情報特別演習最終レポート

連続最適化問題・制御タスクに対する進化計算の比較研究

田邊　拓実

平成 31 年 2 月 8 日

1　概要

進化計算とは，最適化において，個体群である生成した多数の探索点を事前に設定した評価法により順位付けを行いそれを元に次の個体群を生成，という流れを繰り返すことにより最適解を求めるというものである．この進化計算という枠組みの中にも遺伝的アルゴリズム (GA) や遺伝的プログラミング (GP)，進化戦略 (ES) などの進化的アルゴリズム (EA) や粒子群最適化 (PSO) や蟻コロニー最適化 (ACO) などの群知能，また差分進化 (DE) などが存在する．EA とは個体群から個体に付けられた順位に応じて最適な解を選ぶことで個体群を最適なものに近づけることを目的とした選択，個体群の解の探索範囲を狭めすぎないようにするために多様性を維持することを目的とした突然変異，現在の個体から新たな個体を生み出す交叉を繰り返し行うことで最適解を探索するというものである．群知能とは個体ごとの最適化に加え個体群の中の個体同士で情報をやり取りすることにより，最適解を求めていくというものである．
今回は様々な最適化問題の中から連続最適化問題である関数最適化と動的関数最適化，強化学習などでよく用いられる制御タスクに対して比較を行った．関数最適化に関しては，各手法が決めた世代数までに最適解を求めることがどれぐらいできるかという観点から比較する実験を行った．制御タスクに関しては各手法が最終的に得られる最良解の評価値を比較した．それぞれの手法の設定 (GA における交叉法や共分散進化戦略における共分散行列の更新方法) などは事前の調べや実験により一番結果が良好なものを用いた．
このレポートは，それぞれの問題での実験問題の設定，使ったアルゴリズムの説明，実験結果，その後にそれぞれの実験結果に対する考察，まとめという流れで書かれている．また，各実験の実装したソースコードは Git Hub(https://github.com/yoshinobc/EA/tree/master/experiment) にまとめた．

2　連続最適化問題

2.1　問題設定

関数最適化問題に対して，各手法が決めた世代数までに最適解を求めることがどれぐらいできるかを比較するにあたり，個体数を 150 個, 最終世代数を 200 世代に設定し，それぞれの関数，手法に対して 500 回実験を行った．また，世代の中での最良解がそれぞれの関数の 最適解 $+ e^{(-10)}$ になったときに最適解を求められたと判断した．
実験には以下の関数を用いた．単峰生関数である Sphere 関数

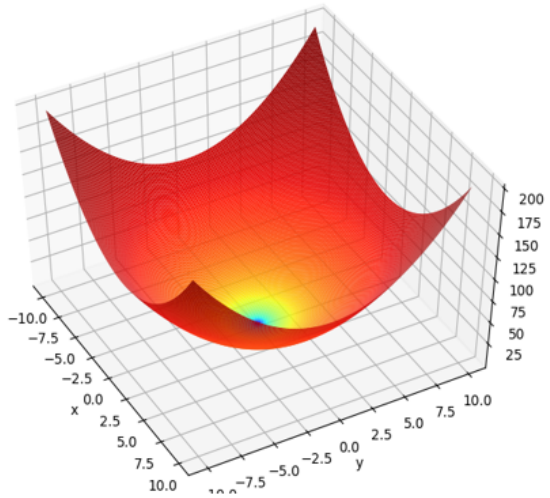

図 1: sphere 関数

，1 つの局所解を持つ Double-Cone 関数 [図 2]，

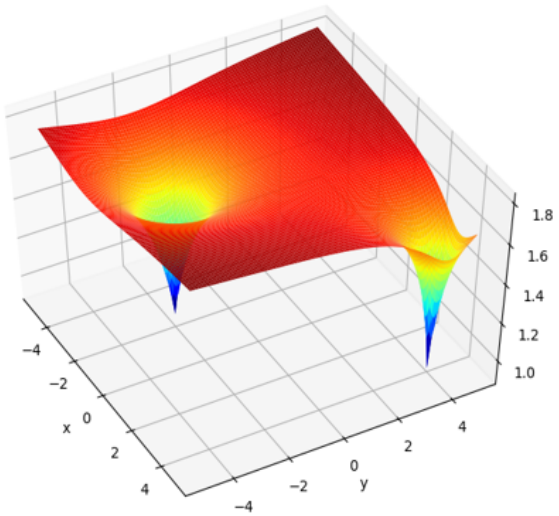

図 2: double-cone 関数

大域的に見たら単峰生関数だが多数の局所解を持つ Griewank 関数 [図 3]，

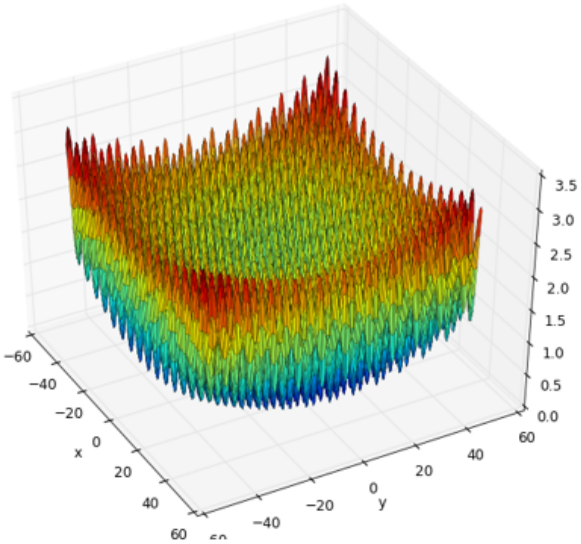

図 3: Griewank 関数

4 つの最適解を持つ Himmelblau 関数 [図 4]

