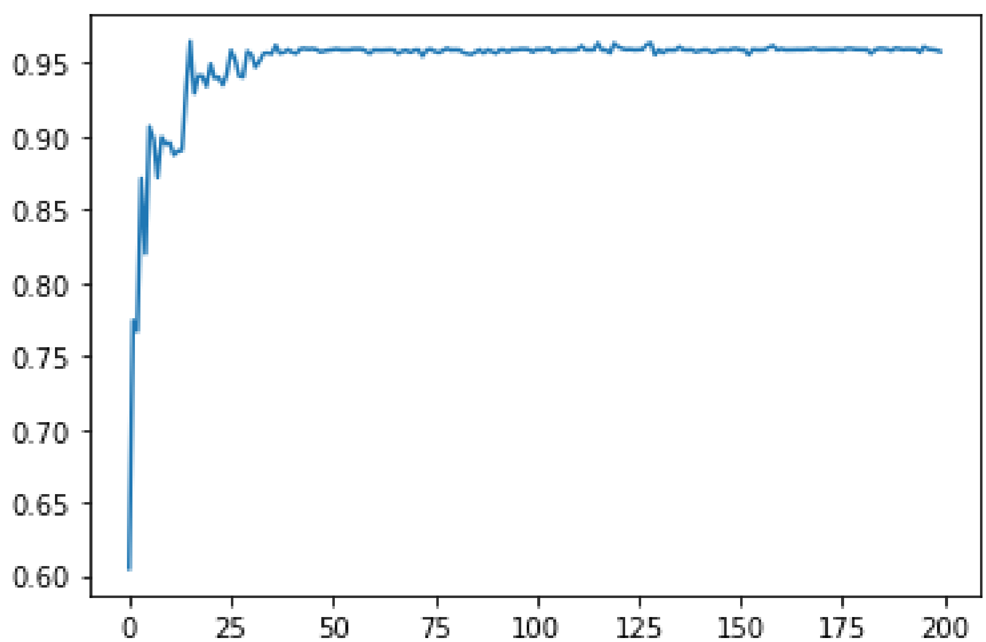



*Best Fitness1 Figure*

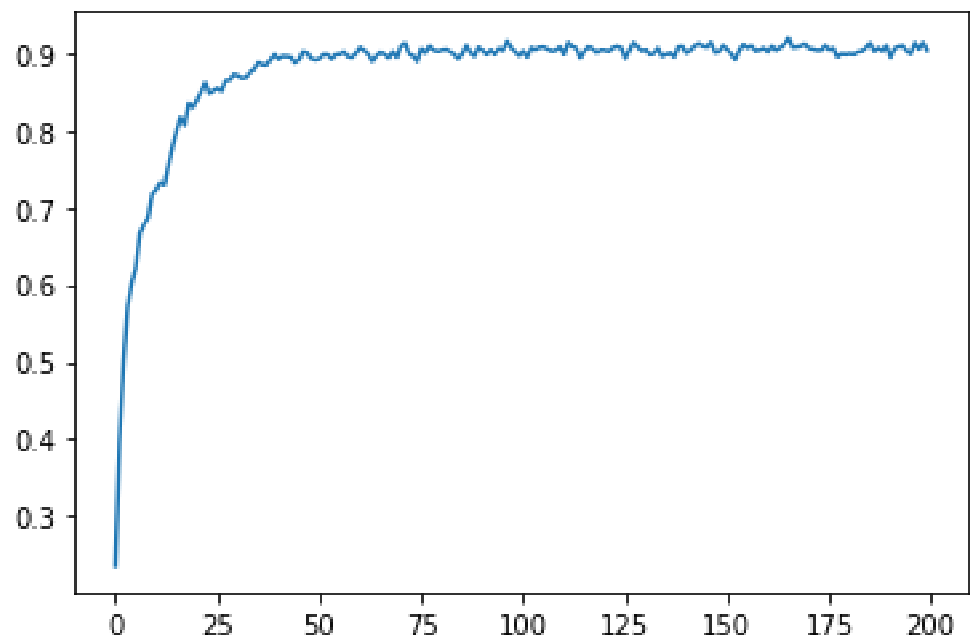



*Mean Fitness2 Figure*

در سیستم های سری نیز همین روند صعودی وجود دارد ولی بهترین فیتنس در 0.95 متوقف شده و میانگین فیتنس ها نیز به 0.9 میرسد. همچنین نوسان سیستم سری نیز بیشتر از پیچیده است.




*Best Fitness3 Figure*




*Mean Fitness4 Figure*