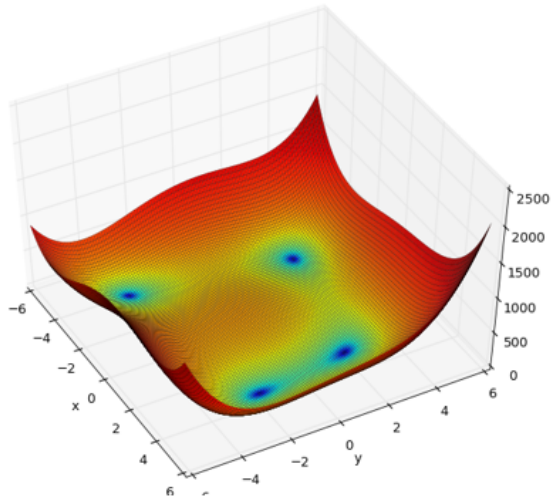

図 4: Himmelblau 関数

を用いた．これらの関数の最適解などについては表 1 にまとめた．これらの静的な関数に加えて動的な関数 [図 5][図 6] も比較に用いた．

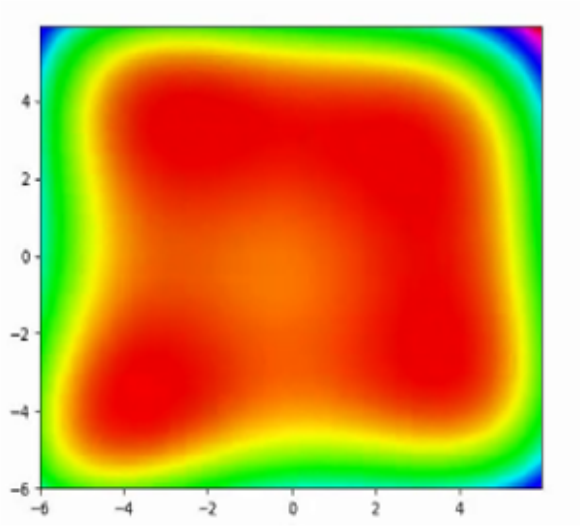

図 5: 動的関数

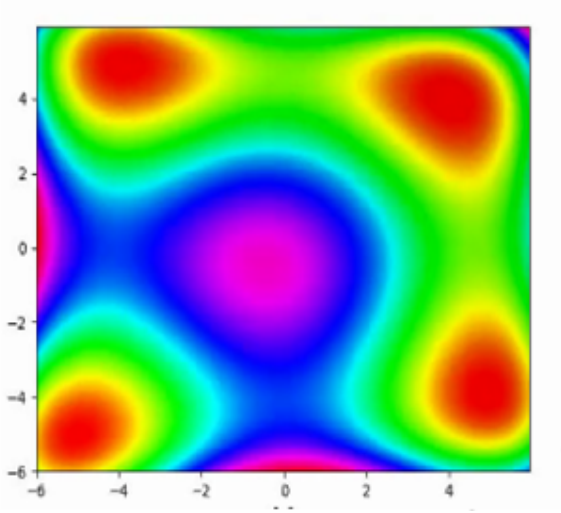

図 6: 動的関数

この関数は最適解の位置が時間ごとに変化する関数である．この動的関数を用いることにより各手法が常に変化する環境に対しての適応力を比較することができると考えられる．また，動的な関数については以下のようなプログラムで実装されている．

ソースコード 1: 動的な関数

```
def mapping(gen):
    if gen <= 22:
        num = gen
    elif 22 < gen <= 44:
        num = 44 - gen
    elif 44 < gen <= 66:
        num = gen - 45
    elif 66 < gen <= 88:
        num = 88 - gen
    elif 88 < gen:
        num = gen - 89

    return num
def dynamic_func(pop):
            pop = numpy.clip(pop
                ,-6,6)
            X = pop[0]
            Y = pop[1]
            return numpy.power((numpy.
                power(X,2) + Y -
                mapping(gen)),2) +
                numpy.power((X +
                numpy.power(Y,2) -
                mapping(gen)),2)
```

2.2 アルゴリズム

2.2.1 遺伝的アルゴリズム

遺伝的アルゴリズムは探索点を遺伝子型で表現したものを持つ解の候補を複数用意し，その個体群に対して事前に決めた評価値を割り当て，評価値の高い個体を優先的に選択して交叉，突然変異を行うことで解を探索するというものである．(Algorithm 1)

今回の関数最適化問題では，関数への入力を遺伝子に割り当てその出力の値を評価値とした．選択方法にはトーナメント選択を用いた．これはあらかじめ決めておいたトーナメントサイズ分ランダムで個体を取り出し，その中で最も適応度の高い個体を取り出すというものである．これを使うことで比較的早くに解を収束させることができると

考え使用した．交叉方法にはBLX-αを用いた．これ個体p1,p2を交差させ個体cを生成するときに,

$$c_i = u(min(p_{1,i}, p_{2,i}) - \alpha I, max(p_{1,i}, p_{2,i} + \alpha I)$$

$$I = |p_{1,i} - p_{2,i}|, u(x, y)\text{は区間}[x, y]\text{の一様乱数}$$

とするものであり，これを用いることで親個体間の中から一様に子個体を取り出せることに加え，範囲の外からも子個体が取り出されることがあるため，個体群の多様性を保つことができると考え使用した．突然変異にはガウス突然変異を使用した．これは遺伝子の要素を平均とする正規分布に従うように乱数で変化させるものでこれを用いることで局所解にはまってしまうことを防ぐができると考えられる．

Algorithm 1 Genetic Algorithm

Require: cxpb = 交叉率,mutpb = 突然変異率,endgeneration = 最終世代数,population = 個体集団

Ensure:

```
for generation = 1 to endgeneration do
  calcFitness(population)
  for _=1 to population/2 do
    parent1,parent2 = select(population)
    if cxpb then
      child1, child2 = crossover(parant1, parent2)
      population.remove(parent1, parent2)
      population.append(child1, child2)
    end if
  end for
  for mutant in population do
    if mutpb then
      mutation(mutant)
    end if
  end for
end for
```

2.2.2 進化戦略

進化戦略とは，個体が探索点のみを持つGAとは違い，個体が探索点，突然変異パラメータ，調整パラメータを持ち突然変異を主に使うことで最適解を探索するものである．今回は新しく生成したλ個の個体から上位μ個を選び次の個体群の生成に利用する(μ, λ)-ES(Algorithm 2)を用いた．また，補助的な探索に使う交叉の方法はGAの時と同じBLX-αを使用した．また，突然変異については突然変異パラメータ(σ), 調整パラメータは(α)以下のような操作を行った．

$$\sigma_i^{'} = \sigma_i \exp(\sqrt{2n}^{-1} + \sqrt{2\sqrt{n}}^{-1} \xi_i)$$

$$\alpha_{i,j}^{'} = \alpha_{i,j} + 0.0873\xi_{i,j}$$

このときの，ξはすべて独立した標準正規分布．探索点xの突然変異には

$$r_{i,i} = r_{j,j} = cos\alpha_{i,j}$$

$$r_{i,j} = -r_{j,i} = -sin\alpha_{i,j}$$

$$r_{k,k} = 1(k = 1 \cdots n \text{ かつ } k \neq i, k \neq j)$$

を要素に持つ行列$R(\alpha_{i,j})$に対して，正規分布$N(0, \sigma_i^2)$に従う正規乱数である$\eta = (\eta_1, \eta_2, \cdots \eta_n)$をかけ合わせたベクトル

$$\eta = \prod_{i=1}^{n-1} \prod_{j=i+1}^{n} R(\alpha_{i,j}\eta)$$

をもともとの探索点との和をとった．

$$x^{'} = x + \eta$$

Algorithm 2 Evolution Strategy

Require:

Ensure:

```
for generation = 1 to endgeneration do
    population = generation()
    calcFitness(population)
    children = generateChildren(population)
    population.append(children)
    sorted(population)
    nextPopulation = select(population)
    population = nextPopulation
end for
```

2.2.3 共分散進化戦略

共分散進化戦略とは ES に共分散行列を適応させたもので，個体群が持つ平均ベクトル，ステップサイズ，共分散行列から個体を生成し，生成した個体の評価を元に個体を選択，選択した個体を元に平均ベクトル (m)，ステップサイズ (σ)，共分散行列 (C) を更新するという流れを繰り返すことにより最適解を求めるというものである．平均ベクトルの更新は以下のようにして評価値の高い個体には高い重みが割り当てられて更新される．

$$m^{g+1} = \sum_{i=1}^{\mu} w_i x_{i:\lambda}^{g+1}, where \sum_{i=1}^{\mu} w_i = 1, w_i > 0$$

このときの $x_i i:\lambda$ は λ 個の個体の内 i 番目に良いものを表す．

ステップサイズと共分散行列の更新は以下のようにして行われる．

$$p_\sigma^{(g+1)} = (1-c_\sigma)p_\sigma^{(g)} + \sqrt{c_\sigma(2-c_\sigma)\mu_w}\sqrt{C^g}^{-1}\sum_{i=1}^{\lambda} w_i x_{i:\lambda}$$

$$\sigma^{g+1} = \sigma^g \exp(\frac{c_\sigma}{d_\sigma}(\frac{||p_\sigma^{g+1}||}{E||N(0,\boldsymbol{I})||} - 1))$$

$$p_c^{(g+1)} = (1-c_c)p_c^t + \sqrt{(c_c(2-c_c)\mu_w)}\sum_{i=1}^{\lambda} w_i x_{i:\lambda}$$

$$C^{(g+1)} = C^g + c_1[OP((p_c - (g+1)) - C^t] +$$

$$c_\mu \sum_{i=1}^{\lambda} w_i [OP(y_{i:\lambda} - C^t]$$

$OP(\boldsymbol{v}) = \boldsymbol{v}\boldsymbol{v}^T$,$c_1$,$c_\mu$ は rank-one,rank-μ 更新の学習率

2.2.4 粒子群最適化

粒子群最適化とは各粒子の集まりを群としたときに各粒子の位置情報，速度の他に群の中の他の粒子の情報を使って最適解を求めるというものである．

$$X_i(g+1) = X_i(g) + V_i(g+1)$$

$$V_i(g+1) = wV_i(g) + c_1 r_1(X_i^{pbest}(g) - X_i(g)) +$$

$$c_2 r_2(X^{pbest}(g) - X_i(g))$$

ここでの $X_i(g)$,$V_i(g)$ はそれぞれ，粒子の位置，速度であり，$X_i^{pbest}(g)$ は世代 g までの粒子の最良解であり $X_i^{pbest}(g)$ は世代 g までの粒子群全体の最良解を表している．この時の $wV_i(g)$ は慣性項の役割を持っていてこれを用いることで局所探索を防ぐことができる．また，c_1,c_2 では個人の粒子の最適解か粒子群全体の最適解のどちらを重要視するかの係数の設定を行うことができ，今回はどちらも 0.5 とすることで等しくした．

2.3 結果

静的な関数最適化の結果は表 2 のようになった．

表 1: 静的関数

関数・手法	GA	ES	CMA-ES	PSO
Sphere	496	15	500	0
Double-Cone	44	0	0	0
Himmelblau	484	17	500	6
Griewank	136	0	500	0

これを見ると，CMA-ES はひとつの局所解を持つ Double-Cone 関数以外では非常に良好な結果を示している．GA は局所解を多数持つ多峰生関数である griewank 関数で CMA-ES ほど良い結果ではないが，Double-Cone 関数では今回用いた方法の中で一番良い結果を示した．一方 ES,PSO はあまり最適解を求めることができなかった．
動的な関数最適化の結果は表 3 のようになった．

表 2: 動的関数

手法	GA	CMA-ES	PSO
E	11.67	0.08	0.44

$$E = \frac{\sum_{g=1}^{n}(y_g - f(x_g))^2}{n}$$

y_gは世代 g での最適解, x_gは世代 g での最良解, $g = 1 \cdots 100$

これを見ると,CMA-ES,PSO,GA の順番で結果が良かった．また，図 7, 図 8, 図 9, 図 10 が各手法がどのようにして最適化されていったかを表すグラフである．

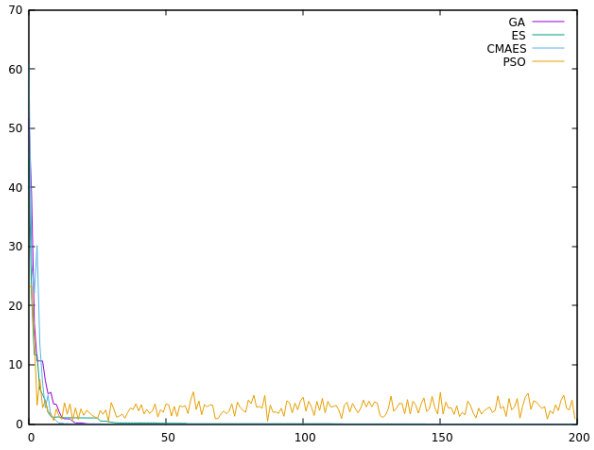

図 7: Sphere_graph

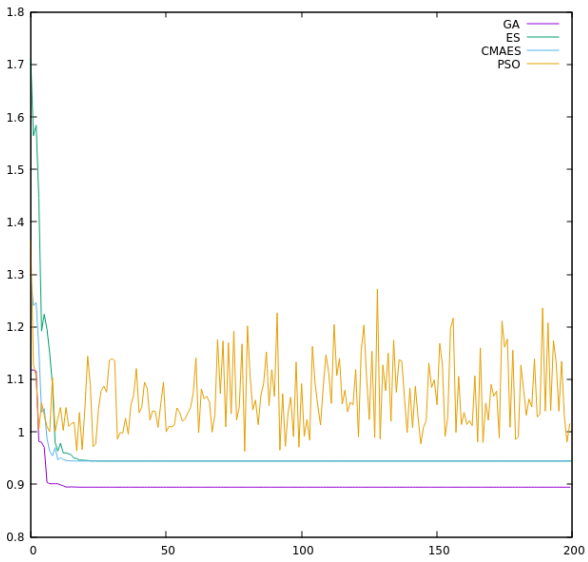

図 8: Double-Cone_graph

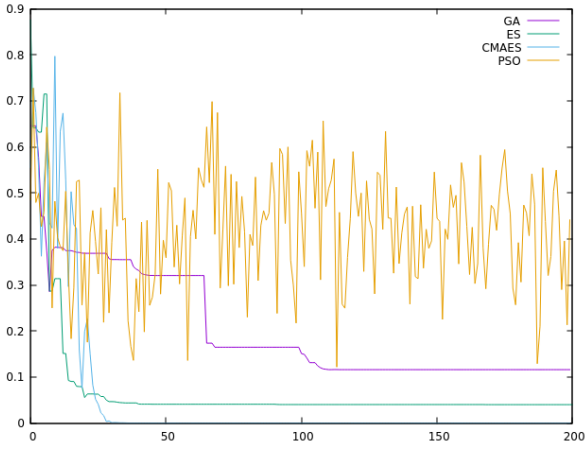

図 9: griewank_graph

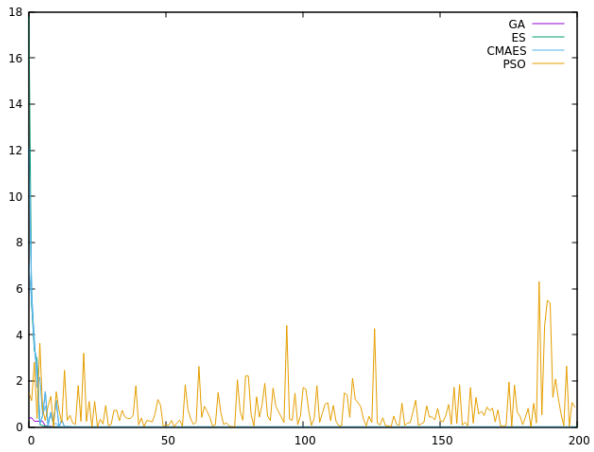

図 10: himmelblau_graph

3　制御タスク

3.1　問題設定

制御タスクに対して，最終世代までに求められた最良解の評価値を比較した．この時，各手法の個体数を 250 個, 最終世代数を 300 世代に統一した．制御タスク問題としては OpenAI が提供している OpenAIGym の中の CartPole-Balancing[図 11],BipedalWalker[図 12] 問題を選んだ．

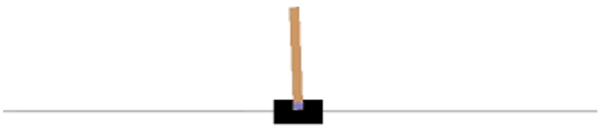

図 11: CartPole-Balancing

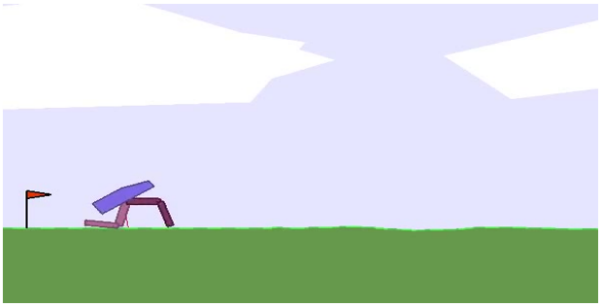

図 12: BipedalWalker

CartPole-Balancing は台車に乗った振り子が倒れないように制御するもので，棒が一定の角度以上に傾くか，台車が範囲外に移動した場合に失敗になる．また，1 ステップごとに報酬を 1 得ることができ 5000 ステップで強制終了となる．この問題に対する入力は台車を左に押すか右に押すかの 2 個で，状態は台車の位置，ポールの角度，台車の速度，ポールの角速度の 4 個である．
BipedalWalker は起伏のある地形を二足歩行ロボットが歩けるように制御するもので，ロボットが倒れるか，一定ステップ経過するか，300 進むと強制終了となる．また，ロボットが 1 進むと 1 の報酬を得ることができ，ロボットが倒れると-100 の報酬を得る．この問題に対する入力はそれぞれ二つの足．太ももに対するトルクの 4 つであり，状態は 2 つの足，太ももの角度やスピード，地面との設置の有無，目の前の地形の 24 個である．CartPole-Balancing は 4 次元の状態から左に押すか右に押すかの 2 つの離散値を選ぶ比較的単純な問題であり，BipedalWalker は 24 次元の状態から 4 次元の-1 から 1 までの連続値を選ぶという CartPole-Balancing に比べて複雑な問題である．また，BipedalWalker は毎回地形の起伏が異なるという点で各手法がどれぐらい外乱に影響されずに最適化できるかを比較できると考えられる．
今回は GA，CMA-ES,GP,NEAT を用いて比較をした．

3.2 アルゴリズム

3.2.1 遺伝的アルゴリズム

GA は関数を直接進化させる GP,NEAT と違い，実数の配列を遺伝子とした個体を進化させる方法であるため，人口ニューラルネットワーク (ANN) をコントローラとし，各ノード間の重みを遺伝子として持たせた．入力と出力の次元の関係から CartPole-Valancing では入力層が 4, 隠れ層が 3, 出力層が 1 の 3 層 ANN を実装し，出力値が 0.5 以上なら右に押す，出力値が 0.5 よりも小さいなら左に押すようにした．BipedalWalker では入力層が 24, 隠れ層が 16, 出力層が 4 の 3 層 ANN を実装し，出力値においては-1 から 1 の値に clip した．選択，交叉，突然変異については前章の関数最適化のときの GA と同じくトーナメント選択，BLX-α, ガウス突然変異を用いた．しかし，今回の問題は関数最適化問題と違い探索空間が広いため，突然変異率を 0.05 から 0.1 に変更した．

3.2.2 共分散進化戦略

CMA-ES においても GA と同様の 3 層 ANN をコントローラとした．

3.2.3 遺伝的プログラミング

遺伝的プログラミング (GP) は配列を遺伝子としている GA と違い，木構造を遺伝子としている．それ以外の流れは GA と同じである．

Algorithm 3 Genetic Programming

Require: cxpb = 交叉率,mutpb = 突然変異率,endgeneration = 最終世代数,population = 個体集団
Ensure:
  for *generation* = 1 to endgeneration **do**
    calcFitness(population)
    for _=1 to population/2 **do**
      parent1,parent2 = select(population)
      if cxpb **then**
        $child1, child2 = crossover(parant1, parent2)$
        $population.remove(parent1, parent2)$
        $population.append(child1, child2)$
      end if
    end for
    for mutant in population **do**
      if mutpb **then**
        mutation(mutant)
      end if
    end for
  end for

GP の選択には GA と同じくトーナメント選択を用いた．交叉は無作為に選択した個体の部分木を交換する一様交叉を用いた．突然変異には個体のランダムな位置に新たな木を挿入する mutInsert を用いた．また，CartPole-Balancing での Primitive 関数には add,sub,mul,SafeDiv,neg,cos,sin,-1,0,1 を用いた．BipedalWalker では探索範囲がさらに広くなるため，CartPole-Balancing の Primitive 関数に加え，sigmoid,relu を追加した．また，出力値が離散値で 1 次元である CartPole-Balaning に対して，BipedalWalker は出力値が連続値で 4 次元であるため，遺伝子の木構造に強い型付けを行い，木の根に 4 つの入力値をとり，入力値を-1 からに 1 に clip する end ノードを付け加えた．また，木が過剰に大きくなりメモリ不足に陥ってしまうことを防ぐために木の大きさを 17 に制限した．

3.2.4 NEAT

NEAT は ANN のノード間の重みに加えてトポロジーも変化させるというものである．NEAT は遺伝的アルゴリズムの遺伝子に NEAT は，ANN の重みだけを調整する上の二つと比べてトポロジーも変化させるため，これを使うことでより柔軟に解を表現するができる．また，NEAT では他の個体とは違う構造や重みをもつ個体が生まれたときは，その個体の評価値が低くても淘汰されないように保存をすること個体群の多様性を維持している．NEAT では，突然変異を用いてノードの追加・削除，トポロジーの追加・削除を行うが，この時のノードは全結合層と活性化関数を持っている．今回は活性化関数を CartPole-Balancing では出力値を 0.5 よりも大きいか小さいかで分類したため，sigmoid,BipedalWalker では-1 から 1 の範囲にするために tanh を使用した．その他の詳しいハイパーパラメータは以下のプログラムに示した．

ソースコード 2: CartPole

```
[NEAT]
fitness_criterion = max
fitness_threshold = 5000
pop_size = 250
reset_on_extinction = 0
no_fitness_termination = 1
[DefaultGenome]
activation_default = sigmoid
activation_mutate_rate = 0.01
activation_options = sigmoid
aggregation_default = sum
aggregation_mutate_rate = 0
aggregation_options = sum
bias_init_mean = 0
bias_init_stdev = 1.0
bias_max_value = 30.0
bias_min_value = -30.0
bias_mutate_power = 0.5
bias_mutate_rate = 0.7
bias_replace_rate = 0.1
compatibility_disjoint_coefficient =
    1.0
compatibility_weight_coefficient = 0.6
conn_add_prob = 0.2
conn_delete_prob = 0.2
enabled_default = True
enabled_mutate_rate = 0.01
feed_forward = False
initial_connection = partial_direct
    0.5
node_add_prob = 0.2
node_delete_prob = 0.2
num_hidden = 1
num_inputs = 4
num_outputs = 1
response_init_mean = 0
response_init_stdev = 1
response_max_value = 30.0
response_min_value = -30.0
response_mutate_power = 0
response_mutate_rate = 0
response_replace_rate = 0
weight_init_mean = 0
weight_init_stdev = 1
weight_max_value = 30
weight_min_value = -30
weight_mutate_power = 0.5
weight_mutate_rate = 0.8
weight_replace_rate = 0.1
[DefaultSpeciesSet]
compatibility_threshold = 3.0
[DefaultStagnation]
species_fitness_func = max
max_stagnation = 20
species_elitism = 5
[DefaultReproduction]
elitism = 10
survival_threshold = 0.2
```

ソースコード 3: BipedalWalker

```
[NEAT]
fitness_criterion = max
fitness_threshold = 300
pop_size = 250
reset_on_extinction = 0
no_fitness_termination = 1
[DefaultGenome]
activation_default = tanh
```

```
activation_mutate_rate = 0.01
activation_options = tanh
aggregation_default = sum
aggregation_mutate_rate = 0
aggregation_options = sum
bias_init_mean = 0
bias_init_stdev = 1.0
bias_max_value = 30.0
bias_min_value = -30.0
bias_mutate_power = 0.5
bias_mutate_rate = 0.7
bias_replace_rate = 0.1
compatibility_disjoint_coefficient =
    1.0
compatibility_weight_coefficient = 0.6
conn_add_prob = 0.2
conn_delete_prob = 0.2
enabled_default = True
enabled_mutate_rate = 0.01
feed_forward = False
initial_connection = partial_direct
    0.5
node_add_prob = 0.2
node_delete_prob = 0.2
num_hidden = 1
num_inputs = 24
num_outputs = 4
response_init_mean = 0
response_init_stdev = 1
response_max_value = 30.0
response_min_value = -30.0
response_mutate_power = 0
response_mutate_rate = 0
response_replace_rate = 0
weight_init_mean = 0
weight_init_stdev = 1
weight_max_value = 30
weight_min_value = -30
weight_mutate_power = 0.5
weight_mutate_rate = 0.8
weight_replace_rate = 0.1
[DefaultSpeciesSet]
compatibility_threshold = 3.0
[DefaultStagnation]
species_fitness_func = max
max_stagnation = 20
species_elitism = 3
[DefaultReproduction]
elitism = 3
survival_threshold = 0.2
```

また，CartPole-Balancinig 問題に対する NEAT による進化の様子を 3.3.2 に示した．この時の灰色の四角形で囲われている-1,-2,-3,-4 はそれぞれ x 座標，速度，棒の角度，棒の角速度の状態の入力を表していて，白いまるで囲われているものが中間層，青いまるで囲われているものが出力層を表している．

3.3 結果

CartPole-Balancing,BipealWalker に関する結果は図 13, 図 14 のようになった．

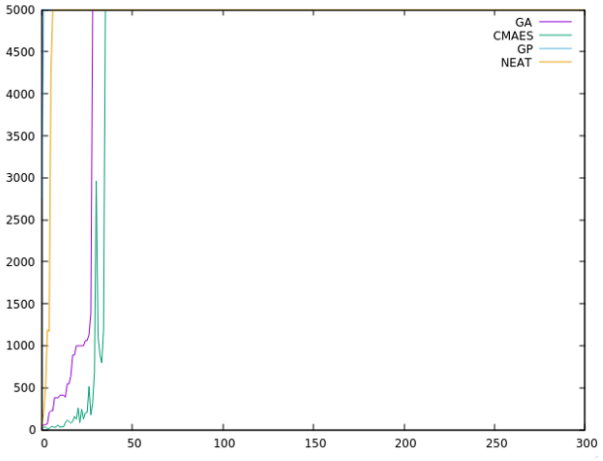

図 13: CartPole-Balancing

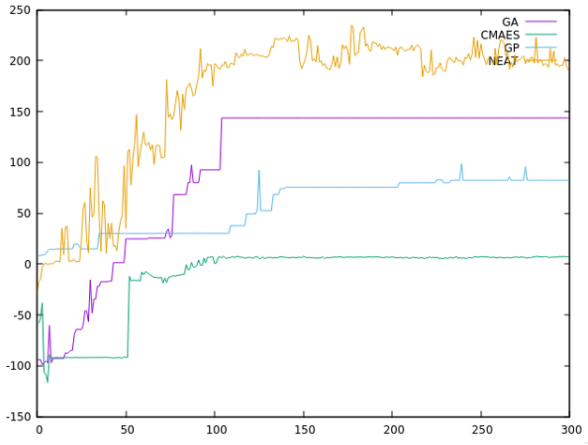

図 14: BipedalWalker

これを見ると，CartPole-Balancing ではどの手法も比較的早い段階で最適解を求められていることがわかる．また BipedalWalker では NEAT,GA,CMA-ES,GP の順番でよい解を得られていることがわかる．

3.3.1 GP によって得られた木

図 15,16 は GP によって得られた最良解である木のグラフである．

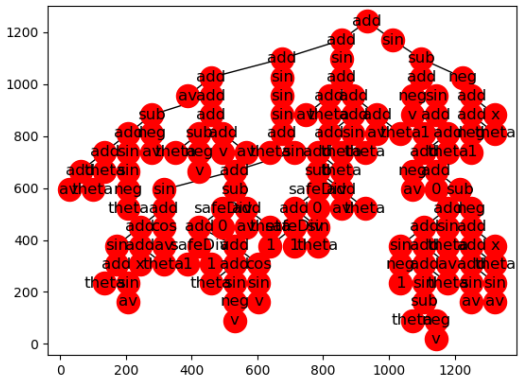

図 15: CartPole

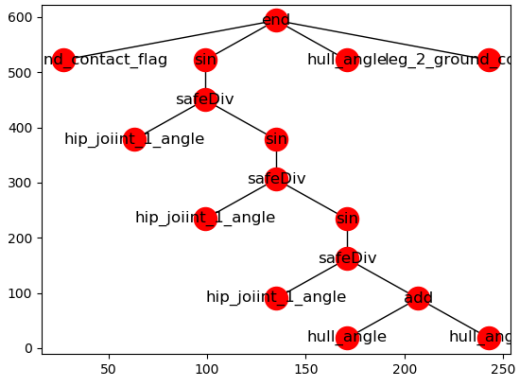

図 16: BipedalWalker

3.3.2 NEAT における進化の様子

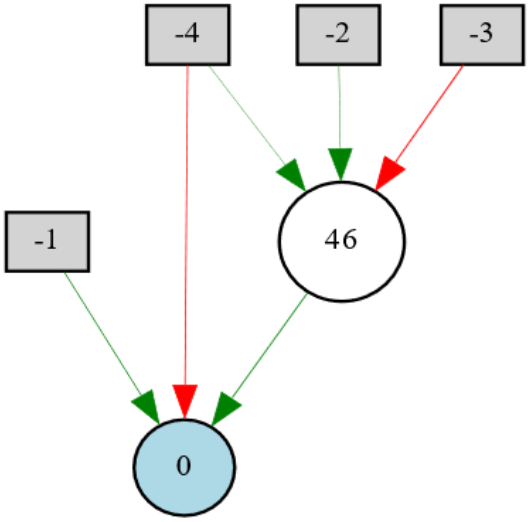

図 17: CartPole:0 世代

この時の評価値は 140 であった．

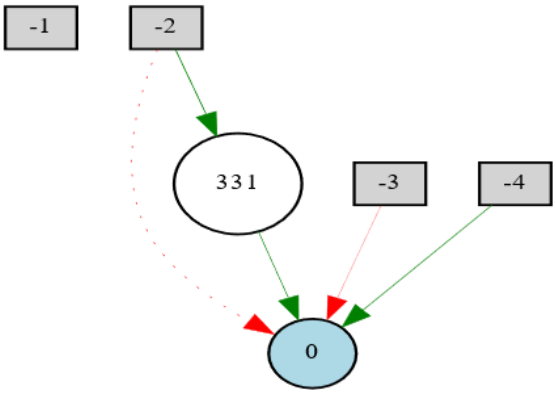

図 18: CartPole:2 世代

この時の評価値は 173 であった.

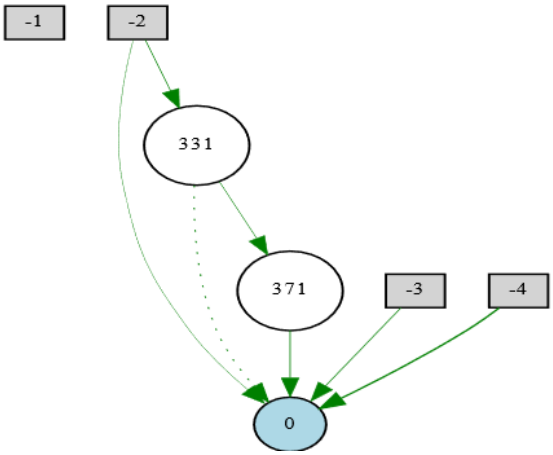

図 19: CartPole:10 世代

この時の評価値は 1786 だった.

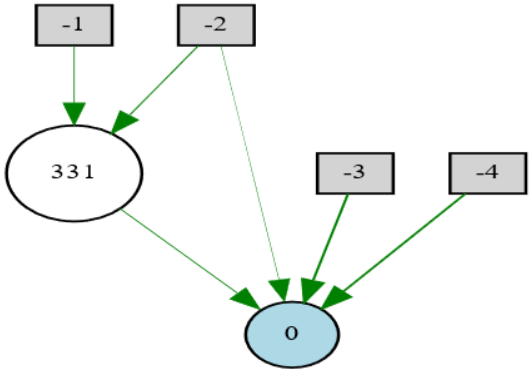

図 20: CartPole:300 世代

この時の評価値は 5000 だった.
以下は BipedalWalker での最良解のネットワークである.

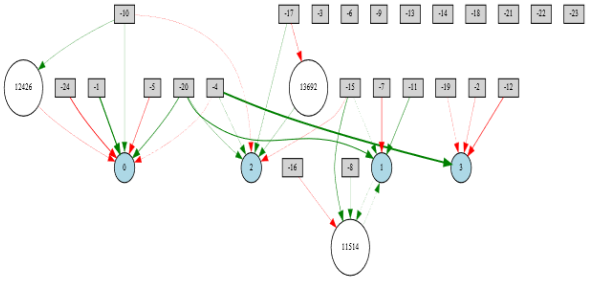

図 21: BipedalWalker:300 世代

この時の評価値は 204 だった.

4 考察

4.1 関数最適化

まず，静的な関数最適化について，GA に関しては CMA-ES 程ではないが比較的多くの回数最適解を求めることができており，Double-Cone 関

数においては今回の実験では一番多く最適解を求めることができている．CMA-ES は Double-Cone 関数以外の関数においては，すべての実験で最適解を求めることができた．しかし，Double-Cone 関数で最適解を 1 度も求めることができなかった．この理由としては CMA-ES の初期ベクトルを $\mathbf{0}$, 初期の共分散行列を $[5.05.0]^T$ と設定したため，初期解が $\mathbf{0}$ 付近に集まってしまい初期解付近にある局所解にすぐにはまってしまったからだと考えた．そのため，後に共分散行列の初期値を変更した．その結果，500 回すべて最適解を求めることができた．

ES,PSO に関してはこの実験ではよい結果が出なかった．これは GA や CMA-ES に比べて解の収束が遅いために 200 世代では最適解を求められなかった，または関数最適化における ES，PSO の初期値や進化方法の設定がうまくいっていなかったことが理由として考えられる．

また，GA による静的な関数の最適化において，局所解にはまってしまうことを回避することを狙いとして，現在の探索点がどれぐらい珍しいかを評価値として採用してみた．評価の方法には，今までの探索点をすべて記録しておく Novelty Map[図 22] と k 個の点からどれぐらい離れているかを表す最近傍法を用いた．

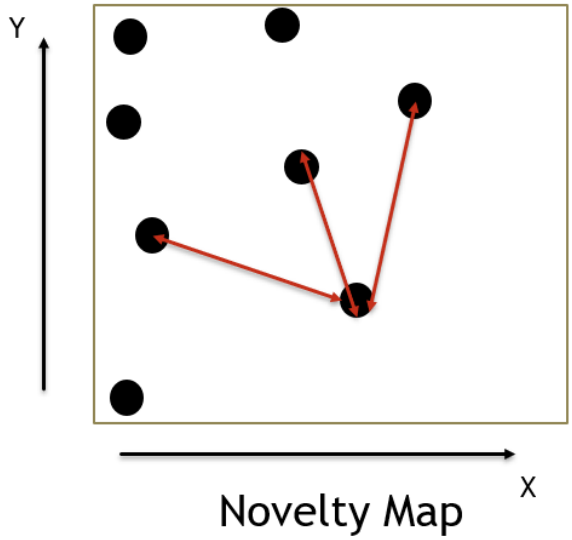

図 22: NoveltyMap

しかし，この評価方法では最適解までは求めるきることができなかったため，通常は今まで通りの評価値で 8 世代に 1 回ごとにこの評価方法を使用した．これを用いて，再び実験を行った結果が以下である．

表 3: 遺伝的アルゴリズム

関数・手法	GA	GA with Novelty
Sphere	496	491
Double-Cone	44	67
Himmelblau	484	474
Griewank	136	135

また，BipedalWalker にてこの評価方法を用いた結果が以下である．BipedalWalker では Novelty の評価値をその個体が個体群の中央値の個体に比べてどれぐらい離れているかで測った．

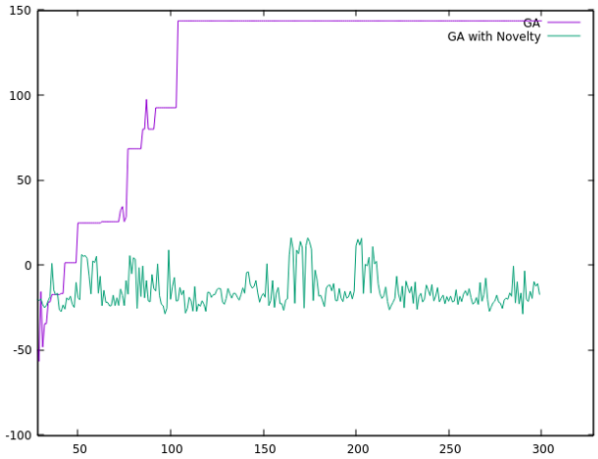

図 23: NoveltyMap

これを見ると，Double-Cone 関数において最適解を求められた回数が増加させることができていることがわかる．しかし，BipedalWalker においては最適化がうまく進まずに逆に結果が悪くなってしまった．

動的な関数最適化について，CMA-ES,PSO,GA の順番で結果が良かった．GA の結果があまり良くなかった原因として，一度収束して分散が小さくなってしまったために解の探索範囲が狭まってしまい，かつ分散が小さくなってしまった個体群の分散を再び大きくすることができなかったために，最適解の位置が移り変わる動的な関数に適応できなかったからであると推測した．個体群の分散を再び広げられなかった要因として，交叉で用いた BLX-α がある．これは，前章で説明した通り，2 つの親個体からなる超直方体の内部もしくは周辺に子個体を生成するというものであるために，親個体よりも離れた位置の探索ができなかったことが上げられる．また，突然変異率も結果に大きく関わってくることが分かった．最初は突然変異率を 0.01 にしていたが，解の探索範囲を広げることを狙いとして 0.1 に上げた結果，誤差の平均は 26.45 から 11.67 に下がった．しかし，あまり突然変異率を上げてしまうと解が全く収束しなくなってしまうことを懸念して 0.1 以上には上げなかった．一方，CMA-ES は平均ベクトルに加えて共分散行列も更新される．このため，一度収束した個体群も再び分散を大きくし解の探索範囲を広げられているため，良好な結果が出たと推測した．PSO については，なかなか解が収束せず常に分散が広い状態だった．そのため，どの世代でもあまり誤差が大きくならずに GA よりも上，CMA-ES よりも下の結果になった．これらの考察は各世代における解の分布の様子を見るとよくわかる．以下の図のほかに，GitHub の方に各世代での解の分布を繋げた gif をのせておく．

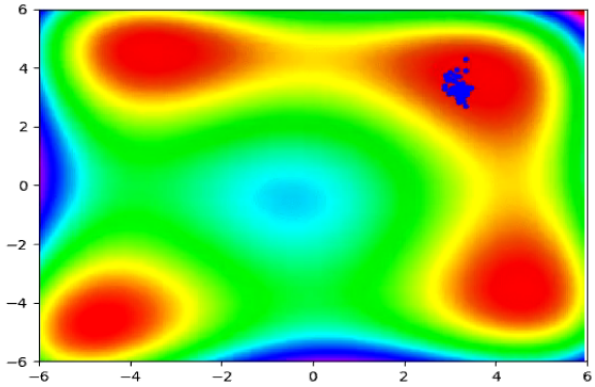

図 24: GA1

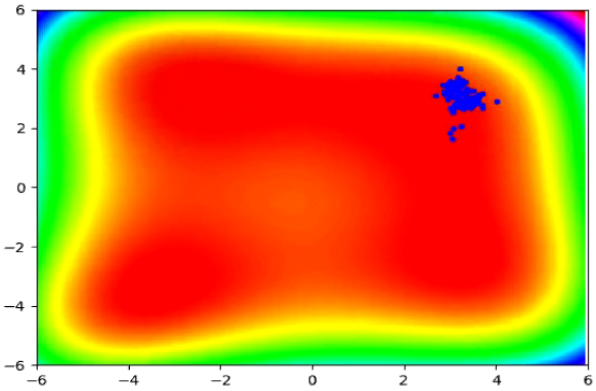

図 25: GA2

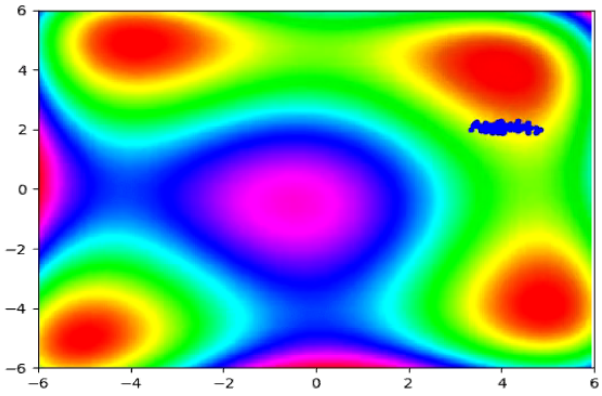

図 26: GA3

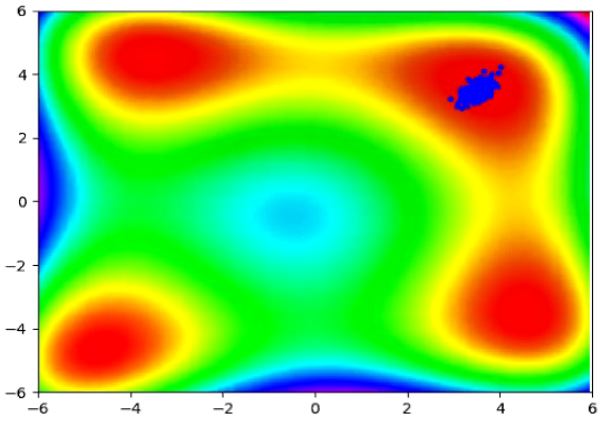

図 27: CMA-ES1

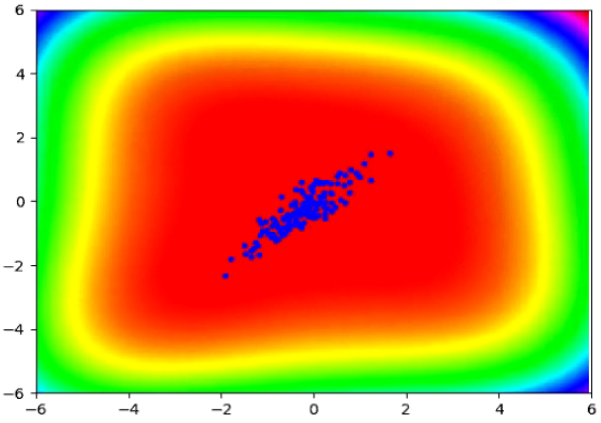

図 28: CMA-ES2

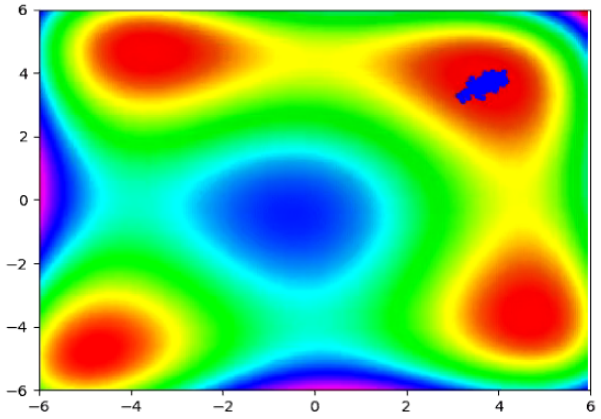

図 29: CMA-ES3

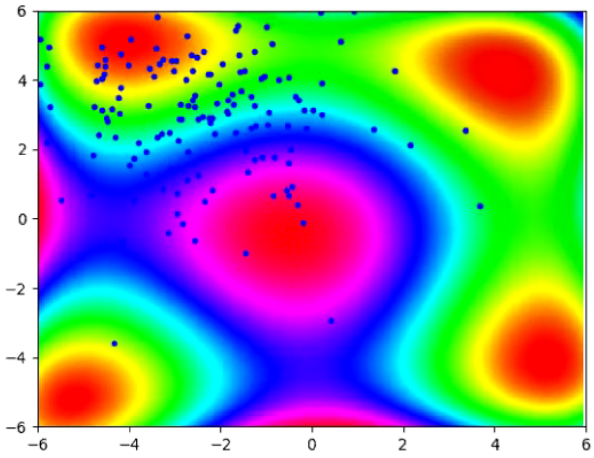

図 30: PSO1

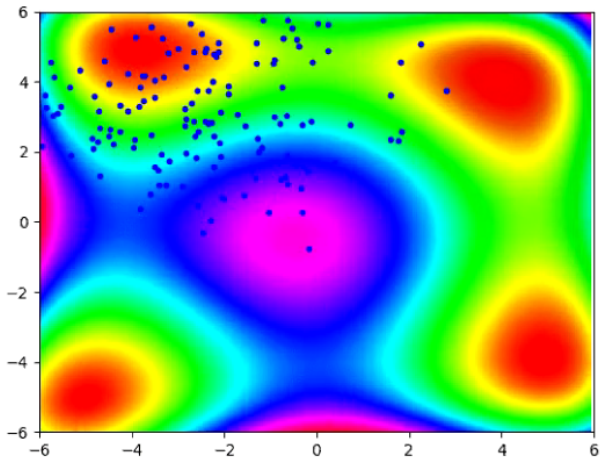

図 32: PSO3

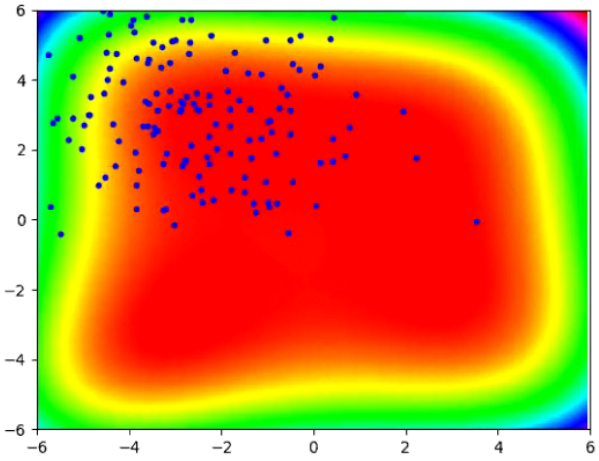

図 31: PSO2

4.2 制御タスク

比較的簡単な制御タスクである CartPole-Balancing についてはどの手法も早い段階で最適解を求めることができている．その中でも GP が一番早くに最適解を求めることができた．逆に Bipedal-Walker では GP が評価値が一番低くなった．そのため，GP では複雑な問題に適応することが難しいことがわかった．

また，関数最適化において良好な結果を示した CMA-ES が良い結果を示せなかった原因として，問題の次元数が関数最適化などに比べて大幅に大きく (468 次元)CMA-ES が最適化をするのに必要な最低個体数に満たさなかったためだと推測した．GA においては，交叉の方法を変数間に強い依存関係があるため，SPX 交叉や UNDX 交叉を用いればさらに結果が良くなると考えられる．

NEAT はどちらの制御タスクも良好な結果を示すことができたが，調整しなければならないパラメータ数が多くその調整に苦労した．例えば，NEAT は小さい ANN からだんだんと探索範囲を広げるようにして最適化をしていくというものであるため，NEAT の初期構造をトポロジーも 1 つも持たない構造にしていたが，これでは上手く最適化が

できなかった．これは初期個体の多様性が一切ないことが問題であると推測した．そのため，初期構造を全結合状態のトポロジーからランダムで半分に減らしたものを用いた結果最適化がうまく進んだ．また，収束を早めることを目的として突然変異によりノード，トポロジーを追加する確率を0.6 ほどにしていたが，最適化が進まなかった．これは世代が進むにつれて ANN の構造が過剰に複雑になってしまうことが原因だと考えた．

5　まとめ

5.1　成果

今回の情報特別演習を通し，進化的計算における各手法の基本的なアルゴリズムや一部の交叉・突然変異アルゴリズムの目的や実装方法，目的各問題に対する最適な調整方法の理解，また，関数最適化・制御タスクに対する一部の手法を最適解を求められた回数・最良解の評価値という観点からの比較を行うことができた．
しかし，良い比較方法が思い浮かばずに実験を行えなかった実行速度や収束までの早さの比較や実装がうまくいかずに試すことができなかった手法や世代交代モデルが存在するため，今後の課題としたい．また，今後は今回の実験で比較的良好な結果を示した GA や CMA-ES,NEAT のアルゴリズムの理解をさらに深め，今回の実験よりも大きい問題の最適化を試みたい．

6　謝辞

本研究を進めるに当たり，ご指導をいただいた指導教官のアランニャ クラウス　デ　カステロ助教に厚く感謝を申し上げます．また，実験のシミュレーション環境などの便利なライブラリを提供している OpenAI や deap に深く感謝いたします．

7　参考文献

1. 知的システムデザイン研究室 GA グループ　卒論・修論作成のための基礎シリーズ　遺伝的アルゴリズム
2.DAN SIMON EVOLUTIONARY OPTIMIZATION ALGORITHMS
3. 樋口　隆英　実数値 GA におけるシンプレクス交叉の提案
4. 秋本　洋平　Evolution Strategies による連続最適化
5.http://www.sist.ac.jp/ kanakubo/research/evolutionary_computing/lbx_spx.html 実数型 GA に於ける交叉法の改良
6.Nikolaus Hansen The CMA Evolution Strategy:A Comparing Review
7.Kenneth O.Stanley Evolving Neural Networks through Augmenting Topologies
8. 喜多　一　実数値 GA の ための正規分布交叉の多数の親 を用いた拡張法の提案
9. 松村　嘉之　進化戦略における事故適応の拡張に関する研究
10.https://www.msi.co.jp/s4/introduction/pso.html 粒子群最適化
11.　Joel Lehman Revising the Evolutionary Computation Abstraction:　Minimal Criteria Novelty Search